Contents

Preface xxxv
Who This Book Is For xxxv
Organization of This Book xxxvi
Orchestrate-Inserted Partition and Sort Components xxxvi
Typographic Conventions xxxvii
Syntax Conventions xxxviii
The Orchestrate Documentation Set xxxviii
Using the Adobe Acrobat Reader xxxix
Searching for Text in Orchestrate Documents xxxix
Assistance and Additional Information xl

PART 1 The General Library

Chapter 1 The changeapply Operator 1 1
Data Flow Diagram 1 2
Properties 1 2
Schemas 1 4
Transfer Behavior 1 4
Key Comparison Fields 1 4
Syntax and Options 1 5
Example 1 8

Chapter 2 The changecapture Operator 2 1
Data Flow Diagram 2 2
Chapter 6  The encode Operator  6 1

  Data Flow Diagram  6 2
  Properties  6 2
  Syntax and Options  6 3
  Encoding Orchestrate Data Sets  6 4
    Processing Encoded Data Sets  6 4
    Encoded Data Sets and Partitioning  6 4
  Example  6 5

Chapter 7  The filter Operator  7 1

  Data Flow Diagram  7 2
  Properties  7 2
  Syntax and Options  7 3
  Job Monitoring Information  7 5
    Example MetaData and Summary Messages  7 5
    Customizing Job Monitor Messages  7 6
  Expressions  7 6
    Input Data Types  7 6
    Supported Boolean Expressions and Operators  7 7
  Example 1: Comparing Two Fields  7 8
  Example 2: Testing for a Null  7 8
  Example 3: Evaluating Input Records  7 8
  Application Scenario: Mailing List for a Wine Auction  7 9
    Schema for Implicit Import  7 9
    OSH Syntax  7 9
    Input Data  7 10
    Outputs  7 10

Chapter 8  The Funnel Operators  8 1

  Data Flow Diagram  8 2
  Properties  8 2
  funnel Operator  8 3
    Default Input Sequencing  8 3
    Non-Deterministic Input Sequencing  8 4
    Syntax and Option  8 4
  sortfunnel Operator  8 5
# Contents

## Input Requirements
- Primary and Secondary Keys 8 5
- Syntax and Option 8 6

## Chapter 9 The generator Operator 9 1
- Data Flow Diagram 9 2
- Properties 9 2
- Syntax and Options 9 3
- Using the generator Operator 9 4
  - Supported Data Types 9 4
  - Example 1: Using the generator Operator 9 5
  - Example 2: Executing the Operator in Parallel 9 6
  - Example 3: Using generator with an Input Data Set 9 6
- Defining the Schema for the Operator 9 7

## Chapter 10 The head Operator 10 1
- Data Flow Diagram 10 2
- Properties 10 2
- Syntax and Options 10 2
- Examples 10 3
  - Example 1: head Operator Default Behavior 10 3
  - Example 2: Extracting Records from a Large Data Set 10 4
  - Example 3: Locating a Single Record 10 4

## Chapter 11 The lookup Operator 11 1
- Data Flow Diagrams 11 3
  - Create-Only Mode 11 3
  - Lookup Mode 11 3
- Properties 11 4
- Syntax and Options 11 5
- Lookup Table Characteristics 11 7
- Partitioning 11 8
- Create-Only Mode 11 8
- Examples 11 9
  - Single Lookup Table Record 11 9
  - Multiple Lookup Table Records 11 10
  - Interest Rate Lookup Example 11 11
Contents

Handling Duplicate Fields Example 11 12

Chapter 12 The merge Operator 12 1

Data Flow Diagram 12 2
Properties 12 3
Syntax and Options 12 4
Merging Records 12 5
  Merging with a Single Update Data Set 12 6
  Merging With Multiple Update Data Sets 12 7
Understanding the merge Operator 12 8
  The Merging Operation 12 9
  Example: Updating National Data with State Data 12 11
  Example: Handling Duplicate Fields 12 14
  Application Scenario: Galactic Industries 12 16
  Missing Records 12 18

Chapter 13 The modify Operator 13 1

Data Flow Diagram 13 3
Properties 13 3
Syntax and Options 13 3
Transfer Behavior 13 4
Performing Conversions 13 5
  Keeping and Dropping Fields 13 6
  Renaming Fields 13 6
  Duplicating a Field and Giving It a New Name 13 6
  Changing a Field’s Data Type 13 7
  Default Data Type Conversion 13 7
  Data Type Conversion Errors 13 9
  How to Convert a Data Type 13 10
  Date Field Conversions 13 10
date Formats 13 13
  Decimal Field Conversions 13 14
  Raw Field Length Extraction 13 17
  String and Ustring Field Conversions 13 18
  String Conversions and Lookup Tables 13 23
  Time Field Conversions 13 26
time Formats 13 27
  Timestamp Field Conversions 13 29
timestamp Formats 13 30
  The modify Operator and Nulls 13 31
  The modify Operator and Partial Schemas 13 35
Contents

Example Sampling of a Data Set  17 4

Chapter 18 The sequence Operator  18 1
  Data Flow Diagram  18 2
  Properties  18 2
  Syntax and Options  18 3
  Using the sequence Operator  18 3

Chapter 19 The switch Operator  19 1
  Data Flow Diagram  19 2
  Properties  19 3
  Syntax and Options  19 3
  Using the switch Operator  19 6
  Job Monitoring Information  19 8
    Example MetaData and Summary Messages  19 8
    Customizing Job Monitor Messages  19 9

Chapter 20 The tail Operator  20 1
  Data Flow Diagram  20 2
  Properties  20 2
  Syntax and Options  20 2
  Examples  20 3
    Example 1: tail Operator Default Behavior  20 3
    Example 2: tail Operator with Both Options  20 4

Chapter 21 The transform Operator  21 1
  Running Your Job on a non-NFS MPP  21 4
  Data Flow Diagram  21 4
  Properties  21 5
  Syntax and Options  21 5
  Transfer Behavior  21 12
    The Transformation Language  21 14
      General structure  21 14
      Names and Keywords  21 14
      External Global C-Function Support  21 14
      Code Segmentation Keywords  21 16
PART 2 The Utility Operators

Chapter 22 Import/Export Introduction 22 1

Introduction 22 2

Record Schemas 22 3
  Example 1: Import Schema 22 3
  Example 2: Export Schema 22 4

Field and Record Properties 22 5

Complete and Partial Schemas 22 7
  Defining Partial Record Schemas 22 7
  Exporting with Partial Record Schemas 22 9

Implicit Import and Export 22 11
  Implicit Import/Export Operations with No Schemas Specified 22 11
  The Default Import Schema 22 12
  The Default Export Schema 22 13
Overriding the Defaults 22 13
Error Handling during Import/Export 22 14
  Failure 22 14
  Warning 22 15
ASCII and EBCDIC Conversion Tables 22 15
  EBCDIC to ASCII 22 16
  ASCII to EBCDIC 22 20
Conversion Table Irregularities 22 22

Chapter 23 The import Operator 23 1
Data Flow Diagram 23 2
Properties 23 2
Syntax and Options 23 3
How to Import Data 23 9
  Specify the Layout of the Imported Data 23 9
  Specify the Source of the Imported Data 23 9
Example 1: Importing from a Single Data File 23 12
Example 2: Importing from Multiple Data Files 23 13

Chapter 24 The export Operator 24 1
Data Flow Diagram 24 2
Properties 24 2
Syntax and Options 24 4
How to Export Data 24 9
  Specify Record and Field Layout of Exported Data 24 9
  Specifying Destination Files or Named Pipes for Exported Data 24 11
Example 1: Data Set Export to a Single File 24 14
Example 2: Data Set Export to Multiple Files 24 16

Chapter 25 Import/Export Properties 25 1
Setting Properties 25 3
Properties 25 5
  Record-Level Properties 25 5
  Field Properties 25 6
Properties: Reference Listing 25 14
  actual_length 25 17
  ascii 25 18
big_endian 25 19
binary 25 19
c_format 25 21
charset 25 22
check_intact 25 23
date_format 25 23
days_since 25 25
decimal_separator 25 25
default 25 26
default_date_format 25 26
default_time_format 25 27
delim 25 27
delim_string 25 28
don 25 30
ebcdic 25 31
export_ebcdic_as_ascii 25 31
fill 25 32
final_delim 25 32
final_delim_string 25 33
fix_zero 25 34
generate 25 34
import_ascii_as_ebcdic 25 35
in_format 25 36
intact 25 37
julian 25 38
link 25 38
little_endian 25 39
max_width 25 40
midnight_seconds 25 40
native_endian 25 41
nofix_zero 25 41
null_field 25 42
null_length 25 43
out_format 25 44
overpunch 25 45
packed 25 45
padchar 25 46
position 25 47
precision 25 49
prefix 25 50
print_field 25 51
quote 25 52
record_delim 25 53
record_delim_string 25 53
record_format 25 54
record_length 25 54
record_prefix 25 55
Chapter 26 COBOL Schema Conversion 26

Using the COBOL Schema Conversion Utility 26
Using the Generated Schema 26
Supported COBOL Data Storage Formats 26
Invoking readcobol 26
Handling COBOL Redefines 26

PART 3 The Operator Libraries

Chapter 27 The Partitioning Library 27

Overview of Partitioners 27
The entire Partitioner 27
Using the Partitioner 27
Data Flow Diagram 27
Properties 27
Syntax 27

The hash Partitioner 27
Specifying Hash Keys 27
Example 27
Using the Partitioner 27
Data Flow Diagram 27
Properties 27
Syntax and Option 27

The modulus Partitioner 27
Data Flow Diagram 27
Properties 27
Syntax and Option 27
Example 27
The field_export Operator 30 9
  Data Flow Diagram 30 10
  Properties 30 10
  Syntax and Options 30 10
  Example 30 11

The field_import Operator 30 13
  Data Flow Diagram 30 13
  Properties 30 14
  Syntax and Options 30 14
  Example 30 15

The makesubrec Operator 30 17
  Data Flow Diagram 30 17
  Properties 30 18
  Transfer Behavior 30 18
  Subrecord Length 30 19
  Syntax and Options 30 19
  Example 30 20

The makevect Operator 30 21
  Data Flow Diagram 30 22
  Properties 30 22
  Transfer Behavior 30 22
  Non-Consecutive Fields 30 23
  Syntax and Option 30 23
  Example 1: The makevect Operator 30 24
  Example 2: The makevect Operator with Missing Input Fields 30 24

The promotesubrec Operator 30 25
  Data Flow Diagram 30 26
  Properties 30 26
  Syntax and Option 30 26
  Example 30 27

The splitsubrec Operator 30 27
  Data Flow Diagram 30 28
  Properties 30 28
  Syntax and Option 30 29
  Example 30 29

The splitvect Operator 30 30
  Data Flow Diagram 30 31
  Properties 30 31
  Syntax and Option 30 32
  Example 30 32

The tagbatch Operator 30 33
  Tagged Fields and Operator Limitations 30 33
  Operator Action and Transfer Behavior 30 34
Chapter 31 The Sorting Library 31 1

Overview of the Sort Operators 31 2

The tsort Operator 31 2
  Configuring the tsort Operator 31 4
  Using a Sorted Data Set 31 4
  Specifying Sorting Keys 31 5
  Data Flow Diagram 31 7
  Properties 31 7
  Syntax and Options 31 8
  Example: Using a Sequential tsort Operator 31 11
  Example: Using a Parallel tsort Operator 31 12
  Performing a Total Sort 31 13
  Example: Performing a Total Sort 31 16

The psort Operator 31 17
  Introduction 31 17
  Performing a Partition Sort 31 18
  Configuring the Partition Sort Operator 31 19
  Using a Sorted Data Set 31 20
  Data Flow Diagram 31 22
  Properties 31 23
  Syntax and Options 31 24
  Example: Using a Sequential Partition Sort Operator 31 27
  Example: Using a Parallel Partition Sort Operator 31 27
  Performing a Total Sort 31 29
  Range Partitioning 31 30
Example: Performing a Total Sort 31

Chapter 32 The SAS Interface Library 32

Overview of the SAS Interface Library 32

Using Orchestrate to Run SAS Code 32
  Writing SAS Programs 32
  Using SAS on Sequential and Parallel Systems 32
  Pipeline Parallelism and SAS 32
  Configuring Your System to Use the SAS Interface Operators 32
  An Example Data Flow 32
  Representing SAS and Non-SAS Data in Orchestrate 32
  Getting Input from a SAS Data Set 32
  Getting Input from an Orchestrate Data Set or an Orchestrated SAS Data Set 32
  Converting between Data Set Types 32
  An Orchestrate Example 32
  Parallelizing SAS Steps 32
  Executing PROC Steps in Parallel 32
  Some Points to Consider in Parallelizing SAS code 32

Using Orchestrate to Do ETL 32

The SAS Interface Operators 32
  Specifying a Character Set and SAS Mode 32
  Parallel SAS Data Sets and SAS International 32
  Specifying an Output Schema 32
  Controlling ustring Truncation 32
  Generating a Proc Contents Report 32
  Orchestrate-Inserted Partition and Sort Components 32
  Long Name Support 32
  Environment Variables 32
  The sasin Operator 32
  The sas Operator 32
  The sasout Operator 32
  The sascontents Operator 32

Chapter 33 The Statistics Library 33

Overview of the Statistics Operators 33

The crosstab Operator 33
  Introduction 33
  Grouping Overview 33
  Data Flow Diagram 33
  Properties 33
  Syntax and Options 33
  Modes of Aggregation 33
# Contents

Specifying a Weighting Field  33 12  
Re-aggregating and Two-phase Grouping  33 13  
Limiting Memory Consumption  33 13  
Generating Statistics  33 13  
Example 1: Using the crosstab Operator without Clustered Keys  33 13  
Example 2: Using the crosstab Operator with Clustered Keys  33 14

The group Operator  33 16  
Introduction  33 16  
The Grouping Operation  33 17  
Defining a Group  33 18  
Creating a Summary  33 19  
Alternatives: The crosstab and statistics Operators  33 20  
Operator Action  33 21  
Data Flow Diagram  33 22  
Properties  33 22  
group Operator Modes  33 23  
Running the group Operator  33 23  
Syntax and Options  33 24  
Examples  33 29

The statistics Operator  33 38  
Statistics Calculation and Data Types  33 38  
Using the statistics Operator  33 39  
Data Flow Diagram  33 40  
Properties  33 40  
Syntax and Options  33 40  
Using the readstats Utility to Retrieve Statistics Results  33 41  
Syntax and Options  33 41  
Example: Using the Statistics Tools  33 43  
Calculating Statistics on String Fields  33 44

The fieldtransform Operator  33 46  
Overview  33 46  
General Transformation Process  33 47  
Using the fieldtransform Operator  33 49  
Syntax and Options  33 50  
Performing Z-score Transforms  33 57  
Using the Z-score Transform  33 58  
Configuring Z-score Transforms  33 61  
Performing Z-sigmoid and Z-symmetric Transforms  33 62  
Using the Transforms  33 63  
Standard Deviation of Zero  33 66  
Performing Identity Transforms  33 67  
Transforming Ordinal Fields  33 73
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example 4: Writing to a DB2 Table Containing an Unmatched Column</td>
<td>34</td>
</tr>
<tr>
<td>The db2upsert Operator</td>
<td>34</td>
</tr>
<tr>
<td>Partitioning for db2upsert</td>
<td>34</td>
</tr>
<tr>
<td>Data Flow Diagram</td>
<td>34</td>
</tr>
<tr>
<td>Properties</td>
<td>34</td>
</tr>
<tr>
<td>Operator Action</td>
<td>34</td>
</tr>
<tr>
<td>Syntax and Options</td>
<td>34</td>
</tr>
<tr>
<td>The db2part Operator</td>
<td>34</td>
</tr>
<tr>
<td>Syntax and Options</td>
<td>34</td>
</tr>
<tr>
<td>Example</td>
<td>34</td>
</tr>
<tr>
<td>The db2lookup Operator</td>
<td>34</td>
</tr>
<tr>
<td>Data Flow Diagram</td>
<td>34</td>
</tr>
<tr>
<td>Properties</td>
<td>34</td>
</tr>
<tr>
<td>Syntax and Options</td>
<td>34</td>
</tr>
<tr>
<td>Example</td>
<td>34</td>
</tr>
<tr>
<td>Considerations for Reading and Writing DB2 Tables</td>
<td>34</td>
</tr>
<tr>
<td>Data Translation Anomalies</td>
<td>34</td>
</tr>
<tr>
<td>Using a Node Map</td>
<td>34</td>
</tr>
</tbody>
</table>

### Chapter 35 The Informix Interface Library 35 1

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of the INFORMIX Interface Operators</td>
<td>35</td>
</tr>
<tr>
<td>Configuring the INFORMIX User Environment</td>
<td>35</td>
</tr>
<tr>
<td>Read Operations</td>
<td>35</td>
</tr>
<tr>
<td>Data Flow Diagram</td>
<td>35</td>
</tr>
<tr>
<td>Read Operator Action</td>
<td>35</td>
</tr>
<tr>
<td>Execution Mode</td>
<td>35</td>
</tr>
<tr>
<td>Column Name Conversion</td>
<td>35</td>
</tr>
<tr>
<td>Data Type Conversion</td>
<td>35</td>
</tr>
<tr>
<td>Example 1: Reading All Data from an INFORMIX Table</td>
<td>35</td>
</tr>
<tr>
<td>Write Operations</td>
<td>35</td>
</tr>
<tr>
<td>Data Flow Diagram</td>
<td>35</td>
</tr>
<tr>
<td>Execution Mode</td>
<td>35</td>
</tr>
<tr>
<td>Column Name Conversion</td>
<td>35</td>
</tr>
<tr>
<td>Data Type Conversion</td>
<td>35</td>
</tr>
<tr>
<td>Write Modes</td>
<td>35</td>
</tr>
<tr>
<td>Writing Fields</td>
<td>35</td>
</tr>
<tr>
<td>Limitations</td>
<td>35</td>
</tr>
<tr>
<td>Example 2: Appending Data to an Existing INFORMIX Table</td>
<td>35</td>
</tr>
<tr>
<td>Example 3: Writing Data to an INFORMIX Table in Truncate Mode</td>
<td>35</td>
</tr>
<tr>
<td>Example 4: Handling Unmatched Orchestrate Fields in an INFORMIX Write Operation</td>
<td>35</td>
</tr>
</tbody>
</table>
Contents

The oraread Operator  36 6
  Data Flow Diagram  36 6
  Properties  36 6
  Operator Action  36 7
  Where the oraread Operator Runs  36 7
  Column Name Conversion  36 8
  Data Type Conversion  36 9
  Oracle Record Size  36 9
  Targeting the Read Operation  36 9
  Join Operations  36 11
  Syntax and Options  36 11
  Example 1: Reading an Oracle Table and Modifying a Field Name  36 15
  Example 2: Reading from an Oracle Table in Parallel with the query Option  36 16

The orawrite Operator  36 17
  Writing to a Multibyte Database  36 17
  Data Flow Diagram  36 17
  Properties  36 17
  Operator Action  36 18
  Data Type Conversion  36 20
  Syntax and Options  36 23
  Example 1: Writing to an Existing Oracle Table  36 29
  Example 2: Creating an Oracle Table  36 30
  Example 3: Writing to an Oracle Table Using the modify Operator  36 31

The oraupsert Operator  36 32
  Data Flow Diagram  36 32
  Properties  36 32
  Operator Action  36 33
  Associated Environment Variables  36 34
  Syntax and Options  36 35
  Example  36 37

The oralookup Operator  36 39
  Data Flow Diagram  36 40
  Properties  36 40
  Syntax and Options  36 41
  Example  36 42

Chapter 37 The Teradata Interface Library  37 1

  Overview of the Teradata Interface Operators  37 1
  Summary of the Teradata Interface Operators  37 2
  National Language Support  37 2

  The teraread Operator  37 4
  Data Flow Diagram  37 4
  Properties  37 4
### Contents

- Specifying the Query  37 5
- Column Name and Data Type Conversion  37 5
- teraread Restrictions  37 7
- Syntax and Options  37 7

The terawrite Operator  37 10
- Data Flow Diagram  37 11
- Properties  37 11
- Column Name and Data Type Conversion  37 12
- Correcting Load Errors  37 13
- Write Modes  37 13
- Writing Fields  37 14
- Limitations  37 14
- Restrictions  37 15
- Syntax and Options  37 15

### Index

- Index 1
<table>
<thead>
<tr>
<th>Table</th>
<th>Operator</th>
<th>Properties/Options</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>changeapply</code></td>
<td>Properties</td>
<td>1, 2</td>
</tr>
<tr>
<td>2</td>
<td><code>changeapply</code></td>
<td>Options</td>
<td>1, 6</td>
</tr>
<tr>
<td>3</td>
<td><code>changecapture</code></td>
<td>Properties</td>
<td>2, 2</td>
</tr>
<tr>
<td>4</td>
<td><code>changecapture</code></td>
<td>Options</td>
<td>2, 5</td>
</tr>
<tr>
<td>5</td>
<td><code>compare</code></td>
<td>Properties</td>
<td>3, 2</td>
</tr>
<tr>
<td>6</td>
<td><code>compare</code></td>
<td>Results Value</td>
<td>3, 4</td>
</tr>
<tr>
<td>7</td>
<td><code>compare</code></td>
<td>Options</td>
<td>3, 5</td>
</tr>
<tr>
<td>8</td>
<td><code>copy</code></td>
<td>Properties</td>
<td>4, 2</td>
</tr>
<tr>
<td>9</td>
<td><code>copy</code></td>
<td>Options</td>
<td>4, 3</td>
</tr>
<tr>
<td>10</td>
<td><code>diff</code></td>
<td>Properties</td>
<td>5, 2</td>
</tr>
<tr>
<td>11</td>
<td><code>diff</code></td>
<td>Options</td>
<td>5, 6</td>
</tr>
<tr>
<td>12</td>
<td><code>encode</code></td>
<td>Properties</td>
<td>6, 2</td>
</tr>
<tr>
<td>13</td>
<td><code>encode</code></td>
<td>Options</td>
<td>6, 3</td>
</tr>
<tr>
<td>14</td>
<td><code>filter</code></td>
<td>Properties</td>
<td>7, 2</td>
</tr>
<tr>
<td>15</td>
<td><code>filter</code></td>
<td>Options</td>
<td>7, 3</td>
</tr>
<tr>
<td>16</td>
<td><code>funnel</code></td>
<td>Properties</td>
<td>8, 2</td>
</tr>
<tr>
<td>17</td>
<td><code>funnel</code></td>
<td>Option</td>
<td>8, 4</td>
</tr>
<tr>
<td>18</td>
<td><code>sortfunnel</code></td>
<td>Options</td>
<td>8, 7</td>
</tr>
<tr>
<td>19</td>
<td><code>generator</code></td>
<td>Properties</td>
<td>9, 2</td>
</tr>
<tr>
<td>20</td>
<td><code>generator</code></td>
<td>Options</td>
<td>9, 3</td>
</tr>
<tr>
<td>21</td>
<td><code>generator</code></td>
<td>Options for Orchestrate Data Types</td>
<td>9, 8</td>
</tr>
<tr>
<td>22</td>
<td><code>head</code></td>
<td>Properties</td>
<td>10, 2</td>
</tr>
<tr>
<td>23</td>
<td><code>head</code></td>
<td>Options</td>
<td>10, 3</td>
</tr>
<tr>
<td>24</td>
<td><code>lookup</code></td>
<td>Properties</td>
<td>11, 4</td>
</tr>
<tr>
<td>25</td>
<td><code>lookup</code></td>
<td>Options</td>
<td>11, 5</td>
</tr>
<tr>
<td>26</td>
<td><code>merge</code></td>
<td>Properties</td>
<td>12, 3</td>
</tr>
<tr>
<td>27</td>
<td><code>merge</code></td>
<td>Options</td>
<td>12, 4</td>
</tr>
<tr>
<td>28</td>
<td><code>modify</code></td>
<td>Properties</td>
<td>13, 3</td>
</tr>
</tbody>
</table>
Table 29  modify Operator Change Name Example 13 5
Table 30  Data Type Conversions 13 9
Table 31  date Type Conversions Performed by modify 13 11
Table 32  Conversions of Decimal Fields performed by modify 13 15
Table 33  raw Field Functions 13 18
Table 34  string and ustring Conversions Performed by the modify Operator 13 19
Table 35  A String Lookup Table 13 23
Table 36  Lookup Table for gender Field 13 25
Table 37  time Type Conversions Provided by modify 13 26
Table 38  timestamp Field Conversions Provided by modify 13 29
Table 39  Rules for Handling Nullable Fields 13 32
Table 40  Schemas with nested aggregates 13 37
Table 41  Conversions Provided by the modify Operator 13 38
Table 42  pcompress Operator Properties 14 2
Table 43  pcompress Operator Options 14 3
Table 44  peek Operator Properties 15 2
Table 45  peek Operator Options 15 3
Table 46  remdup Operator Properties 16 2
Table 47  remdup Operator Options 16 3
Table 48  Sample Input Data Set 16 5
Table 49  Sample Output Data set 16 5
Table 50  Sample Output Data Set with -last Option 16 6
Table 51  Sample Input 16 6
Table 52  Sample Output from a Case-Sensitive Sort 16 7
Table 53  Sample Output from a Case-Insensitive Sort 16 7
Table 54  Output from remdup after Case-Insensitive Sort 16 8
Table 55  sample Operator Properties 17 2
Table 56  sample Operator Options 17 3
Table 57  sequence Operator Properties 18 2
Table 58  switch Operator Properties 19 3
Table 59  switch Operator Options 19 4
Table 60  tail Operator Properties 20 2
Table 61  tail Operator Options 20 2
Table 62  tail Operator Input and Output for Example 1 20 3
Table 63  tail Operator Input and Output for Example 2 20 4
Table 64  transform Operator Properties 21 5
Table 65  transform Operator Options 21 6
Table 66  C Function Schema Types and Associated C Types 21 15
List of Tables

Table 67 Simple Fieldtypes 21 19
Table 68 Unary Operators 21 22
Table 69 Binary and Ternary Operators 21 22
Table 70 Precedence and Associativity of Operators 21 24
Table 71 date Field Functions 21 29
Table 72 decimal and float Field Functions 21 32
Table 73 raw Field Functions 21 34
Table 74 time Field and timestamp Field Functions 21 35
Table 75 Null Handling Functions 21 38
Table 76 Mathematical Functions 21 39
Table 77 A String Lookup Table 21 41
Table 78 string Field Functions 21 42
Table 79 ustring Field Functions 21 48
Table 80 Bit Manipulation Functions 21 54
Table 81 Job Monitoring Functions 21 56
Table 82 Miscellaneous Functions 21 57
Table 83 Partial Record Schemas versus Complete Record Schemas 22 7
Table 84 EBCDIC to ASCII Conversion 22 16
Table 85 ASCII to EBCDIC Conversion 22 20
Table 86 Conversion Table Irregularities 22 22
Table 87 import Operator Properties 23 2
Table 88 import Operator Options 23 4
Table 89 export Operator Properties 24 2
Table 90 export Operator Options 24 5
Table 91 Example: Data Set Schema Versus Export Schema 24 10
Table 92 Setting Record Level Properties 25 6
Table 93 Numeric Field Properties 25 8
Table 94 String Field Properties 25 8
Table 95 ustring Field Properties 25 9
Table 96 Decimal Field Properties 25 9
Table 97 Date Field Properties 25 10
Table 98 Time Field Properties 25 10
Table 99 Timestamp Field Properties 25 11
Table 100 Variable-Length Vector Field Properties 25 12
Table 101 Nullable Field Properties 25 12
Table 102 Setting Record Level Properties 25 13
Table 103 Tagged Subrecord Properties 25 14
Table 104 Record and Field Properties 25 14
List of Tables

Table 105  Sign of Unpacked Decimal  25 58
Table 106  Sign Representation in ASCII and EBCDIC  25 66
Table 107  COBOL Storage Format Conversions  26 5
Table 108  Arguments for readcobol  26 6
Table 109  entire Partitioner Properties  27 5
Table 110  hash Operator Properties  27 8
Table 111  hash Partitioner Option  27 9
Table 112  modulus Operator Properties  27 11
Table 113  modulus Partitioner Option  27 12
Table 114  random Operator Properties  27 14
Table 115  range Operator Properties  27 21
Table 116  range Partitioner Options  27 22
Table 117  writerangemap Operator Properties  27 24
Table 118  writerangemap Operator Options  27 25
Table 119  makerangemap Utility Options  27 27
Table 120  roundrobin Operator Properties  27 30
Table 121  same Operator Properties  27 32
Table 122  Join Operators Properties  28 4
Table 123  Comparison of Joins, Lookup, and Merge  28 6
Table 124  innerjoin Operator option  28 8
Table 125  leftouterjoin Operator Option  28 10
Table 126  rightouterjoin Operator Options  28 12
Table 127  fullouterjoin Operator Option  28 16
Table 128  ordered Collector Properties  29 3
Table 129  roundrobin Collector Properties  29 5
Table 130  sortmerge Collector Properties  29 8
Table 131  sortmerge Collection Options  29 9
Table 132  Restructure Operators Listing  30 3
Table 133  aggtorec Operator Properties  30 4
Table 134  aggtorec Operator Options  30 5
Table 135  field_export Operator Properties  30 10
Table 136  field_export Operator Options  30 11
Table 137  field_import Operator Properties  30 14
Table 138  field_import Operator Properties  30 14
Table 139  makesubrec Operator Options  30 18
Table 140  makesubrec Operator Options  30 20
Table 141  makevect Operator Options  30 22
Table 142  makevect Operator Option  30 24
Table 143  promotesubrec Operator Properties  30  26
Table 144  promotesubrec Operator Option  30  27
Table 145  splitsubrec Operator Properties  30  28
Table 146  splitsubrec Operator Options  30  29
Table 147  splitvect Operator Properties  30  31
Table 148  splitvect Operator Option  30  32
Table 149  tagbatch Operator Properties  30  35
Table 150  tagbatch Operator Options  30  37
Table 151  tagswitch Operator Properties  30  43
Table 152  tagswitch Operator Options  30  45
Table 153  tsort Operator Properties  31  7
Table 154  tsort Operator Options  31  9
Table 155  psort Operator Properties  31  23
Table 156  psort Operator Options  31  25
Table 157  Conversions to SAS Data Types  32  12
Table 158  sasin Operator Properties  32  38
Table 159  sasin Operator Options  32  40
Table 160  sas Operator Properties  32  44
Table 161  sas Operator Options  32  46
Table 162  sasout Operator Properties  32  52
Table 163  sasout Operator Options  32  53
Table 164  sascontents Operator Properties  32  55
Table 165  sascontents Operator Options  32  56
Table 166  crosstab Operator Properties  33  7
Table 167  crosstab Operator Options  33  9
Table 168  group Operator Properties  33  22
Table 169  group Operator Options  33  24
Table 170  statistics Operator Properties  33  40
Table 171  statistics Operator Options  33  41
Table 172  readstats utility Options  33  42
Table 173  fieldtransform Operator Properties  33  50
Table 174  fieldtransform Operator Options  33  52
Table 175  Required DB2 Privileges  34  4
Table 176  db2read Operator Properties  34  9
Table 177  DB2 Interface Operator Data Type Conversions  34  11
Table 178  db2read Options  34  15
Table 179  db2write and db2load Operator Properties  34  20
Table 180  Operators that modify Orchestrate Types for DB2  34  23
List of Tables

Table 181  db2write and db2load Type Conversions  34 24
Table 182  Privileges Required for db2write and db2load Write Modes  34 26
Table 183  db2write and db2load Options  34 28
Table 184  db2upsert Operator Properties  34 39
Table 185  db2upsert Operator Options  34 41
Table 186  dbpart Operator Options  34 44
Table 187  db2lookup Operator Properties  34 48
Table 188  db2lookup Operator Options  34 49
Table 189  DB2 Table Write Operations Translation Anomalies  34 52
Table 190  DB2 Table Read Operations Translation Anomaly  34 52
Table 191  INFORMIX Interface Read Operators Data-Type Conversion  35 5
Table 192  Schemas for Orchestrate Read of INFORMIX Result Set  35 7
Table 193  INFORMIX Interface Write Operators Data Conversion  35 9
Table 194  Schemas for Orchestrate Append to INFORMIX Table  35 12
Table 195  Schemas for Orchestrate Write to Table with Unmatched Column  35 15
Table 196  hplread Operator Properties  35 17
Table 197  hplread Operator Options  35 18
Table 198  hplwrite Operator Properties  35 20
Table 199  hplwrite Operator Options  35 21
Table 200  infxread Operator Properties  35 23
Table 201  infxread Operator Options  35 24
Table 202  infxread Operator Properties  35 26
Table 203  infxwrite Operator Properties  35 27
Table 204  xpsread Operator Options  35 29
Table 205  xpsread Operator Options  35 30
Table 206  xpswrite Operator Properties  35 32
Table 207  xpswrite Operator Options  35 33
Table 208  oraread Operator Properties  36 6
Table 209  oraread Operator Data Type Conversion  36 9
Table 210  oraread Operator Options  36 12
Table 211  orawrite Operator Properties  36 17
Table 212  Operators for Oracle Data Type Compatibility  36 20
Table 213  Orchestrate to Oracle Data Type Conversion  36 20
Table 214  Required Oracle Privileges for Oracle Interface Write Operators  36 22
Table 215  orawrite Operator Options  36 24
Table 216  oraupsert Operator Properties  36 32
Table 217  oraupsert Operator Options  36 35
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Operator Type</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 218</td>
<td>oralookup Operator Properties</td>
<td>36 40</td>
</tr>
<tr>
<td>Table 219</td>
<td>oralookup Operator Options</td>
<td>36 41</td>
</tr>
<tr>
<td>Table 220</td>
<td>teraread Operator Properties</td>
<td>37 4</td>
</tr>
<tr>
<td>Table 221</td>
<td>teraread Operator Data Type Conversions</td>
<td>37 6</td>
</tr>
<tr>
<td>Table 222</td>
<td>teraread Operator Options</td>
<td>37 8</td>
</tr>
<tr>
<td>Table 223</td>
<td>terawrite Operator Properties</td>
<td>37 11</td>
</tr>
<tr>
<td>Table 224</td>
<td>terawrite Operator Data Type Conversions</td>
<td>37 12</td>
</tr>
<tr>
<td>Table 225</td>
<td>terawrite Operator Options</td>
<td>37 16</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1  Data Flow for changeapply Example 1 9
Figure 2  Data Flow for changecapture Example 2 2 9
Figure 3  Data Flow Diagram for diff Example 2 5 10
Figure 4  Data Flow Example for funnel 8 3
Figure 5  Data Flow for generator Operator Example 1 9 5
Figure 6  Effect of lookup with Single Lookup Table Record 11 9
Figure 7  Effect of lookup with Two Lookup Table Records 11 10
Figure 8  Data Flow for Interest Rate Lookup Example 11 11
Figure 9  Using the modify Operator with the lookup Operator 11 12
Figure 10  Merge of a Master Record and an Update Record 12 6
Figure 11  Merge of a Master Record and Two Update Records 12 7
Figure 12  Typical Data Flow for merge Operator 12 8
Figure 13  Alternate Data Flow for merge Operator 12 9
Figure 14  Schemas for merge Example 12 11
Figure 15  Data Flow for merge Example 12 13
Figure 16  Data Sets Containing the Same Field Name 12 14
Figure 17  Data Flow for Resolving Field Name Conflicts 12 15
Figure 18  merge Operator Showing Reject Data Set 12 20
Figure 19  merge Operator Showing Multiple Reject Data Sets 12 20
Figure 20  Data Flow With Default Data Type Conversion 13 8
Figure 21  Converting a Date Field to Three Integer Fields 13 14
Figure 22  Conversion with Non-Default Rounding 13 17
Figure 23  Finding the Length of a Raw Field 13 18
Figure 24  Data Flow Diagram Showing Substring Extraction 13 22
Figure 25  Extracting the Length of a String 13 22
Figure 26  String Lookup Table Example 13 25
Figure 27 Converting a date  13  28
Figure 28 Conversion of a Date Field to a Timestamp Field  13  31
Figure 29 Converting an Out-of-Band Null to an In-Band Null  13  33
Figure 30 Converting to Out-of-Band Null Representation  13  34
Figure 31 Using Aggregates with the modify Operator  13  36
Figure 32 Compressing a Data Set with pcompress  14  5
Figure 33 Expanding a Data Set with pcompress  14  5
Figure 34 Use of peek Operator  15  4
Figure 35 Data Flow for Sampling Example  17  4
Figure 36 Example for sequence Operator  18  3
Figure 37 Example for switch Operator  19  7
Figure 38 Data Flow for Example 1  22  4
Figure 39 Data Flow for Example 2  22  5
Figure 40 Insertion of an Implicit import Operator  22  12
Figure 41 Data Flow for Example 1  23  13
Figure 42 Data Flow for Example 2  23  14
Figure 43 Data Flow for Example 1  24  15
Figure 44 Data Flow for Example 2  24  16
Figure 45 Using the entire Partitioner  27  4
Figure 46 Hash Partitioning Example  27  6
Figure 47 Using the hash Partitioner  27  7
Figure 48 Using the random Partitioner  27  13
Figure 49 Range Partitioning Example  27  15
Figure 50 Two-Step Data Flow with Range Partitioner  27  19
Figure 51 Using the random Partitioner  27  29
Figure 52 Using the random Partitioner  27  31
Figure 53 Using the ordered Collector  29  3
Figure 54 Using the roundrobin Collector  29  4
Figure 55 Sample Input to sortmerge Collector  29  7
Figure 56 Sample Outputs from sortmerge Collector  29  7
Preface

Describes this document and the conventions that it uses.

Who This Book Is For  xxxvii
Organization of This Book  xxxviii
Orchestrate-Inserted Partition and Sort Components  xxxviii
Typographic Conventions  xxxix
Syntax Conventions  xl
The Orchestrate Documentation Set  xli
Using the Adobe Acrobat Reader  xli
Searching for Text in Orchestrate Documents  xli
Assistance and Additional Information  xlii

Who This Book Is For

This book is a reference for anyone who uses Orchestrate. To use this book, you should have a basic familiarity with Orchestrate; for example, by reading the Orchestrate 7.0 User Guide. The book is particularly valuable for developers who have a working copy of Orchestrate available.
Organization of This Book

This book is sectioned into three parts. Part 1 consists of twenty chapters which describe the operators in the General Library. Part 2 has five chapters that describe the import/export utility and the COBOL converter utility. The remainder of the book describes the operators in various special libraries, one library per chapter.

Orchestrate-Inserted Partition and Sort Components

Orchestrate automatically inserts partition and sort components in your application to optimize the performance of the operators in your data flow. This makes it possible for you to write correct data flows without having to deal directly with issues of parallelism.

When you want to explicitly control the partitioning and sorting behavior of an operator, you can insert a predefined Orchestrate partitioner component or create a custom component using the C++ API. Orchestrate may issue a warning, but does not remove an unnecessary or inappropriate partitioning component.

There are two ways to prevent Orchestrate from automatically inserting partition and sort components in your data flow:

- Include the top-level command-line options, `-nopartinsertion` and `-nosortinsertion` in your `osh` command. For example:
  
  `osh -nosortinsertion "step-specification-string"`

- Set the environment variables `APT_NO_PART_INSERTION` and `APT_NO_SORT_INSERTION`.

For information on Orchestrate's predefined partitioners, see Chapter 27, “The Partitioning Library” on page 27-1 of this Reference and the Orchestrate 7.0 User Guide. For information on creating a custom partitioner, see the Orchestrate 7.0 Developer Guide.
## Typographic Conventions

<table>
<thead>
<tr>
<th>Format</th>
<th>Terms</th>
<th>Examples</th>
</tr>
</thead>
</table>
| **bold serif**   | Orchestrate-defined terms including class and function names, data types, and default field names. | APT_Operator
runLocally()
APT_Operator::runLocally()
class_predicted
uint64                                                        |
|                  | Orchestrate-defined macros                                          | APT_DECLARE_PERSISTENT                                                   |
|                  | Orchestrate-defined operators, operator options, and collection and partition methods | The *clusterquality* operator takes the results and flatten options.
Use the any or round robin collection method.                  |
| **bold italic serif** | Orchestrate technical terms within the text that defines them               | In *pipeline parallelism*, each operation runs when it has input data available. |
| **sans serif**   | Orchestrate menus                                                    | Tools > Check Config                                                     |
| **fixed width**  | C++ code                                                            | return APT_StatusOk;                                                     |
|                  | operating-system commands                                          | $ cd $APT_ORCHHOME/bin
$ rm aptserv2                                                            |
|                  | osh commands                                                        | $ osh "clusterquality < kmeans.ds"                                    |
|                  | non-Orchestrate-defined C++ functions                              | sprintf()                                                               |
| **italic**       | user-defined classes, functions, and field names                    | MyOperator
selectState()
MyOperator::selectState()
Price                                                        |
|                  | Orchestrate variables                                               | Specify an insert_statement                                             |
Syntax Conventions

Operator syntax is presented as you would enter it as part of an *osh* command. For a description of the general syntax of an *osh* command, refer to the Orchestrate 7.0 User Guide.

The following syntax conventions are used throughout this book:

- A vertical bar (|) separates mutually exclusive alternatives.
- Braces ({}) are used with vertical bars (|) to indicate that one of several mutually exclusive alternatives are required, for example {a | b} indicates that a or b is required.
- If one or more items are enclosed in braces ({}) and separated by commas the items are synonymous. Thus {a, b} indicates that a and b have exactly the same effect.
- Brackets ([ ]) indicate that the item(s) inside are optional. For example, [a | b] indicates a, or b, or neither. Occasionally, brackets do not indicate optional items but are part of the syntax. In these cases, the special nature of the brackets is explicitly stated.
- Ellipsis (...) indicates that the preceding item occurs zero or more times. If a user-provided string is indicated, it may represent a different item in each occurrence. For example:
  - [key field ...] means zero or more occurrences of -key field, where field may be a different string in each occurrence.
  - To indicate one or more occurrences of an item, the item occurs first without brackets and ellipsis and then with, for example
    - key field [key field ...]

The Orchestrate Documentation Set

These documents are available both in hardcopy and online PDF format.

- Orchestrate 7.0 User Guide
- Orchestrate 7.0 Operators Reference
- Orchestrate 7.0 Installation and Administration Manual
- Orchestrate 7.0 Developer Guide
- Orchestrate 7.0 WebHouse User Guide
- Orchestrate 7.0 Record Schema
- Orchestrate 7.0 C++ Classes and Macros Sorted by Header File
Using the Adobe Acrobat Reader

To read the document set online, you need the Adobe Acrobat Reader. If you already have Adobe Acrobat Reader installed, make sure it is version 4.05 or later and is the version of Reader with Search. If your version of Reader does not have Search, you will not be able to search across all the documents in the Orchestrate set.

To see if your copy of Reader has Search, look for the Search icon: on the Reader toolbar, and make sure it is present and not dimmed. The Search icon should be located alongside the somewhat similar Find icon: .

If you do not have the appropriate version of Acrobat installed, you may use the Acrobat Reader 4.05 included with Orchestrate. Find the version you need in one of the following platform-specific directories:

$APT_ORCHHOME/etc/acroread-sun-405.tar.gz
$APT_ORCHHOME/etc/acroread-aix-405.tar.gz
$APT_ORCHHOME/etc/acroread-osf1-405.tar.gz
$APT_ORCHHOME/etc/acroread-hpux-405.tar.gz

Use the UNIX gunzip and tar commands to unpack the files. Then cd to the directory*.install (where * contains an abbreviation of your platform name) and follow the instructions in the INSTGUID.TXT file.

The Orchestrate online documentation set and the full-text search index is located in $APT_ORCHHOME/doc.

Searching for Text in Orchestrate Documents

You can find specific words or phrases in this Guide and across all Orchestrate online documents using the Adobe Acrobat Reader.

To find text in a single document:

1. Open the document in the Adobe Acrobat Reader.
2. Choose Edit > Find or Click Find on the Adobe Toolbar
3. Enter the text you want to find in the Find What field and click Find.
4. Use Ctrl+G to find the next occurrence of the text.
To find text across all documents:

1. Choose Edit > Search > Query
   or
   Click Search on the Acrobat toolbar

2. In the Adobe Acrobat Search window, type the text you want to find in the Find Results Containing Text box, then click Search.
   a. If the following message appears at the bottom of the Adobe Acrobat Search window:
      
      No selected indexes are available for search

      Click Indexes... to bring up the Index Selection window.

   b. Click Add, then navigate the Add Index window to find the full-text search index, Orchestrate.pdx, located in $APT_ORCHHOME/doc.

   c. Select the index, then click OK on the Index Selection window.

3. The Search Results window lists the documents that contain your search text, ordered according to the number of search hits.

4. Select a book in the list.

5. Use the Previous Highlight and Next Highlight buttons to move to the previous or next instances of your search text.

Assistance and Additional Information

If you require assistance or have questions about Orchestrate, you can contact Ascential Customer Support by:

- Calling (866) INFONOW
- Writing support@ascential.com for any Ascential Software product or orch-support@ascential.com for Orchestrate-specific help.

For current information about Ascential and its products log onto:

http://www.ascential.com/
PART 1

The General Library

This part of the Orchestrate 7.0 Operators Reference describes the Orchestrate general library operators as listed below.

Part 2 describes the Orchestrate Import/Export and Cobol schema conversion utility operators.

Part 3 describes the Orchestrate Join, Partitioning, Collection, Restructure, SAS, Sorting, and Statistics libraries, and the four RDBMS libraries that provide interfaces to DB2, INFORMIX, Oracle, and Teradata.

Chapter 1 The changeapply Operator 1 1
Chapter 2 The changecapture Operator 2 1
Chapter 3 The compare Operator 3 1
Chapter 4 The copy Operator 4 1
Chapter 5 The diff Operator 5 1
Chapter 6 The encode Operator 6 1
Chapter 7 The filter Operator 7 1
Chapter 8 The Funnel Operators 8 1
Chapter 9 The generator Operator 9 1
Chapter 10 The head Operator 10 1
Chapter 11 The lookup Operator 11 1
Chapter 12 The merge Operator 12 1
Chapter 13 The modify Operator 13 1
Chapter 14 The pcompress Operator 14 1
Chapter 15 The peek Operator 15 1
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Operator</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>The remdup Operator</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>The sample Operator</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>The sequence Operator</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>The switch Operator</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>The tail Operator</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>The transform Operator</td>
<td>21</td>
</tr>
</tbody>
</table>
The changeapply Operator

The changeapply operator uses the output from the changecapture operator and a before data set to compute an after data set.

Data Flow Diagram  1 2
Properties  1 2
Schemas  1 4
  Transfer Behavior  1 4
  Key Comparison Fields  1 4
Syntax and Options  1 5
Example  1 8

The changeapply operator takes the change data set output from the changecapture operator and applies the encoded change operations to a before data set to compute an after data set. If the before data set is identical to the before data set that was input to the changecapture operator, then the output after data set for changeapply is identical to the after data set that was input to the changecapture operator. That is:

\[
\text{change} := \text{chancgecapture (before, after)}
\]
\[
\text{after} := \text{changeapply (before, change)}
\]

You use the companion operator changecapture to provide a data set that contains the changes in the before and after data sets.
Data Flow Diagram

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>2</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>beforeRec:* CHANGE, changeRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>afterRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>changeRec:* \rightarrow \text{afterRec:<em>}, dropping the change_code field; beforeRec:</em> \rightarrow \text{afterRec:*} with type conversions</td>
</tr>
<tr>
<td>Input partitioning style</td>
<td>keys in same partition</td>
</tr>
<tr>
<td>Output partitioning style</td>
<td>keys in same partition</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

The before input to changeapply must have the same fields as the before input that was input to changecapture, and an automatic conversion must exist between the types of corresponding fields. In addition, results are only guaranteed if the contents of the before input to changeapply are identical (in value and record order in each partition) to the before input that was fed to changecapture, and if the keys are unique.
The change input to changeapply must have been output from changecapture without modification. Because preserve-partitioning is set on the change output of changecapture (under normal circumstances you should not override this), the changeapply operator has the same number of partitions as the changecapture operator. Additionally, both inputs of changeapply are designated as same partitioning by the operator logic.

The changeapply operator performs these actions for each change record: If the before keys come before the change keys in the specified sort order, the before record is consumed and transferred to the output; no change record is consumed. This is a copy.

- If the before keys are equal to the change keys, the behavior depends on the code in the change_code field of the change record:
  - **Insert**: The change record is consumed and transferred to the output; no before record is consumed.
    
    If key fields are not unique, and there is more than one consecutive insert with the same key, then changeapply applies all the consecutive inserts before existing records. This record order may be different from the after data set given to changecapture.
  
  - **Delete**: The value fields of the before and change records are compared. If they are not the same, the before record is consumed and transferred to the output; no change record is consumed (copy). If the value fields are the same or if ignoreDeleteValues is specified, the change and before records are both consumed; no record is transferred to the output.
    
    If key fields are not unique, the value fields ensure that the correct record is deleted. If more than one record with the same keys have matching value fields, the first encountered is deleted. This may cause different record ordering than in the after data set given to the changecapture operator.
  
  - **Edit**: The change record is consumed and transferred to the output; the before record is just consumed.
    
    If key fields are not unique, then the first before record encountered with matching keys is edited. This may be a different record from the one that was edited in the after data set given to the changecapture operator, unless the -keepCopy option was used.
  
  - **Copy**: The change record is consumed. The before record is consumed and transferred to the output.

- If the before keys come after the change keys, behavior also depends on the change_code field.
  - **Insert**: The change record is consumed and transferred to the output; no before record is consumed (the same as when the keys are equal).
Chapter 1  The changeapply Operator Schemas

- **Delete:** A warning is issued and the change record is consumed; no record is transferred to the output; no before record is consumed.
- **Edit or Copy:** A warning is issued and the change record is consumed and transferred to the output; no before record is consumed. This is an insert.

If the before input of `changeapply` is identical to the before input of `changecapture` and either keys are unique or copy records are used, then the output of `changeapply` is identical to the after input of `changecapture`. However, if the before input of `changeapply` is not the same (different record contents or ordering), or keys are not unique and copy records are not used, this fact is not detected and the rules described above are applied anyway, producing a result that might or might not be useful.

**Schemas**

The `changeapply` output data set has the same schema as the change data set, with the `change_code` field removed.

The before interface schema is:

```plaintext
record (key:type; ... value:type; ... beforeRec:*;)
```

The change interface schema is:

```plaintext
record (change_code:int8; key:type; ... value:type; ... changeRec:*;)
```

The after interface schema is:

```plaintext
record (afterRec:*;)
```

**Transfer Behavior**

The change to after transfer uses an internal transfer adapter to drop the `change_code` field from the transfer. This transfer is declared first, so the schema of the change data set determines the schema of the after data set.

**Key Comparison Fields**

An internal, generic comparison function compares key fields. An internal, generic equality function compares non-key fields. You adjust the comparison with parameters and equality functions for individual fields using the `-param` suboption of the `-key`, `-allkeys`, `-allvalues`, and `-value` options.
Syntax and Options

You must specify at least one -key field or specify the -allkeys option. Terms in italic typeface are option strings you supply. When your option string contains a space or a tab character, you must enclose it in single quotes.

```
changeapply
  -key inpt_field_name [-cs | ci] [-asc | -desc] [-nulls first | last]
  [param params]
  [-key inpt_field_name [-cs | ci] [-asc | -desc] [-nulls first | last]
  [param params...]
  |
  -allkeys [-cs | ci] [-asc | -desc] [-nulls first | last]
  [param params]
  [-allvalues [-cs | ci] [param params]]
  [-codeField field_name]
  [-copyCode n]
  [-collation_sequence locale| collation_file_pathname] OFF]
  [-deleteCode n]
  [-doStats]
  [-dropkey input_field_name...]
  [-dropvalue input_field_name...]
  [-editCode n]
  [ignoreDeleteValues]
  [-insertCode n]
  [-value inpt_field_name [-ci | -cs] ...]
```

**Note** The -checkSort option has been deprecated. By default, partitioner and sort components are now inserted automatically.
Table 2  changeapply Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
</table>

Specify one or more key fields.

You must specify at least one key for this option or specify the **-allkeys** option. These options are mutually exclusive. You cannot use a vector, subrecord, or tagged aggregate field as a value key.

The **-ci** suboption specifies that the comparison of value keys is case insensitive. The **-cs** suboption specifies a case-sensitive comparison, which is the default.

**-asc** and **-desc** specify ascending or descending sort order.

**-nulls first | last** specifies the position of nulls.

The **-params** suboption allows you to specify extra parameters for a key. Specify parameters using property=value pairs separated by commas.

**-allkeys**


Specify that all fields not explicitly declared are key fields. The suboptions are the same as the suboptions described for the **-key** option above.

You must specify either the **-allkeys** option or the **-key** option. They are mutually exclusive.

**-allvalues**

- allvalues [-cs | ci] [-param params]

Specify that all fields not otherwise explicitly declared are value fields.

The **-ci** suboption specifies that the comparison of value keys is case insensitive. The **-cs** suboption specifies a case-sensitive comparison, which is the default.

The **-param** suboption allows you to specify extra parameters for a key. Specify parameters using property=value pairs separated by commas.

The **-allvalues** option is mutually exclusive with the **-value** and **-allkeys** options.

**-codeField**

- codeField field-name

The name of the change code field. The default is change_code. This should match the field name used in changecapture.
### Table 2  changeapply Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-collation_sequence</code></td>
<td>`-collation_sequence locale</td>
</tr>
<tr>
<td></td>
<td>This option determines how your string data is sorted. You can:</td>
</tr>
<tr>
<td></td>
<td>• Specify a predefined IBM ICU locale</td>
</tr>
<tr>
<td></td>
<td>• Write your own collation sequence using ICU syntax, and supply its</td>
</tr>
<tr>
<td></td>
<td>collation_file_pathname</td>
</tr>
<tr>
<td></td>
<td>• Specify OFF so that string comparisons are made using Unicode</td>
</tr>
<tr>
<td></td>
<td>code-point value order, independent of any locale or custom</td>
</tr>
<tr>
<td></td>
<td>sequence.</td>
</tr>
<tr>
<td></td>
<td>By default, Orchestrate sorts strings using byte-wise comparisons.</td>
</tr>
<tr>
<td></td>
<td>For more information, see Chapter 7: National Language Support in</td>
</tr>
<tr>
<td></td>
<td>the Orchestrate 7.0 User Guide; and reference this IBM ICU site:</td>
</tr>
</tbody>
</table>
| `-copyCode`    | `
copyCode n`
|               | Specifies the value for the change_code field in the change record  |
|               | for the copy result. The n value is an `int8`. The default value is 0|
|               | A copy record means that the before record should be copied to the  |
|               | output without modification.                                         |
| `-deleteCode`  | `
deleteCode n`
|               | Specifies the value for the change_code field in the change record  |
|               | for the delete result. The n value is an `int8`. The default value is 2|
|               | A delete record means that a record in the before data set must be  |
|               |   deleted to produce the after data set.                           |
| `-doStats`     | `-doStats`
|               | Configures the operator to display result information containing the|
|               | number of input records and the number of copy, delete, edit, and   |
|               | insert records.                                                    |
| `-dropkey`     | `-dropkey input_field_name`
|               | Optionally specify that the field is not a key field. If you specify |
|               | this option, you must also specify the `-allkeys` option.           |
|               | There can be any number of occurrences of this option.              |
| `-dropvalue`   | `-dropvalue input_field_name`
|               | Optionally specify that the field is not a value field. If you specify |
|               | this option, you must also specify the `-allvalues` option.         |
|               | There can be any number of occurrences of this option.              |
Table 2  changeapply Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-editCode n</code></td>
<td>Specifies the value for the <code>change_code</code> field in the change record for the edit result. The <code>n</code> value is an <code>int8</code>. The default value is 3. An edit record means that the value fields in the before data set must be edited to produce the after data set.</td>
</tr>
<tr>
<td><code>-ignoreDeleteValues</code></td>
<td>Do not check value fields on deletes. Normally, <code>changeapply</code> compares the value fields of delete change records to those in the before record to ensure that it is deleting the correct record. The <code>-ignoreDeleteValues</code> option turns off this behavior.</td>
</tr>
<tr>
<td><code>-insertCode n</code></td>
<td>Specifies the value for the <code>change_code</code> field in the output record for the insert result. The <code>n</code> value is an <code>int8</code>. The default value is 1. An insert means that a record must be inserted into the before data set to reproduce the after data set.</td>
</tr>
<tr>
<td>`-value field [ -ci</td>
<td>-cs ] [ param params ]`</td>
</tr>
</tbody>
</table>

Example

This example assumes that the input data set records contain `customer`, `month`, and `balance` fields. The operation examines the `customer` and `month` fields of each input record for differences. By default, Orchestrates inserts partition and sort components to meet the partitioning and sorting needs of the `changeapply` operator and other operators. See the section “Orchestrate-Inserted Partition and Sort Components” on page xxxvi of this Reference for information on this facility.
**Figure 1** shows the data flow diagram for the example.

Here is the `osh` command:

```bash
$ osh 'hash -key month -key customer < beforeRaw.ds |
  tsort -key month -key customer | copy > before_capture.v >
  before_apply.v;
hash -key month -key customer < afterRaw.ds |
  tsort -key month -key customer > after.v;
changepage -key month -key customer -value balance <
  before_capture.v < after.v > change.v;
changepage -key month -key customer -value balance <
  before_apply.v < change.v > after.ds'
```
The changecapture Operator

This chapter describes the changecapture operator which inputs two data sets and outputs a single data set whose records represent the changes made to the first input data set to obtain the second input data set.

Data Flow Diagram 2 2
Properties 2 2
Key and Value Fields 2 3
Transfer Behavior 2 3
Determining Differences 2 3
Syntax and Options 2 4
Example 1: All Output Results 2 8
Example 2: Dropping Output Results 2 8

The changecapture operator takes two input data sets, denoted before and after, and outputs a single data set whose records represent the changes made to the before data set to obtain the after data set. The operator produces a change data set, whose schema is transferred from the schema of the after data set with the addition of one field: a change code with values encoding the four actions: insert, delete, copy, and edit. The preserve-partitioning flag is set on the change data set.

You can use the companion operator changeapply to combine the changes from the changecapture operator with the original before data set to reproduce the after data set.
The `changecapture` operator is very similar to the `diff` operator described in Chapter 5, “The diff Operator.”

**Data Flow Diagram**

```
before data set                    after data set

key:... value:... beforeRec:*;     key:... value:... afterRec:*;

change:capture
change_code:int8; changeRec:*
```

**Properties**

Table 3  `changecapture` Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>2</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>beforeRec:<em>,afterRec:</em></td>
</tr>
<tr>
<td>Output interface schema</td>
<td>changeRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>afterRec:* -&gt; changeRec:* without record modification,</td>
</tr>
<tr>
<td></td>
<td>beforeRec:* -&gt; changeRec:* with changes to match changeRec:*'s schema</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
<tr>
<td>Input partitioning style</td>
<td>keys in same partition</td>
</tr>
<tr>
<td>Output partitioning style</td>
<td>keys in same partition</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>any (parallel mode)</td>
</tr>
<tr>
<td>Collection method</td>
<td>any (sequential mode)</td>
</tr>
</tbody>
</table>
Key and Value Fields

Records from the two input data sets are compared using key and value fields which must be top-level non-vector fields and can be nullable. Using the -param suboption of the -key, -allkeys, -allvalues, and -value options, you can provide comparison arguments to guide the manner in which key and value fields are compared. In the case of equal key fields, the value fields are compared to distinguish between the copy and edit cases.

Transfer Behavior

In the insert and edit cases, the after input is transferred to output. In the delete case, an internal transfer adapter transfers the before keys and values to output. In the copy case, the after input is optionally transferred to output.

Because an internal transfer adapter is used, no user transfer or view adapter can be used with the changecapture operator.

Determining Differences

The changecapture output data set has the same schema as the after data set, with the addition of a change_code field. The contents of the output depend on whether the after record represents an insert, delete, edit, or copy to the before data set:

- **Insert**: a record exists in the after data set but not the before data set as indicated by the sorted key fields. The after record is consumed and transferred to the output. No before record is consumed.

  If key fields are not unique, changecapture may fail to identify an inserted record with the same key fields as an existing record. Such an insert may be represented as a series of edits, followed by an insert of an existing record. This has consequences for changeapply.

- **Delete**: a record exists in the before data set but not the after data set as indicated by the sorted key fields. The before record is consumed and the key and value fields are transferred to the output; no after record is consumed.

  If key fields are not unique, changecapture may fail to identify a deleted record if another record with the same keys exists. Such a delete may be

---

Table 3  changecapture Operator Properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>set</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

Orchestrate 7.0 Operators Reference

23
represented as a series of edits, followed by a delete of a different record. This has consequences for \texttt{changeapply}.

**Edit:** a record exists in both the before and after data sets as indicated by the sorted key fields, but the before and after records differ in one or more value fields. The before record is consumed and discarded; the after record is consumed and transferred to the output.

If key fields are not unique, or sort order within a key is not maintained between the before and after data sets, spurious edit records may be generated for those records whose sort order has changed. This has consequences for \texttt{changeapply}.

**Copy:** a record exists in both the before and after data sets as indicated by the sorted key fields, and furthermore the before and after records are identical in value fields as well. The before record is consumed and discarded; the after record is consumed and optionally transferred to the output. If no after record is transferred, no output is generated for the record; this is the default.

The operator produces a change data set, whose schema is transferred from the schema of the after data set, with the addition of one field: a change code with values encoding insert, delete, copy, and edit. The preserve-partitioning flag is set on the change data set.

**Syntax and Options**

\texttt{changeapply}

```
-key \texttt{input\_field\_name} \{-cs | ci\} \{-asc | -desc\} \{-nulls first | last\}
  \{-param \texttt{params}\}

-allkeys \{-cs | ci\} \{-asc | -desc\} \{-nulls first | last\}
  \{-param \texttt{params}\}

-allvalues \{-cs | ci\} \{-param \texttt{params}\}

-codeField \texttt{field\_name}

-copyCode n

-collation_sequence \texttt{locale|collation\_file\_pathname|OFF}

-deleteCode n

-doStats

-dropkey \texttt{input\_field\_name}...

-dropvalue \texttt{input\_field\_name}...

-editCode n

-insertCode n

-keepCopy | -dropCopy

-keepDelete | -dropDelete

-keepEdit | -dropEdit
```
The changecapture Operator Syntax and Options

Syntax:

```java
[-keepinsert | -dropinsert]
[-value input_field_name [-ci | -cs] [-param params] ...]
```

Terms in italic typeface are option strings you supply. When your option string contains a space or a tab character, you must enclose it in single quotes.

You must specify either one or more `-key` fields or the `-allkeys` option. You can parameterize each key field's comparison operation and specify the expected sort order (the default is ascending).

**Note**
The `-checkSort` option has been deprecated. By default, partitioner and sort components are now inserted automatically.

### Table 4  changecapture Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-key</code></td>
<td>`key input_field_name [-cs</td>
</tr>
<tr>
<td><code>-allkeys</code></td>
<td>`allkeys [-cs</td>
</tr>
</tbody>
</table>
Table 4  

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-allvalues</td>
<td>Specify that all fields not otherwise explicitly declared are value fields. The -ci option specifies that the comparison of value keys is case insensitive. The -cs option specifies a case-sensitive comparison, which is the default. The -param option allows you to specify extra parameters for a key. Specify parameters using property=value pairs separated by commas. The -allvalues option is mutually exclusive with the -value and -allkeys options. You must specify the -allvalues option when you supply the -dropkey option.</td>
</tr>
<tr>
<td>-codeField</td>
<td>Optionally specify the name of the change code field. The default is change_code.</td>
</tr>
</tbody>
</table>
| -collation_sequence | This option determines how your string data is sorted. You can:  

  • Specify a predefined IBM ICU locale     
  • Write your own collation sequence using ICU syntax, and supply its collation_file_pathname  
  • Specify OFF so that string comparisons are made using Unicode code-point value order, independent of any locale or custom sequence.  

  By default, Orchestrate sorts strings using byte-wise comparisons. For more information, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site: http://oss.software.ibm.com/icu/userguide/Collate_Intro.html |
| -copyCode       | Optionally specify the value of the change_code field in the output record for the copy result. The n value is an int8. The default value is 0. A copy result means that all keys and all values in the before data set are equal to those in the after data set. |
| -deleteCode     | Optionally specify the value for the change_code field in the output record for the delete result. The n value is an int8. The default value is 2. A delete result means that a record exists in the before data set but not in the after data set as defined by the key fields. |
Table 4  changecapture Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-doStats</td>
<td>Optionally configure the operator to display result information</td>
</tr>
<tr>
<td></td>
<td>containing the number of input records and the number of copy,</td>
</tr>
<tr>
<td></td>
<td>delete, edit, and insert records.</td>
</tr>
<tr>
<td>-dropkey</td>
<td>-dropkey input_field_name</td>
</tr>
<tr>
<td></td>
<td>Optionally specify that the field is not a key field. If you specify</td>
</tr>
<tr>
<td></td>
<td>this option, you must also specify the -allkeys option.</td>
</tr>
<tr>
<td></td>
<td>There can be any number of occurrences of this option.</td>
</tr>
<tr>
<td>-dropvalue</td>
<td>-dropvalue input_field_name</td>
</tr>
<tr>
<td></td>
<td>Optionally specify that the field is not a value field. If you specify</td>
</tr>
<tr>
<td></td>
<td>this option, you must also specify the -allvalues option.</td>
</tr>
<tr>
<td></td>
<td>There can be any number of occurrences of this option.</td>
</tr>
<tr>
<td>-editCode</td>
<td>-editCode n</td>
</tr>
<tr>
<td></td>
<td>Optionally specify the value for the change_code field in the output</td>
</tr>
<tr>
<td></td>
<td>record for the edit result. The n value is an int8. The default value</td>
</tr>
<tr>
<td></td>
<td>is 3. A n edit result means all key fields are equal but one or more</td>
</tr>
<tr>
<td></td>
<td>value fields are different.</td>
</tr>
<tr>
<td>-insertCode</td>
<td>-insertCode n</td>
</tr>
<tr>
<td></td>
<td>Optionally specify the value for the change_code field in the output</td>
</tr>
<tr>
<td></td>
<td>record for the insert result. The n value is an int8. The default value</td>
</tr>
<tr>
<td></td>
<td>is 1. A n insert result means that a record exists in the after data</td>
</tr>
<tr>
<td></td>
<td>set but not in the before data set as defined by the key fields.</td>
</tr>
<tr>
<td>-keepCopy</td>
<td>-keepCopy</td>
</tr>
<tr>
<td>-dropCopy</td>
<td>-keepDelete</td>
</tr>
<tr>
<td>-keepDelete</td>
<td>-keepEdit</td>
</tr>
<tr>
<td>-dropDelete</td>
<td>-keepInsert</td>
</tr>
<tr>
<td>-keepEdit</td>
<td>-dropEdit</td>
</tr>
<tr>
<td>-keepInsert</td>
<td>-dropInsert</td>
</tr>
<tr>
<td></td>
<td>Optionally specifies whether to keep or drop copy records at output.</td>
</tr>
<tr>
<td></td>
<td>By default, the operator creates an output record for all differences</td>
</tr>
<tr>
<td></td>
<td>except copy.</td>
</tr>
</tbody>
</table>
Example 1: All Output Results

This example assumes that the input data set records contain customer, month, and balance fields. The operation examines the customer and month fields of each input record for differences. By default, Orchestrates inserts partition and sort components to meet the partitioning and sorting needs of the changecapture operator and other operators. See the section “Orchestrate-Inserted Partition and Sort Components” on page xxxvi of this Reference for information on this facility.

Here is the osh command:

```
$ osh "changecapture -key month -key customer -value balance <
    before_capture.v < after.v > change.ds"
```

Example 2: Dropping Output Results

In some cases, you may be interested only in some results of the changecapture operator. In this example, you keep only the output records of the edit, delete and insert results. That is, you explicitly drop the copy results so that the output data set contains records only when there is a difference between the before and after data records.

As in Example 1, this example assumes that the before and after data sets are already sorted. Shown below is the data flow diagram for this example:
You specify these key and value fields to the `changecapture` operator:

- `key month`
- `key customer`
- `value balance`

After you run the `changecapture` operator, you invoke the `switch` operator to divide the output records into data sets based on the result type. The `switch` operator in this example creates three output data sets: one for delete results, one for edit results, and one for insert results. It creates only three data sets, because you have explicitly dropped copy results from the `changecapture` operator by specifying `-dropCopy`. By creating a separate data set for each of the three remaining result types, you can handle each one differently:

- `deleteCode 0`
- `editCode 1`
- `insertCode 2`

Here is the `osh` command:

```bash
$ osh "changecapture -key month -key customer -value balance
   -dropCopy -deleteCode 0 -editCode 1 -insertCode 2
   < before.ds < after.ds | switch -key changecapture
   > outDelete.ds > outEdit.ds > outInsert.ds"
```
The compare Operator

The compare operator performs a field-by-field comparison of records in two presorted input data sets. It is analogous to the UNIX comp command.

Data Flow Diagram  3 2
Properties  3 2
   Restrictions  3 3
   Results Field  3 4
Syntax and Options  3 4
Example 1: Running the compare Operator in Parallel  3 6
Example 2: Running the compare Operator Sequentially  3 7

The compare operator performs a field-by-field comparison of records in two presorted input data sets. This operator compares the values of top-level non-vector data types such as strings. All appropriate comparison parameters are supported, for example, case sensitivity and insensitivity for string comparisons.

The compare operator does not change the schema, partitioning, or content of the records in either input data set. It transfers both data sets intact to a single output data set generated by the operator. The comparison results are also recorded in the output data set.

By default, Orchestrates inserts partition and sort components to meet the partitioning and sorting needs of the changecapture operator and other
operators. See the section “Orchestrate-Inserted Partition and Sort Components” on page xxxvi of this Reference for information on this facility.

**Data Flow Diagram**

![Data Flow Diagram](image)

**Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>2</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>key0:type0; ... keyN:typeN; inRec:*;</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>result:int8; first:subrec(rec:<em>); second:subrec(rec:</em>);</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>The first input data set is transferred to first.rec, The second input data set is transferred to second.rec</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
<tr>
<td>Input partitioning style</td>
<td>keys in same partition</td>
</tr>
</tbody>
</table>
Table 5  compare Operator Properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning method</td>
<td>same (parallel mode)</td>
</tr>
<tr>
<td>Collection method</td>
<td>any (sequential mode)</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

The compare operator:

- Compares only scalar data types. See “Restrictions” on page 3-3.
- Takes two presorted data sets as input.
  - Outputs one data set.
  - Has an input interface schema consisting of the key fields and the schema variable inRec, and an output interface schema consisting of the result field of the comparison and a subrecord field containing each input record.
  - Performs a field-by-field comparison of the records of the input data sets.
  - Transfers the two input data sets to the single output data set without altering the input schemas, partitioning, or values.
  - Writes to the output data set signed integers that indicate comparison results.

Restrictions

The compare operator:

- Compares only scalar data types, specifically string, integer, float, decimal, raw, date, time, and timestamp; you cannot use the operator to compare data types such as tagged aggregate, subrec, vector, and so on.
- Compares only fields explicitly specified as key fields, except when you do not explicitly specify any key field. In that case, the operator compares all fields that occur in both records.
Results Field

The operator writes the following default comparison results to the output data set. In each case, you can specify an alternate value:

Table 6  Default Results Value of the compare Operator

<table>
<thead>
<tr>
<th>Description of Comparison Results</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>The record in the first input data set is greater than the corresponding record in the second data set.</td>
<td>1</td>
</tr>
<tr>
<td>The record in the first input data set is equal to the corresponding record in the second data set.</td>
<td>0</td>
</tr>
<tr>
<td>The record in the first input data set is less than the corresponding record in the second data set.</td>
<td>-1</td>
</tr>
<tr>
<td>The number of records in the first input data set is greater than the number of records in the second data set.</td>
<td>2</td>
</tr>
<tr>
<td>The number of records in the first input data set is less than the number of records in the second data set.</td>
<td>-2</td>
</tr>
</tbody>
</table>

When this operator encounters any of the mismatches described in the table shown above, you can force it to take one or both of the following actions:

- Terminate the remainder of the current comparison
- Output a warning message to the screen

Syntax and Options

```
compare [-abortOnDifference]
  [-field fieldname [-ci | -cs] [-param params] ...]
  | [-key fieldname [-ci | -cs] [-param params] ...]
  | [-collation_sequence locale|collation_file_pathname| OFF]
  | [-gt n | -eq n | -lt n]
  | [-second n]
  | [-warnRecordCountMismatch]
```

No options are required.
Table 7  Options to the compare Operator

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-abortOnDifference</td>
<td>Forces the operator to abort its operation each time a difference is encountered between two corresponding fields in any record of the two input data sets. This option is mutually exclusive with -warnRecordCountMismatch, -lt, -gt, -first, and -second.</td>
</tr>
<tr>
<td>-collation_sequence</td>
<td>locale</td>
</tr>
<tr>
<td>-field or -key</td>
<td>-field fieldname [-ci</td>
</tr>
</tbody>
</table>

- field or -key is a key field to compare in the two input data sets. The maximum number of fields is the number of fields in the input data sets. If no key fields are explicitly specified, all fields shared by the two records being processed are compared.

fieldname specifies the name of the field.

-ci specifies that the comparison of strings is case-insensitive.

-cs specifies case-sensitive string comparison, which is the default.

The -param suboption allows you to specify extra parameters for a field. Specify parameters using property=value pairs separated by commas.
Example 1: Running the compare Operator in Parallel

Each record has the fields name, age, and gender. All operations are performed on the key fields, age and gender. By default, Orchestrates inserts partition and sort components to meet the partitioning and sorting needs of the compare operator and other operators. See the section “Orchestrate-Inserted Partition and Sort Components” on page xxxvi of this Reference for information on this facility.

The compare operator runs in parallel mode which is the default mode for this operator; and the -abortOnDifference option is selected to force the operator to abort at the first indication of mismatched records.
Here is the `osh` code corresponding to these operations:

```
$ osh '"compare -abortOnDifference -field age -field gender
< sortedDS0.v < sortedDS1.v > outDS.ds"
```

The output record format for a successful comparison of records looks like this, assuming all default values are used:

`result:0 first:name; second:age; third:gender;`

**Example 2: Running the compare Operator Sequentially**

By default, the `compare` operator executes in parallel on all processing nodes defined in the default node pool.

However, you might want to run an application the operator sequentially on a single node. This could be useful when you intend to store a persistent data set to disk in a single partition. For example, your parallel application may perform data cleansing and data reduction on its input to produce an output data set that is much smaller than the input. Before storing the results to disk, or passing the result to a sequential application, you can use a sequential `compare` operator to store the data set to disk with a single partition.

To force the operator to execute sequentially specify the `[-seq]` framework argument. When executed sequentially, the operator uses a collection method of `any`. A sequential operator using this collection method can have its collection method overridden by an input data set to the operator.

Suppose you want to run same application as shown in “Example 1: Running the compare Operator in Parallel” on page 3-6 but you want the `compare` operator to run sequentially.

Issue this `osh` command:

```
$ osh '"compare -field gender -field age [-seq] < inDS0.ds < inDS1.ds
> outDS.ds"
```
The copy Operator

The copy operator copies each record of an input data set to zero or more output data sets without modification. It is analogous to the UNIX cp command. It is useful for backing up a data set.

Data Flow Diagram 4 2
Properties 4 2
Syntax and Options 4 3
Preventing Orchestrate from Removing a copy Operator 4 3
Example 1: The copy Operator 4 5
Example 2: Running the copy Operator Sequentially 4 5

You can use the modify operator with the copy operator to modify the data set as the operator performs the copy operation. See the chapter on the modify operator for more information on modifying data.
Data Flow Diagram

Properties

Table 8  **copy Operator Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>0 or more (0 - n) set by user</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec -&gt; outRec without record modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>any (parallel mode)</td>
</tr>
<tr>
<td>Collection method</td>
<td>any (sequential mode)</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
<tr>
<td>Combinable operator</td>
<td>yes</td>
</tr>
</tbody>
</table>

The **copy** operator:

- Takes any single data set as input
- Has an input interface schema consisting of a single schema variable inRec and an output interface schema consisting of a single schema variable outRec
• Copies the input data set to the output data sets without affecting the record schema or contents

Syntax and Options

```
copy [-checkpoint n] [-force]
```

Table 9  copy Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-checkpoint n</td>
<td>Specifies the number of records copied from the input persistent data set to each segment of each partition of the output data set. The value of n must be positive. Its default value is 1. In order for this option to be specified, the input data set to the copy operator must be persistent and the operator must be run in parallel. The step containing the copy operator must be checkpointed, that is, you must have specified the keyword -restartable as part of the step definition.</td>
</tr>
<tr>
<td>-force</td>
<td>Specifies that Orchestrate cannot attempt to optimize the step by removing the copy operator. In some cases, Orchestrate can remove a copy operator if it determines that the copy operator is unnecessary. However, your application may require the copy operator to execute. In this case, you use the -force option. See “Preventing Orchestrate from Removing a copy Operator” on page 4-3.</td>
</tr>
</tbody>
</table>

Preventing Orchestrate from Removing a copy Operator

Before running an application, Orchestrate optimizes each step. As part of this optimization, Orchestrate removes unnecessary copy operators. However, this optimization can sometimes remove a copy operator that you do not want removed.
For example, the following data flow imports a single file into a virtual data set, then copies the resulting data set to a new data set:

Here is the `osh` command:

```
$ osh "import -file inFile.dat -schema recordSchema | copy > outDS.ds"
```

This occurs in the step:

1. The `import` operator reads the data file, `inFile.data`, into a virtual data set. The virtual data set is written to a single partition because it reads a single data file. In addition, the `import` operator executes only on the processing node containing the file.

2. The `copy` operator runs on all processing nodes in the default node pool, because no constraints have been applied to the input operator. Thus, it writes one partition of `outDS.ds` to each processing node in the default node pool.

However, if Orchestrate removes the `copy` operator as part of optimizing the step, the resultant persistent data set, `outDS.ds`, would be stored only on the processing node executing the `import` operator. In this example, `outDS.ds` would be stored as a single partition data set on one node.

To prevent removal specify the `-force` option. The operator explicitly performs the repartitioning operation to spread the data over the system.
Example 1: The copy Operator

In this example, you sort the records of a data set. However, before you perform the sort, you use the **copy** operator to create two copies of the data set: a persistent copy, which is saved to disk, and a virtual data set, which is passed to the sort operator. Here is a data flow diagram of the operation:

```
step
  ...
  ↓
  copy
  ↓
  tsort
```

Output data set 0 from the copy operator is written to `outDS1.ds` and output data set 1 is written to the tsort operator.

The syntax is as follows:

```
$ osh "... | copy > outDS1.ds | tsort options ... "  
```

Example 2: Running the copy Operator Sequentially

By default, the **copy** operator executes in parallel on all processing nodes defined in the default node pool. However, you may have an application in which you want to run the operator sequentially, that is, on a single node. For example, you might want to store a persistent data set to disk in a single partition.

You can run the operator sequentially by specifying the `[seq]` framework argument to the **copy** operator.

When run sequentially, the operator uses a collection method of **any**. However, you can override the collection method of a sequential operator. This can be useful when you want to store a sorted data set to a single partition. Shown below is a data flow example using the **ordered** collection operator with a sequential **copy** operator:

```
Here is the osh command:
$ osh "... op1 | ordered | copy [seq] > outDS.ds "  
```
The diff Operator

The diff operator has been replaced by the changecapture operator. Ascential recommends you use the changecapture operator for new development.

Data Flow Diagram 5 2
Properties 5 2
   Transfer Behavior 5 3
   Key Fields 5 4
   Identical Field Names 5 4
   Determining Differences 5 4
Syntax and Options 5 6
Example 1: General Example 5 9
Example 2: Dropping Output Results 5 9

Important The diff operator has been superseded by the changecapture operator. While the diff operator has been retained for backwards compatibility, Ascential recommends that you use the changecapture operator for new development.

The diff operator performs a record-by-record comparison of two versions of the same data set (the before and after data sets) and outputs a data set whose records represent the difference between them. The operator assumes that the input data sets are hash-partitioned and sorted in ascending order on the key fields you specify for the comparison.
The comparison is performed based on a set of difference key fields. Two records are copies of one another if they have the same value for all difference keys. In addition, you can specify a set of value key fields. If two records are copies based on the difference key fields, the value key fields determine if one record is a copy or an edited version of the other.

The `diff` operator is very similar to the `changecapture` operator described in Chapter 2, “The changecapture Operator.” In most cases, you should use the `changecapture` operator rather than the `diff` operator.

By default, Orchestrates inserts partition and sort components to meet the partitioning and sorting needs of the `diff` operator and other operators. See the section “Orchestrate-Inserted Partition and Sort Components” on page -xxxvi of this Reference for information on this facility.

**Note** The Orchestrate `diff` operator does not behave like the UNIX `diff`.

### Data Flow Diagram

The input data sets are known as the before and after data sets.

![Data Flow Diagram](image)

### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>2</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
</tbody>
</table>
The operator produces a single output data set, whose schema is the concatenation of the before and after input schemas. Each record of the output data set has the following format:

The usual name conflict resolution rules apply.

The output data set contains a number of records in the range:

\[
\text{num\_in\_before} \leq \text{num\_in\_output} \leq (\text{num\_in\_before} + \text{num\_in\_after})
\]
The number of records in the output data set depends on how many records are copies, edits, and deletes. If the before and after data sets are exactly the same, the number of records in the output data set equals the number of records in the before data set. If the before and after data sets are completely different, the output data set contains one record for each before and one record for each after data set record.

**Key Fields**

The before data set’s schema determines the difference key type. You can use an upstream modify operator to alter it. The after data set’s key field(s) must have the same name as the before key field(s) and be either of the same data type or of a compatible data type.

The same rule holds true for the value fields: The after data set’s value field(s) must be of the same name and data type as the before value field(s). You can use an upstream modify operator to bring this about.

Only top-level, non-vector, non-nullable fields may be used as difference keys. Only top-level, non-vector fields may be used as value fields. Value fields may be nullable.

**Identical Field Names**

When the two input data sets have the same field name, the diff operator retains the field of the first input, drops the identically named field from the second output, and issues a warning for each dropped field. Override the default behavior by modifying the second field name so that both versions of the field are retained in the output. (See Chapter 13, “The modify Operator”.) You can then write a custom operator to select the version you require for a given application.

**Determining Differences**

The diff operator reads the current record from the before data set, reads the current record from the after data set, and compares the records of the input data sets using the difference keys. The comparison results are classified as follows:

- **Insert**: A record exists in the after data set but not the before data set.
  - The operator transfers the after record to the output.
  - The operator does not copy the current before record to the output but retains it for the next iteration of the operator.
  - The data type’s default value is written to each before field in the output.
  - By default the operator writes a 0 to the diff field of the output record.

- **Delete**: A record exists in the before data set but not the after data set.
- The operator transfers the before record to the output.
- The operator does not copy the current after record to the output but retains it for the next iteration of the operator.
- The data type's default value is written to each after field in the output.
- By default, the operator writes a 1 to the **diff** field of the output record.

**Copy:** The record exists in both the before and after data sets and the specified value field values have not been changed.
- The before and after records are both transferred to the output.
- By default, the operator writes a 2 to the **diff** (first) field of the output record.

**Edit:** The record exists in both the before and after data sets; however, one or more of the specified value field values have been changed.
- The before and after records are both transferred to the output.
- By default, the operator writes a 3 to the **diff** (first) field of the output record.

Options are provided to drop each kind of output record and to change the numerical value written to the **diff** (first) field of the output record.

In addition to the difference key fields, you can optionally define one or more value key fields. If two records are determined to be copies because they have equal values for all the difference key fields, the operator then examines the value key fields.

- Records whose difference and value key fields are equal are considered copies of one another. By default, the operator writes a 2 to the **diff** (first) field of the output record.
- Records whose difference key fields are equal but whose value key fields are not equal are considered edited copies of one another. By default, the operator writes a 3 to the **diff** (first) field of the output record.
Syntax and Options

You must specify at least one difference key to the operator using `-key`.

diff
   -key field [-ci | -cs] [-param params]
   [-key field [-ci | -cs] [-param params] ...]
   [-allValues [-ci | -cs] [-param params]]
   [-collation_sequence locale|collation_file_pathname| OFF]
   [-copyCode n]
   [-deleteCode n]
   [-dropCopy]
   [-dropDelete]
   [-dropEdit]
   [-dropInsert]
   [-editCode n]
   [-insertCode n]
   [-stats]
   [-tolerateUnsorted]
   [-value field [-ci | -cs] [-param params] ...]

Table 11  diff Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-key</code></td>
<td>-key field [-ci</td>
</tr>
</tbody>
</table>

   Specifies the name of a difference key field. The `-key` option may be repeated if there are multiple key fields.

   Note that you cannot use a nullable, vector, subrecord, or tagged aggregate field as a difference key.

   The `-ci` option specifies that the comparison of difference key values is case insensitive. The `-cs` option specifies a case-sensitive comparison, which is the default.

   The `-params` suboption allows you to specify extra parameters for a key. Specify parameters using property=value pairs separated by commas.
- **allValues**: Specifies that all fields other than the difference key fields identified by \(-key\) are used as value key fields. The operator does not use vector, subrecord, and tagged aggregate fields as value keys and skips fields of these data types.

When a before and after record are determined to be copies based on the difference keys, the value keys can then be used to determine if the after record is an edited version of the before record. The \(-ci\) option specifies that the comparison of value keys is case insensitive. The \(-cs\) option specifies a case-sensitive comparison, which is the default.

The \(-params\) suboption allows you to specify extra parameters for a key. Specify parameters using property=value pairs separated by commas.

- **collation_sequence**: This option determines how your string data is sorted. You can:
  - Specify a predefined IBM ICU locale
  - Write your own collation sequence using ICU syntax, and supply its collation_file_pathname
  - Specify OFF so that string comparisons are made using Unicode code-point value order, independent of any locale or custom sequence.

By default, Orchestrate sorts strings using byte-wise comparisons.

For more information, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:


- **copyCode**: Specifies the value for the diff field in the output record when the before and after records are copies. The value is an int8. The default value is 2.

A copy means all key fields and all optional value fields are equal.

- **deleteCode**: Specifies the value for the diff field in the output record for the delete result. The value is an int8. The default value is 1.

A delete result means that a record exists in the before data set but not in the after data set as defined by the difference key fields.
Table 11  **diff Operator Options (continued)**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-dropCopy</td>
<td>Specifies to drop the output record, meaning not generate it, for any one of the four difference result types. By default, an output record is always created by the operator. You can specify any combination of these four options.</td>
</tr>
<tr>
<td>-dropDelete</td>
<td></td>
</tr>
<tr>
<td>-dropEdit</td>
<td></td>
</tr>
<tr>
<td>-dropInsert</td>
<td></td>
</tr>
<tr>
<td>-editCode n</td>
<td>Specifies the value for the <strong>diff</strong> field in the output record for the edit result. The <strong>n</strong> value is an int8. The default value is 3. An edit result means all difference key fields are equal but one or more value key fields are different.</td>
</tr>
<tr>
<td>-insertCode n</td>
<td>Specifies the value for the <strong>diff</strong> field in the output record for the insert result. The <strong>n</strong> value is an int8. The default value is 0. An insert result means that a record exists in the after data set but not in the before data set as defined by the difference key fields.</td>
</tr>
<tr>
<td>-stats</td>
<td>Configures the operator to display result information containing the number of input records and the number of copy, delete, edit, and insert records.</td>
</tr>
<tr>
<td>-tolerateUnsorted</td>
<td>Specifies that the input data sets are not sorted. By default, the operator generates an error and aborts the step when detecting unsorted inputs. This option allows you to process groups of records that may be arranged by the difference key fields but not sorted. The operator consumes input records in the order in which they appear on its input. If you use this option, no automatic partitioner or sort insertions are made.</td>
</tr>
</tbody>
</table>
Example 1: General Example

The following example assumes that the input data set records contain a customer and month field. The operator examines the customer and month fields of each input record for differences. By default, Orchestrates inserts partition and sort components to meet the partitioning and sorting needs of the diff operator and other operators. See the section “Orchestrate-Inserted Partition and Sort Components” on page xxxvi of this Reference for information on this facility.

Here is the osh command:

```
$ osh "diff -key month -key customer < before.v < after.v > outDS.ds"
```

Example 2: Dropping Output Results

In some cases, you may be interested only in some results of the diff operator. In this example, you keep only the output records of the edit, delete and insert results. That is, you explicitly drop the copy results so that the output data set contains records only when there is a difference between the before and after data records.
Figure 3 shows the data flow for this example.

Figure 3  **Data Flow Diagram for diff Example 2**

You specify these **key** and **value** fields to the **diff** operator:

```plaintext
key=month
key=customer
value=balance
```

After you run the **diff** operator, you invoke the **switch** operator to divide the output records into data sets based on the result type. The **switch** operator in this example creates three output data sets: one for delete results, one for edit results, and one for insert results. It creates only three data sets, because you have explicitly dropped copy results from the **diff** operation by specifying `-dropCopy`. By creating a separate data set for each of the three remaining result types, you can handle each one differently:

```plaintext
delEteCode=0
edItCode=1
insErtCode=2
```

Here is the **osh** command:

```bash
$ osh " diff -key month -key customer -value balance 
   -dropCopy -deleteCode 0 -editCode 1 -insertCode 2 
< before.ds < after.ds | switch -key diff 
> outDelete.ds > outEdit.ds > outInsert.ds"
```
The encode Operator

The encode operator encodes or decodes an Orchestrate data set using a UNIX encoding command that you supply.

- Data Flow Diagram 6 2
- Properties 6 2
- Syntax and Options 6 3
- Encoding Orchestrate Data Sets 6 4
  - Processing Encoded Data Sets 6 4
  - Encoded Data Sets and Partitioning 6 4
- Example 6 5

The `encode` operator encodes or decodes an Orchestrate data set using a UNIX encoding command that you supply. The operator can convert an Orchestrate data set from a sequence of records into a stream of raw binary data. The operator can also reconver the data stream to an Orchestrate data set.
Data Flow Diagram

In the figure shown above, the mode argument specifies whether the operator is performing an encoding or decoding operation. Possible values for mode are:

- **encode**: encode the input data set
- **decode**: decode the input data set

Properties

Table 12  **encode Operator Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
</tbody>
</table>
| Input interface schema    | mode = encode: in:*;  
                          | mode = decode: none                                                  |
| Output interface schema   | mode = encode: none                                                  |
                          | mode = decode: out:*;                                                |
| Transfer behavior         | in -> out without record modification for an encode/ decode cycle   |
| Execution mode            | parallel (default) or sequential                                     |
| Partitioning method       | mode = encode: any                                                   |
                          | mode = decode: same                                                  |
Table 12  encode Operator Properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection method</td>
<td>any</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>mode = encode: sets</td>
</tr>
<tr>
<td></td>
<td>mode = decode: propagates</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
<tr>
<td>Combinable operator</td>
<td>no</td>
</tr>
</tbody>
</table>

Syntax and Options

Terms in italic typeface are option strings you supply. When your option string contains a space or a tab character, you must enclose it in single quotes.

```
encode -command command_line
    [-direction encode | decode]
    
    [-mode [encode | decode]]
    
    [-encode | -decode]
```

Table 13  encode Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-command</td>
<td>Specifies the command line used for encoding/decoding. The command line must configure the UNIX command to accept input from stdin and write its results to stdout. The command must be located in the search path of your application and be accessible by every processing node on which the encode operator executes.</td>
</tr>
<tr>
<td>-direction or -mode</td>
<td>Specifies the mode of the operator. If you do not select a direction, it defaults to encode.</td>
</tr>
<tr>
<td>-encode</td>
<td>Specify encoding of the data set. Encoding is the default mode.</td>
</tr>
<tr>
<td>-decode</td>
<td>Specify decoding of the data set.</td>
</tr>
</tbody>
</table>
Encoding Orchestrate Data Sets

Each record of an Orchestrate data set has defined boundaries that mark its beginning and end. The **encode** operator lets you invoke a UNIX command that encodes an Orchestrate data set, which is in record format, into raw binary data and vice versa.

Processing Encoded Data Sets

An encoded data set is similar to an Orchestrate data set. An encoded, persistent data set is stored on disk in the same way as a normal data set, by two or more files:

- A single descriptor file
- One or more data files

See the chapter on Orchestrate data sets in the Orchestrate 7.0 User Guide for more information on persistent data sets.

However, an encoded data set cannot be accessed like a standard Orchestrate data set, because its records are in an encoded binary format. Nonetheless, you can specify an encoded data set to any operator that does no field-based processing or reordering of the records. For example, you can invoke the **copy** operator to create a copy of the encoded data set.

You can further encode a data set using a different encoding operator to create an encoded-compressed data set. For example, you might compress the encoded file using Orchestrate's **pcompress** operator (see Chapter 14, “The pcompress Operator”), then invoke the **unencode** command to convert a binary file to an emailable format. You would then restore the data set by first decompressing and then decoding the data set.

Encoded Data Sets and Partitioning

When you encode a data set, you remove its normal record boundaries. The encoded data set cannot be repartitioned, because partitioning in Orchestrate is performed record-by-record. For that reason, the **encode** operator sets the preserve-partitioning flag in the output data set. This prevents an Orchestrate operator that uses a partitioning method of any from repartitioning the data set and causes Orchestrate to issue a warning if any operator attempts to repartition the data set.

For a decoding operation, the operator takes as input a previously encoded data set. The preserve-partitioning flag is propagated from the input to the output data set.
See the chapter on partitioning in the Orchestrate 7.0 User Guide for more information on the preserve-partitioning flag.

**Example**

In the following example, the **encode** operator compresses a data set using the UNIX **gzip** utility. By default, **gzip** takes its input from **stdin**. You specify the **-c** switch to configure the operator to write its results to **stdout** as required by the operator:

Here is the **osh** code for this example:

```
$ osh " ... op1 | encode -command 'gzip' > encodedDS.ds"
```

The following example decodes the previously compressed data set so that it may be used by other Orchestrate operators. To do so, you use an instance of the **encode** operator with a mode of **decode**. In a converse operation to the encoding, you specify the same operator, **gzip**, with the **-cd** option to decode its input.

Here is the **osh** command for this example:

```
$ osh "encode -decode -command 'gzip -d' < inDS.ds | op2 ..."
```

In this example, the command line uses the **-d** switch to specify the decompress mode of **gzip**.
The filter Operator

The filter operator transfers the records of the input data set which satisfy your requirements, and filters out all other records without modification.

Data Flow Diagram  7 2
Properties  7 2
Syntax and Options  7 3
Job Monitoring Information  7 5
  Example MetaData and Summary Messages  7 5
  Customizing Job Monitor Messages  7 6
Expressions  7 6
  Input Data Types  7 6
  Supported Boolean Expressions and Operators  7 7
    Order of Association  7 7
    String Comparison  7 8
Example 1: Comparing Two Fields  7 8
Example 2: Testing for a Null  7 8
Example 3: Evaluating Input Records  7 8
Application Scenario: Mailing List for a Wine Auction  7 9
  Schema for Implicit Import  7 9
  OSH Syntax  7 9
  Input Data  7 10
  Outputs  7 10
The **filter** operator transfers the input records in which one or more fields meet the requirements you specify. If you request a reject data set, the **filter** operator transfers records that do not meet the requirements to the reject data set.

### Data Flow Diagram

```
input data set

inRec:*;

filter

outRec:*

... output data sets

optional reject data set
```

### Properties

**Table 14  filter Operator Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1 or more, and, optionally, a reject data set</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*;</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*;</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec -&gt; outRec without record modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel by default, or sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>any (parallel mode)</td>
</tr>
<tr>
<td>Collection method</td>
<td>any (sequential mode)</td>
</tr>
<tr>
<td>Preserve-partitioning flag in</td>
<td>propagated</td>
</tr>
<tr>
<td>output data set</td>
<td></td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
<tr>
<td>Combinable operator</td>
<td>yes</td>
</tr>
</tbody>
</table>
Syntax and Options

The `-where` option is required. Terms in italic typeface are option strings you supply. When your option string contains a space or a tab character, you must enclose it in single quotes.

```plaintext
filter
  -where 'P' [-target dsNum] [-where 'P' [-target dsNum] ... ]
  [-collation_sequence locale|collation_file_pathname|OFF]
  [-first]
  [-nulls first | last]
  [-reject]
```

Table 15  filter Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-where</code></td>
<td><code>-where P [-target dsNum]</code></td>
</tr>
<tr>
<td></td>
<td>Specifies the predicate which determines the filter. In SQL, a predicate is an expression which evaluates as TRUE, FALSE, or UNKNOWN and whose value depends on the value of one or more field values.</td>
</tr>
<tr>
<td></td>
<td>Enclose the predicate in single quotes. Single quotes within the predicate must be preceded by the backslash character (), as in “Example 3: Evaluating Input Records” below. If a field is formatted as a special Orchestrate data type, such as date or timestamp, enclose it in single quotes.</td>
</tr>
<tr>
<td></td>
<td>Multi-byte Unicode character data is supported in predicate field names, constants, and literals.</td>
</tr>
<tr>
<td></td>
<td>Multiple <code>-where</code> options are allowed. Each occurrence of <code>-where</code> causes the output data set to be incremented by one, unless you use the <code>-target</code> suboption.</td>
</tr>
<tr>
<td><code>-first</code></td>
<td><code>-first</code></td>
</tr>
<tr>
<td></td>
<td>Records are output only to the data set corresponding to the first <code>-where</code> clause they match. The default is to write a record to the data sets corresponding to all <code>-where</code> clauses they match.</td>
</tr>
</tbody>
</table>
Table 15  filter Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-collation_sequence</td>
<td>-collation_sequence locale</td>
</tr>
<tr>
<td></td>
<td>This option determines how your string data is sorted. You can:</td>
</tr>
<tr>
<td></td>
<td>• Specify a predefined IBM ICU locale</td>
</tr>
<tr>
<td></td>
<td>• Write your own collation sequence using ICU syntax, and supply its collation_file_pathname</td>
</tr>
<tr>
<td></td>
<td>• Specify OFF so that string comparisons are made using Unicode code-point value order, independent of any locale or custom sequence.</td>
</tr>
<tr>
<td></td>
<td>By default, Orchestrate sorts strings using byte-wise comparisons.</td>
</tr>
<tr>
<td></td>
<td>For more information, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:</td>
</tr>
<tr>
<td>-nulls</td>
<td>-nulls first</td>
</tr>
<tr>
<td></td>
<td>By default, nulls are evaluated first, before other values. To override this default, specify -nulls last.</td>
</tr>
<tr>
<td>-reject</td>
<td>-reject</td>
</tr>
<tr>
<td></td>
<td>By default, records that do not meet specified requirements are dropped. Specify this option to override the default. If you do, attach a reject output data set to the operator.</td>
</tr>
<tr>
<td>-target</td>
<td>-target dsNum</td>
</tr>
<tr>
<td></td>
<td>An optional sub-property of where. Use it to specify the target data set for a where clause. Multiple -where clauses can direct records to the same output data set. If a target data set is not specified for a particular -where clause, the output data set for that clause is implied by the order of all -where properties that do not have the -target sub-property. For example:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Output Data Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>-where &quot;field1 &lt; 4&quot;</td>
<td>0</td>
</tr>
<tr>
<td>-where &quot;field2 like 'bb'&quot;</td>
<td>1</td>
</tr>
<tr>
<td>-where &quot;field3 like 'aa'&quot;</td>
<td>2</td>
</tr>
<tr>
<td>-target 2</td>
<td></td>
</tr>
<tr>
<td>-where &quot;field4 &gt; 10&quot; -target 0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>-where &quot;field5 like 'c.*'&quot;</td>
<td></td>
</tr>
</tbody>
</table>
Job Monitoring Information

The filter operator reports business logic information which can be used to make decisions about how to process data. It also reports summary statistics based on the business logic.

The business logic is included in the metadata messages generated by Orchestrate as custom information. It is identified with:

```
name="BusinessLogic"
```

The output summary per criterion is included in the summary messages generated by Orchestrate as custom information. It is identified with:

```
name="CriterionSummary"
```

The XML tags, criterion, case and where, are used by the filter operator when generating business logic and criterion summary custom information. These tags are used in the example information below.

Example MetaData and Summary Messages

```
<response type="metadata">
 <component ident="filter">
  <componentstats startTime="2002-08-08 14:41:56"/>
  <linkstats portNum="0" portType="in"/>
  <linkstats portNum="0" portType="out/>
  <linkstats portNum="1" portType="out/>
  <custom_info Name="BusinessLogic" Desc="User-supplied logic to filter operator">
   <criterion name="where">
    <where value="true" output_port="0"/>
    <where value="false" output_port="1"/>
   </criterion>
  </custom_info>
 </component>
</response>

<response type="summary">
 <component ident="filter" pid="2239">
  <componentstats startTime="2002-08-08 14:41:59" stopTime="2002-08-08 14:42:40" percentCPU="99.5"/>
  <linkstats portNum="0" portType="in" recProcessed="1000000"/>
  <linkstats portNum="0" portType="out" recProcessed="500000"/>
  <linkstats portNum="1" portType="out" recProcessed="500000"/>
  <custom_info Name="CriterionSummary" Desc="Output summary per criterion">
   <where value="true" output_port="0" recProcessed="500000"/>
   <where value="false" output_port="0" recProcessed="500000"/>
  </custom_info>
 </component>
</response>
```
The Orchestrate Job Monitor is documented in the Orchestrate 7.0 User Guide.

### Customizing Job Monitor Messages

Orchestrate specifies the business logic and criterion summary information for the `filter` operator using the functions `addCustomMetadata()` and `addCustomSummary()`. You can also use these functions to generate similar information for the operators you write. See the section "Customizing Job Monitoring Messages" in Chapter 1 of the Orchestrate 7.0 Developer Guide.

### Expressions

The behavior of the filter operator is governed by expressions that you set. You can use the following elements to specify the expressions:

- Fields of the input data set
- Requirements involving the contents of the fields
- Optional constants to be used in comparisons
- The Boolean operators AND and OR to combine requirements

When a record meets the requirements, it is written unchanged to an output data set. Which of the output data sets it is written to is either implied by the order of your `-where` options or explicitly defined by means of the `-target` suboption.

The `filter` operator supports standard SQL expressions, except when comparing strings.

### Input Data Types

If you specify a single field for evaluation, that field can be of any data type. Note that Orchestrate's treatment of strings differs slightly from that of standard SQL.

If you compare fields they must be of the same or compatible data types. Otherwise, the operation terminates with an error. Compatible data types are those that Orchestrate converts by default. Regardless of any conversions the whole record is transferred unchanged to the output. If the fields are not compatible upstream of the filter, you can convert the types by using the `modify` operator prior to using the filter.

Field data type conversion is based on the following rules:

- Any integer, signed or unsigned, when compared to a floating-point type, is converted to floating-point.
- Comparisons within a general type convert the smaller to the larger size (`sfloat` to `dfloat`, `uint8` to `uint16`, etc.)
• When signed and unsigned integers are compared, unsigned are converted to signed.
• Decimal, raw, string, time, date, and timestamp do not figure in type conversions. When any of these is compared to another type, filter returns an error and terminates.

**Note**
The conversion of numeric data types may result in a loss of range and cause incorrect results. Orchestrate displays a warning message to that effect when range is lost.

The input field can contain nulls. If it does, null values are less than all non-null values, unless you specify the operators’s nulls last option.

### Supported Boolean Expressions and Operators
The following list summarizes the Boolean expressions that Orchestrate supports. In the list, BOOLEAN denotes any Boolean expression.

1. **true**
2. **false**
3. Six comparison operators: =, <>, <, >, <=, >=
4. **is null**
5. **is not null**
6. **like 'abc'** (the second operand must be a regular expression)
7. **between** (for example, A between B and C is equivalent to B <= A and A => C)
8. **not BOOLEAN**
9. BOOLEAN is true
10. BOOLEAN is false
11. BOOLEAN is not true
12. BOOLEAN is not false

Any of these can be combined using AND or OR.

### Order of Association
As in SQL, expressions are associated left to right. AND and OR have the same precedence. You may group fields and expressions in parentheses to affect the order of evaluation.
String Comparison

Orchestrate operators sort string values according to these general rules:

- Characters are sorted in lexicographic order
- Strings are evaluated by their ASCII value
- Sorting is case sensitive, that is, uppercase letters appear before lowercase letter in sorted data
- Null characters appear before non-null characters in a sorted data set, unless you specify the **nulls last** option
- Byte-for-byte comparison is performed

Example 1: Comparing Two Fields

You want to compare fields A and O. If the data in field A is greater than the data in field O, the corresponding records are to be written to the output data set.

Use the following **osh** command:

```
$ osh "... | filter -where 'A > O' ...
```

Example 2: Testing for a Null

You want to test field A to see if it contains a null. If it does, you want to write the corresponding records to the output data set.

Use the following **osh** command:

```
$ osh "... | filter -where 'A is null' ...
```

Example 3: Evaluating Input Records

You want to evaluate each input record to see if these conditions prevail:

- EITHER all the following are true
  - Field A does not have the value 0
  - Field a does not have the value 3
  - Field o has the value 0
- OR field q equals the string 'ZAG'

Here is the **osh** command for this example:

```
$ osh "... | filter -where 'A <> 0 and a <> 3 and o=0 or q = 'ZAG'' ...
```
Application Scenario: Mailing List for a Wine Auction

The following extended example illustrates the use of the filter operator to extract a list of prospects who should be sent a wine auction catalog, drawn from a large list of leads. A \texttt{where} clause selects individuals at or above legal drinking age (adult) with sufficient income to be likely to respond to such a catalog (rich).

The example illustrates the use of the where clause by not only producing the list of prospects, but by also producing a list of all individuals who are either adult or rich (or both) and a list of all individuals who are adult.

Schema for Implicit Import

The example assumes you have created the following schema and stored it as \texttt{filter_example.schema}:

\begin{verbatim}
record {
  first_name: string[max=16];
  last_name: string[max=20];
  gender: string[1];
  age: uint8;
  income: decimal[9,2];
  state: string[2];
}
\end{verbatim}

OSH Syntax

\begin{verbatim}
osh " filter
  -where 'age >= 21 and income > 50000.00'
  -where 'income > 50000.00'
  -where 'age >= 21' -target 1
  -where 'age >= 21'
  < [record@'filter_example.schema'] all12.txt
  0>| AdultAndRich.txt
  1>| AdultOrRich.txt
  2>| Adult.txt
"
\end{verbatim}

The first \texttt{-where} option directs all records that have age $\geq 21$ and income $> 50000.00$ to output 0, which is then directed to the file \texttt{AdultAndRich.txt}.

The second \texttt{-where} option directs all records that have income $> 50000.00$ to output 1, which is then directed to \texttt{AdultOrRich.txt}.

The third \texttt{-where} option directs all records that have age $\geq 21$ also to output 1 (because of the expression \texttt{-target 1}) which is then directed to \texttt{AdultOrRich.txt}.

The result of the second and third \texttt{-where} options is that records that satisfy either of the two conditions income $> 50000.00$ or age $\geq 21$ are sent to output 1. A
record that satisfies multiple -where options that are directed to the same output are only written to output once, so the effect of these two options is exactly the same as:

-where 'income > 50000.00' or age >= 21'

The fourth -where option causes all records satisfying the condition age >= 21 to be sent to the output 2, because the last -where option without a -target suboption directs records to output 1. This output is then sent to Adult.txt.

**Input Data**

As a test case, the following twelve data records exist in an input file all12.txt.

John Parker M 24  0087228.46 MA  
Susan Calvin F 24  0091312.42 IL  
William Mandella M 67  0040676.94 CA  
Ann Claybourne F 29  0061774.32 FL  
Frank Chalmers M 19  0004881.94 NY  
Jane Studdock F 24  0075990.80 TX  
Seymour Glass M 18  0051531.56 NJ  
Laura Engels F 57  0015280.31 KY  
John Boone M 16  0042729.03 CO  
Jennifer Sarandon F 58  0081319.09 ND  
William Tell M 73  0021008.45 SD  
Ann Dillard F 21  0004552.65 MI  
Jennifer Sarandon F 58  0081319.09 ND

**Outputs**

The following output comes from running Orchestrate. Because of parallelism, the order of the records may be different for your installation. If order matters, you can apply the psort or tsort operator to the output of the filter operator.

**After the Orchestrate job is run, the file AdultAndRich.txt contains:**

John Parker M 24  0087228.46 MA  
Susan Calvin F 24  0091312.42 IL  
Ann Claybourne F 29  0061774.32 FL  
Jane Studdock F 24  0075990.80 TX  
Jennifer Sarandon F 58  0081319.09 ND

**After the Orchestrate job is run, the file AdultOrRich.txt contains:**

John Parker M 24  0087228.46 MA  
Susan Calvin F 24  0091312.42 IL  
William Mandella M 67  0040676.94 CA  
Ann Claybourne F 29  0061774.32 FL
After the Orchestrate job is run, the file `Adult.txt` contains:

```
John Parker M 24  0087228.46 MA
Susan Calvin F 24  0091312.42 IL
William Mandella M 67  0040676.94 CA
Ann Claybourne F 29  0061774.32 FL
Jane Studdock F 24  0075990.80 TX
Laura Engels F 57  0015280.31 KY
Jennifer Sarandon F 58  0081319.09 ND
William Tell M 73  0021008.45 SD
Ann Dillard F 21  0004552.65 MI
```
The Funnel Operators

The funnel operators, funnel and sortfunnel, copy multiple input data sets to a single output data set.

Data Flow Diagram  8 2
Properties  8 2
funnel Operator  8 3
   Default Input Sequencing  8 3
   Non-Deterministic Input Sequencing  8 4
   Syntax and Option  8 4
sortfunnel Operator  8 5
   Input Requirements  8 5
   Primary and Secondary Keys  8 5
   Syntax and Option  8 6

The funnel operators copy multiple input data sets to a single output data set. This operation is useful for combining separate data sets into a single large data set.

Orchestrate provides two funnel operators:

- The **funnel** operator combines the records of the input data in no guaranteed order.
- The **sortfunnel** operator combines the input records in the order defined by the value(s) of one or more key fields and the order of the output records is determined by these sorting keys. By default, Orchestrate inserts partition
and sort components to meet the partitioning and sorting needs of the sortfunnel operator and other operators. See the section “Orchestrate-Inserted Partition and Sort Components” on page xxxvi of this Reference for information on this facility.

Data Flow Diagram

Properties

Table 16  \textit{funnel} and sortfunnel Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>N (set by user)</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec -&gt; outRec without record modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
<tr>
<td>Input partitioning style</td>
<td>sortfunnel operator: keys in same partition</td>
</tr>
<tr>
<td>Output partitioning style</td>
<td>sortfunnel operator: distributed keys</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>funnel operator: round robin (parallel mode)</td>
</tr>
<tr>
<td>Collection method</td>
<td>sortfunnel operator: hash</td>
</tr>
<tr>
<td></td>
<td>any (sequential mode)</td>
</tr>
</tbody>
</table>
funnel Operator

Default Input Sequencing

The funnel operator transfers records from the input data sets to the output data set in round-robin order:

1. It transfers one record from one input data set, then one record from the next input data set, until it has transferred one record from all input data sets to the output.

2. It begins again, transferring the next record from one input data set, then the next record from the next input data set, and so on.

   If the operator encounters an end-of-file in one input data set, it continues and transfers the record from the next data set to the output.

3. The operator stops after it has transferred all records from all input data sets to the output data set.

Figure 4 shows an example of a data flow using the funnel operator without the -asAvailable option.

---

Table 16  funnel and sortfunnel Operator Properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

---

Figure 4  Data Flow Example for funnel
Here are the contents of input data sets 0 and 1 and the output data set:

<table>
<thead>
<tr>
<th>inDS0.ds Name</th>
<th>Age</th>
<th>inDS1.ds Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>David</td>
<td>16</td>
<td>Ellen</td>
<td>53</td>
</tr>
<tr>
<td>Mary</td>
<td>32</td>
<td>Jane</td>
<td>17</td>
</tr>
</tbody>
</table>

The operator proceeds from `inDS0.ds` to `inDS1.ds`, one record at a time. After Jane 17, the operator encounters no record in `inDS0.ds` and moves on to `inDS1.ds`.

### Non-Deterministic Input Sequencing

Using the `-asAvailable` option, you can specify that a funnel operator processes its inputs using non-deterministic selection based on record availability.

With this option, a funnel operator still examines its input data sets in round-robin order. If the current record in a data set is ready for processing, the operator processes it. However, if the current record in a data set is not ready for processing, the operator does not halt execution. Instead, it moves on to the next data set and examines its current record for availability. This process continues until all the records have been transferred to output.

Currently, a funnel operator that is specified with the `-asAvailable` option cannot be combined.

### Syntax and Option

```
funnel [-asAvailable]
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-asAvailable</code></td>
<td><code>-asAvailable</code></td>
</tr>
</tbody>
</table>
The sortfunnel operator guarantees the order of the output records, because it combines the input records in the order defined by the value(s) of one or more key fields. The default partitioner and sort operator with the same keys are automatically inserted before the sortfunnel operator.

The sortfunnel operator requires that the record schema of all input data sets be identical.

A parallel sortfunnel operator uses the default partitioning method local keys. See Chapter 27, “The Partitioning Library” for more information on partitioning styles.

Primary and Secondary Keys

The sortfunnel operator allows you to set one primary key and multiple secondary keys. The sortfunnel operator first examines the primary key in each input record. For multiple records with the same primary key value, the sortfunnel operator then examines secondary keys to determine the order of records it outputs.

For example, the following figure shows the current record in each of three input data sets:

<table>
<thead>
<tr>
<th></th>
<th>data set 0</th>
<th>data set 1</th>
<th>data set 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>current</td>
<td>&quot;Jane&quot; &quot;Smith&quot; 42</td>
<td>&quot;Paul&quot; &quot;Smith&quot; 34</td>
<td>&quot;Mary&quot; &quot;Davis&quot; 42</td>
</tr>
<tr>
<td>record</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Ascential Software nonetheless does not guarantee the output order of records transferred by means of the funnel operator. Use the sortfunnel operator to guarantee transfer order.
If the data set shown above is sortfunneled on the primary key, LastName, and then on the secondary key, Age, here is the result:

```
primary key
↓
"Mary" "Davis" 42
"Paul" "Smith" 34
"Jane" "Smith" 42
```

secondary key

**Syntax and Option**

The `-key` option is required. Multiple key options are allowed.

```
sortfunnel
.key field [-cs | -ci] [-asc | -desc] [-nulls first | last] [-ebcdic]
[-param params]
.key field [-cs | -ci] [-asc | -desc] [-nulls first | last] [-ebcdic]
[-param params] ...
[.collation_sequence locale|collation_file_pathname] OFF
```
Table 18  sortfunnel Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-collation_sequence</td>
<td>-collation_sequence locale</td>
</tr>
<tr>
<td></td>
<td>This option determines how your string data is sorted. You can:</td>
</tr>
<tr>
<td></td>
<td>• Specify a predefined IBM ICU locale</td>
</tr>
<tr>
<td></td>
<td>• Write your own collation sequence using ICU syntax, and supply its</td>
</tr>
<tr>
<td></td>
<td>collation_file_pathname</td>
</tr>
<tr>
<td></td>
<td>• Specify OFF so that string comparisons are made using Unicode code-</td>
</tr>
<tr>
<td></td>
<td>point value order, independent of any locale or custom sequence.</td>
</tr>
<tr>
<td></td>
<td>By default, Orchestrate sorts strings using byte-wise comparisons.</td>
</tr>
<tr>
<td></td>
<td>For more information, see Chapter 7: National Language Support in the</td>
</tr>
<tr>
<td></td>
<td>Orchestrate 7.0 User Guide; and reference this IBM ICU site:</td>
</tr>
<tr>
<td></td>
<td><a href="http://oss.software.ibm.com/icu/userguide/Collate_Intro.htm">http://oss.software.ibm.com/icu/userguide/Collate_Intro.htm</a></td>
</tr>
<tr>
<td>-key</td>
<td>-key field [ -cs</td>
</tr>
<tr>
<td></td>
<td>Specifies a key field of the sorting operation. The first -key</td>
</tr>
<tr>
<td></td>
<td>defines the primary key field of the sort; lower-priority key fields</td>
</tr>
<tr>
<td></td>
<td>are supplied on subsequent -key specifications.</td>
</tr>
<tr>
<td></td>
<td>You must define a single primary key to the sortfunnel operator. Yo-</td>
</tr>
<tr>
<td></td>
<td>u can define as many secondary keys as are required by your applica-</td>
</tr>
<tr>
<td></td>
<td>tion. For each key, select the option and supply the field name. Ea-</td>
</tr>
<tr>
<td></td>
<td>ch record field can be used only once as a key. Therefore, the to-</td>
</tr>
<tr>
<td></td>
<td>tal number of primary and secondary keys must be less than or equal</td>
</tr>
<tr>
<td></td>
<td>to the total number of fields in the record.</td>
</tr>
<tr>
<td></td>
<td>-cs</td>
</tr>
<tr>
<td></td>
<td>insensitive sorting. By default, the operator uses a case-sensitive</td>
</tr>
<tr>
<td></td>
<td>algorithm for sorting, that is, uppercase strings appear before lo-</td>
</tr>
<tr>
<td></td>
<td>wercase strings in the sorted data set. Specify -ci  to override th-</td>
</tr>
<tr>
<td></td>
<td>is default and perform case-insensitive sorting of string fields.</td>
</tr>
<tr>
<td></td>
<td>-asc</td>
</tr>
<tr>
<td></td>
<td>desing sorting. By default, the operator uses ascending sorting or-</td>
</tr>
<tr>
<td></td>
<td>der, that is, smaller values appear before larger values in the sor-</td>
</tr>
<tr>
<td></td>
<td>ted data set. Specify desc  to sort in descending sorting order in-</td>
</tr>
<tr>
<td></td>
<td>stead, so that larger values appear before smaller values in the so-</td>
</tr>
<tr>
<td></td>
<td>rted data set.</td>
</tr>
<tr>
<td></td>
<td>-nulls first</td>
</tr>
<tr>
<td></td>
<td>sorted data set. To override this default so that fields containing</td>
</tr>
<tr>
<td></td>
<td>null values appear last in the sorted data set, specify nulls last.</td>
</tr>
</tbody>
</table>
|                         | continued on next page
In this `osh` example, the `sortfunnel` operator combines two input data sets into one sorted output data set:

```bash
$ osh "sortfunnel -key Lastname -key Age < out0.v < out1.v > combined.ds"
```
The generator Operator

The Orchestrate generator operator lets you create a data set with a specified schema. It is used for creating synthetic data for prototyping purposes.

Data Flow Diagram 9 2
Properties 9 2
Syntax and Options 9 3
Using the generator Operator 9 4
  Supported Data Types 9 4
  Example 1: Using the generator Operator 9 5
  Example 2: Executing the Operator in Parallel 9 6
  Example 3: Using generator with an Input Data Set 9 6
Defining the Schema for the Operator 9 7
  Schema Syntax for Generator Options 9 7
Numeric Fields 9 8
  Date Fields 9 10
  Decimal Fields 9 11
  Raw Fields 9 11
  String Fields 9 12
  Time Fields 9 13
  Timestamp Fields 9 14
  Null Fields 9 14

Often during the development of an Orchestrate application, you will want to test the application using valid data. However, you may not have any data available
to run the test, your data may be too large to execute the test in a timely manner, or you may not have data with the characteristics required to test the application.

The Orchestrate generator operator lets you create a data set with the record layout that you pass to the operator. In addition, you can control the number of records in the data set, as well as the value of all record fields. You can then use this generated data set while developing, debugging, and testing your Orchestrate application.

To generate a data set, you pass to the operator a schema defining the field layout of the data set and any information used to control generated field values. This chapter describes how to use the generator operator, including information on the schema options you use to control the generated field values.

### Data Flow Diagram

![Data Flow Diagram](image)

### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>0 or 1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>supplied_schema; outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec -&gt; outRec without record modification</td>
</tr>
</tbody>
</table>
Syntax and Options

generator -schema schema | -schemafile filename [-records num_recs]

You must use either the -schema or the -schemafile argument to specify a schema to the operator. Terms in italic typeface are option strings you supply. When your option string contains a space or a tab character, you must enclose it in single quotes.

Table 19  generator Operator Properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution mode</td>
<td>sequential (default) or parallel</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>any (parallel mode)</td>
</tr>
<tr>
<td>Collection method</td>
<td>any (sequential mode)</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
</tbody>
</table>

Table 20  generator Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-schema schema</td>
<td>Specifies the schema for the generated data set.</td>
</tr>
<tr>
<td></td>
<td>You must specify either -schema or -schemafile to the operator.</td>
</tr>
<tr>
<td></td>
<td>If you supply an input data set to the operator, new fields with the specified schema are prepended to the beginning of each record.</td>
</tr>
<tr>
<td>-schemafile</td>
<td>Specifies the name of a file containing the schema for the generated data set.</td>
</tr>
<tr>
<td></td>
<td>You must specify either -schema or -schemafile to the operator.</td>
</tr>
<tr>
<td></td>
<td>If you supply an input data set to the operator, new fields with the supplied schema are prepended to the beginning of each record.</td>
</tr>
<tr>
<td>-records num_recs</td>
<td>Specifies the number of records to generate. By default the operator generates an output data set with 10 records (in sequential mode) or 10 records per partition (in parallel mode).</td>
</tr>
<tr>
<td></td>
<td>If you supply an input data set to the operator, any specification for -records is ignored. In this case, the operator generates one record for each record in the input data set.</td>
</tr>
</tbody>
</table>
Using the generator Operator

During the development of an Orchestrate application, you may find it convenient to execute your application against a data set with a well-defined content. This may be necessary because you want to:

- Run the program against a small number of records to test its functionality
- Control the field values of the data set to examine application output
- Test the program against a variety of data sets
- Run the program but have no available data

You pass to the generator operator a schema that defines the field layout of the data set. By default, the generator operator initializes record fields using a well-defined generation algorithm. For example, an 8-bit unsigned integer field in the first record of a generated data set is set to 0. The field value in each subsequently generated record is incremented by 1 until the generated field value is 255. The field value then wraps back to 0.

However, you can also include information in the schema passed to the operator to control the field values of the generated data set. See “Schema Syntax for Generator Options” on page 9-7 for more information on these options.

By default, the operator executes sequentially to generate a data set with a single partition containing 10 records. However, you can configure the operator to generate any number of records. If you configure the operator to execute in parallel, you control the number of records generated in each partition of the output data set.

You can also pass an input data set to the operator. In this case, the operator prepends the generated record fields to the beginning of each record of the input data set to create the output.

Supported Data Types

The generator operator supports the creation of data sets containing most Orchestrate data types, including fixed-length vectors and nullable fields. However, the generator operator does not support the following data types:

- Variable-length string and ustring types (unless you include a maximum-length specification)
- Variable-length raws (unless you include a maximum-length specification)
- Subrecords
- Tagged aggregates
- Variable-length vectors


**Example 1: Using the generator Operator**

In this example, you use the `generator` operator to create a data set with 1000 records where each record contains five fields. You also allow the operator to generate default field values.

Here is the schema for the generated data set for this example:

```plaintext
record (
    a: int32;
    b: int16;
    c: sfloat;
    d: string[10];
    e: dfloat;
)
```

Figure 5 shows the data flow diagram for this example:

![Data Flow for generator Operator Example 1]

This example consists of a single operator writing to an output data set.

To use the `generator` operator, first configure the schema:

```plaintext
$ rec_schema="record
    a: int32;
    b: int16;
    c: sfloat;
    d: string[10];
    e: dfloat;
)"
```

Then issue the generator command:

```plaintext
$ osh "generator -schema $rec_schema -records 1000 > newDS.ds"
```

This example defines an environment variable (`$rec_schema`) to hold the schema passed to the operator.

Alternatively you can specify the name of a file containing the schema, as shown below:

```plaintext
$ osh "generator -schemafile s_file.txt -records 1000 > newDS.ds"
```

where the text file `s_file.txt` contains the schema.
Example 2: Executing the Operator in Parallel

In the previous example, the operator executed sequentially to create an output data set with 1000 records in a single partition. You can also execute the operator in parallel. When executed in parallel, each partition of the generated data set contains the same number of records as determined by the setting for the -records option.

For example, the following osh command executes the operator in parallel to create an output data set with 500 records per partition:

```
$ osh 'generator -schemafile s_file -records 500 [par] > newDS.ds'
```

Note that the keyword [par] has been added to the example to configure the generator operator to execute in parallel.

The number of partitions of the output data set from a parallel generator operator depends on the Orchestrate configuration file. If the configuration file defines four processing nodes, the output data set contains four partitions. In addition, you can specify constraints to the operator to control the processing nodes used to execute the operator and the disk drives used to store the data set. See the chapter on using constraints in the Orchestrate 7.0 User Guide for more information on this subject.

Example 3: Using generator with an Input Data Set

You can pass an input data set to the generator operator. In this case, the generated fields are prepended to the beginning of each input record. The operator generates an output data set with the same number of records as the input data set; you cannot specify a record count.

The following command creates an output data set from an input data set and a schema file:

```
$ osh 'generator -schemafile s_file [par] < oldDS.ds > newDS.ds'
```

The figure below shows the output record of the generator operator:

For example, you can enumerate the records of a data set by appending an int32 field that cycles from 0 upward.
The generated fields are prepended to the beginning of each record. This means conflicts caused by duplicate field names in the generator schema and the input data set result in the field from the input data set being dropped. Note that Orchestrate issues a warning message to inform you of the naming conflict.

You can use the modify operator to rename the fields in the input data set to avoid the name collision. See Chapter 21, “The transform Operator” for more information.

Defining the Schema for the Operator

The schema passed to the generator operator defines the record layout of the output data set. For example, the previous section showed examples using the following schema:

```
record (  
a:int32;  
b:int16;  
c:sfloat;  
d:string[10];  
e:dfloat; )
```

In the absence of any other specifications in the schema, the operator assigns default values to the fields of the output data set. However, you can also include information in the schema to control the values of the generated fields. This section describes the default values generated for all Orchestrate data types and the use of options in the schema to control field values.

Schema Syntax for Generator Options

You specify generator options within the schema in the same way you specify import/export properties. The following example shows the basic syntax of the generator properties:

```
record (  
a:int32 {generator_options};  
b:int16 {generator_options};  
c:sfloat {generator_options};  
d:string[10] {generator_options};  
e:dfloat {generator_options}; )
```

Note that you include the generator options as part of the schema definition for a field. The options must be included within braces and before the trailing semicolon. Use commas to separate options for fields that accept multiple options.

Table 21 lists all options for the different Orchestrate data types. Detailed information on these options follows the table.
Table 21  generator Options for Orchestrate Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Generator Options for the Schema</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>numeric (also decimal, date, time, timestamp)</td>
<td>cycle = {init = init_val, incr = incr_val, limit = limit_val} random = {limit = limit_val, seed = seed_val, signed}</td>
<td>8</td>
</tr>
<tr>
<td>date</td>
<td>epoch = 'date'</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>invalids = percentage</td>
<td></td>
</tr>
<tr>
<td></td>
<td>function = rundate</td>
<td></td>
</tr>
<tr>
<td>decimal</td>
<td>zeros = percentage</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>invalids = percentage</td>
<td></td>
</tr>
<tr>
<td>raw</td>
<td>no options available</td>
<td>11</td>
</tr>
<tr>
<td>string</td>
<td>cycle = {value = 'string_1', value = 'string_2', ...} alphabet = 'alpha_numeric_string'</td>
<td>12</td>
</tr>
<tr>
<td>ustring</td>
<td>cycle = {value = 'ustring_1', value = 'ustring_2', ...} alphabet = 'alpha_numeric_ustring'</td>
<td>12</td>
</tr>
<tr>
<td>time</td>
<td>scale = factor</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>invalids = percentage</td>
<td></td>
</tr>
<tr>
<td>timestamp</td>
<td>epoch = 'date'</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>scale = factor</td>
<td></td>
</tr>
<tr>
<td></td>
<td>invalids = percentage</td>
<td></td>
</tr>
<tr>
<td>nullable fields</td>
<td>nulls = percentage</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>nullseed = number</td>
<td></td>
</tr>
</tbody>
</table>

**Numeric Fields**

By default, the value of an integer or floating point field in the first record created by the operator is 0 (integer) or 0.0 (float). The field in each successive record generated by the operator is incremented by 1 (integer) or 1.0 (float).

The `generator` operator supports the use of the `cycle` and `random` options that you can use with integer and floating point fields (as well as with all other fields except raw and string). The `cycle` option generates a repeating pattern of values.
for a field. The random option generates random values for a field. These options are mutually exclusive; that is, you can only use one option with a field.

- **cycle** generates a repeating pattern of values for a field. Shown below is the syntax for this option:

```
cycle = {init = init_val, incr = incr_val, limit = limit_val}
```

where:

- `init_val` is the initial field value (value of the first output record). The default value is 0.
- `incr_val` is the increment value added to produce the field value in the next output record. The default value is 1 (integer) or 1.0 (float).
- `limit_val` is the maximum field value. When the generated field value is greater than `limit_val`, it wraps back to `init_val`. The default value of `limit_val` is the maximum allowable value for the field’s data type.

You can specify the keyword `part` or `partcount` for any of these three option values. Specifying `part` uses the partition number of the operator on each processing node for the option value. The partition number is 0 on the first processing node, 1 on the next, etc. Specifying `partcount` uses the number of partitions executing the operator for the option value. For example, if the operator executes on four processing nodes, `partcount` corresponds to a value of 4.

- **random** generates random values for a field. Shown below is the syntax for this option (all arguments to random are optional):

```
random = {limit = limit_val, seed = seed_val, signed}
```

where:

- `limit_val` is the maximum generated field value. The default value of `limit_val` is the maximum allowable value for the field’s data type.
- `seed_val` is the seed value for the random number generator used by the operator for the field. You do not have to specify `seed_val`. By default, the operator uses the same seed value for all fields containing the random option.
- `signed` specifies that signed values are generated for the field (values between -`limit_val` and +`limit_val`). Otherwise, the operator creates values between 0 and +`limit_val`.

You can also specify the keyword `part` for `seed_val` and `partcount` for `limit_val`.

For example, the following schema generates a repeating cycle of values for the AccountType field and a random number for Balance:

```
record {
    AccountType:int8 {cycle={init=0, incr=1, limit=24}};
```
Balance: dfloat {random={limit=100000, seed=34455}};

### Date Fields

By default, a date field in the first record created by the operator is set to January 1, 1960. The field in each successive record generated by the operator is incremented by one day.

You can use the cycle and random options for date fields as shown above. When using these options, you specify the option values as a number of days. For example, to set the increment value for a date field to seven days, you use the following syntax:

```plaintext
record {
  transDate: date {cycle={incr=7}};
  transAmount: dfloat {random={limit=100000, seed=34455}};
}
```

In addition, you can use the following options: epoch, invalids, and functions.

The epoch option sets the earliest generated date value for a field. You can use this option with any other date options. The syntax of epoch is:

```plaintext
epoch = 'date'
```

where date sets the earliest generated date for the field. The date must be in yyyy-mm-dd format and leading zeros must be supplied for all portions of the date. If an epoch is not specified, the operator uses 1960-01-01.

For example, the following schema sets the initial field value of transDate to January 1, 1998:

```plaintext
record {
  transDate: date {epoch='1998-01-01'};
  transAmount: dfloat {random={limit=100000, seed=34455}};
}
```

You can also specify the invalids option for a date field. This option specifies the percentage of generated fields containing invalid dates:

```plaintext
invalids = percentage
```

where percentage is a value between 0.0 and 100.0. Orchestrate operators that process date fields can detect an invalid date during processing.

The following example causes approximately 10% of transDate fields to be invalid:

```plaintext
record {
  transDate: date {epoch='1998-01-01', invalids=10.0};
  transAmount: dfloat {random={limit=100000, seed=34455}};
}
```

You can use the function option to set date fields to the current date:

```plaintext
function = rundate
```
There must be no other options specified to a field using `function`. The following schema causes `transDate` to have the current date in all generated records:

```plaintext
record {
        transDate: date {function=rundate};
        transAmount: dfloat {random={limit=100000, seed=34455}};
    }
```

**Decimal Fields**

By default, a decimal field in the first record created by the operator is set to 0. The field in each successive record generated by the operator is incremented by 1. The maximum value of the decimal is determined by the decimal’s scale and precision. When the maximum value is reached, the decimal field wraps back to 0.

You can use the `cycle` and `random` options with decimal fields. See “Numeric Fields” on page 9-8 for information on these options. In addition, you can use the `zeros` and `invalids` options with decimal fields. These options are described below.

The `zeros` option specifies the percentage of generated decimal fields where all bytes of the decimal are set to binary zero (0x00). Many operations performed on a decimal can detect this condition and either fail or return a flag signifying an invalid decimal value. The syntax for the `zeros` option is:

```plaintext
zeros = percentage
```

where `percentage` is a value between 0.0 and 100.0.

The `invalids` options specify the percentage of generated decimal fields containing and invalid representation of 0xFF in all bytes of the field. Any operation performed on an invalid decimal detects this condition and either fails or returns a flag signifying an invalid decimal value. The syntax for `invalids` is:

```plaintext
invalids = percentage
```

where `percentage` is a value between 0.0 and 100.0.

If you specify both `zeros` and `invalids`, the percentage for `invalids` is applied to the fields that are not first made zero. For example, if you specify `zeros=50` and `invalids=50`, the operator generates approximately 50% of all values to be all zeros and only 25% (50% of the remainder) to be invalid.

**Raw Fields**

You can use the `generator` operator to create fixed-length raw fields or raw fields with a specified maximum length; you cannot use the operator to generate variable-length raw fields. If the field has a maximum specified length, the length of the string is a random number between 1 and the maximum length.

Maximum-length raw fields are variable-length fields with a maximum length defined by the `max` parameter in the form:
max_r: raw [max=10];

By default, all bytes of a raw field in the first record created by the operator are set to 0x00. The bytes of each successive record generated by the operator are incremented by 1 until a maximum value of 0xFF is reached. The operator then wraps byte values to 0x00 and repeats the cycle.

You cannot specify any options to raw fields.

**String Fields**

You can use the generator operator to create fixed-length string and ustring fields or string and ustring fields with a specified maximum length; you cannot use the operator to generate variable-length string fields. If the field has a maximum specified length, the length of the string is a random number between 0 and the maximum length.

Note that maximum-length string fields are variable-length fields with a maximum length defined by the max parameter in the form:

max_s: string [max=10];

In this example, the field max_s is variable length up to 10 bytes long.

By default, the generator operator initializes all bytes of a string field to the same alphanumeric character. When generating a string field, the operators uses the following characters, in the following order:

abcdefghijklmnopqrstuvwxyz0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ

For example, the following field specification:

s: string[5];

produces successive string fields with the values:

aaaaa
bbbba
cccccc
ddddd
...

After the last character, capital Z, values wrap back to lowercase a and the cycle repeats.

### Note

The alphabet property for ustring values accepts Unicode characters.

You can use the alphabet property to define your own list of alphanumeric characters used for generated string fields:

alphabet = 'alpha_numeric_string'

This option sets all characters in the field to successive characters in the alphabet_numeric_string. For example, this field specification:

s: string[3] {alphabet='abc'};
produces strings with the following values:

```
aaa
bbb
ccc
aaa
...
```

**Note**
The `cycle` option for `ustring` values accepts Unicode characters.

The `cycle` option specifies the list of string values assigned to generated string field:

```plaintext
cycle = { value = 'string_1', value = 'string_2', ... }
```

The operator assigns `string_1` to the string field in the first generated record, `string_2` to the field in the second generated record, etc. In addition:

- If you specify only a single `value`, all string fields are set to that value.
- If the generated string field is fixed length, the `value` string is truncated or padded with the default pad character `0x00` to the fixed length of the string.
- If the string field contains a maximum length setting, the length of the string field is set to the length of the `value` string. If the length of the `value` string is longer than the maximum string length, the `value` string is truncated to the maximum length.

**Time Fields**

By default, a time field in the first record created by the operator is set to 00:00:00 (midnight). The field in each successive record generated by the operator is incremented by one second. After reaching a time of 23:59:59, time fields wrap back to 00:00:00.

You can use the `cycle` and `random` options with time fields. See “**Numeric Fields**” on page 9-8 for information on these options. When using these options, you specify the options values in numbers of seconds. For example, to set the value for a time field to a random value between midnight and noon, you use the following syntax:

```plaintext
record (
    transTime:time {random={limit=43200, seed=83344}};
)
```

For a time field, midnight corresponds to an initial value of 0 and noon corresponds to 43,200 seconds (12 hours * 60 minutes * 60 seconds).

In addition, you can use the `scale` and `invalids` options with time fields.

The `scale` option allows you to specify a multiplier to the increment value for time. The syntax of this options is:

```plaintext
scale = factor
```
The increment value is multiplied by factor before being added to the field. For example, the following schema generates two time fields:

```plaintext
record {
  timeMinutes: time {scale=60};
  timeSeconds: time;
}
```

In this example, the first field increments by 60 seconds per record (one minute), and the second field increments by seconds.

You use the `invalids` option to specify the percentage of invalid time fields generated:

```plaintext
invalids = percentage
```

where percentage is a value between 0.0 and 100.0. The following schema generates two time fields with different percentages of invalid values:

```plaintext
record {
  timeMinutes: time {scale=60, invalids=10};
  timeSeconds: time {invalids=15};
}
```

**Timestamp Fields**

A timestamp field consists of both a time and date portion. Timestamp fields support all valid options for both date and time fields. See “Date Fields” on page 9-10 or “Time Fields” on page 9-13 for more information.

By default, a timestamp field in the first record created by the operator is set to 00:00:00 (midnight) on January 1, 1960. The time portion of the timestamp is incremented by one second for each successive record. After reaching a time of 23:59:59, the time portion wraps back to 00:00:00 and the date portion increments by one day.

**Null Fields**

By default, schema fields are not nullable. Specifying a field as nullable allows you to use the `nulls` and `nullseed` options within the schema passed to the `generator` operator.

<table>
<thead>
<tr>
<th>Important</th>
<th>If you specify these options for a non-nullable field, the operator issues a warning and the field is set to its default value.</th>
</tr>
</thead>
</table>

The `nulls` option specifies the percentage of generated fields that are set to null:

```plaintext
nulls = percentage
```

where percentage is a value between 0.0 and 100.0.

The following example specifies that approximately 15% of all generated records contain a null for field `a`:
record (  
  a:nullable int32 {random={limit=100000, seed=34455}, nulls=15.0};  
  b:int16;  
)

The **nullseed** options sets the seed for the random number generator used to decide whether a given field will be null.

```
nullseed = seed
```

where **seed** specifies the seed value and must be an integer larger than 0.

In some cases, you may have multiple fields in a schema that support nulls. You can set all nullable fields in a record to null by giving them the same **nulls** and **nullseed** values. For example, the following schema defines two fields as nullable:

```
record (  
  a:nullable int32 {nulls=10.0, nullseed=5663};  
  b:int16;  
  c:nullable sfloat {nulls=10.0, nullseed=5663};  
  d:string[10];  
  e:dfloat;  
)
```

Since both fields `a` and `c` have the same settings for **nulls** and **nullseed**, whenever one field in a record is null the other is null as well.
The head Operator

The head operator copies the first \( N \) records from each partition of an input data set to an output data set. The behavior is analogous to the UNIX command of the same name.

Data Flow Diagram 10 2
Properties 10 2
Syntax and Options 10 2
Examples 10 3
  Example 1: head Operator Default Behavior 10 3
  Example 2: Extracting Records from a Large Data Set 10 4
  Example 3: Locating a Single Record 10 4

The head operator selects the first \( N \) records from each partition of an input data set and copies the selected records to an output data set. By default, \( N \) is 10 records. However, you can determine the following by means of options:

- The number of records to copy
- The partition from which the records are copied
- The location of the records to copy
- The number of records to skip before the copying operation begins.

This control is helpful in testing and debugging applications with large data sets. For example, the -part option lets you see data from a single partition to ascertain
if the data is being partitioned as you want. The -skip option lets you access a portion of a data set.

The tail operator performs a similar operation, copying the last \( N \) records from each partition. See Chapter 20, “The tail Operator”.

**Data Flow Diagram**

```plaintext
input data set

head

| inRec:*;

| outRec:*;

output data set
```

**Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec -&gt; outRec without record modification</td>
</tr>
</tbody>
</table>

**Syntax and Options**

```plaintext
head
[-all | -nrecs count]
[-part partition_number]
[-period P]
[-skip recs]
```
This section contains three examples illustrating the `head` operator:

- “Example 1: head Operator Default Behavior” on page 10-3
- “Example 2: Extracting Records from a Large Data Set” on page 10-4
- “Example 3: Locating a Single Record” on page 10-4

### Example 1: head Operator Default Behavior

In this example, no options have been specified to the `head` operator. The input data set consists of 100 sorted positive integers hashed into four partitions. The output data set consists of the first ten integers of each partition. The table below lists the input and output data sets by partition.
The osh command is:

$osh "head < in.ds > out.ds"

<table>
<thead>
<tr>
<th>Partition 0</th>
<th>Partition 1</th>
<th>Partition 2</th>
<th>Partition 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Output</td>
<td>Input</td>
<td>Output</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>36</td>
<td>36</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>37</td>
<td>37</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>47</td>
<td>47</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>51</td>
<td>42</td>
<td>43</td>
<td>43</td>
</tr>
<tr>
<td>46</td>
<td>48</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>49</td>
<td>55</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>50</td>
<td>56</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>53</td>
<td>58</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>57</td>
<td>52</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>54</td>
<td>54</td>
<td></td>
</tr>
</tbody>
</table>

Example 2: Extracting Records from a Large Data Set

In this example you use the head operator to extract the first 1000 records of each partition of a large data set, in.ds.

To perform this operation, use the osh command:

$ osh "head -nrecs 1000 < in.ds > out0.ds"

For example, if in.ds is a data set of one megabyte, with 2500 K records, out0.ds is a dataset of 15.6 kilobytes with 4K records.

Example 3: Locating a Single Record

In this example you use head to extract a single record from a particular partition to diagnose the record.

The osh command is:

$ osh "head -nrecs 1 -skip 1234 -part 2 < in.ds > out0.ds"

Since nrecs = 1, there is just one output record.
The lookup Operator

The lookup operator lets you create and use lookup tables to modify the input data set. This is useful, for example, for mapping long strings to or from short codes and for checking the correctness of data.

Data Flow Diagrams 11 3
Create-Only Mode 11 3
Lookup Mode 11 3
Properties 11 4
Syntax and Options 11 5
Lookup Table Characteristics 11 7
Partitioning 11 8
Create-Only Mode 11 8
Examples 11 9
  Single Lookup Table Record 11 9
  Multiple Lookup Table Records 11 10
  Interest Rate Lookup Example 11 11
  Handling Duplicate Fields Example 11 12

With the lookup operator, you can create and use lookup tables to modify your input data set. For example, you could map a field that should contain a two letter U. S. state postal code to the name of the state, adding a FullStateName field to the output schema.
The operator performs in two modes: lookup mode and create-only mode:

- In lookup mode, the operator takes as input a single source data set, one or more lookup tables represented as Orchestrate data sets, and one or more file sets. A file set is a lookup table that contains key-field information. There must be at least one lookup table or file set.

  For each input record of the source data set, the operator performs a table lookup on each of the input lookup tables. The table lookup is based on the values of a set of lookup key fields, one set for each table. A source record and lookup record correspond when all of the specified lookup key fields have matching values.

  Each record of the output data set contains all of the fields from a source record. Concatenated to the end of the output records are the fields from all the corresponding lookup records where corresponding source and lookup records have the same value for the lookup key fields.

  The reject data set is an optional second output of the operator. This data set contains source records that do not have a corresponding entry in every input lookup table.

- In create-only mode, you use the `-createOnly` option to create lookup tables without doing the lookup processing step. This allows you to make and save lookup tables that you expect to need at a later time, making for faster start-up of subsequent lookup operations.

The `lookup` operator is similar in function to the `merge` operator and the `join` operators. To understand the similarities and differences see “Comparison with Other Operators” on page 28-5.
Data Flow Diagrams

Create-Only Mode

```
| table0.ds | table1.ds | tableN.ds |
```

```
file-set1  file-set2  ...  file-setN
```

Look-up Mode

```
| source.ds | fileset1  ... file-setN | table0.ds  ... tableN.ds |
```

```
key0; ... keyN; inRec:*;  key0; ... keyN; tableRec1:*;
```

```
outRec:*; tableRec1:*; ... tableRecN:*;  rejectRec:*;
```

```
output.ds     reject ds  file-set0  ...  file-setN
```

(when the save suboption is used)
## Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Normal mode</th>
<th>Create-only mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>T + 1</td>
<td>T</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1 or 2 (output and optional reject)</td>
<td>0</td>
</tr>
</tbody>
</table>

### Input interface schema
- input data set: `key0:datatype; ... keyN:datatype; inRec:*;`  
- lookup data sets: `key0:datatype; ... keyM:datatype`  
- tableRec:*;

### Output interface schema
- output data set: `outRec:*;` with lookup fields missing from the input data set concatenated  
- reject data sets: `rejectRec:*`

### Transfer behavior
- source to output: `inRec -> outRec` without record modification  
- lookup to output: `tableRecN -> tableRecN`, minus lookup keys and other duplicate fields  
- source to reject: `inRec -> rejectRec` without record modification (optional)  
- table to file set: `key-field information is added` to the table  

### Partitioning method
- any (parallel mode); the default for table inputs is entire  
- any (default is entire)

### Collection method
- any (sequential mode)  
- any

### Preserve-partitioning flag in output data set
- propagated  
- n/a

### Composite operator
- yes  
- yes
Terms in italic typeface are option strings you supply. When your option string contains a space or a tab character, you must enclose it in single quotes.

The syntax for the `lookup` operator in an `osh` command has two forms, depending on whether you are only creating one or more lookup tables or doing the lookup (matching) itself and creating file sets or using existing file sets.

```
lookup -createOnly
   -table -key field [-cs | -ci] [-key field [-cs | -ci] ...]
   [-allow_dups] -save lookup_fileset [-diskpool pool]
[-table -key field [-cs | -ci] [-key field [-cs | -ci] ...]
   [-allow_dups] -save fileset_descriptor [-diskpool pool] ...
```

or

```
lookup [-fileset fileset_descriptor] [-collation_sequence locale | collation_file_pathname | OFF]
[-table key_specifications [-allow_dups] -save fileset_descriptor]
   [-diskpool pool] ...
[-ifNotFound continue | drop | fail | reject]
```

where a fileset, or a table, or both, must be specified, and key_specifications is a list of one or more strings of this form:

```
-key field [-cs | -ci]
```

### Table 25 lookup Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-collation_sequence</code></td>
<td>`-collation_sequence locale</td>
</tr>
<tr>
<td></td>
<td>This option determines how your string data is sorted. You can:</td>
</tr>
<tr>
<td></td>
<td>• Specify a predefined IBM ICU locale</td>
</tr>
<tr>
<td></td>
<td>• Write your own collation sequence using ICU syntax, and supply its</td>
</tr>
<tr>
<td></td>
<td>collation_file_pathname</td>
</tr>
<tr>
<td></td>
<td>• Specify OFF so that string comparisons are made using Unicode code-</td>
</tr>
<tr>
<td></td>
<td>point value order, independent of any locale or custom sequence.</td>
</tr>
<tr>
<td></td>
<td>By default, Orchestrate sorts strings using byte-wise comparisons.</td>
</tr>
<tr>
<td></td>
<td>For more information, see Chapter 7: National Language Support in the</td>
</tr>
<tr>
<td></td>
<td>Orchestrate 7.0 User Guide, and reference this IBM ICU site:</td>
</tr>
<tr>
<td><code>-createOnly</code></td>
<td><code>-createOnly</code></td>
</tr>
<tr>
<td></td>
<td>Specifies the creation of one or more lookup tables; no lookup processing</td>
</tr>
<tr>
<td></td>
<td>is to be done.</td>
</tr>
</tbody>
</table>
Specify the name of a fileset containing one or more lookup tables to be matched. These are tables that have been created and saved by an earlier execution of the `lookup` operator using the `-createOnly` option.

In lookup mode, you must specify either the `-fileset` option, or a table specification, or both, in order to designate the lookup table(s) to be matched against. There can be zero or more occurrences of the `-fileset` option. It cannot be specified in create-only mode.

**Warning:** The fileset already contains key specifications. When you follow `-fileset fileset_descriptor ...` by `key_specifications`, the keys specified do not apply to the fileset; rather, they apply to the first lookup table. For example, `lookup -fileset file -key field`, is the same as:

```
lookup -fileset file1 -table -key field
```

**Option** | **Use**
---|---
`-fileset` | `[·fileset fileset_descriptor ...]`

Specifies the name of a fileset containing one or more lookup tables to be matched. These are tables that have been created and saved by an earlier execution of the `lookup` operator using the `-createOnly` option.

Specifying the operator action when a record of an input data set does not have a corresponding record in every input lookup table. The default action of the operator is to fail and terminate the step.

**continue** tells the operator to continue execution when a record of an input data set does not have a corresponding record in every input lookup table. The input record is transferred to the output data set along with the corresponding records from the lookup tables that matched. The fields in the output record corresponding to the lookup table(s) with no corresponding record are set to their default value or null if the field supports nulls.

**drop** tells the operator to drop the input record (refrain from creating an output record).

**fail** sets the operator to abort. This is the default.

**reject** tells the operator to copy the input record to the reject data set. In this case, a reject output data set must be specified.
Chapter 11  The lookup Operator

Lookup Table Characteristics

The lookup tables input to the operator are created from Orchestrate data sets. The lookup tables do not have to be sorted and should be small enough that all tables fit into physical memory on the processing nodes in your system. Lookup tables larger than physical memory do not cause an error, but they adversely affect the execution speed of the operator.

The memory used to hold a lookup table is shared among the lookup processes running on each machine. Thus, on an SMP, all instances of a lookup operator...

Table 25  lookup Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
</table>
| -table    | -table-key field [-ci | -cs] [-key field [-ci | cs] ...]
|           | [-allow_dups] -save fileset_descriptor [-diskpool pool] ...          |
|           | Specifies the beginning of a list of key fields and other specifications for a lookup table. The first occurrence of -table marks the beginning of the key field list for lookup table1; the next occurrence of -table marks the beginning of the key fields for lookup table2, etc. For example: lookup-table-key field-table-key field |
|           | The -key option specifies the name of a lookup key field. The -key option must be repeated if there are multiple key fields. You must specify at least one key for each table. You cannot use a vector, subrecord, or tagged aggregate field as a lookup key. |
|           | The -ci suboption specifies that the string comparison of lookup key values is to be case insensitive; the -cs option specifies case-sensitive comparison, which is the default. |
|           | In create-only mode, the -allow_dups option causes the operator to save multiple copies of duplicate records in the lookup table without issuing a warning. Two lookup records are duplicates when all lookup key fields have the same value in the two records. If you do not specify this option, Orchestrate issues a warning message when it encounters duplicate records and discards all but the first of the matching records. |
|           | In normal lookup mode, only one lookup table (specified by either -table or -fileset) can have been created with -allow_dups set. |
|           | The -save option lets you specify the name of a fileset to write this lookup table to; if -save is omitted, tables are written as scratch files and deleted at the end of the lookup. In create-only mode, -save is, of course, required. |
|           | The -diskpool option lets you specify a disk pool in which to create lookup tables. By default, the operator looks first for a "lookup" disk pool, then uses the default pool (""). Use this option to specify a different disk pool to use. |
share a single copy of the lookup table, rather than having a private copy of the
table for each process. This reduces memory consumption to that of a single
sequential lookup process. This is why partitioning the data, which in a non-
shared-memory environment saves memory by creating smaller tables, also has
the effect of disabling this memory sharing, so that there is no benefit to
partitioning lookup tables on an SMP or cluster.

**Partitioning**

Normally (and by default), lookup tables are partitioned using the entire
partitioning method so that each processing node receives a complete copy of the
lookup table. You can partition lookup tables using another partitioning method,
such as hash, as long as you ensure that all records with the same lookup keys are
partitioned identically. Otherwise, source records may be directed to a partition
that doesn’t have the proper table entry.

For example, if you are doing a lookup on keys a, b, and c, having both the source
data set and the lookup table hash partitioned on the same keys would permit the
lookup tables to be broken up rather than copied in their entirety to each
partition. This explicit partitioning disables memory sharing, but the lookup
operation consumes less memory, since the entire table is not duplicated. Note,
though, that on a single SMP, hash partitioning does not actually save memory.
On MPPs, or where shared memory can be used only in a limited way, or not at
all, it can be beneficial.

**Create-Only Mode**

In its normal mode of operation, the lookup operator takes in a source data set
and one or more data sets from which the lookup tables are built. The lookup
tables are actually represented as file sets, which can be saved if you wish but
which are normally deleted as soon as the lookup operation is finished. There is
also a mode, selected by the -createOnly option, in which there is no source data
set; only the data sets from which the lookup tables are to be built are used as
input. The resulting file sets, containing lookup tables, are saved to persistent
storage.

This create-only mode of operation allows you to build lookup tables when it is
convenient, and use them for doing lookups at a later time. In addition,
initialization time for the lookup processing phase is considerably shorter when
lookup tables already exist.

For example, suppose you have data sets data1.ds and data2.ds and you want to
create persistent lookup tables from them using the name and ID fields as lookup
keys in one table and the name and accountType fields in the other.
For this use of the `lookup` operator, you specify the `-createOnly` option and two `-table` options. In this case, two suboptions for the `-table` options are specified: `-key` and `-save`.

In `osh`, use the following command:

```
$ osh ' lookup -createOnly -table -key name -key ID -save fs1.fs -table -key name -key accountType -save fs2.fs < data1.ds < data2.ds''
```

**Examples**

**Single Lookup Table Record**

Figure 6 shows the lookup of a source record and a single lookup table record.

![Diagram of a source record, lookup record, and output record](image)

**Figure 6  Effect of lookup with Single Lookup Table Record**

This figure shows the source and lookup record and the resultant output record. A source record and lookup record are matched if they have the same values for the key field(s). In this example, both records have `John` as the name and `27` as the ID number. In this example, the lookup keys are the first fields in the record. You can use any field in the record as a lookup key.

Note that fields in a lookup table that match fields in the source record are dropped. That is, the output record contains all of the fields from the source record plus any fields from the lookup record that were not in the source record. Whenever any field in the lookup record has the same name as a field in the source record, the data comes from the source record and the lookup record field is ignored.
Here is the command for this example:

```
$ osh "lookup -table -key Name -key ID
     < inSrc.ds < inLU1.ds > outDS.ds"
```

**Multiple Lookup Table Records**

When there are multiple lookup tables as input, the lookup tables can all use the same key fields, or they can use different sets. **Figure 6** shows the lookup of a source record and two lookup records where both lookup tables have the same key fields.

![Figure 6 Effect of lookup with Two Lookup Table Records](image)

Any fields in the first lookup record not in the source record are concatenated to the output record. Then, any fields in the second lookup record not in the source record or the first lookup record are concatenated to the output record. For each additional lookup data set, any fields not already in the output record are concatenated to the end of the output record.

The **osh** command for this example is:

```
$ osh "lookup -table -key name -key ID
     -table -key name -key ID
     < inSrc.ds < inLU1.ds < inLU2.ds > outDS.ds"
```

Note that in this example you specify the same key fields for both lookup tables. Alternatively, you can specify a different set of lookup keys for each lookup table. For example, you could use name and ID for the first lookup table and the fields accountType and minBalance (not shown in the figure) for the second lookup table. Each of the resulting output records would contain those four fields, where the values matched appropriately, and the remaining fields from each of the three input records.
Here is the `osh` command for this example:

```
$ osh "lookup -table -key name -key ID -table -key accountType
    -key minBalance < inSrc.ds < inLU1.ds < inLU2.ds > outDS.ds"
```

**Interest Rate Lookup Example**

*Figure 7* shows the schemas for a source data set `customer.ds` and a lookup data set `interest.ds`. This operator looks up the interest rate for each customer based on the customer’s account type. In this example, Orchestrate inserts the `entire` partitioner (this happens automatically; you do not need to explicitly include it in your program) so that each processing node receives a copy of the entire lookup table.

Since the interest rate is not represented in the source data set record schema, the `interestRate` field from the lookup record has been concatenated to the source record.

Here is the `osh` code for this example:

```
$ osh "lookup -table -key accountType < customers.ds < interest.ds
    > outDS.ds"
```

*Figure 7  Data Flow for Interest Rate Lookup Example*
Handling Duplicate Fields Example

If, in the previous example, the record schema for `customer.ds` also contained a field named `interestRate`, both the source and the lookup data sets would have a non-lookup-key field with the same name. By default, the `interestRate` field from the source record is output to the lookup record and the field from the lookup data set is ignored.

If you want the `interestRate` field from the lookup data set to be output, rather than the value from the source record, you can use a `modify` operator before the `lookup` operator to drop the `interestRate` field from the source record.

The following diagram shows record schemas for the `customer.ds` and `interest.ds` in which both schemas have a field named `interestRate`.

To make the lookup table's `interestRate` field the one that is retained in the output, use a `modify` operator to drop `interestRate` from the source record. The `interestRate` field from the lookup table record is propagated to the output data set, because it is now the only field of that name.

Figure 8 shows how to use a `modify` operator to drop the `interestRate` field:
The \texttt{osh} command for this example is:

\begin{verbatim}
$ osh ' modify -spec 'drop interestRate;' < customer.ds |
lookup -table -key accountType < interest.ds > outDS.ds'
\end{verbatim}

Note that this is unrelated to using the \texttt{-allow_dups} option on a table, which deals with the case where two records in a lookup table are identical in all the key fields.
The merge Operator

The merge operator combines a sorted master data set with one or more sorted update data sets.

Data Flow Diagram 12 2
Properties 12 3
Syntax and Options 12 4
Merging Records 12 5
   Merging with a Single Update Data Set 12 6
   Merging With Multiple Update Data Sets 12 7
Understanding the merge Operator 12 8
   The Merging Operation 12 9
   Example: Updating National Data with State Data 12 11
   Example: Handling Duplicate Fields 12 14
      Renaming a Duplicate Field 12 15
      Dropping Duplicate Fields 12 15
   Application Scenario: Galactic Industries 12 16
      Why the merge Operator Is Used 12 17
      Why the lookup Operator Is Used 12 17
      Why the entire Operator is Used 12 18
Missing Records 12 18
   Handling Bad Master Records 12 18
   Handling Bad Update Records 12 19
The **merge** operator combines a sorted master data set with one or more sorted update data sets. The fields from the records in the master and update data sets are merged so that the output record contains all the fields from the master record plus any additional fields from matching update record.

A master record and an update record are merged only if both of them have the same values for the merge key field(s) that you specify. Merge key fields are one or more fields that exist in both the master and update records.

By default, Orchestrates inserts partition and sort components to meet the partitioning and sorting needs of the **merge** operator and other operators. See the section "Orchestrate-Inserted Partition and Sort Components" on page xxxvi of this Reference for information on this facility.

As part of preprocessing your data for the **merge** operator, you must remove duplicate records from the master data set. If you have more than one update data set, you must remove duplicate records from the update data sets as well. This chapter describes how to use the **merge** operator. Included in this chapter are examples using the **remdup** operator to preprocess your data.

The **merge** operator is similar in function to the **lookup** operator and the **join** operators. To understand the similarities and differences see "Comparison with Other Operators" on page 28-5.

**Data Flow Diagram**

The **merge** operator merges a master data set with one or more update data sets.
## Properties

### Table 26  **merge Operator Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1 master; 1-n update</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1 output; 1-n reject (optional)</td>
</tr>
<tr>
<td>Input interface schema</td>
<td></td>
</tr>
<tr>
<td>master data set:</td>
<td>mKey0: data_type; ... mKeyk: data_type;</td>
</tr>
<tr>
<td></td>
<td>masterRec:*;</td>
</tr>
<tr>
<td>update data sets:</td>
<td>mKey0: data_type; ... mKeyk: data_type;</td>
</tr>
<tr>
<td></td>
<td>updateRec:<em>; rejectRec:</em>;</td>
</tr>
<tr>
<td>Output interface schema</td>
<td></td>
</tr>
<tr>
<td>output data set:</td>
<td>masterRec:<em>; updateRec1:</em>; updateRec2:<em>; ... updateRecn:</em>;</td>
</tr>
<tr>
<td></td>
<td>rejectRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td></td>
</tr>
<tr>
<td>master to output:</td>
<td>masterRec -&gt; masterRec without record modification</td>
</tr>
<tr>
<td>update to output:</td>
<td>updateRecn -&gt; outputRec rejectRecn -&gt;</td>
</tr>
<tr>
<td>update to reject:</td>
<td>updateRecn -&gt; rejectRecn without record modification (optional)</td>
</tr>
<tr>
<td>Input partitioning style</td>
<td>keys in same partition</td>
</tr>
<tr>
<td>Output partitioning style</td>
<td>distributed keys</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
</tbody>
</table>
Syntax and Options

merge
   -key field [-ci | -cs] [-asc | -desc] [-ebcdic] [-nulls first | last] [param params] ...
   [-collation_sequence locale|collation_file_pathname|OFF]
   [-dropBadMasters | -keepBadMasters]
   [-nowarnBadMasters | -warnBadMasters]
   [-nowarnBadUpdates | -warnBadUpdates]

Table 27  merge Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
</table>
| -key            | -key field [-ci | -cs] [-asc | -desc] [-ebcdic] [-nulls first | last] [param params] ...
|                 | Specifies the name of a merge key field. The -key option may be     |
|                 | repeated if there are multiple merge key fields.                   |
|                 | The -ci option specifies that the comparison of merge key values is |
|                 | case insensitive. The -cs option specifies a case-sensitive        |
|                 | comparison, which is the default.                                  |
| -asc | -desc   | are optional arguments for specifying ascending or                 |
|       |         | descending sorting. By default, the operator uses ascending       |
|       |         | sorting order, that is, smaller values appear before larger       |
|       |         | values in the sorted data set. Specify -desc to sort in descending |
|       |         | sorting order instead, so that larger values appear before        |
|       |         | smaller values in the sorted data set.                            |
| -nulls first | last   | By default fields containing null values appear first in the      |
|             |         | sorted data set. To override this default so that fields         |
|             |         | containing null values appear last in the sorted data set,       |
|             |         | specify nulls last.                                              |
| -ebcdic       | By default data is represented in the ASCII character set. To     |
|               | represent data in the EBCDIC character set, specify this option.   |
|               | The param suboption allows you to specify extra parameters for a   |
|               | field. Specify parameters using property=value pairs separated by  |
|               | commas.                                                             |
The *merge* operator combines a master and one or more update data sets into a single, merged data set based upon the values of a set of merge key fields. Each record of the output data set contains all of the fields from a master record. Concatenated to the end of the output records are any fields from the corresponding update records that are not already in the master record.

### Merging Records

The *merge* operator combines a master and one or more update data sets into a single, merged data set based upon the values of a set of merge key fields. Each record of the output data set contains all of the fields from a master record. Concatenated to the end of the output records are any fields from the corresponding update records that are not already in the master record.

### Table 27  merge Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-collation_sequence</code></td>
<td>`collation_sequence locale</td>
</tr>
<tr>
<td></td>
<td>This option determines how your string data is sorted. You can:</td>
</tr>
<tr>
<td></td>
<td>• Specify a predefined IBM ICU locale</td>
</tr>
<tr>
<td></td>
<td>• Write your own collation sequence using ICU syntax, and supply its <code>collation_file_pathname</code></td>
</tr>
<tr>
<td></td>
<td>• Specify <code>OFF</code> so that string comparisons are made using Unicode code-point value order, independent of any locale or custom sequence.</td>
</tr>
<tr>
<td></td>
<td>By default, Orchestrate sorts strings using byte-wise comparisons.</td>
</tr>
<tr>
<td></td>
<td>For more information, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide, and reference this IBM ICU site:</td>
</tr>
<tr>
<td><code>-dropBadMasters</code></td>
<td><code>dropBadMasters</code></td>
</tr>
<tr>
<td></td>
<td>Rejected masters are not output to the merged data set.</td>
</tr>
<tr>
<td><code>-keepBadMasters</code></td>
<td><code>keepBadMasters</code></td>
</tr>
<tr>
<td></td>
<td>Rejected masters are output to the merged data set. This is the default.</td>
</tr>
<tr>
<td><code>-nowarnBadMasters</code></td>
<td><code>nowarnBadMasters</code></td>
</tr>
<tr>
<td></td>
<td>Do not warn when rejecting bad masters.</td>
</tr>
<tr>
<td><code>-nowarnBadUpdates</code></td>
<td><code>nowarnBadUpdates</code></td>
</tr>
<tr>
<td></td>
<td>Do not warn when rejecting bad updates.</td>
</tr>
<tr>
<td><code>-warnBadMasters</code></td>
<td><code>warnBadMasters</code></td>
</tr>
<tr>
<td></td>
<td>Warn when rejecting bad masters. This is the default.</td>
</tr>
<tr>
<td><code>-warnBadUpdates</code></td>
<td><code>warnBadUpdates</code></td>
</tr>
<tr>
<td></td>
<td>Warn when rejecting bad updates. This is the default.</td>
</tr>
</tbody>
</table>
Corresponding master and update records have the same value for the specified merge key fields.

The action of the **merge** operator depends on whether you specify multiple update data sets or a single update data set. When merging a master data set with multiple update data sets, each update data set may contain only one record for each master record. When merging with a single update data set, the update data set may contain multiple records for a single master record. The following sections describe merging a master data set with a single update data set and with multiple update data sets.

**Merging with a Single Update Data Set**

Figure 10 shows the merge of a master record and a single update record.

![Figure 10](merge_example.png)

**Figure 10  Merge of a Master Record and an Update Record**

The figure shows the master and update records and the resultant merged record. A master record and an update record are merged only if they have the same values for the key field(s). In this example, both records have “John” as the Name and 27 as the ID value. Note that in this example the merge keys are the first fields in the record. You can use any field in the record as a merge key, regardless of its location.

The schema of the master data set determines the data types of the merge key fields. The schemas of the update data sets may be dissimilar but they must contain all merge key fields (either directly or through adapters).

The merged record contains all of the fields from the master record plus any fields from the update record which were not in the master record. Thus, if a field in the update record has the same name as a field in the master record, the data comes from the master record and the update field is ignored.

The master data set of a merge must not contain duplicate records where duplicates are based on the merge keys. That is, no two master records can have the same values for all merge keys.
For a merge using a single update data set, you can have multiple update records, as defined by the merge keys, for the same master record. In this case, you get one output record for each master/update record pair. In the figure above, if you had two update records with “John” as the Name and 27 as the value of ID, you would get two output records.

**Merging With Multiple Update Data Sets**

In order to merge a master and multiple update data sets, all data sets must be sorted and contain no duplicate records where duplicates are based on the merge keys. That is, there must be at most one update record in each update data set with the same combination of merge key field values for each master record. In this case, the **merge** operator outputs a single record for each unique combination of merge key fields.

By default, Orchestrates inserts partition and sort components to meet the partitioning and sorting needs of the **merge** operator and other operators. See the section “Orchestrate-Inserted Partition and Sort Components” on page xxxvi of this Reference for information on this facility.

The following figure shows the merge of a master record and two update records (one update record from each of two update data sets):

![Figure 9 Merge of a Master Record and Two Update Records](image)

Any fields in the first update record not in the master record are concatenated to the output record. Then, any fields in the second update record not in the master record or the first update record are concatenated to the output record. For each additional update data set, any fields not already in the output record are concatenated to the end of the output record.
Understanding the merge Operator

Figure 10 shows the overall data flow for a typical use of the merge operator.

This diagram shows the overall process as one step. Note that the remdup operator is required only if you have multiple update data sets. If you have only a single update data set, the data set may contain more than one update record for each master record.
Another method is to save the output of the `remdup` operator to a data set and then pass that data set to the `merge` operator, as shown in Figure 11.

![Diagram of Alternate Data Flow for merge Operator]

Figure 11  Alternate Data Flow for merge Operator

This method has the disadvantage that you need the disk space to store the preprocessed master and update data sets and the merge must be in a separate step from the remove duplicates operator. However, the intermediate files can be checked for accuracy before merging them, or used by other processing steps that require records without duplicates.

### The Merging Operation

When data sets are merged, one is the master and all other are update data sets. The master data set is always connected to input 0 of the operator. The merged output data set always contains all of the fields from the records in the master data set. In addition, it contains any additional fields from the update data sets.
Chapter 12  The merge Operator

Understanding the merge Operator

The following diagram shows the record schema of the output data set of the merged operator, based on the record schema of the master and update data sets:

This data-flow diagram shows the record schema of the master and update data sets. The record schema of the master data set has five fields and all five of these appear in the record schema of the output data set. The update data set also has five fields, but only two of these (f and g) are copied to the output data set because the remaining fields (a, b, and c) already exist in the master data set.

If the example above is extended to include a second update data set with a record schema containing the following fields:

```
    a  b  d  h  i
```

Then the fields in the merged output record are now:

```
    a  b  c  d  e  f  g  h  i
```

because the last two fields (h and i) occur only in the second update data set and not in the master or first update data set. The unique fields from each additional update data set are concatenated to the end of the merged output.

If there is a third update data set with a schema that contains the fields:

```
    a  b  d  h
```

it adds nothing to the merged output since none of the fields is unique. Thus if master and five update data sets are represented as:

```
M U1 U2 U3 U4 U5
```
where \( M \) represents the master data set and \( U_n \) represent update data set \( n \), and if the records in all six data sets contain a field named \( b \), the output record has a value for \( b \) taken from the master data set. If a field named \( e \) occurs in the \( U_2, U_3 \), and \( U_5 \) update data sets, the value in the output comes from the \( U_2 \) data set since it is the first one encountered.

Therefore, the record schema of the merged output record is the sequential concatenation of the master and update schema(s) with overlapping fields having values taken from the master data set or from the first update record containing the field. The values for the merge key fields are taken from the master record, but are identical values to those in the update record(s).

**Example: Updating National Data with State Data**

Figure 12 shows the schemas for a master data set named National.ds and an update data set named California.ds. The merge operation is performed to combine the two; the output is saved into a new data set named Combined.ds.

National.ds schema:
- Customer:int16;
- Month:string[3];
- Name:string[21];
- Balance:sfloat;
- Salesman:string[8];
- AccountType:int8;

California.ds schema:
- Customer:int16;
- Month:string[3];
- Name:string[21];
- CalBalance:sfloat;
- Status:string[8];

Combined.ds schema:
- Customer:int16;
- Month:string[3];
- Name:string[21];
- Balance:sfloat;
- Salesman:string[8];
- AccountType:int8;
- CalBalance:sfloat;
- Status:string[8];
The *National.ds* master data set contains the following record:

National.ds  
schema:
- Customer:int16;
- Month:string[3];
- Name:string[21];
- Balance:sfloat;
- Salesman:string[8];
- AccountType:int8;

<table>
<thead>
<tr>
<th>86111</th>
<th>JUN</th>
<th>Jones, Bob</th>
<th>345.98</th>
<th>Steve</th>
<th>12</th>
</tr>
</thead>
</table>

master record in National.ds

The *Customer* and *Month* fields are used as the merge key fields. You also have a record in the update data set named *California.ds* that contains the following record:

California.ds  
schema:
- Customer:int16;
- Month:string[3];
- Name:string[21];
- CalBalance: sfloat;
- Status:string[8];

<table>
<thead>
<tr>
<th>86111</th>
<th>JUN</th>
<th>Jones, Bob</th>
<th>637.04</th>
<th>Normal</th>
</tr>
</thead>
</table>

update record in California.ds

After you merge these records, the result is:

<table>
<thead>
<tr>
<th>86111</th>
<th>JUN</th>
<th>Jones, Bob</th>
<th>345.98</th>
<th>Steve</th>
<th>12</th>
<th>637.04</th>
<th>Normal</th>
</tr>
</thead>
</table>

record in Combined.ds data set

This example shows that the *CalBalance* and *Status* fields from the update record have been concatenated to the fields from the master record. The combined record has the same values for the key fields as do both the master and the update records since they must be the same for the records to be merged.

Figure 13 shows the data flow for this example. The original data comes from the data sets *NationalRaw.ds* and *CaliforniaRaw.ds*. *National.ds* and *California.ds* are created by first sorting and then removing duplicates from *NationalRaw.ds* and *CaliforniaRaw.ds*. 
For the `remdup` operators and for the `merge` operator you specify the same key two fields:

- **Option:** `key` **Value:** `Month`
- **Option:** `key` **Value:** `Customer`

The steps for this example have been written separately so that you can check the output after each step. Because each step is separate, it is easier to understand the entire process. Later all of the steps are combined together into one step.

The separate steps, shown as `osh` commands, are:

```bash
# Produce National.ds
$ osh "remdup -key Month -key Customer < NationalRaw.ds > National.ds"

# Produce California.ds
$ osh "remdup -key Month -key Customer < CaliforniaRaw.ds > California.ds"

# Perform the merge
$ osh "merge -key Month -key Customer < National.ds < California.ds > Combined.ds"
```

This example takes `NationalRaw.ds` and `CaliforniaRaw.ds` and produces `Combined.ds` without creating the intermediate files.

When combining these three steps into one, you use a named virtual data sets to connect the operators.

```bash
$ osh "remdup -key Month -key Customer < CaliforniaRaw.ds > California.v;
  remdup -key Month -key Customer < NationalRaw.ds |
  merge -key Month -key Customer < California.v > Combined.ds"
```

In this example, `California.v` is a named virtual data set used as input to `merge`. 

![Data Flow for merge Example](image)

Figure 13  **Data Flow for merge Example**
**Example: Handling Duplicate Fields**

If the record schema for `CaliforniaRaw.ds` from the previous example is changed so that it now has a field named `Balance`, both the master and the update data sets will have a field with the same name. By default, the `Balance` field from the master record is output to the merged record and the field from the update data set is ignored.

**Figure 14** shows record schemas for the `NationalRaw.ds` and `CaliforniaRaw.ds` in which both schemas have a field named `Balance`:

<table>
<thead>
<tr>
<th>Master Data Set</th>
<th>Update Data Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>NationalRaw.ds</td>
<td>CaliforniaRaw.ds</td>
</tr>
<tr>
<td>schema:</td>
<td>schema:</td>
</tr>
<tr>
<td>Customer:int16;</td>
<td>Customer:int16;</td>
</tr>
<tr>
<td>Month:string[3];</td>
<td>Month:string[3];</td>
</tr>
<tr>
<td>Balance:sfloat;</td>
<td>Balance:sfloat;</td>
</tr>
<tr>
<td>Salesman:string[8];</td>
<td>CalBalance:sfloat;</td>
</tr>
<tr>
<td>AccountType:int8;</td>
<td>Status:string[8];</td>
</tr>
</tbody>
</table>

**Figure 14** *Data Sets Containing the Same Field Name*

If you want the `Balance` field from the update data set to be output by the `merge` operator, you have two alternatives, both using the `modify` operator:

- Rename the `Balance` field in the master data set.
- Drop the `Balance` field from the master record.

In either case, the `Balance` field from the update data set propagates to the output record because it is the only `Balance` field.

The following figure shows the data flow for both methods.
Renaming a Duplicate Field

The `osh` command for this approach is:

```
$ osh 'remdup -key Month -key Customer < CaliforniaRaw.ds > California.v;
    remdup -key Month -key Customer < NationalRaw.ds |
    modify 'OldBalance = Balance' |
    merge -key Month -key Customer < California.v > Combined.ds'
```

The name of the `Balance` field has been changed to `OldBalance`. The `Balance` field from the update data set no longer conflicts with a field in the master data set and is added to records by the `merge`.

Dropping Duplicate Fields

Another method of handling duplicate field names is to drop `Balance` from the master record. The `Balance` field from the update record is written out to the merged record because it is now the only field with that name.

The `osh` command for this approach is:

```
$ osh 'remdup -key Month -key Customer < CaliforniaRaw.ds > California.v';
    remdup -key Month -key Customer < NationalRaw.ds |
    modify 'DROP Balance' |
    merge -key Month -key Customer < California.v > Combined.ds'
```
Application Scenario: Galactic Industries

This section contains an extended example that illustrates the use of the merge operator in a semi-realistic data flow. The example is followed by an explanation of why the operators were chosen. Files have been provided to allow you to run this example yourself. The files are in `SAP_OCH_HOME/examples/doc/mergeop` subdirectory of the Orchestrate installation directory.

Galactic Industries stores certain customer data in one database table and orders received for a given month in another table.

The customer table contains one entry per customer, indicating the location of the customer, how long she has been a customer, the customer contact, and other customer data. Each customer in the table is also assigned a unique identifier, `cust_id`. However, the customer table contains no information concerning what the customer has ordered.

The order table contains details about orders placed by customers; for each product ordered, the table lists the product name, amount, price per unit, and other product information. The order table can contain many entries for a given customer. However, the only information about customers in the order table is the customer identification field, indicated by a `cust_id` field which matches an entry in the customer table.

Each month Galactic Industries needs to merge the customer information with the order information to produce reports, such as how many of a given product were ordered by customers in a given region. Because the reports are reviewed by human eyes, they also need to perform a lookup operation which ties a description of each product to a `product_id`. Galactic Industries performs this merge and lookup operation using Orchestrate.

The Orchestrate solution is based on the fact that Galactic Industries has billions of customers, trillions of orders, and needs the reports fast.

The `osh` script for the solution follows.

```bash
# import the customer file; store as a virtual dataset.
import -schema $CUSTOMER_SCHEMA -file customers.txt -readers 4 |
peek -name -nrecs 1 >customers.v;

# import the order file; store as a virtual dataset.
import -schema $ORDER_SCHEMA -file orders.txt -readers 4 |
peek -name -nrecs 1 >orders.v;
```
# import the product lookup table; store as a virtual dataset.
import -schema $LOOKUP_SCHEMA -file lookup_product_id.txt | entire | # entire partitioning only necessary in MPP environments peek -name -nrecs 1 >lookup_product_id.v;

# merge customer data with order data; lookup product descriptions; # store as a persistent dataset.
merge -key cust_id -dropBadMasters # customer did not place an order this period < customers.v < orders.v 1>| orders_without_customers.ds | # if not empty, we have a problem

lookup -key product_id -ifNotFound continue # allow products that don't have a description < lookup_product_id.v | peek -name -nrecs 10 >| customer_orders.ds;

Why the merge Operator Is Used

The **merge** operator is not the only component in the Orchestrate library capable of merging the customer and order tables. An identical merged output dataset could be produced with either the **lookup** or **innerjoin** operator. Furthermore, if the **-dropBadMasters** behavior was not chosen, merging could also be performed using the **leftouterjoin** operator.

Galactic Industries' needs make the **merge** operator the best choice. If the **lookup** operator were used the customer table would be used as the lookup table. Since Galactic Industries has billions of customers and only a few Gigabytes of RAM on its SMP the data would have to spill over onto paging space, resulting in a dramatic decrease in processing speed. Because Galactic Industries is interested in identifying entries in the order table that do not have a corresponding entry in the customer table, the **merge** operator is a better choice than the **innerjoin** or **leftouterjoin** operator, because the **merge** operator allows for the capture of bad update records in a reject dataset (**orders_without_customers.ds** in the script above).

Why the lookup Operator Is Used

Similar functionality can be obtained from **merge**, **lookup**, or one of the join operators. For the task of appending a descriptions of product field to each record the **lookup** operator is most suitable in this case for the following reasons.

- Since there are around 500 different products and the length of each description is approximately 50 bytes, a lookup table consists of only about 25 Kilobytes of data. The size of the data makes the implementation of a lookup
table in memory feasible, and means that the scan of the lookup table based on the key field is a relatively fast operation.

- Use of either the merge or one of the join operators would necessitate a repartition and resort of the data based on the lookup key. In other words, having partitioned and sorted the data by cust_id to accomplish the merge of customer and order tables, Galactic industries would then have to perform a second partitioning and sorting operation based on product_id in order to accomplish the lookup of the product description using either merge or innerjoin. Given the small size of the lookup table and the huge size of the merged customer/ order table, the lookup operator is clearly the more efficient choice.

Why the entire Operator is Used

The lookup dataset lookup_product_id.v is entire partitioned. The entire partitioner copies all records in the lookup table to all partitions ensuring that all values are available to all records for which lookup entries are sought. The entire partitioner is only required in MPP environments, due to the fact that memory is not shared between nodes of an MPP. In an SMP environment, a single copy of the lookup table is stored in memory that can be accessed by all nodes of the SMP. Using the entire partitioner in the flow makes this example portable to MPP environments, and due to the small size of the lookup table is not particularly wasteful of resources.

Missing Records

The merge operator expects that for each master record there exists a corresponding update record, based on the merge key fields, and vice versa. If the merge operator takes a single update data set as input, the update data set may contain multiple update records for a single master record.

By using command-line options to the operator, you can specify the action of the operator when a master record has no corresponding update record (a bad master record) or when an update record has no corresponding master record (a bad update record).

Handling Bad Master Records

A master record with no corresponding update record is called a bad master. When a master record is encountered which has no corresponding update record, you can specify whether the master record is to be copied to the output or dropped. You can also request that you get a warning message whenever this happens.

By default, the merge operator writes a bad master to the output data set and issues a warning message. Default values are used for fields in the output record which would normally have values taken from the update data set. You can
specify `-nowarnBadM asters` to the `merge` operator to suppress the warning message issued for each bad master record.

Suppose the data in the master record is:

```
86111 | JUN | Lee, Mary | 345.98 | Steve | 12
```

record in National.ds data set

The first field, `Customer`, and the second field, `Month`, are the key fields.

If the `merge` operator cannot find a record in the update data set for customer `86111` for the month of `June`, then the output record is:

```
86111 | JUN | Lee, Mary | 345.98 | Steve | 12 | 0.0 | 0x00
```

record in Combined.ds data set

The last two fields in the output record, `OldBalance` and `Status`, come from the update data set. Since there is no update record from which to get values for the `OldBalance` and `Status` fields, default values are written to the output record. This default value of a field is the default for that particular data type. Thus, if the value is an `sfloat`, the field in the output data set has a value of 0.0. For a fixed-length string, the default value for every byte is `0x00`.

If you specify `-dropBadM asters`, master records with no corresponding update record are discarded (not copied to the output data set).

**Handling Bad Update Records**

When an update record is encountered which has no associated master record, you can control whether the update record is dropped or is written out to a separate reject data set.

In order to collect bad update records from an update data set, you attach one output data set, called a reject data set, for each update data set. The presence of a reject data set configures the `merge` operator to write bad update records to the reject data set.

In the case of a `merge` operator taking as input multiple update data sets, you must attach a reject data set for each update data set if you want to save bad update records. You cannot selectively collect bad update records from a subset of the update data sets.

By default, the `merge` operator issues a warning message when it encounters a bad update record. You can use the `-nowarnBadM asters` option to the operator to suppress this warning.

For example, suppose you have a data set named `National.ds` that has one record per the key field `Customer`. You also have an update data set named `California`. 
ds, which also has one record per Customer. If you now merge these two data sets, and include a reject data set, bad update records are written to the reject data set for all customer records from California.ds that are not already in National.ds. If the reject data set is empty after the completion of the operator, it means that all of the California.ds customers already have National.ds records.

Figure 16 shows an example using a reject data set.

In osh, the command is:

```
$ osh "merge -key customer < National.ds < California.ds
   > Combined.ds > CalReject.ds"
```

After this step executes, CalReject.ds contains all records from update data set that did not have a corresponding record in the master data set.

Figure 17 shows the merge operator with multiple update sets (U1 and U2) and reject data sets (R1 and R2). In the figure, M indicates the master data set and O indicates the merged output data set.

As you can see, you must specify a reject data set for each update data set in order to save bad update records. You must also specify the output reject data sets in the same order as you specified the input update data sets.
For example:

$ osh 'merge -key customer
  < National.ds < California.ds < NewYork.ds
  > Combined.ds > CalRejects.ds > NewYorkRejects.ds"
The modify Operator

The modify operator alters the record schema of its input data set.

Data Flow Diagram 3
Properties 3
Syntax and Options 3
Transfer Behavior 4
Performing Conversions 5
  Keeping and Dropping Fields 6
  Renaming Fields 6
  Duplicating a Field and Giving It a New Name 6
  Changing a Field's Data Type 7
  Default Data Type Conversion 7
  Data Type Conversion Errors 9
  How to Convert a Data Type 10
  Date Field Conversions 10
date Formats 13
    uformat date Format 13
date Format 13
  Decimal Field Conversions 14
    Rounding Type 16
  Raw Field Length Extraction 17
  String and Ustring Field Conversions 18
  String Conversions and Lookup Tables 23
  Time Field Conversions 26
time Formats 27
    uformat time Format 27
time Format 27
    Converting Time Fields to Integers Example 28
  Timestamp Field Conversions 29
timestamp Formats 30
timestamp Format 30
The `modify` operator takes a single data set as input and alters (modifies) the record schema of the input data set to create the output data set. The `modify` operator changes the representation of data before or after it is processed by another operator, or both. Use it to modify:

- Elements of the record schema of an input data set to the interface required by the operator to which it is input
- Elements of an operator’s output to those required by the data set that receive the results

The operator performs the following modifications:

- Keeping and dropping fields
- Renaming fields
- Changing a field’s data type
- Changing the null attribute of a field

The `modify` operator has no usage string.
Data Flow Diagram

Properties

Table 28  modify Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>any (parallel mode)</td>
</tr>
<tr>
<td>Collection method</td>
<td>any (sequential mode)</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

Syntax and Options

Terms in italic typeface are option strings you supply. When your option string contains a space or a tab character, you must enclose it in single quotes.

```
modify ' modify_spec1; modify_spec2; ... modify_specn;
```

where each modify_spec specifies a conversion you want to perform. “Performing Conversions” on page 13-5 describes the conversions that the modify operator can perform.

- Enclose the list of modifications in single quotation marks.
- Separate the modifications with a semi-colon.
• If modify_spec takes more than one argument, separate the arguments with a comma and terminate the argument list with a semi-colon, as in the following example:

```plaintext
modify 'keep field1, field2, ... fieldn;
```

Multi-byte Unicode character data is supported for fieldnames in the modify specifications below.

The modify_spec can be one of the following:

• DROP
• KEEP
• replacement_spec
• NOWARN

To drop a field:

```plaintext
DROP fieldname [, fieldname ... ]
```

To keep a field:

```plaintext
KEEP fieldname [, fieldname ... ]
```

To change the name or data type of a field, or both, specify a replacement-spec, which takes the form:

```plaintext
new-fieldname [: new-type] = [ explicit-conversion-spec ] old-fieldname
```

Replace the old field name with the new one. The default type of the new field is the same as that if the old field unless it is specified by the output type of the conversion-spec if provided. Multiple new fields can be instantiated based on the same old field.

When there is an attempt to put a null in a field that has not been defined as nullable, Orchestrate issues an error message and terminates the application. However, a warning is issued at step-check time. To disable the warning specify the NOWARNF option.

### Transfer Behavior

Fields of the input data set that are not acted on by the modify operator are transferred to the output data set unchanged.

In addition, changes made to fields are permanent. Thus:

• If you drop a field from processing by means of the modify operator, it does not appear in the output of the operation for which you have dropped it.
• If you use an upstream modify operator to change the name, type, or both of an input field, the change is permanent in the output unless you restore the
field name, type, or both by invoking the modify operator downstream of the operation for whose sake the field name was changed.

In the following example, the modify operator changes field names upstream of an operation and restores them downstream of the operation, as indicated in Table 29.

Table 29  modify Operator Change Name Example

<table>
<thead>
<tr>
<th>Source Field Name</th>
<th>Destination Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upstream of Operator</td>
<td></td>
</tr>
<tr>
<td>aField</td>
<td>field1</td>
</tr>
<tr>
<td>bField</td>
<td>field2</td>
</tr>
<tr>
<td>cField</td>
<td>field3</td>
</tr>
<tr>
<td>Downstream of Operator</td>
<td></td>
</tr>
<tr>
<td>field1</td>
<td>aField</td>
</tr>
<tr>
<td>field2</td>
<td>bField</td>
</tr>
<tr>
<td>field3</td>
<td>cField</td>
</tr>
</tbody>
</table>

You set these fields with the command:
```
$ osh " ... | modify 'field1=aField; field2=bField; field3=cField; ' | op | modify 'aField=field1; bField=field2; cField=field3; ' ...
```

Performing Conversions

The section “Allowed Conversions” on page 13-38 provides a complete list of conversions you can effect using the modify operator.

This section discusses these topics:

- “Keeping and Dropping Fields” on page 13-6
- “Renaming Fields” on page 13-6
- “Duplicating a Field and Giving It a New Name” on page 13-6
- “Changing a Field’s Data Type” on page 13-7
- “Default Data Type Conversion” on page 13-7
- “Date Field Conversions” on page 13-10
- “Decimal Field Conversions” on page 13-14
- “Raw Field Length Extraction” on page 13-17
- “String and Ustring Field Conversions” on page 13-18
- “String Conversions and Lookup Tables” on page 13-23
- “Time Field Conversions” on page 13-26
Keeping and Dropping Fields

Invoke the `modify` operator to keep fields in or drop fields from the output. Here are the effects of keeping and dropping fields:

- If you choose to drop a field or fields, all fields are retained except those you explicitly drop.
- If you chose to keep a field or fields, all fields are excluded except those you explicitly keep.

In `osh` you specify either the keyword `keep` or the keyword `drop` to keep or drop a field, as follows:

```
modify 'keep field1, field2, ... fieldn;
modify 'drop field1, field2, ... fieldn;
```

Renaming Fields

To rename a field specify the attribution operator (=), as follows:

```
modify 'newField1=oldField1; newField2=oldField2; ... newFieldn=oldFieldn;
```

Duplicating a Field and Giving It a New Name

You can duplicate a field and give it a new name, that is, create multiple new names for the same old name. You can also convert the data type of a field and give it a new name.

```
To duplicate and rename a field or duplicate it and change its data type use the attribution operator (=). The operation must be performed by one `modify` operator, that is, the renaming and duplication must be specified in the same command as follows:
```

```
$ osh "modify 'a_1 = a; a_2 = a;'
$ osh "modify c_1 = conversionSpec(c); c_2 = conversionSpec(c);"
```

```
Note: This does not work with aggregates.
```

```
changing a field’s data type

sometimes, although field names are the same, an input field is of a type that differs from that of the same field in the output, and conversion must be performed. orchestrate often automatically changes the type of the source field to match that of the destination field. sometimes, however, you must invoke the modify operator to perform explicit conversion. the next sections discuss default data type conversion and data type conversion errors. the subsequent sections discuss non-default conversions of orchestrate data types.

default data type conversion

for a data set to be used as input to or output from an operator, its record schema must be compatible with that of the operator’s interface. that is:

• the names of the data set’s fields must be identical to the names of the corresponding fields in the operator interface. use the modify operator to change them if they are not (see “renaming fields” on page 13-6).

• the data type of each field in the data set must be compatible with that of the corresponding field in the operator interface. data types are compatible if orchestrate can perform a default data type conversion, translating a value in a source field to the data type of a destination field.
Figure 20 shows an input data set schema in which the data types of some fields do not match those of the corresponding fields and Orchestrate’s default conversion of these types:

input data set
schema:
field1:int8;
field2:int16;
field3:int16;

default conversion

field1:int32; field2:int16; field3:sfloat;

modify

field1:int32; field2:int16; field3:sfloat;

output data set

Figure 20  Data Flow With Default Data Type Conversion

See Table 30 which shows the default data conversion types.

In this example, the disparate fields are compatible and:

• field1’s data type is automatically converted from int8 to int32.
• field3’s data type is automatically converted from int16 to sfloat.

Orchestrate performs default type conversions on Orchestrate built-in numeric types (integer and floating point) as defined in C: A Reference Manual (3rd edition) by Harbison and Steele. Orchestrate also performs default data conversions involving decimal, date, time, and timestamp fields. The remaining allowable data type conversions are performed explicitly, using the modify operator, as described in this chapter.
Table 30 shows the default data type conversions performed by Orchestrate and the conversions that you can perform with the modify operator.

**Table 30  Data Type Conversions**

<table>
<thead>
<tr>
<th>Source Field</th>
<th>Destination Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8</td>
<td>d m m m m m m m m</td>
</tr>
<tr>
<td>uint8</td>
<td>d m m m m m m m m</td>
</tr>
<tr>
<td>int16</td>
<td>d m m m m m m m m</td>
</tr>
<tr>
<td>uint16</td>
<td>d m m m m m m m m</td>
</tr>
<tr>
<td>int32</td>
<td>d m m m m m m m m</td>
</tr>
<tr>
<td>uint32</td>
<td>d m m m m m m m m</td>
</tr>
<tr>
<td>int64</td>
<td>d m m m m m m m m</td>
</tr>
<tr>
<td>uint64</td>
<td>d m m m m m m m m</td>
</tr>
<tr>
<td>sfloat</td>
<td>d m m m m m m m m</td>
</tr>
<tr>
<td>dfloat</td>
<td>d m m m m m m m m</td>
</tr>
<tr>
<td>decimal</td>
<td>d m m m m m m m m</td>
</tr>
<tr>
<td>string</td>
<td>d m m m m m m m m</td>
</tr>
<tr>
<td>ustring</td>
<td>d m m m m m m m m</td>
</tr>
<tr>
<td>raw</td>
<td>m m m m m m m m m</td>
</tr>
<tr>
<td>date</td>
<td>m m m m m m m m m</td>
</tr>
<tr>
<td>time</td>
<td>m m m m m m m m m</td>
</tr>
<tr>
<td>timestamp</td>
<td>m m m m m m m m m</td>
</tr>
</tbody>
</table>

**Data Type Conversion Errors**

A data type conversion error occurs when a conversion cannot be performed. Orchestrate's action when it detects such an error differs according to whether the
destination field has been defined as nullable, according to the following three rules:

- If the destination field has been defined as nullable, Orchestrate sets it to null.
- If the destination field has not been defined as nullable but you have directed modify to convert a null to a value, Orchestrate sets the destination field to the value. To convert a null to a value supply the handle_null conversion specification.

For complete information on converting a null to a value, see “Out-of-Band to Normal Representation” on page 13-32.

- If the destination field has not been defined as nullable, Orchestrate issues an error message and terminates the application. However, a warning is issued at step-check time. To disable the warning specify the nowarn option.

How to Convert a Data Type

To convert the data type of a field, pass the following argument to the modify operator:

\[ \text{destField[:dataType]} = \text{[conversionSpec]}(\text{sourceField}); \]

where:

- destField is the field in the output data set
- dataType optionally specifies the data type of the output field. This option is allowed only when the output data set does not already have a record schema, which is typically the case.
- sourceField specifies the field in the input data set
- conversionSpec specifies the data type conversion specification; you need not specify it if a default conversion exists (see “Default Data Type Conversion” on page 13-7). A conversion specification can be double quoted, single quoted, or not quoted, but it cannot be a variable.

Note that once you have used a conversion specification to perform a conversion, Orchestrate performs the necessary modifications to translate a conversion result to the numeric data type of the destination. For example, you can use the conversion hours_from_time to convert a time to an int8, or to an int16, int32, dfloat, and so on.

Date Field Conversions

Orchestrate performs no automatic type conversion of date fields. Either an input data set must match the operator interface or you must effect a type conversion by means of the modify operator.
The following table lists the conversions involving the date field. For a description of the formats, refer to “date Formats” on page 13-13.

Table 31  date Type Conversions Performed by modify

<table>
<thead>
<tr>
<th>Conversion Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateField = date_from_days_since[ date] (int32Field)</td>
<td>date from days since</td>
</tr>
<tr>
<td>dateField = date_from_julian_day(uint32Field)</td>
<td>date from Julian day</td>
</tr>
<tr>
<td>dateField = date_from_string[date_format</td>
<td>uformat] (stringField)</td>
</tr>
<tr>
<td>dateField = date_from_ustring[date_format</td>
<td>uformat] (ustringField)</td>
</tr>
<tr>
<td>dateField = date_from_timestamp(tsField)</td>
<td>date from timestamp</td>
</tr>
<tr>
<td>int8Field = month_day_from_date(dateField)</td>
<td>day of month from date</td>
</tr>
<tr>
<td>int8Field = weekday_from_date<a href="dateField">originDay</a></td>
<td>day of week from date</td>
</tr>
<tr>
<td>int16Field = year_day_from_date(dateField)</td>
<td>day of year from date (returned value 1–366)</td>
</tr>
<tr>
<td>int32Field = days_since_from_date[source_date] (dateField)</td>
<td>days since date</td>
</tr>
<tr>
<td>uint32Field = julian_day_from_date(dateField)</td>
<td>Julian day from date</td>
</tr>
</tbody>
</table>

Orchestrate 7.0 Operators Reference  13 11
A date conversion to or from a numeric field can be specified with any Orchestrate numeric data type. Orchestrate performs the necessary modifications and either translates a numeric field to the source data type shown above or

Table 31  date Type Conversions Performed by modify (continued)

<table>
<thead>
<tr>
<th>Conversion Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int8Field = month_from_date(dateField)</code></td>
<td>month from date</td>
</tr>
<tr>
<td><code>dateField = next_weekday_from_date[day](dateField)</code></td>
<td>next weekday from date</td>
</tr>
<tr>
<td><code>dateField = previous_weekday_from_date[day](dateField)</code></td>
<td>previous weekday from date</td>
</tr>
<tr>
<td>`stringField = string_from_date[date_format</td>
<td>uformat](dateField)`</td>
</tr>
<tr>
<td><code>ustringField = ustring_from_date(dateField)</code></td>
<td>By default, the string format is <code>yyyy-mm-dd</code>. date_format and uformat are described in “date Formats” on page 13-13.</td>
</tr>
<tr>
<td><code>tsField = timestamp_from_date[time](dateField)</code></td>
<td>timestamp from date</td>
</tr>
<tr>
<td><code>int16Field = year_from_date(dateField)</code></td>
<td>year from date</td>
</tr>
<tr>
<td><code>int8Field = year_week_from_date(dateField)</code></td>
<td>week of year from date</td>
</tr>
</tbody>
</table>
translates a conversion result to the numeric data type of the destination. For example, you can use the conversion `month_day_from_date` to convert a date to an int8, or to an int16, int32, dfloat, etc.

**date Formats**

Four conversions, `string_from_date`, `ustring_from_date`, `date_from_string`, and `ustring_from_date`, take as a parameter of the conversion a date format or a uformat date format. These formats are described below. The default format of the date contained in the string is \texttt{yyyy-mm-dd}.

The format string requires that you provide enough information for Orchestrate to determine a complete date (either day, month, and year, or year and day of year). The date components of a source string (date, month, and year) must be zero-padded to the character length specified by the format string; Orchestrate zero pads the date components of a destination string to the specified length.

**uformat date Format**

The uformat date format provides support for international components in date fields. It’s syntax is:

\texttt{String\%macroString\%macroString\%macroString\%macroString}

where \%macro is a date formatting macro such as \%mmm for a 3-character English month. See “date Format” below for a description of the date format macros. Only the String components of uformat can include multi-byte Unicode characters.

**date Format**

The format string requires that you provide enough information for Orchestrate to determine a complete date (either day, month, and year, or year and day of year). The date components of a source string (date, month, and year) must be zero-padded to the character length specified by the format string; Orchestrate zero pads the date components of a destination string to the specified length.

The possible format components are:

- \%dd: A two digit day.
- \%mm: A two digit month.
- \%yy: A two digit year derived from a year cutoff of 1900.
- \%year_cutoff\%yy: A two digit year derived from \%yy and the specified year cutoff.
  (For example, if you specify the year format as \%2000\%yy, two-digit values which represent years from 2000-2099 is be imported and exported.)
- \%yyyy: A four digit year.
- \%d\%d: Day of year in three digit form (range of 1 - 366)
The default date format is as follows:
%yyyy-%mm-%dd

When you specify a date format string, prefix each component with the percent symbol (%). Separate the string’s components with any character except the percent sign (%).

For example, the format string %mm/ %d d/ %yyyy specifies that slashes separate the string’s date components; the format %dd - %yy specifies that the string stores the date as a value from 1 to 366, derives the year from the current year cutoff of 1900, and separates the two components with a dash (-).

Figure 18 shows the modification of a date field to three integers. The modify operator takes:
- The day of the month portion of a date field and writes it to an 8-bit integer
- The month portion of a date field and writes it to an 8-bit integer
- The year portion of a date field and writes it to a 16-bit integer.

Use the following osh command:

```
$ osh "... | modify 'dayField = month_day_from_date(dField);
                monthField = month_from_date(dField);
                yearField = year_from_date(dField);' | ..."
```

Decimal Field Conversions

By default Orchestrate converts decimal fields to and from all numeric data types and to and from string fields. The default rounding method of these conversion is truncate toward zero. However, the modify operator can specify a different rounding method. See “Rounding Type” on page 13-16.
The operator can specify **fix_zero** so that a source decimal containing all zeros (by default illegal) is treated as a valid decimal with a value of zero.

**Orchestrate** does not perform range or representation checks of the fields when a source and destination decimal have the same precision and scale. However, you can specify the **decimal_from_decimal** conversion to force **Orchestrate** to perform an explicit range and representation check. This conversion is useful when one decimal supports a representation of zeros in all its digits (normally illegal) and the other does not.

The following table lists the conversions involving decimal fields:

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Conversion Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal from decimal</td>
<td><code>decimalField = decimal_from_decimal[r_type](decimalField)</code></td>
</tr>
<tr>
<td>decimal from dfloat</td>
<td><code>decimalField = decimal_from_dfloat[r_type](dfloatField)</code></td>
</tr>
<tr>
<td>decimal from string</td>
<td><code>decimalField = decimal_from_string[r_type](stringField)</code></td>
</tr>
<tr>
<td>decimal from ustring</td>
<td><code>decimalField = decimal_from_ustring[r_type](ustringField)</code></td>
</tr>
<tr>
<td>dfloat from decimal</td>
<td><code>dfloatField = dfloat_from_decimal[fix_zero](decimalField)</code></td>
</tr>
<tr>
<td>dfloat from decimal</td>
<td><code>dfloatField = mantissa_from_decimal(decimalField)</code></td>
</tr>
<tr>
<td>int32 from decimal</td>
<td><code>int32Field = int32_from_decimal[r_type,fix_zero](decimalField)</code></td>
</tr>
<tr>
<td>int64 from decimal</td>
<td><code>int64Field = int64_from_decimal[r_type,fix_zero](decimalField)</code></td>
</tr>
<tr>
<td>string from decimal</td>
<td><code>stringField = string_from_decimal[fix_zero] [suppress_zero](decimalField)</code></td>
</tr>
<tr>
<td>ustring from decimal</td>
<td><code>ustringField = ustring_from_decimal[fix_zero] [suppress_zero](decimalField)</code></td>
</tr>
<tr>
<td>uint64 from decimal</td>
<td><code>uint64Field = uint64_from_decimal[r_type,fix_zero](decimalField)</code></td>
</tr>
</tbody>
</table>

A decimal conversion to or from a numeric field can be specified with any **Orchestrate** numeric data type. **Orchestrate** performs the necessary modification. For example, **int32_from_decimal** converts a decimal either to an int32 or to any numeric data type, such as int16, or uint32.

The **fix_zero** specification causes a decimal field containing all zeros (normally illegal) to be treated as a valid zero. Omitting **fix_zero** causes **Orchestrate** to issue a conversion error when it encounters a decimal field containing all zeros. “Data Type Conversion Errors” on page 13-9 discusses conversion errors.
The suppress_zero argument specifies that the returned string value will have no leading or trailing zeros. Examples:

000.100 -> 0.1; 001.000 -> 1; -001.100 -> -1.1

Rounding Type

You can optionally specify a value for the rounding type (r_type) of many conversions. The values of r_type are:

- **ceil**: Round the source field toward positive infinity. This mode corresponds to the IEEE 754 Round Up mode.
  
  **Examples**: 1.4 -> 2, -1.6 -> -1

- **floor**: Round the source field toward negative infinity. This mode corresponds to the IEEE 754 Round Down mode.
  
  **Examples**: 1.6 -> 1, -1.4 -> -2

- **round_inf**: Round or truncate the source field toward the nearest representable value, breaking ties by rounding positive values toward positive infinity and negative values toward negative infinity. This mode corresponds to the COBOL ROUNDED mode.
  
  **Examples**: 1.4 -> 1, 1.5 -> 2, -1.4 -> -1, -1.5 -> -2

- **trunc_zero (default)**: Discard any fractional digits to the right of the right-most fractional digit supported in the destination, regardless of sign. For example, if the destination is an integer, all fractional digits are truncated. If the destination is another decimal with a smaller scale, round or truncate to the scale size of the destination decimal. This mode corresponds to the COBOL INTEGER-PART function.
  
  **Examples**: 1.6 -> 1, -1.6 -> -1

Figure 19 shows the conversion of a decimal field to a 32-bit integer with a rounding mode of ceil rather than the default mode of truncate to zero:
The syntax for this conversion is:

'field1 = int32_from_decimal[ceil,fix_zero](dField);'

where `fix_zero` ensures that a source decimal containing all zeros is treated as a valid representation.

**Raw Field Length Extraction**

Invoke the `modify` operator and the `raw_length` option to extract the length of a raw field. This specification returns an int32 containing the length of the raw field and optionally passes through the source field.

Figure 20 shows how to find the length of `aField` using the `modify` operator and the `raw_length` option:
Use the following `osh` commands to specify the `raw_length` conversion of a field:

```bash
$ modifySpec="field1 = raw_length(aField); field2 = aField;"
$ osh " ... | modify '$modifySpec' |... "
```

Notice that a shell variable (`modifySpec`) has been defined containing the specifications passed to the operator.

**Table 33 raw Field Functions**

<table>
<thead>
<tr>
<th>Conversion Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rawField = raw_from_string(string)</code></td>
<td>Returns string in raw representation.</td>
</tr>
<tr>
<td><code>rawField = u_raw_from_string(ustring)</code></td>
<td>Returns ustring in raw representation.</td>
</tr>
<tr>
<td><code>int32Field = raw_length(raw)</code></td>
<td>Returns the length of the raw field.</td>
</tr>
</tbody>
</table>

**String and Ustring Field Conversions**

Use the `modify` operator to perform the following modifications involving `string` and `ustring` fields:

- Extract the length of a string
- Convert long strings to shorter strings by string extraction
- Convert strings to and from numeric values using lookup tables (see “String Conversions and Lookup Tables” on page 13-23)
Table 34  **string and ustring Conversions Performed by the modify Operator**

<table>
<thead>
<tr>
<th>Conversion Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stringField=substring(string, starting_position, length)</code></td>
<td>Copies parts of strings and ustrings to shorter strings by string extraction. The starting_position specifies the starting location of the substring; length specifies the substring length. The arguments starting_position and length are uint16 types and must be positive (&gt;= 0).</td>
</tr>
<tr>
<td><code>ustringField=u_substring(ustring, starting_position, length)</code></td>
<td></td>
</tr>
<tr>
<td><code>stringField=lookup_string_from_int16[tableDefinition](int16Field)</code></td>
<td>Converts numeric values to strings and ustrings by means of a lookup table.</td>
</tr>
<tr>
<td><code>ustringField=lookup_ustring_from_int16[tableDefinition](int16Field)</code></td>
<td></td>
</tr>
<tr>
<td><code>int16Field=lookup_int16_from_string[tableDefinition](stringField)</code></td>
<td>Converts strings and ustrings to numeric values by means of a lookup table.</td>
</tr>
<tr>
<td><code>int16Field=lookup_int16_from_ustring[tableDefinition](ustringField)</code></td>
<td></td>
</tr>
<tr>
<td><code>uint32=lookup_uint32_from_string[tableDefinition](stringField)</code></td>
<td></td>
</tr>
<tr>
<td><code>uint32=lookup_uint32_from_ustring[tableDefinition](ustringField)</code></td>
<td></td>
</tr>
<tr>
<td><code>stringField=lookup_string_from_uint32[tableDefinition](uint32Field)</code></td>
<td>Converts numeric values to strings and ustrings by means of a lookup table.</td>
</tr>
<tr>
<td><code>ustringField=lookup_ustring_from_uint32[tableDefinition](uint32Field)</code></td>
<td></td>
</tr>
<tr>
<td><code>stringField=string_from_ustring(ustring)</code></td>
<td>Converts ustrings to strings.</td>
</tr>
<tr>
<td><code>ustringField=string_from_string(string)</code></td>
<td>Converts strings to ustrings.</td>
</tr>
<tr>
<td><code>decimalField=decimal_from_string(stringField)</code></td>
<td>Converts strings to decimals.</td>
</tr>
<tr>
<td><code>decimalField=decimal_from_ustring(ustringField)</code></td>
<td>Converts ustrings to decimals.</td>
</tr>
</tbody>
</table>
Table 34  **string and ustring Conversions Performed by the modify Operator**

<table>
<thead>
<tr>
<th>Conversion Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stringField = string_from_decimal[fix_zero] [suppress_zero] (decimalField)</td>
<td>Converts decimals to strings. fix_zero causes a decimal field containing all zeros to be treated as a valid zero. suppress_zero specifies that the returned ustring value will have no leading or trailing zeros. Examples: 000.100 -&gt; 0.1; 001.000 -&gt; 1; -001.100 -&gt; -1.1</td>
</tr>
<tr>
<td>ustringField = ustring_from_decimal[fix_zero] [suppress_zero] (decimalField)</td>
<td>Converts decimals to ustrings. See string_from_decimal above for a description of the fix_zero and suppress_zero arguments.</td>
</tr>
<tr>
<td>dateField = date_from_string[date_format</td>
<td>uformat] (stringField)</td>
</tr>
<tr>
<td>dateField = date_from_ustring[date_format</td>
<td>uformat] (ustringField)</td>
</tr>
<tr>
<td>stringField = string_from_date[date_format</td>
<td>uformat] (dateField)</td>
</tr>
<tr>
<td>ustringField = ustring_from_date[date_format</td>
<td>uformat] (dateField)</td>
</tr>
<tr>
<td>int32Field = string_length(stringField)</td>
<td>Returns an int32 containing the length of a string or ustring.</td>
</tr>
<tr>
<td>int32Field = ustring_length(ustringField)</td>
<td></td>
</tr>
<tr>
<td>stringField = substring [startPosition, len] (stringField)</td>
<td>Converts long strings/ustrings to shorter strings/ustrings by string extraction. The startPosition specifies the starting location of the substring; len specifies the substring length. If startPosition is positive, it specifies the byte offset into the string from the beginning of the string. If startPosition is negative, it specifies the byte offset from the end of the string.</td>
</tr>
<tr>
<td>ustringField = substring [startPosition, len] (ustringField)</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 13  The modify Operator  Performing Conversions

The following osh command converts a string field to lowercase:

```
osh "... | modify "lname=lowercase_string(lname)" | peek"
```

### Table 34  string and ustring Conversions Performed by the modify Operator

<table>
<thead>
<tr>
<th>Conversion Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stringField=uppercase_string (stringField)</code></td>
<td>Convert strings and ustrings to all upper case. Non-alphabetic characters are ignored in the conversion.</td>
</tr>
<tr>
<td><code>stringField=lowercase_string (stringField)</code></td>
<td>Convert strings and ustrings to all lower case. Non-alphabetic characters are ignored in the conversion.</td>
</tr>
<tr>
<td>`stringField = string_from_time [time_format</td>
<td>uformat] (timeField)`</td>
</tr>
<tr>
<td>`stringField = string_from_timestamp [timestamp_format</td>
<td>uformat] (tsField)`</td>
</tr>
<tr>
<td>`tsField = timestamp_from_string [timestamp_format</td>
<td>uformat] (stringField)`</td>
</tr>
<tr>
<td><code>timeField = time_from_string [time_format] (stringField)</code></td>
<td>string and ustring from time&lt;br&gt;Converts the time to a string or ustring representation using the specified time_format.&lt;br&gt;The time_format options are described below.</td>
</tr>
</tbody>
</table>
Figure 21 shows a modification that converts the name of aField to field1 and produces field2 from bField by extracting the first eight bytes of bField:

Figure 21  Data Flow Diagram Showing Substring Extraction

The following osh command performs the substring extraction:

```
modify 'field1 = aField; field2 = substring[0,8](bField);'
```

Figure 22 shows the extraction of the string_length of aField. The length is included in the output as field1.

Figure 22  Extracting the Length of a String

The following osh commands extract the length of the string in aField and place it in field1 of the output:

```
$ modifySpec="field1 = string_length(aField); field2 = aField;"
$ osh "... | modify "$modifySpec" |...
```
Notice that a shell variable (modifySpec) has been defined containing the specifications passed to the operator.

**String Conversions and Lookup Tables**

You can construct a string lookup table to use when default conversions do not yield satisfactory results. A string lookup table is a table of two columns and as many rows as are required to perform a conversion to or from a string as shown in Table 35.

Table 35 **A String Lookup Table**

<table>
<thead>
<tr>
<th>Numeric Value</th>
<th>String or Ustring</th>
</tr>
</thead>
<tbody>
<tr>
<td>numVal1</td>
<td>string1</td>
</tr>
<tr>
<td>numVal2</td>
<td>string2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>numVal3</td>
<td>stringN</td>
</tr>
</tbody>
</table>

Each row of the lookup table specifies an association between a 16-bit integer or unsigned 32-bit integer value and a string or ustring. Orchestrate scans the **Numeric Value** or the **String or Ustring** column until it encounters the value or string to be translated. The output is the corresponding entry in the row.

The numeric value to be converted may be of the int16 or the uint32 data type. Orchestrate converts strings to values of the int16 or uint32 data type using the same table.

If the input contains a numeric value or string that is not listed in the table, Orchestrate operates as follows:

- If a numeric value is unknown, an empty string is returned by default. However, you can set a default string value to be returned by the string lookup table.
- If a string has no corresponding value, 0 is returned by default. However, you can set a default numeric value to be returned by the string lookup table.

Here are the options and arguments passed to the **modify** operator to create a lookup table:

```plaintext
intField = lookup_int16_from_string[ tableDefinition ]( source_stringField ); | intField = lookup_int16_from_ustring[ tableDefinition ]( source_ustringField );
```

OR:

```plaintext
intField = lookup_uint32_from_string[ tableDefinition ]( source_stringField ); | intField = lookup_uint32_from_ustring[ tableDefinition ]( source_ustringField );
```
stringField = lookup_string_from_int16(tableDefinition)(source_intField); |
ustringField = lookup_ustring_from_int16(tableDefinition)(source_intField);
OR:
stringField = lookup_string_from_uint32(tableDefinition)(source_intField);
ustringField = lookup_ustring_from_uint32(tableDefinition)(source_intField);

where:

tableDefinition defines the rows of a string or ustring lookup table and has the
following form:

{propertyList} ('string' | 'ustring' = value, 'string' | 'ustring' = value, ... )

where:

• propertyList is one or more of the following options; the entire list is
  enclosed in braces and properties are separated by commas if there are
  more than one:
    • case_sensitive: perform a case-sensitive search for matching strings;
      the default is case-insensitive.
    • default_value = defVal: the default numeric value returned for a
      string that does not match any of the strings in the table.
    • default_string = defString: the default string returned for numeric
      values that do not match any numeric value in the table.
    • string or ustring specifies a comma-separated list of strings or ustrings
      associated with value; enclose each string or ustring in quotes.
    • value specifies a comma-separated list of 16-bit integer values associated
      with string or ustring.
Figure 23 shows an operator and data set requiring type conversion:

Whereas gender is defined as a string in the input data set, the SampleOperator defines the field as an 8-bit integer. The default conversion operation cannot work in this case, because by default Orchestrate converts a string to a numeric representation and gender does not contain the character representation of a number. Instead the gender field contains the string values "male", "female", "m", or "f". You must therefore specify a string lookup table to perform the modification.

The gender lookup table required by the example shown above is as follows:

Table 36  Lookup Table for gender Field

<table>
<thead>
<tr>
<th>Numeric Value</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;f&quot;</td>
</tr>
<tr>
<td>0</td>
<td>&quot;female&quot;</td>
</tr>
<tr>
<td>1</td>
<td>&quot;m&quot;</td>
</tr>
<tr>
<td>1</td>
<td>&quot;male&quot;</td>
</tr>
</tbody>
</table>

The value "female" is translated to a numeric value of 0; the value "male" is translated to a numeric value of 1.
The following osh code performs the conversion:

```osh
modify 'gender = lookup_int16_from_string[default_value = 2]
    {'f' = 0; 'female' = 0; 'm' = 1; 'male' = 1;} (gender);
```

In this example, `gender` is the name of both the source and the destination fields of the translation. In addition, the string lookup table defines a default value of 2; if `gender` contains a string that is not one of "f", "female", "m", or "male", the lookup table returns a value of 2.

## Time Field Conversions

Orchestrate performs no automatic conversions to or from the time data type. You must invoke the `modify` operator if you want to convert a source or destination time field. Most time field conversions extract a portion of the time, such as hours or minutes, and write it into a destination field.

### Table 37  time Type Conversions Provided by modify

<table>
<thead>
<tr>
<th>Conversion Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int8Field = hours_from_time(timeField)</code></td>
<td>hours from time</td>
</tr>
<tr>
<td><code>int32Field = microseconds_from_time(timeField)</code></td>
<td>microseconds from time</td>
</tr>
<tr>
<td><code>int8Field = minutes_from_time(timeField)</code></td>
<td>minutes from time</td>
</tr>
<tr>
<td><code>dfloatField = seconds_from_time(timeField)</code></td>
<td>seconds from time</td>
</tr>
<tr>
<td><code>dfloatField = midnight_seconds_from_time(timeField)</code></td>
<td>seconds-from-midnight from time</td>
</tr>
<tr>
<td><code>stringField = string_from_time[time_format](timeField)</code></td>
<td>string and ustring from time</td>
</tr>
<tr>
<td><code>ustringField = ustring_from_time[time_format](timeField)</code></td>
<td>Converts the time to a string or ustring</td>
</tr>
<tr>
<td></td>
<td>representation using the specified</td>
</tr>
<tr>
<td></td>
<td>time_format. The time_format options are</td>
</tr>
<tr>
<td></td>
<td>described below.</td>
</tr>
<tr>
<td><code>timeField = time_from_midnight_seconds(dfloatField)</code></td>
<td>time from seconds-from-midnight</td>
</tr>
<tr>
<td><code>timeField = time_from_string[time_format](stringField)</code></td>
<td>time from string</td>
</tr>
<tr>
<td><code>timeField = time_from_ustring[time_format](ustringField)</code></td>
<td>Converts the string or ustring to a time</td>
</tr>
<tr>
<td></td>
<td>representation using the specified</td>
</tr>
<tr>
<td></td>
<td>time_format. The time_format options are</td>
</tr>
<tr>
<td></td>
<td>described below.</td>
</tr>
</tbody>
</table>


Time conversion to a numeric field can be used with any Orchestrate numeric data type. Orchestrate performs the necessary modifications to translate a conversion result to the numeric data type of the destination. For example, you can use the conversion `hours_from_time` to convert a time to an int8, or to an int16, int32, dfloat, and so on.

**Time Formats**

Four conversions, `string_from_time`, `ustring_from_time`, `time_from_string`, and `ustring_from_time`, take as a parameter of the conversion a time format or a uformat time format. These formats are described below. The default format of the time contained in the string is `hh:mm:ss`.

**uformat time Format**

The uformat date format provides support for international components in time fields. It’s syntax is:

```
String%macroString%macroString%macroString
```

where `%macro` is a time formatting macro such as `%hh` for a two-digit hour. See “time Format” below for a description of the date format macros. Only the String components of uformat can include multi-byte Unicode characters.

**time Format**

The `string_from_time` and `time_from_string` conversions take a format as a parameter of the conversion. The default format of the time in the string is `hh:mm:ss`. However, you can specify an optional format string defining the time format of the string field. The format string must contain a specification for hours, minutes, and seconds.

The time components of a source string (hours, minutes, and seconds) must be zero padded to the character length specified by the format string; Orchestrate zero pads the time components of a destination string to the specified length.
The components of the format string are:

- `%hh`: A two-digit hours component.
- `%mm`: A two-digit minutes component.
- `%ss`: A two-digit seconds component.
- `%ss.n`: A two-digit seconds component plus a fractional part, where \( n \) is the number of fractional digits with a maximum value of 6. If \( n \) is 0, no decimal point is printed as part of the seconds component. Trailing zeros are not suppressed. For example, a format string of `%hh:%mm:%ss.4` indicates that the string contains the seconds to four decimal places.

The default time format is as follows:

```
%hh:%mm:%ss
```

When you specify a time format string, prefix each component with the percent symbol (`%`). Separate the string's components with any character except the percent sign (`%`).

**Converting Time Fields to Integers Example**

The following figure shows the conversion of time field to two 8-bit integers, where:

- The `hours_from_time` conversion specification extracts the hours portion of a time field and writes it to an 8-bit integer.
- The `minutes_from_time` conversion specification extracts the minutes portion of a time field and writes it to an 8-bit integer.

```
input data set schema:
tField:time;

hoursField = hours_from_time(tField);
minField = minutes_from_time(tField);

modify

output data set schema:
hoursField:int8;
minField:int8;
```

Figure 24  **Converting a date**
Chapter 13 The modify Operator

Performing Conversions

The following osh code converts the hours portion of tField to the int8 hoursField and the minutes portion to the int8 minField:

```osh
modify 'hoursField = hours_from_time(tField);
  minField = minutes_from_time(tField);'
```

**Timestamp Field Conversions**

By default Orchestrate converts a source timestamp field only to either a time or date destination field. However, you can invoke the `modify` operator to perform other conversions.

Table 38 timestamp Field Conversions Provided by modify

<table>
<thead>
<tr>
<th>Conversion Specification</th>
<th>Description</th>
</tr>
</thead>
</table>
| `dfloatField =
  seconds_since_from_timestamp
  [timestamp](tsField)` | seconds_since from timestamp |
| `tsField =
  timestamp_from_seconds_since
  [timestamp](dfloatField)` | timestamp from seconds_since |
| `stringField =
  string_from_timestamp
  [timestamp_format | uformat](tsField)` | strings and ustrings from timestamp |
| `ustringField =
  ustring_from_timestamp
  [timestamp_format | uformat](tsField)` | Converts the timestamp to a string or ustring representation using the specified timestamp_format or uformat. By default, the string format is %Y %m %d %H:%M:%S. The timestamp_format and uformat options are described in “timestamp Formats” on page 13-30. |
| `int32Field =
  timet_from_timestamp
  (tsField)` | time_t from timestamp |
| `dateField =
  date_from_timestamp(tsField)` | int32Field contains a timestamp as defined by the UNIX time_t representation. |
| `tsField =
  timestamp_from_string
  [timestamp_format | uformat](stringField)` | date from timestamp |
| `tsField =
  timestamp_from_ustring
  [timestamp_format | uformat](ustringField)` | Converts the timestamp to a date representation. |
| `tsField =
  timestamp_from_timet
  (int32Field)` | timestamp from strings and ustrings |
| `tsField =
  timestamp_from_timet
  (int32Field)` | Converts the string or ustring to a timestamp representation using the specified timestamp_format. By default, the string format is %Y %m %d %H:%M:%S. The timestamp_format and uformat options are described in “timestamp Formats” on page 13-30. |
| `tsField =
  timestamp_from_timet
  (int32Field)` | timestamp from time_t |
| `tsField =
  timestamp_from_timet
  (int32Field)` | int32Field must contain a timestamp as defined by the UNIX time_t representation. |
Chapter 13  The modify Operator  Performing Conversions

Table 38  timestamp Field Conversions Provided by modify (continued)

<table>
<thead>
<tr>
<th>Conversion Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsField = timestamp_from_date [time] dateField</td>
<td>timestamp from date</td>
</tr>
<tr>
<td></td>
<td>The time argument optionally specifies the time to be used in building the</td>
</tr>
<tr>
<td></td>
<td>timestamp result and must be in the form hh:mm:ss.</td>
</tr>
<tr>
<td></td>
<td>If omitted, the time defaults to midnight.</td>
</tr>
<tr>
<td>tsField = timestamp_from_time [date] timeField</td>
<td>timestamp from time</td>
</tr>
<tr>
<td></td>
<td>The date argument is required. It specifies the date portion of the</td>
</tr>
<tr>
<td></td>
<td>timestamp and must be in the form</td>
</tr>
<tr>
<td></td>
<td>yyyy-mm-dd.</td>
</tr>
<tr>
<td>tsField = timestamp_from_date_time (date, time)</td>
<td>Returns a timestamp from date and time. The date</td>
</tr>
<tr>
<td></td>
<td>specifies the date portion (yyyy-mm-dd) of the</td>
</tr>
<tr>
<td></td>
<td>timestamp. The time argument specifies the time to</td>
</tr>
<tr>
<td></td>
<td>be used when building the timestamp. The time</td>
</tr>
<tr>
<td></td>
<td>argument must be in the hh:mm:ss format.</td>
</tr>
<tr>
<td>timeField = time_from_timestamp(tsField)</td>
<td>time from timestamp</td>
</tr>
</tbody>
</table>

Timestamp conversion of a numeric field can be used with any Orchestrate numeric data type. Orchestrate performs the necessary conversions to translate a conversion result to the numeric data type of the destination. For example, you can use the conversion `timet_from_timestamp` to convert a timestamp to an int32, dfloat, and so on.

**timestamp Formats**

The `string_from_timestamp`, `ustring_from_timestamp`, `timestamp_from_string`, and `timestamp_from_ustring` conversions take a `timestamp_format` or `uformat` argument. The default format of the timestamp contained in the string is `yyyymm-dd hh:mm:ss`. However, you can specify an optional format string defining the data format of the string field.

**timestamp Format**

The timestamp components (month, day, year, hour, minutes, and seconds) of a source string, must be zero padded to the character length specified by the format string; Orchestrate zero pads the timestamp components of a destination string to the specified length.

The format options of timestamp combine the formats of the date and time data types. The default timestamp format is as follows:

%yyyy  %mm  %dd  %h  %m  %s
**uformat timestamp Format**

For uformat format, concatenate the uformat date format with the uformat time format. The two formats can be in any order, but their components cannot be mixed. These formats are described in “uformat date Format” on page 13-13 and “uformat time Format” on page 13-27.

See “Date Field Conversions” on page 13-10 and “Time Field Conversions” on page 13-26 for more information on these options.

**Figure 25** shows the conversion of a date field to a timestamp field. As part of the conversion, the operator sets the time portion of the timestamp to 10:00:00.

```plaintext
input data set schema:
dField:date;

modify 'tsField=timestamp_from_date[10:00:00](dField);

output data set schema:
tsField:timestamp;
```

**The modify Operator and Nulls**

All Orchestrate data types support nulls. As part of processing a record, an operator can detect a null and take the appropriate action, for example, it can omit the null field from a calculation or signal an error condition.

Orchestrate represents nulls in two ways.

- It allocates a single bit to mark a field as null. This type of representation is called an out-of-band null.
- It designates a specific field value to indicate a null, for example a numeric field’s most negative possible value. This type of representation is called an in-band null. In-band null representation can be disadvantageous because you must reserve a field value for nulls and this value cannot be treated as valid data elsewhere.
The **modify** operator can change a null representation from an out-of-band null to an in-band null and from an in-band null to an out-of-band null.

The record schema of an operator’s input or output data set can contain fields defined to support out-of-band nulls. In addition, fields of an operator’s interface may also be defined to support out-of-band nulls. The next table lists the rules for handling nullable fields when an operator takes a data set as input or writes to a data set as output.

<table>
<thead>
<tr>
<th>Source Field</th>
<th>Destination Field</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>not_nullable</td>
<td>not_nullable</td>
<td>Source value propagates to destination.</td>
</tr>
<tr>
<td>not_nullable</td>
<td>nullable</td>
<td>Source value propagates; destination value is never null.</td>
</tr>
<tr>
<td>nullable</td>
<td>not_nullable</td>
<td>If the source value is not null, the source value propagates.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the source value is null, a fatal error occurs, unless you apply the <strong>modify</strong> operator, as in “Out-of-Band to Normal Representation” on page 13-32.</td>
</tr>
<tr>
<td>nullable</td>
<td>nullable</td>
<td>Source value or null propagates.</td>
</tr>
</tbody>
</table>

See the chapter on Orchestrate data sets in the Orchestrate 7.0 User Guide for more information on Orchestrate null representation.

**Out-of-Band to Normal Representation**

The **modify** operator can change a field’s null representation from a single bit to a value you choose, that is, from an out-of-band to an in-band representation. Use this feature to prevent fatal data type conversion errors that occur when a destination field has not been defined as supporting nulls. See “Data Type Conversion Errors” on page 13-9.

To change a field’s null representation from a single bit to a value you choose, use the following **osh** syntax:

```
destField[:dataType] = handle_null(sourceField, value)
```

where:

- `destField` is the destination field’s name.
- `dataType` is its optional data type; use it if you are also converting types.
• sourceField is the source field's name
• value is the value you wish to represent a null in the output. The destField is converted from an Orchestrate out-of-band null to a value of the field's data type. For a numeric field value can be a numeric value, for decimal, string, time, date, and timestamp fields, value can be a string.

Conversion specifications are described in:
- “Date Field Conversions” on page 13-10
- “Decimal Field Conversions” on page 13-14
- “String and Ustring Field Conversions” on page 13-18
- “Time Field Conversions” on page 13-26
- “Timestamp Field Conversions” on page 13-29

For example, Figure 26 shows the modify operator converting Orchestrate's out-of-band null representation in the input to an output value that is written when a null is encountered:

![Diagram of modify operator conversion](image)

Figure 26 Converting an Out-of-Band Null to an In-Band Null

While in the input fields a null takes Orchestrate's out-of-band representation, in the output a null in aField is represented by -128 and a null in bField is represented by ASCII XXXX (0x59 in all bytes).

To make the output aField contain a value of -128 whenever the input contains an out-of-band null, and the output bField contain a value of 'XXXX' whenever the input contains an out-of-band null, use the following osh code:

```osh
$ modifySpec = 'aField = handle_null(aField, -128);
   bField = handle_null(bField, 'XXXX'); 
$ osh ' ... | modify $modifySpec | ... '
```
Notice that a shell variable (modifySpec) has been defined containing the specifications passed to the operator.

**Normal to Out-of-Band Representation**

The `modify` operator can change a field’s null representation from a normal field value to a single bit, that is, from an in-band to an out-of-band representation.

To change a field’s null representation to out-of-band use the following `osh` syntax:

```
destField[:dataType] = make_null(sourceField, value);
```

Where:

- `destField` is the destination field’s name.
- `dataType` is its optional data type; use it if you are also converting types.
- `sourceField` is the source field’s name.
- `value` is the value of the source field when it is null.

A conversion result of `value` is converted from an Orchestrate out-of-band null to a value of the field’s data type. For a numeric field `value` can be a numeric value, for decimal, string, time, date, and timestamp fields, `value` can be a string.

For example, Figure 27 shows a `modify` operator converting the value representing a null in an input field (-128 or ‘XXXX’) to Orchestrate’s single-bit null representation in the corresponding field of the output data set:

```
input data set schema:
 aField:int8;
bField:string[4];

field1 = make_null(aField, -128);
field2 = make_null(bField, 'XXXX');

modify

output data set schema:
 field1:nullable int8;
 field2:nullable string[4];
```

Figure 27 **Converting to Out-of-Band Null Representation**

In the input a null value in `aField` is represented by -128 and a null value in `bField` is represented by ASCII XXXX, but in both output fields a null value if represented by Orchestrate’s single bit.
The following osh syntax causes the aField of the output data set to be set to Orchestrate’s single-bit null representation if the corresponding input field contains -128 (in-band-null), and the bField of the output to be set to Orchestrate’s single-bit null representation if the corresponding input field contains 'XXXX' (in-band-null).

```bash
$modifySpec = 'aField = make_null(aField, -128); 
  bField = make_null(bField, 'XXXX'); 
$ osh ' ... | modify '$modifySpec' | ... ' 
```

Notice that a shell variable (modifySpec) has been defined containing the specifications passed to the operator.

### The null and notnull Conversions

Orchestrate supplies two other conversions to use with nullable fields, called null and notnull.

- The null conversion sets the destination field to 1 if the source field is null and to 0 otherwise.
- The notnull conversion sets the destination field to 1 if the source field is not null and to 0 if it is null.

In osh, define a null or notnull conversion as follows:

```bash
destField[:dataType] = null(sourceField); 
destField[:dataType] = notnull(sourceField); 
```

By default, the data type of the destination field is int8. Specify a different destination data type to override this default. Orchestrate issues a warning if the source field is not nullable or the destination field is nullable.

### The modify Operator and Partial Schemas

You can invoke a modify operator to change certain characteristics of a data set containing a partial record schema. ("Complete and Partial Schemas" on page 227 discusses partial schemas and their definition.) When the modify operator drops a field from the intact portion of the record, it drops only the field definition. The contents of the intact record are not altered. Dropping the definition means you can no longer access that portion of the intact record.

### The modify Operator and Vectors

The modify operator cannot change the length of a vector or the vector’s length type (fixed or variable). However, you can use the operator either to translate the name of a vector or to convert the data type of the vector elements.
The modify Operator and Aggregate Schema Components

Data set and operator interface schema components can contain aggregates (subrecords and tagged aggregates). You can apply modify adapters to aggregates, with these restrictions:

- Subrecords may be translated only to subrecords.
- Tagged fields may be translated only to tagged fields.
- Within subrecords and tagged aggregates, only elements of the same level can be bound by the operator.

Figure 28 shows an operation in which both the input data set and the output contain aggregates:

```plaintext
input data set schema:
 fName:string;
 lName:string;
 purchase:subrec(
   itemNum:int32;
   price:sfloat;
 )
 date:tagged(
   intDate:int32;
   stringDate:string;
 )

subField:subrec(
  subF1:int32;
  subF2:sfloat;
 )
tagField:tagged(
  tagF1:int32;
  tagF2:string;
 )

subField.subF1 = purchase.itemNum
subField.subF2 = purchase.price
tagField.tagF1 = date.intDate
tagField.tagF2 = date.stringDate

modify

output data set schema:
 fName:string;
 lName:string;
 subField:subrec(
   subF1:int32;
   subF2:sfloat;
 )
tagField:tagged(
   tagF1:int32;
   tagF2:string;
 )
```

Figure 28  Using Aggregates with the modify Operator

In this example, purchase contains an item number and a price for a purchased item; date contains the date of purchase represented as either an integer or a string. You must translate the aggregate purchase to the interface component subField and the tagged component date to tagField.
**To translate aggregates:**

1. Translate the aggregate of an input data set to an aggregate of the output.  
   
   To translate purchase, the corresponding output component must be a compatible aggregate type. The type is subField and the component is subField. The same principle applies to the elements of the subrecord.

2. Translate the individual fields of the data set’s aggregate to the individual fields of the operator’s aggregate.

   If multiple elements of a tagged aggregate in the input are translated, they must all be bound to members of a single tagged component of the output’s record schema. That is, all elements of tagField must be bound to a single aggregate in the input.

   Here is the osh code to rename `purchase.price` to `subField.subF2`.

   ```
   $ modifySpec = 'subField = purchase;  
   subField.subF1 = purchase.itemNum;  
   subField.subF2 = purchase.price;  
   tagField = date;  
   tagField.tagF1 = date.intDate;  
   tagField.tagF2 = date.stringDate; '); 
   $ osh '... | modify "$modifySpec" | ...'
   ```

   Notice that a shell variable (modifySpec) has been defined containing the specifications passed to the operator.

   Aggregates may contain nested aggregates. When you translate nested aggregates, all components at one nesting level in an input aggregate must be bound to all components at one level in an output aggregate.

   Table 40 shows sample input and output data sets containing nested aggregates. In the input data set, the record `purchase` contains a subrecord `description` containing a description of the item:

<table>
<thead>
<tr>
<th>Level</th>
<th>Schema 1 (for input data set)</th>
<th>Level</th>
<th>Schema 2 (for output data set)</th>
</tr>
</thead>
</table>
   | 0     | `purchase: subrec (
   | 1     | `itemNum: int32;
   | 1     | `price: sfloat;
   | 1     | `description: subrec; {  
   | 2     | `color: int32;
   | 2     | `size: int8; });`  
   |       |                               |       |                               |
   | n     | `subField (  
   | n+1   | `subF1;
   | n+1   | `subF2: );`  
   |       |                               |       |                               |

   Note that:
   
   - `itemNum` and `price` are at the same nesting level in `purchase`. 
• color and size are at the same nesting level in purchase.
• subF1 and subF2 are at the same nesting level in subField.

You can bind:
• purchase.itemNum and purchase.price (both level 1) to subField.subF1 and subField.subF2, respectively
• purchase.description.color and purchase.description.size (both level 2) to subField.subF1 and subField.subF2, respectively

You cannot bind two elements of purchase at different nesting levels to subF1 and subF2. Therefore, you cannot bind itemNum (level 1) to subF1 and size (level 2) to subF2.

Note Orchestrate features several operators that modify the record schema of the input data set and the level of fields within records. Two of them act on tagged subrecords. See the chapter on restructure operators.

Allowed Conversions

Table 41 lists all allowed data type conversions arranged alphabetically. The form of each listing is:

conversion_name(source_type, destination_type)

Table 41 Conversions Provided by the modify Operator

<table>
<thead>
<tr>
<th>Conversion Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>date_from_days_since (int32, date)</td>
</tr>
<tr>
<td>date_from_julian_day (uint32, date)</td>
</tr>
<tr>
<td>date_from_string (string, date)</td>
</tr>
<tr>
<td>date_from_timestamp (timestamp, date)</td>
</tr>
<tr>
<td>date_from_ustring (ustring, date)</td>
</tr>
<tr>
<td>days_since_from_date (date, int32)</td>
</tr>
<tr>
<td>decimal_from_decimal (decimal, decimal)</td>
</tr>
<tr>
<td>decimal_from_dfloat (dfloat, decimal)</td>
</tr>
<tr>
<td>decimal_from_string (string, decimal)</td>
</tr>
<tr>
<td>decimal_from_ustring (ustring, decimal)</td>
</tr>
</tbody>
</table>
### Table 41 Conversions Provided by the modify Operator (continued)

<table>
<thead>
<tr>
<th>Conversion Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>dfloat_from_decimal(decimal, dfloat)</td>
</tr>
<tr>
<td>hours_from_time(time, int8)</td>
</tr>
<tr>
<td>int32_from_decimal(decimal, int32)</td>
</tr>
<tr>
<td>int64_from_decimal(decimal, int64)</td>
</tr>
<tr>
<td>julian_day_from_date(date, uint32)</td>
</tr>
<tr>
<td>lookup_int16_from_string(string, int16)</td>
</tr>
<tr>
<td>lookup_int16_from_ustring(ustring, int16)</td>
</tr>
<tr>
<td>lookup_string_from_int16(int16, string)</td>
</tr>
<tr>
<td>lookup_string_from_uint32(uint32, string)</td>
</tr>
<tr>
<td>lookup_uint32_from_string(string, uint32)</td>
</tr>
<tr>
<td>lookup_uint32_from_ustring(ustring, uint32)</td>
</tr>
<tr>
<td>lookup_ustring_from_int16(int16, ustring)</td>
</tr>
<tr>
<td>lookup_ustring_from_int32(int32, ustring)</td>
</tr>
<tr>
<td>lowercase_string(string, string)</td>
</tr>
<tr>
<td>lowercase_ustring(ustring, ustring)</td>
</tr>
<tr>
<td>mantissa_from_dfloat(dfloat, dfloat)</td>
</tr>
<tr>
<td>mantissa_from_decimal(decimal, dfloat)</td>
</tr>
<tr>
<td>microseconds_from_time(time, int32)</td>
</tr>
<tr>
<td>midnight_seconds_from_time(time, dfloat)</td>
</tr>
<tr>
<td>minutes_from_time(time, int8)</td>
</tr>
<tr>
<td>month_day_from_date(date, int8)</td>
</tr>
<tr>
<td>month_from_date(date, int8)</td>
</tr>
<tr>
<td>next_weekday_from_date(date, date)</td>
</tr>
<tr>
<td>notnull(any, int8)</td>
</tr>
<tr>
<td>null(any, int8)</td>
</tr>
<tr>
<td>previous_weekday_from_date(date, date)</td>
</tr>
</tbody>
</table>
Table 41  **Conversions Provided by the modify Operator (continued)**

<table>
<thead>
<tr>
<th>Conversion Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>raw_from_string (string, raw)</code></td>
</tr>
<tr>
<td><code>raw_length (raw, int32)</code></td>
</tr>
<tr>
<td><code>seconds_from_time (time, dfloat)</code></td>
</tr>
<tr>
<td><code>seconds_since_from_timestamp (timestamp, dfloat)</code></td>
</tr>
<tr>
<td><code>string_from_date (date, string)</code></td>
</tr>
<tr>
<td><code>string_from_decimal (decimal, string)</code></td>
</tr>
<tr>
<td><code>string_from_time (time, string)</code></td>
</tr>
<tr>
<td><code>string_from_timestamp (timestamp, string)</code></td>
</tr>
<tr>
<td><code>string_from_ustring (ustring, string)</code></td>
</tr>
<tr>
<td><code>string_length (string, int32)</code></td>
</tr>
<tr>
<td><code>substring (string, string)</code></td>
</tr>
<tr>
<td><code>time_from_midnight_seconds (dfloat, time)</code></td>
</tr>
<tr>
<td><code>time_from_string (string, time)</code></td>
</tr>
<tr>
<td><code>time_from_timestamp (timestamp, time)</code></td>
</tr>
<tr>
<td><code>time_from_ustring (ustring, time)</code></td>
</tr>
<tr>
<td><code>timestamp_from_date (date, timestamp)</code></td>
</tr>
<tr>
<td><code>timestamp_from_seconds_since (dfloat, timestamp)</code></td>
</tr>
<tr>
<td><code>timestamp_from_string (string, timestamp)</code></td>
</tr>
<tr>
<td><code>timestamp_from_time (time, timestamp)</code></td>
</tr>
<tr>
<td><code>timestamp_from_datetime (int32, timestamp)</code></td>
</tr>
<tr>
<td><code>timestamp_from_ustring (ustring, timestamp)</code></td>
</tr>
<tr>
<td><code>timet_from_timestamp (timestamp, int32)</code></td>
</tr>
<tr>
<td><code>uint64_from_decimal (decimal, uint64)</code></td>
</tr>
<tr>
<td><code>uppercase_string (string, string)</code></td>
</tr>
<tr>
<td><code>uppercase_ustring (ustring, ustring)</code></td>
</tr>
<tr>
<td><code>u_raw_from_string (ustring, raw)</code></td>
</tr>
<tr>
<td><code>ustring_from_date (date, ustring)</code></td>
</tr>
</tbody>
</table>


Table 41 Conversions Provided by the modify Operator (continued)

<table>
<thead>
<tr>
<th>Conversion Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>ustring_from_decimal (decimal, ustring)</td>
</tr>
<tr>
<td>ustring_from_string (string, ustring)</td>
</tr>
<tr>
<td>ustring_from_time (time, ustring)</td>
</tr>
<tr>
<td>ustring_from_timestamp (timestamp, ustring)</td>
</tr>
<tr>
<td>ustring_length (ustring, int32)</td>
</tr>
<tr>
<td>u_substring (ustring, ustring)</td>
</tr>
<tr>
<td>weekday_from_date (date, int8)</td>
</tr>
<tr>
<td>year_day_from_date (date, int16)</td>
</tr>
<tr>
<td>year_from_date (date, int16)</td>
</tr>
<tr>
<td>year_week_from_date (date, int8)</td>
</tr>
</tbody>
</table>
The pcompress Operator

The pcompress operator can convert an Orchestrate data set from a sequence of records into a stream of compressed binary data and can reconvert the data stream into an Orchestrate data set.

The pcompress operator uses the UNIX compress utility to compress or expand a data set. The operator converts an Orchestrate data set from a sequence of records into a stream of raw binary data; conversely, the operator reconverts the data stream into an Orchestrate data set.
The mode of the `pcompress` operator determines its action. Possible values for the mode are:

- **compress**: compress the input data set
- **expand**: expand the input data set

### Properties

Table 42  **pcompress Operator Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>mode = compress: in:*;</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>mode = expand: out:*;</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>in -&gt; out without record modification for a compress/ decompress cycle</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>mode = compress: any</td>
</tr>
<tr>
<td></td>
<td>mode = expand: same</td>
</tr>
<tr>
<td>Collection method</td>
<td>any</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>mode = compress: sets</td>
</tr>
<tr>
<td></td>
<td>mode = expand: propagates</td>
</tr>
<tr>
<td>Composite operator</td>
<td>yes</td>
</tr>
</tbody>
</table>
Syntax and Options

\texttt{pcompress [-compress | -expand] [-command compress | gzip]}

Table 43 \textit{pcompress} Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-compress</td>
<td>\texttt{-compress}</td>
</tr>
<tr>
<td></td>
<td>This option is the default mode of the operator. The operator takes a data set as input and produces a compressed version as output.</td>
</tr>
<tr>
<td>-expand</td>
<td>\texttt{-expand}</td>
</tr>
<tr>
<td></td>
<td>This option puts the operator in expand mode. The operator takes a compressed data set as input and produces an uncompressed data set as output.</td>
</tr>
<tr>
<td>-command</td>
<td>\texttt{-command &quot;compress&quot;</td>
</tr>
<tr>
<td></td>
<td>Optionally specifies the UNIX command to be used to perform the compression or expansion. When you specify \texttt{&quot;compress&quot;} the operator uses the UNIX command, \texttt{compress -f}, for compression and the UNIX command, \texttt{uncompress}, for expansion. When you specify \texttt{&quot;gzip&quot;}, the operator uses the UNIX command, \texttt{gzip -d}, for compression and the UNIX command, \texttt{gzip -d}, for expansion.</td>
</tr>
</tbody>
</table>

The default mode of the operator is \texttt{-compress}, which takes a data set as input and produces a compressed version as output. Specifying \texttt{-expand} puts the command in expand mode, which takes a compressed data set as input and produces an uncompressed data set as output.

Compressed Data Sets

Each record of an Orchestrate data set has defined boundaries that mark its beginning and end. The \texttt{pcompress} operator invokes the UNIX \texttt{compress} utility to change an Orchestrate data set, which is in record format, into raw binary data and vice versa.

Processing Compressed Data Sets

A compressed data set is similar to an Orchestrate data set. A compressed, persistent data set is represented on disk in the same way as a normal data set, by two or more files: a single descriptor file and one or more data files.
A compressed data set cannot be accessed like a standard Orchestrate data set. See the chapter on Orchestrate data sets in the Orchestrate 7.0 User Guide for more information on persistent data sets.

A compressed data set cannot be processed by most Orchestrate operators until it is decoded, that is, until its records are returned to their normal Orchestrate format.

Nonetheless, you can specify a compressed data set to any operator that does not perform field-based processing or reorder the records. For example, you can invoke the copy operator to create a copy of the compressed data set.

You can further encode a compressed data set, using an encoding operator, to create a compressed-encoded data set. (See Chapter 6, “The encode Operator”.) You would then restore the data set by first decoding and then decompressing it.

**Compressed Data Sets and Partitioning**

When you compress a data set, you remove its normal record boundaries. The compressed data set must not be repartitioned before it is expanded, because partitioning in Orchestrate is performed record-by-record. For that reason, the pcompress operator sets the preserve-partitioning flag in the output data set. This prevents an Orchestrate operator that uses a partitioning method of any from repartitioning the data set to optimize performance and causes Orchestrate to issue a warning if any operator attempts to repartition the data set.

For an expand operation, the operator takes as input a previously compressed data set. If the preserve-partitioning flag in this data set is not set, Orchestrate issues a warning message.

See the chapter on partitioning in Orchestrate in the Orchestrate 7.0 User Guide for more information on the preserve-partitioning flag.

**Using orchadmin with a Compressed Data Set**

The orchadmin utility manipulates persistent data sets. However, the records of a compressed data set are not in the normal form. For that reason, you can invoke only a subset of the orchadmin commands to manipulate a compressed data set. These commands are as follows:

- **delete** to delete a compressed data set
- **copy** to copy a compressed data set
- **describe** to display information about the data set before compression

The Orchestrate 7.0 Installation and Administration Manual chapter on maintenance and monitoring tools discusses orchadmin in detail. Also see Chapter 6, “The encode Operator”.
Example

This example consists of two steps. In the first step, the \texttt{pcompress} operator compresses the output of the upstream operator before it is stored on disk:

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure32.png}
\caption{Compressing a Data Set with \texttt{pcompress}}
\end{figure}

In \texttt{osh}, the default mode of the operator is \texttt{-compress}, so you need not specify any option:

\begin{verbatim}
$ osh " ... op1 | pcompress > compressDS.ds "
\end{verbatim}

In the second step, the \texttt{pcompress} operator expands the same data set so that it can be used by another operator.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure29.png}
\caption{Expanding a Data Set with \texttt{pcompress}}
\end{figure}
Use the **osh** command:

```
$ osh "pcompress -expand < compressDS.ds | op2 ... "
```
The peek Operator

The peek operator prints record field values to the screen as the operator copies records from its input data set to one or more output data sets.

Data Flow Diagram 15 2
Properties 15 2
Syntax and Options 15 3
Using the Operator 15 4

The **peek** operator lets you print record field values to the screen as the operator copies records from its input data set to one or more output data sets. This may be helpful for monitoring the progress of your application, or to diagnose a bug in your application.
### Data Flow Diagram

```
input data sets

inRec:*;
peek
outRec:*; outRec:*; outRec:*

output data sets
...```

### Properties

Table 44  **peek Operator Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>N (set by user)</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec -&gt; outRec without record modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>any (parallel mode)</td>
</tr>
<tr>
<td>Collection method</td>
<td>any (sequential mode)</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>
### Syntax and Options

Terms in italic typeface are option strings you supply. When your option string contains a space or a tab character, you must enclose it in single quotes.

```plaintext
peek
   [-all]
   | [nrecs numrec]
   [-dataset]
   [-delim string]
   [-field fieldname ...]
   [-name]
   [-part part_num]
   [-period P]
   [-skip N]
   [-var input_schema_var_name]
```

There are no required options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-all</td>
<td>-all</td>
</tr>
<tr>
<td></td>
<td>Causes the operator to print all records. The default operation is to print 10 records per partition.</td>
</tr>
<tr>
<td>-dataset</td>
<td>-dataset</td>
</tr>
<tr>
<td></td>
<td>Specifies to write the output to a data set. The record schema of the output data set is:</td>
</tr>
<tr>
<td></td>
<td>record(rec:string;)</td>
</tr>
<tr>
<td>-delim</td>
<td>-delim string</td>
</tr>
<tr>
<td></td>
<td>Uses the string string as a delimiter on top-level fields. Other possible values for this are: n1 (newline), tab, and space. The default is the space character.</td>
</tr>
<tr>
<td>-field</td>
<td>-field fieldname</td>
</tr>
<tr>
<td></td>
<td>Specifies the field whose values you want to print. The default is to print all field values. There can be multiple occurrences of this option.</td>
</tr>
<tr>
<td>-name</td>
<td>-name</td>
</tr>
<tr>
<td></td>
<td>Causes the operator to print the field name, followed by a colon, followed by the field value. By default, the operator prints only the field value, followed by a space.</td>
</tr>
<tr>
<td>-nrecs</td>
<td>-nrecs numrec</td>
</tr>
<tr>
<td></td>
<td>Specifies the number of records to print per partition. The default is 10.</td>
</tr>
</tbody>
</table>
The **peek** operator reads the records from a single input data set and copies the records to zero or more output data sets. For a specified number of records per partition, where the default is 10, the record contents are printed to the screen. By default, the value of all record fields is printed. You can optionally configure the operator to print a subset of the input record fields.

For example, Figure 34 shows the **peek** operator, using the **-name** option to dump both the field names and field values for ten records from every partition of its input data set, between two other operators in a data flow:

![Diagram](image)

**Figure 34 Use of peek Operator**
This data flow can be implemented with the **osh** command:

```
$ osh " ... op1 | peek -name | op2 ... "
```

The output of this example is similar to the following:

```
ORCHESTRATE Application Framework
Copyright (C) 1997-2003 Ascential Systems, Inc.
All Rights Reserved
ORCHESTRATE VX.Y
16:30:49 00 APT configuration file: ./config.apt
From[1,0]: 16:30:59 00
Name: Mary Smith Age: 33 Income: 17345 Zip: 02141 Phone: 555-1212

From[1,1]: 16:30:59 00
Name: John Doe Age: 34 Income: 67000 Zip: 02139 Phone: 555-2121
16:30:59 00 Step execution finished with status = OK.
```
The remove-duplicates operator, \texttt{remdup}, takes a single sorted data set as input, removes all duplicate records, and writes the results to an output data set. Removing duplicate records is a common way of cleansing a data set before you perform further processing.

Two records are considered duplicates if they are adjacent in the input data set and have identical values for the key field(s). A key field is any field you designate to be used in determining whether two records are identical.
For example, a direct mail marketer might use the `remdup` operator to aid in **householding**, the task of cleansing a mailing list to prevent multiple mailings going to several people in the same household.

The input data set to the remove duplicates operator must be sorted so that all records with identical key values are adjacent. By default, Orchestrates inserts partition and sort components to meet the partitioning and sorting needs of the `remdup` operator and other operators. See the section “Orchestrator-Inserted Partition and Sort Components” on page xxxvi of this Reference for information on this facility.

### Data Flow Diagram

```
input data set

\[\text{inRec}:*;\]

\[\text{remdup}\]

\[\text{outRec}:*;\]

output data set
```

### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td><code>inRec:*</code></td>
</tr>
<tr>
<td>Output interface schema</td>
<td><code>outRec:*</code></td>
</tr>
<tr>
<td>Transfer behavior</td>
<td><code>inRec -&gt; outRec without record modification</code></td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
<tr>
<td>Input partitioning style</td>
<td>keys in same partition</td>
</tr>
</tbody>
</table>
Syntax and Options

remdup
[ -collation_sequence locale | collation_file_pathname | OFF]
[ -first | -last ]

Table 46  remdup Operator Properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning method</td>
<td>same (parallel mode)</td>
</tr>
<tr>
<td>Collection method</td>
<td>any (sequential mode)</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
<tr>
<td>Restartable</td>
<td>yes</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 47  remdup Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-collation_sequence</td>
<td>-collation_sequence locale</td>
</tr>
</tbody>
</table>

This option determines how your string data is sorted. You can:

- Specify a predefined IBM ICU locale
- Write your own collation sequence using ICU syntax, and supply its collation_file_pathname
- Specify OFF so that string comparisons are made using Unicode code-point value order, independent of any locale or custom sequence.

By default, Orchestrate sorts strings using byte-wise comparisons.

For more information, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide, and reference this IBM ICU site:

http://oss.software.ibm.com/icu/usertguide/Collate_Intro.htm

-first

Specifies that the first record of the duplicate set is retained. This is the default.

-last

Specifies that the last record of the duplicate set is retained. The options -first and -last are mutually exclusive.
Chapter 16 The remdup Operator

Removing Duplicate Records

The remove duplicates operator determines if two adjacent records are duplicates by comparing one-or-more fields in the records. The fields used for comparison are called key fields. When using this operator, you specify which of the fields on the record are to be used as key fields. You can define only one key field or as many as you need. Any field on the input record may be used as a key field. The determination that two records are identical is based solely on the key field values and all other fields on the record are ignored.

If the values of all of the key fields for two adjacent records are identical, then the records are considered duplicates. When two records are duplicates, one of them is discarded and one retained. By default, the first record of a duplicate pair is retained and any subsequent duplicate records in the data set are discarded. This action can be overridden with an option to keep the last record of a duplicate pair.

In order for the operator to recognize duplicate records as defined by the key fields, the records must be adjacent in the input data set. This means that the data set must have been hash partitioned, then sorted, using the same key fields for the hash and sort as you want to use for identifying duplicates. By default, Orchestrates inserts partition and sort components to meet the partitioning and sorting needs of the remdup operator and other operators. See the section “Orchestrate-Inserted Partition and Sort Components” on page xxxvi of this Reference for information on this facility.

For example, suppose you want to sort the data set first by the Month field and then by the Customer field and then use these two fields as the key fields for the remove duplicates operation.

Table 47 remdup Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-key</td>
<td>-key field [-cs</td>
</tr>
</tbody>
</table>

Specifies the name of a key field. The -key option may be repeated for as many fields as are defined in the input data set's record schema.

The -cs option specifies case-sensitive comparison, which is the default. The -ci option specifies a case-insensitive comparison of the key fields.

By default data is represented in the ASCII character set. To represent data in the EBCDIC character set, specify the -ebcdic option.

The -hash option specifies hash partitioning using this key.

The -param suboption allows you to specify extra parameters for a field. Specify parameters using property=value pairs separated by commas.
Use the following **osh** command:

```
$ osh "remdup -key Month -key Customer < inDS.ds > outDS.ds"
```

In this example, Orchestrate-inserted partition and sort components guarantees that all records with the same key field values are in the same partition of the data set. For example, all of the January records for Customer 86111 are processed together as part of the same partition.

## Using Options to the Operator

By default, the **remdup** operator retains the first record of a duplicate pair and discards any subsequent duplicate records in the data set. Suppose you have a data set which has been sorted on two fields: Month and Customer. Each record has a third field for the customer’s current **Balance** and the data set can contain multiple records for a customers balance for any month.

When using the **remdup** operator to cleanse this data set, by default, only the first record is retained for each customer and all the others are discarded as duplicates. For example, if the records in the data set are:

<table>
<thead>
<tr>
<th>Month</th>
<th>Customer</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apr</td>
<td>86111</td>
<td>787.38</td>
</tr>
<tr>
<td>Apr</td>
<td>86111</td>
<td>459.32</td>
</tr>
<tr>
<td>Apr</td>
<td>86111</td>
<td>333.21</td>
</tr>
<tr>
<td>May</td>
<td>86111</td>
<td>134.66</td>
</tr>
<tr>
<td>May</td>
<td>86111</td>
<td>594.26</td>
</tr>
</tbody>
</table>

The default result of removing duplicate records on this data set is:

<table>
<thead>
<tr>
<th>Month</th>
<th>Customer</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apr</td>
<td>86111</td>
<td>787.38</td>
</tr>
<tr>
<td>May</td>
<td>86111</td>
<td>134.66</td>
</tr>
</tbody>
</table>

Using the **-last** option, you can specify that the last duplicate record is to be retained rather than the first. This can be useful if you know, for example, that the last record in a set of duplicates is always the most recent record.
For example, if the `osh` command is:

```
$ osh "remdup -key Month -key Customer -last < inDS.ds > outDS.ds"
```

the output would given by Table 50.

**Table 50  Sample Output Data Set with -last Option**

<table>
<thead>
<tr>
<th>Month</th>
<th>Customer</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apr</td>
<td>86111</td>
<td>333.21</td>
</tr>
<tr>
<td>May</td>
<td>86111</td>
<td>594.26</td>
</tr>
</tbody>
</table>

If a key field is a string, you have a choice about how the value from one record is compared with the value from the next record. The default is that the comparison is case sensitive. If you specify the `-ci` option the comparison is case insensitive. In `osh`, specify the `-key` option with the command:

```
$osh "remdup -key Month -ci < inDS.ds > outDS.ds"
```

With this option specified, month values of “JANUARY” and “January” match, whereas without the case-insensitive option they do not match.

For example, if your input data set is:

**Table 51  Sample Input**

<table>
<thead>
<tr>
<th>Month</th>
<th>Customer</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apr</td>
<td>59560</td>
<td>787.38</td>
</tr>
<tr>
<td>apr</td>
<td>43455</td>
<td>459.32</td>
</tr>
<tr>
<td>apr</td>
<td>59560</td>
<td>333.21</td>
</tr>
<tr>
<td>May</td>
<td>86111</td>
<td>134.66</td>
</tr>
<tr>
<td>may</td>
<td>86111</td>
<td>594.26</td>
</tr>
</tbody>
</table>
The output from a case-sensitive sort is:

Table 52  Sample Output from a Case-Sensitive Sort

<table>
<thead>
<tr>
<th>Month</th>
<th>Customer</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apr</td>
<td>59560</td>
<td>787.38</td>
</tr>
<tr>
<td>May</td>
<td>86111</td>
<td>134.66</td>
</tr>
<tr>
<td>apr</td>
<td>43455</td>
<td>459.32</td>
</tr>
<tr>
<td>apr</td>
<td>59560</td>
<td>333.21</td>
</tr>
<tr>
<td>may</td>
<td>86111</td>
<td>594.26</td>
</tr>
</tbody>
</table>

Thus the two April records for customer 59560 are not recognized as a duplicate pair because they are not adjacent to each other in the input.

To remove all duplicate records regardless of the case of the Month field, use the following statement in osh:

```
$ osh "remdup -key Month -ci -key Customer < inDS.ds > outDS.ds"
```

This causes the result of sorting the input in Table 51 to be:

Table 53  Sample Output from a Case-Insensitive Sort

<table>
<thead>
<tr>
<th>Month</th>
<th>Customer</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>apr</td>
<td>43455</td>
<td>459.32</td>
</tr>
<tr>
<td>Apr</td>
<td>59560</td>
<td>787.38</td>
</tr>
<tr>
<td>apr</td>
<td>59560</td>
<td>333.21</td>
</tr>
<tr>
<td>May</td>
<td>86111</td>
<td>134.66</td>
</tr>
<tr>
<td>may</td>
<td>86111</td>
<td>594.26</td>
</tr>
</tbody>
</table>
The output from the `remdup` operator will then be:

<table>
<thead>
<tr>
<th>Month</th>
<th>Customer</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>apr</td>
<td>43455</td>
<td>459.32</td>
</tr>
<tr>
<td>Apr</td>
<td>59560</td>
<td>787.38</td>
</tr>
<tr>
<td>May</td>
<td>86111</td>
<td>134.66</td>
</tr>
</tbody>
</table>

### Using the Operator

The `remdup` operator takes a single data set as input, removes all duplicate records, and writes the results to an output data set. As part of this operation, the operator copies an entire record from the input data set to the output data set without altering the record. Only one record is output for all duplicate records.

#### Example 1: Using remdup

The following is an example of use of the `remdup` operator. Use the `osh` command:

```
osh "remdup -key Month < indDS.ds > outDS.ds"
```

This example removes all records in the same month except the first record. The output data set thus contains at most 12 records.

#### Example 2: Using the -last Option

In this example, the last record of each duplicate pair is output rather than the first, because of the `-last` option. Use the `osh` command:

```
osh "remdup -key Month -last < indDS.ds > outDS.ds"
```

#### Example 3: Case-Insensitive String Matching

This example shows use of case-insensitive string matching. Use the `osh` command:

```
osh "remdup -key Month -ci -last < indDS.ds > outDS.ds"
```

The results differ from those of the previous example if the `Month` field has mixed-case data values such as “May” and “MAY”. When the case-insensitive comparison option is used these values match and when it is not used they do not.
Example 4: Using remdup with Two Keys

This example retains the first record in each month for each customer. Therefore there are no more than 12 records in the output for each customer.

Use the `osh` command:

```bash
$ osh 'remdup -key Month -ci -key Customer < inDS.ds > outDS.ds'
```
The sample Operator

The sample operator lets you create subsets of an input data set by randomly or specifically selecting records from the input data set and assigning them to output data sets.

Data Flow Diagram  17 2
Properties  17 2
Syntax and Options  17 3
Example Sampling of a Data Set  17 4

The **sample** operator is useful when you are building, testing, or training data sets for use with the Orchestrate data-modeling operators.

The **sample** operator allows you to:

- Create disjoint subsets of an input data set by randomly sampling the input data set to assign a percentage of records to output data sets. Orchestrate uses a pseudo-random number generator to randomly select, or sample, the records of the input data set to determine the destination output data set of a record.

  You supply the initial seed of the random number generator. By changing the seed value, you can create different record distributions each time you sample a data set, and you can recreate a given distribution by using the same seed value.
A record distribution is repeatable if you use the same:
- Seed value
- Number of output data sets
- Percentage of records assigned to each data set

No input record is assigned to more than one output data set. The sum of the percentages of all records assigned to the output data sets must be less than or equal to 100%.

- Alternatively, you can specify that every nth record be written to output dataset 0.

Data Flow Diagram

![Data Flow Diagram](image)

Properties

Table 55  **sample Operator Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>N (set by user)</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec -&gt; outRec without record modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
</tbody>
</table>
Table 55  **sample Operator Properties (continued)**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning method</td>
<td>any (parallel mode)</td>
</tr>
<tr>
<td>Collection method</td>
<td>any (sequential mode)</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

**Syntax and Options**

```plaintext
sample
  -percent percent output_port_num
  [ -percent percent output_port_num ... ]
  | sample sample
  [ -maxoutputrows maxout]
  [ -seed seed_val]
```

Either the **-percent** option must be specified for each output data set or the **-sample** option must be specified.

Table 56  **sample Operator Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>maxoutputrows</strong></td>
<td>- maxoutputrows maxout</td>
</tr>
<tr>
<td></td>
<td>Optionally specifies the maximum number of rows to be output per process.</td>
</tr>
<tr>
<td></td>
<td>Supply an integer &gt;= 1 for maxout.</td>
</tr>
<tr>
<td><strong>-percent</strong></td>
<td>-percent percent output_port_num</td>
</tr>
<tr>
<td></td>
<td>Specifies the sampling percentage for each output data set. You specify the percentage as an integer value in the range of 0, corresponding to 0.0%, to 100, corresponding to 100.0%. The sum of the percentages specified for all output data sets cannot exceed 100.0%. The <code>output_port_num</code> following <code>percent</code> is the output dataset number.</td>
</tr>
<tr>
<td></td>
<td>The <strong>-percent</strong> and <strong>-sample</strong> options are mutually exclusive. One must be specified.</td>
</tr>
</tbody>
</table>
Example Sampling of a Data Set

This example configures the **sample** operator to generate three output data sets from an input data set. The first data set receives 5.0% of the records of the input data set 0, data set 1 receives 10.0%, and data set 2 receives 15.0%. See Figure 35.

![Data Flow for Sampling Example](image)
Use this osh command to implement this example:

```bash
$ osh 'sample -seed 304452 -percent 5 0 -percent 10 1 -percent 15 2
   < inDS.ds > outDS0.ds > outDS1.ds > outDS2.ds'
```

In this example, you specify a seed value of 304452, a sampling percentage for each output data set, and three output data sets.
The sequence Operator

The sequence operator lets you copy multiple input data sets to single output data set with the records ordered according to input data set.

Using the sequence operator, you can copy multiple input data sets to a single output data set. The sequence operator copies all records from the first input data set to the output data set, then all the records from the second input data set, and so forth. This operation is useful when you want to combine separate data sets into a single large data set. This chapter describes how to use the sequence operator.

The sequence operator takes one or more data sets as input and copies all input records to a single output data set. The operator copies all records from the first input data set to the output data set, then all the records from the second input data set, etc. The record schema of all input data sets must be identical.

You can execute the sequence operator either in parallel (the default) or sequentially. Sequential mode allows you to specify a collection method for an input data set to control how the data set partitions are combined by the operator.
This operator differs from the **funnel** operator, described in Chapter 8, “The Funnel Operators”, in that the **funnel** operator does not guarantee the record order in the output data set.

### Data Flow Diagram

![Data Flow Diagram](image)

### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>N (set by user)</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec -&gt; outRec without record modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>round robin (parallel mode)</td>
</tr>
<tr>
<td>Collection method</td>
<td>any (sequential mode)</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>
Syntax and Options

The syntax for the `sequence` operator in an `osh` command is simply:

```
sequence
```

It has no operator-specific options.

Using the sequence Operator

This example uses the `sequence` operator to combine multiple data sets created by multiple steps, before passing the combined data to another operator `op1`. Figure 36 shows data flow for this example:

The following `osh` commands create the data sets:

```
$ osh ' ... > outDS0.ds'
$ osh ' ... > outDS1.ds'
$ osh ' ... > outDS2.ds'
```

The `osh` command for the step beginning with the `sequence` operator is:

```
$ osh 'sequence < outDS0.ds < outDS1.ds < outDS2.ds | op1 ... ''
```
The switch Operator

The switch operator which takes a single data set as input and assigns each input record to an output data set based on the value of a selector field.

Data Flow Diagram 19 2
Properties 19 3
Syntax and Options 19 3
Using the switch Operator 19 6
Job Monitoring Information 19 8
  Example MetaData and Summary Messages 19 8
  Customizing Job Monitor Messages 19 9

The switch operator takes a single data set as input. The input data set must have an integer field to be used as a selector, or a selector field whose values can be mapped, implicitly or explicitly, to int8. The switch operator assigns each input record to one of multiple output data sets based on the value of the selector field.
The **switch** operator is analogous to a C **switch** statement, which causes the flow of control in a C program to branch to one of several cases based on the value of a selector variable, as shown in the following C program fragment.

```c
switch (selector) {
    case 0:  // if selector = 0, 
             // write record to output data set 0 
             break;
    case 1:  // if selector = 1, 
             // write record to output data set 1 
             break;
    ... 
    case discard: // if selector = discard value 
                  // skip record 
                  break;
    case default: // if selector is invalid, 
                   // abort operator and end step
}
```

You can attach up to 128 output data sets to the **switch** operator corresponding to 128 different values for the selector field.

Note that the selector value for each record must be in or be mapped to the range 0 to \(N-1\), where \(N\) is the number of data sets attached to the operator, or be equal to the discard value. Invalid selector values normally cause the **switch** operator to terminate and the step containing the operator to return an error. However, you may set an option that allows records whose selector field does not correspond to that range to be either dropped silently or treated as allowed rejects.

You can set a discard value for the selector field. Records whose selector field contains or is mapped to the discard value is dropped, that is, not assigned to any output data set.
Properties

Table 58 switch Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1 ≤ N ≤ 128</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>selector field: any data type; inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
</tbody>
</table>

Syntax and Options

Terms in italic typeface are option strings you supply. When your option string contains a space or a tab character, you must enclose it in single quotes.

```
switch
   [-allowRejects]
   [-ifNotFound ignore | allow | fail]
   [-ignoreRejects]
   [-hashSelector]

   [-case "selector_value = output_ds"]
   [-collation_sequence locale|collation_file_pathname| OFF]
   [-discard discard_value]
   [-key field_name [-cs | -ci] [-param params]]
```

If the selector field is of type integer and has no more than 128 values, there are no required options, otherwise you must specify a mapping using the -case option.
Table 59  switch Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-allowRejects</td>
<td>-allowRejects</td>
</tr>
<tr>
<td></td>
<td>Rejected records (whose selector value is not in the range 0 to $N$-1, where $N$ is the number of data sets attached to the operator, or equal to the discard value) are assigned to the last output data set. This option is mutually exclusive with the -ignoreRejects, -ifNotFound, and -hashSelector options.</td>
</tr>
<tr>
<td>-case</td>
<td>-case mapping</td>
</tr>
<tr>
<td></td>
<td>Specifies the mapping between actual values of the selector field and the output data sets. mapping is a string of the form &quot;selector_value = output_ds&quot;, where output_ds is the number of the output data set to which records with that selector value should be written (output_ds can be implicit, as shown in the example below). You must specify an individual mapping for each value of the selector field you want to direct to one of the output data sets, thus -case is invoked as many times as necessary to specify the complete mapping.</td>
</tr>
<tr>
<td></td>
<td>Multi-byte Unicode character data is supported for ustring selector values.</td>
</tr>
<tr>
<td></td>
<td>Note: This option is incompatible with the -hashSelector option.</td>
</tr>
<tr>
<td>-collation_sequence</td>
<td>-collation_sequence locale</td>
</tr>
<tr>
<td></td>
<td>This option determines how your string data is sorted. You can:</td>
</tr>
<tr>
<td></td>
<td>• Specify a predefined IBM ICU locale</td>
</tr>
<tr>
<td></td>
<td>• Write your own collation sequence using ICU syntax, and supply its collation_file_pathname</td>
</tr>
<tr>
<td></td>
<td>• Specify OFF so that string comparisons are made using Unicode code-point value order, independent of any locale or custom sequence.</td>
</tr>
<tr>
<td></td>
<td>By default, Orchestrate sorts strings using byte-wise comparisons.</td>
</tr>
<tr>
<td></td>
<td>For more information, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site: <a href="http://oss.software.ibm.com/iciuserguide/Collate_Intro.htm">http://oss.software.ibm.com/iciuserguide/Collate_Intro.htm</a></td>
</tr>
</tbody>
</table>
Table 59  switch Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-discard</td>
<td>- discard discard_value</td>
</tr>
<tr>
<td></td>
<td>Specifies an integer value of the selector field, or the value to</td>
</tr>
<tr>
<td></td>
<td>which it was mapped using case, that causes a record to be</td>
</tr>
<tr>
<td></td>
<td>discarded by the operator.</td>
</tr>
<tr>
<td></td>
<td>Note: This option is mutually exclusive with -hashSelector.</td>
</tr>
<tr>
<td>-hashSelector</td>
<td>- hashSelector</td>
</tr>
<tr>
<td></td>
<td>A boolean; when this is set, records are hashed on the selector</td>
</tr>
<tr>
<td></td>
<td>field modulo the number of output data sets and assigned to an</td>
</tr>
<tr>
<td></td>
<td>output data set accordingly. The selector field must be of a type</td>
</tr>
<tr>
<td></td>
<td>that is convertible to uint32 and may not be nullable.</td>
</tr>
<tr>
<td></td>
<td>Note: This option is incompatible with the -case, -discard, -allowRejects, -ignoreRejects, and -ifNotFound options.</td>
</tr>
<tr>
<td>-ifNotFound</td>
<td>- ifNotFound {allow</td>
</tr>
<tr>
<td></td>
<td>Specifies what the operator should do if a data set corresponding</td>
</tr>
<tr>
<td></td>
<td>to the selector value does not exist:</td>
</tr>
<tr>
<td></td>
<td>allow</td>
</tr>
<tr>
<td></td>
<td>Rejected records (whose selector value is not in the range 0 to N-1</td>
</tr>
<tr>
<td></td>
<td>or equal to the discard value) are assigned to the last output data</td>
</tr>
<tr>
<td></td>
<td>set. If this option value is used, you may not explicitly assign</td>
</tr>
<tr>
<td></td>
<td>records to the last data set.</td>
</tr>
<tr>
<td></td>
<td>fail</td>
</tr>
<tr>
<td></td>
<td>When an invalid selector value is found, return an error and</td>
</tr>
<tr>
<td></td>
<td>terminate. This is the default.</td>
</tr>
<tr>
<td></td>
<td>ignore</td>
</tr>
<tr>
<td></td>
<td>Drop the record containing the out-of-range value and continue.</td>
</tr>
<tr>
<td></td>
<td>Note: This option is incompatible with -allowRejects, -ignoreRejects,</td>
</tr>
<tr>
<td></td>
<td>and -hashSelector options.</td>
</tr>
<tr>
<td>-ignoreRejects</td>
<td>- ignoreRejects</td>
</tr>
<tr>
<td></td>
<td>Drop the record containing the out-of-range value and continue.</td>
</tr>
<tr>
<td></td>
<td>Note: This option is mutually exclusive with the -allowRejects, -ifNotFound, and -hashSelector options.</td>
</tr>
</tbody>
</table>
Using the switch Operator

In this example, you create a switch operator and attach three output data sets numbered 0 through 2. The switch operator assigns input records to each output data set based on the selector field, whose year values have been mapped to the numbers 0 or 1 by means of the -case option. A selector field value that maps to an integer other than 0 or 1 causes the operator to write the record to the last data set. You may not explicitly assign input records to the last data set if the -ifNotFound option is set to allow.

With these settings, records whose year field has the value 1990, 1991, or 1992 go to outDS0.ds. Those whose year value is 1993 or 1994 go to outDS1.ds. Those whose year is 1995 are discarded. Those with any other year value are written to outDS2.ds, since rejects are allowed by the -ifNotFound setting. Note that because the -ifNotFound option is set to allow rejects, switch does not let you map any year value explicitly to the last data set (outDS2.ds), as that is where rejected records are written.

Note also that it was unnecessary to specify an output data set for 1991 or 1992, since without an explicit mapping indicated, case maps values across the output data sets, starting from the first (outDS0.ds). You may map more than one selector field value to a given output data set.

The operator also verifies that if a -case entry maps a selector field value to a number outside the range 0 to N -1, that number corresponds to the value of the -discard option.

---

Table 59  switch Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-key</td>
<td>-key field_name [-cs</td>
</tr>
</tbody>
</table>

Specifies the name of a field to be used as the selector field. The default field name is selector. This field can be of any data type that can be converted to int8, or any non-nullable type if case options are specified. Field names can contain multi-byte Unicode characters.

Use the -ci flag to specify that field_name is case-insensitive. The -cs flag specifies that field_name is treated as case-sensitive, which is the default.
Figure 37 shows the data flow model for this example.

In this example, all records with the selector field mapped to:

- 0 are written to `outDS0.ds`
- 1 is written to `outDS1.ds`
- 5 are discarded
- any other values are treated as rejects and written to `outDS2.ds`.

In most cases, your input data set does not contain an 8-bit integer field to be used as the selector; therefore, you use the `-case` option to map its actual values to the required range of integers. In this example, the record schema of the input data set contains a string field named `year`, which you must map to 0 or 1.

Specify the mapping with the following osh code:

```bash
```
Job Monitoring Information

The `switch` operator reports business logic information which can be used to make decisions about how to process data. It also reports summary statistics based on the business logic.

The business logic is included in the metadata messages generated by Orchestrate as custom information. It is identified with:

```
name="BusinessLogic"
```

The output summary per criterion is included in the summary messages generated by Orchestrate as custom information. It is identified with:

```
name="CriterionSummary"
```

The XML tags `criterion`, `case` and `where` are used by the `switch` operator when generating business logic and criterion summary custom information. These tags are used in the example information below.

Example MetaData and Summary Messages

```xml
<response type="metadata">
  <component ident="switch">
    <componentstats startTime="2002-08-08 14:41:56"/>
    <linkstats portNum="0" portType="in"/>
    <linkstats portNum="0" portType="out/"/>
    <linkstats portNum="1" portType="out/"/>
    <linkstats portNum="2" portType="out/"/>
    <custom_info Name="BusinessLogic" Desc="User-supplied logic to switch operator">
      <criterion name="key">tfield</criterion>
      <criterion name="case">
        <case value=" 0" output_port="0"></case>
        <case value=" 1" output_port="1"></case>
        <case value=" 2" output_port="2"></case>
      </criterion>
      </custom_info>
    </component>
  </response>
```
The Orchestrator Job Monitor is documented in the Orchestrate 7.0 User Guide.

**Customizing Job Monitor Messages**

Orchestrator specifies the business logic and criterion summary information for the `switch` operator using the functions `addCustomMetadata()` and `addCustomSummary()`. You can also use these functions to generate this kind of information for the operators you write. See the section "Customizing Job Monitoring Messages" in Chapter 1: Creating Operators, in the Orchestrate 7.0 Developer Guide.
The tail Operator

The tail operator copies the last N records from each partition of its input data set to its output data set. By default, N is 10 records. However, you can determine the following by means of options:

- The number of records to copy
- The partition from which the records are copied

This control is helpful in testing and debugging applications with large data sets.

The head operator performs a similar operation, copying the first N records from each partition. See Chapter 10, “The head Operator”.

Data Flow Diagram 20 2
Properties 20 2
Syntax and Options 20 2
Examples 20 3
  Example 1: tail Operator Default Behavior 20 3
  Example 2: tail Operator with Both Options 20 4
Data Flow Diagram

```
input data set

```

```
| inRec:*; |
| tail |
| outRec:*; |

```

output data set

Properties

Table 60  tail Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec -&gt; outRec without record modification</td>
</tr>
</tbody>
</table>

Syntax and Options

```
tail
    [-nrecs count]
    [-part partition_number]
```

Table 61  tail Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-nrecs</td>
<td>-nrecs count Specify the number of records (count) to copy from each partition of the input data set to the output data set. The default value of count is 10.</td>
</tr>
</tbody>
</table>
Examples

These are as follows:

- “Example 1: tail Operator Default Behavior” on page 20-3
- “Example 2: tail Operator with Both Options” on page 20-4

Example 1: tail Operator Default Behavior

In this example, no options have been specified to the tail operator. The input data set consists of 60 sorted records (positive integers) hashed into four partitions. The output data set consists of the last ten records of each partition. The osh command for the example is:

```
$osh "tail < in.ds > out.ds"
```

<table>
<thead>
<tr>
<th>Partition</th>
<th>Input</th>
<th>Output</th>
<th>Input</th>
<th>Output</th>
<th>Input</th>
<th>Output</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>9</td>
<td>1</td>
<td>3</td>
<td>16</td>
<td>6</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>18</td>
<td>18</td>
<td>11</td>
<td>35</td>
<td>8</td>
<td>41</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>23</td>
<td>23</td>
<td>12</td>
<td>42</td>
<td>22</td>
<td>43</td>
<td>10</td>
</tr>
<tr>
<td>23</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>13</td>
<td>46</td>
<td>29</td>
<td>44</td>
<td>20</td>
</tr>
<tr>
<td>25</td>
<td>36</td>
<td>36</td>
<td>36</td>
<td>14</td>
<td>49</td>
<td>30</td>
<td>45</td>
<td>21</td>
</tr>
<tr>
<td>36</td>
<td>37</td>
<td>37</td>
<td>37</td>
<td>15</td>
<td>50</td>
<td>33</td>
<td>48</td>
<td>24</td>
</tr>
<tr>
<td>37</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>16</td>
<td>53</td>
<td>41</td>
<td>55</td>
<td>26</td>
</tr>
<tr>
<td>40</td>
<td>47</td>
<td>47</td>
<td>47</td>
<td>17</td>
<td>57</td>
<td>43</td>
<td>56</td>
<td>27</td>
</tr>
<tr>
<td>47</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>35</td>
<td>59</td>
<td>44</td>
<td>58</td>
<td>28</td>
</tr>
<tr>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>42</td>
<td>59</td>
<td>45</td>
<td>58</td>
<td>28</td>
</tr>
</tbody>
</table>

Table 62 tail Operator Input and Output for Example 1
Example 2: tail Operator with Both Options

In this example, both the -nrecs and -part options are specified to the tail operator to request that the last 3 records of Partition 2 be output. The input data set consists of 60 sorted records (positive integers) hashed into four partitions. The output data set contains only the last three records of Partition 2. Table 63 shows the input and output data.

The osh command for this example is:
$ osh "tail -nrecs 3 -part 2 < in.ds > out0.ds"

Table 63  tail Operator Input and Output for Example 2

<table>
<thead>
<tr>
<th>Partition 0</th>
<th>Partition 1</th>
<th>Partition 2</th>
<th>Partition 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Output</td>
<td>Input</td>
<td>Output</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>55</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>7</td>
<td>56</td>
</tr>
<tr>
<td>18</td>
<td>11</td>
<td>8</td>
<td>58</td>
</tr>
<tr>
<td>19</td>
<td>12</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>13</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>14</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>15</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>16</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>17</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>35</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>42</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The transform Operator

The transform operator transforms all records that pass through it based on the value of transformation expressions you supply.

Running Your Job on a non-NFS MPP 21 4
Data Flow Diagram 21 4
Properties 21 5
Syntax and Options 21 5
Transfer Behavior 21 12

The Transformation Language 21 14
   General structure 21 14
   Names and Keywords 21 14
   External Global C-Function Support 21 14
      C Function Schema Types and Associated C Types 21 15
      Specifying the Location of Your C Libraries 21 15
   Code Segmentation Keywords 21 16
   Record Processing Control 21 17
   Specifying Data Sets 21 17
   Data types and record fields 21 18
   Local Variables 21 20
   Expressions 21 21
      Language Elements 21 21
      Operators 21 22
      Conditional Branching 21 25
   Built-in Functions 21 27
      Data Conversion Functions 21 27
      Mathematical Functions 21 39
      string Field Functions 21 41
      ustring Field Functions 21 48
      Bit Manipulation Functions 21 54
      Job Monitoring Functions 21 55
      Miscellaneous Functions 21 57
Chapter 21 The transform Operator

The Transformation Language Versus C 21 57
  Keywords 21 57
  Local Variables 21 58
  Operators 21 58
  Flow Control 21 58
  Other Unsupported C Language Elements 21 58
  References 21 58

Using the transform Operator 21 59
  Example 1: Student-Score Distribution 21 59
    Application Logic 21 59
    Data Flow Diagram 21 59
    Highlighted Transformation Language Components 21 59
    Transformation Language 21 60
    osh Command 21 61
    osh Script 21 61
    Example Input and Output 21 62
  Example 2: Student-Score Distribution With a Letter Grade Added to
  Example 1 21 62
    Application Logic 21 62
    Data Flow Diagram 21 63
    Highlighted Transformation Language Components 21 63
    Transformation Language 21 63
    osh Command 21 64
    osh Script 21 65
    Example Input and Output 21 66
  Example 3: Student-Score Distribution with a Class Field Added to
  Example 2 21 65
    Application Logic 21 65
    Data Flow Diagram 21 66
    Highlighted Transformation Language Components 21 66
    Transformation Language 21 66
    osh Command 21 67
    osh Script 21 67
    Example Input and Output 21 68
  Example 4. Student Record Distribution With Null Score Values And a
  Reject Data Set 21 69
    Application Logic 21 69
    Data Flow Diagram 21 69
    osh Command 21 69
    osh Script 21 69
    Example Input and Output 21 70
  Example 5. Student Record Distribution With Null Score Values
  Handled 21 71
    Application Logic 21 71
    Data Flow Diagram 21 71
    Highlighted Transformation Language Components 21 71
The `transform` operator modifies your input records, or transfers them unchanged, guided by the logic of the transformation expression you supply. You build transformation expressions using the **Transformation Language**, which is the language that defines expression syntax and provides built-in functions.

By using the Transformation Language with the `transform` operator, you can:

- Transfer input records to one or more outputs
- Define output fields and assign values to them based on your application logic
- Use local variables and input and output fields to perform arithmetic operations, make function calls, and construct conditional statements.
- Control record processing behavior by explicitly calling `writerecord`, `droprecord`, and `rejectrecord` as part of a conditional statement.

Running Your Job on a non-NFS MPP

At run time, the `transform` operator distributes its shared library to remote nodes on non-NFS MPP systems. To prevent your job from aborting, these three conditions must be satisfied:

1. The `APT_COPY_TRANSFORM_OPERATOR` environment variable must be set.
2. Users must have create privileges on the project directory paths on all remote nodes at runtime. For example, the `transform` library `trx.so` is created on the conductor node at this location:
   ```plaintext
   /u1/Ascential/DataStage/Projects/simple/RT_BP1.O
   ```
3. Rename `$APT_ORCHHOME/etc/distribute-component.example` to `$APT_ORCHHOME/etc/distribute-component` and make the file executable:
   ```bash
   chmod 755 $APT_ORCHHOME/etc/distribute-component
   ```

Data Flow Diagram

```
input data set

transform

output data sets

optional reject data set
```
Properties

Table 64  transform Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1 or more and, optionally, 1 reject data set</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>See “Transfer Behavior” on page 21-12</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel by default, or sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>any (parallel mode)</td>
</tr>
<tr>
<td>Collection method</td>
<td>any (sequential mode)</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
<tr>
<td>Composite operator</td>
<td>yes</td>
</tr>
<tr>
<td>Combinable operator</td>
<td>yes</td>
</tr>
</tbody>
</table>

Syntax and Options

Terms in italic typeface are option strings you supply. When your option string contains a space or a tab character, you must enclose it in single quotes.

```
transform
    [-argvalue job_parameter_name=job_parameter_value...]
    [-collation_sequence locale|collation_file_pathname|OFF]
    [-expression expression_string|-expressionfile expressionfile_path]
    [-maxrejectlogs integer]
    [-sort [-input |-output [port] -key field_name sort_key_suboptions...]]
    [-part [-input |-output [port] -key field_name part_key_suboptions...]]
    [-flag {compile | run | compileAndRun} [flag_compilation_options]]
    [-inputschema schema |-inputschemafile schema_file]
    [-outputschema schema |-outputschemafile schema_file]
    [-reject [-rejectinfo reject_info_column_name_string]]
```

Where:

- **sort_key_suboptions are:**
  ```
  [-ci | -cs] [-asc | -desc] [-nulls {first | last}] [-param params]
  ```

- **part_key_options are:**
  ```
  [-ci | -cs] [-param params]
  ```
flag_compilation_options are:

- [dir dir_name_for_compilation] [-name library_path_name]
- [optimize | debug] [-verbose] [-compiler cpath]
- [staticobj absolute_path_name] [-sharedobj absolute_path_name]
- [t options]
- [compileopt options] [-linker [path]] [-linkopt options]

Note

The following option values can contain multi-byte Unicode values:

- the field names given to the -inputschema and -outputschema options and the ustring values
- -inputschemafile and -outputschemafile files
- -expression option string and the -expressionfile option filepath
- -sort and -part key-field names
- -compiler, -linker, and -dir pathnames
- -name filename
- -staticobj and -sharedobj pathnames
- -compileopt and -linkopt pathnames

<table>
<thead>
<tr>
<th>Table 65</th>
<th>transform Operator Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option</td>
<td>Use</td>
</tr>
<tr>
<td>-argvalue</td>
<td>-argvalue job_parameter_name = job_parameter_value</td>
</tr>
</tbody>
</table>

This option is similar to the -params top-level osh option, but the initialized variables apply to a transform operator rather than to an entire job. The global variable given by job_parameter_name is initialized with the value given by job_parameter_value.

In your osh script, you reference the job_parameter_value with [ &job_parameter_name] where the job_parameter_value component replaces the occurrence of [ &job_parameter_name].
### Table 65  transform Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-collation_sequence</code></td>
<td>`-collation_sequence locale</td>
</tr>
<tr>
<td></td>
<td>This option determines how your string data is sorted. You can:</td>
</tr>
<tr>
<td></td>
<td>• Specify a predefined IBM ICU <code>locale</code></td>
</tr>
<tr>
<td></td>
<td>• Write your own collation sequence using ICU syntax, and supply its</td>
</tr>
<tr>
<td></td>
<td><code>collation_file_pathname</code></td>
</tr>
<tr>
<td></td>
<td>• Specify <code>OFF</code> so that string comparisons are made using Unicode code-</td>
</tr>
<tr>
<td></td>
<td>point value order, independent of any locale or custom sequence.</td>
</tr>
<tr>
<td></td>
<td>By default, Orchestrate sorts strings using byte-wise comparisons.</td>
</tr>
<tr>
<td></td>
<td>For more information, see Chapter 7: National Language Support in the</td>
</tr>
<tr>
<td></td>
<td>Orchestrate 7.0 User Guide; and reference this IBM ICU site:</td>
</tr>
<tr>
<td></td>
<td><a href="http://oss.software.ibm.com/icu/userguide/">http://oss.software.ibm.com/icu/userguide/</a> Collate_Intro.htm</td>
</tr>
<tr>
<td><code>-expression</code></td>
<td><code>-expression expression_string</code></td>
</tr>
<tr>
<td></td>
<td>This option lets you specify expressions written in the Transformation</td>
</tr>
<tr>
<td></td>
<td>Language. The expression string may contain multi-byte Unicode</td>
</tr>
<tr>
<td></td>
<td>characters.</td>
</tr>
<tr>
<td></td>
<td>Unless you choose the <code>-flag</code> option with <code>run</code>, you must use either</td>
</tr>
<tr>
<td></td>
<td>the <code>-expression</code> or <code>-expressionfile</code> option.</td>
</tr>
<tr>
<td></td>
<td>The <code>-expression</code> and <code>-expressionfile</code> options are mutually exclusive.</td>
</tr>
<tr>
<td><code>-expressionfile</code></td>
<td><code>-expressionfile expression_file</code></td>
</tr>
<tr>
<td></td>
<td>This option lets you specify expressions written in the Transformation</td>
</tr>
<tr>
<td></td>
<td>Language. The expression must reside in an <code>expression_file</code>, which</td>
</tr>
<tr>
<td></td>
<td>includes the name and path to the file which may include multi-byte</td>
</tr>
<tr>
<td></td>
<td>Unicode characters. Use an absolute path, or by default the current</td>
</tr>
<tr>
<td></td>
<td>UNIX directory. Unless you choose the <code>-flag</code> option with <code>run</code>, you</td>
</tr>
<tr>
<td></td>
<td>must choose either the <code>-expression</code> or <code>-expressionfile</code> option.</td>
</tr>
<tr>
<td></td>
<td>The <code>-expressionfile</code> and <code>-expression</code> options are mutually exclusive.</td>
</tr>
</tbody>
</table>
Table 65  transform Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-flag</code> suboptions</td>
<td>`-flag {compile</td>
</tr>
</tbody>
</table>

**compile:** This option indicates that you wish to check the Transformation Language expression for correctness, and compile it. An appropriate version of a C++ compiler must be installed on your computer. Field information used in the expression must be known at compile time; therefore, input and output schema must be specified.

**run:** This option indicates that you wish to use a pre-compiled version of the Transformation Language code. You do not need to specify input and output schemas or an expression because these elements have been supplied at compile time. However, you must add the directory containing the pre-compiled library to your library search path. This is not done by the `transform` operator. You must also use the `-name` suboption to provide the name of the library where the pre-compiled code resides.

**compileAndRun:** This option indicates that you wish to compile and run the Transformation Language expression. This is the default value. An appropriate version of a C++ compiler must be installed on your computer.

You can supply schema information in the following ways:

You can omit all schema specifications. The `transform` operator then uses the up-stream operator’s output schema as its input schema, and the schema for each output dataset contains all the fields from the input record plus any new fields you create for a dataset.

You can omit the input dataset schema, but specify schemas for all output data sets or for selected data sets. The `transform` operator then uses the up-stream operator’s output schema as its input schema. Any output schemas specified on the command line are used unchanged, and output datasets without schemas contain all the fields from the input record plus any new fields you create for a dataset.

You can specify an input schema, but omit all output schemas or omit some output schemas. The `transform` operator then uses the input schema as specified. Any output schemas specified on the command line are used unchanged, and output datasets without schemas contain all the fields from the input record plus any new fields you create for a dataset.

continued on the following page
The flag option has the following suboptions:

- **dir** dir_name lets you specify a compilation directory. By default, compilation occurs in the TMPDIR directory or, if this environment variable does not point to an existing directory, to the /tmp directory. Whether you specify it or not, you must make sure the directory for compilation is in the library search path.

- **name** file_name lets you specify the name of the file containing the compiled code. If you use the **dir** dir_name suboption, this file is in the dir_name directory.

The following examples show how to use the **dir** and **name** options in an osh command line:

For development:

```
osh "transform -inputschema schema -outputschema schema -expression expression -flag compile -dir dir_name -name file_name"
```

For your production machine:

```
osh "... | transform -flag run -name file_name | ..."
```

The library file must be copied to the production machine.

**flag compile** and **flag compileAndRun** have these additional suboptions:

- **optimize** specifies the optimize mode for compilation.
- **debug** specifies the debug mode for compilation.
- **verbose** causes verbose messages to be output during compilation.
- **compiler cpath** lets you specify the compiler path when the compiler is not in the default directory. The default compiler path for each operating system is:
  - **Solaris**: /opt/SUNPRO6/SUNWspro/bin/CC
  - **AIX**: /usr/va/cpp/bin/xlC_r
  - **Tru64**: /bin/cxx
  - **HP-UX**: /opt/ac/1n/ac

- **staticobj absolute_path_name**
- **sharedobj absolute_path_name**

These two suboptions specify the location of your static and dynamic-linking C-object libraries. The file suffix can be omitted. See “External Global C-Function Support” on page 21-14 for details.

continued on the following page
### Table 65  transform Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-flag</code></td>
<td><code>-compileopt</code> options lets you specify additional compiler options. These options are compiler-dependent. Pathnames may contain multi-byte Unicode characters.</td>
</tr>
<tr>
<td></td>
<td><code>-linker lpath</code> lets you specify the linker path when the linker is not in the default directory. The default linker path of each operating system is the same as the default compiler path listed above.</td>
</tr>
<tr>
<td></td>
<td><code>-linkopt</code> options lets you specify link options to the compiler. Pathnames may contain multi-byte Unicode characters.</td>
</tr>
<tr>
<td><code>-inputschema</code></td>
<td><code>-inputschema</code> schema</td>
</tr>
<tr>
<td></td>
<td>Use this option to specify an input schema. The schema may contain multi-byte Unicode characters. An error occurs if an expression refers to an input field not in the input schema.</td>
</tr>
<tr>
<td></td>
<td>The <code>-inputschema</code> and the <code>-inputschemafile</code> options are mutually exclusive.</td>
</tr>
<tr>
<td></td>
<td>The <code>-inputschema</code> option is not required when you specify <code>compileAndRun</code> or <code>run</code> for the <code>-flag</code> option; however, when you specify <code>compile</code> for the <code>-flag</code> option, you must include either the <code>-inputschema</code> or the <code>-inputschemafile</code> option. See the <code>-flag</code> option description in this table for information on the <code>-compile</code> suboption.</td>
</tr>
<tr>
<td><code>-inputschemafile</code></td>
<td><code>-inputschemafile</code> schema_file</td>
</tr>
<tr>
<td></td>
<td>Use this option to specify an input schema. An error occurs if an expression refers to an input field not in the input schema. To use this option, the input schema must reside in a <code>schema_file</code>, where <code>schema_file</code> is the name and path to the file which may contain multi-byte Unicode characters. You can use an absolute path, or by default the current UNIX directory.</td>
</tr>
<tr>
<td></td>
<td>The <code>-inputschemafile</code> and the <code>-inputschema</code> options are mutually exclusive.</td>
</tr>
<tr>
<td></td>
<td>The <code>-inputschemafile</code> option is not required when you specify <code>compileAndRun</code> or <code>run</code> for the <code>-flag</code> option; however, when you specify <code>compile</code> for the <code>-flag</code> option, you must include either the <code>-inputschema</code> or the <code>-inputschemafile</code> option. See the <code>-flag</code> option description in this table for information on the <code>-compile</code> suboption.</td>
</tr>
<tr>
<td><code>-maxrejectlogs</code></td>
<td><code>-maxrejectlogs</code> integer</td>
</tr>
<tr>
<td></td>
<td>An information log is generated every time a record is written to the reject output data set. Use this option to specify the maximum number of output reject logs the <code>transform</code> option generates. The default is 50. When you specify <code>-1</code> to this option, an unlimited number of information logs are generated.</td>
</tr>
</tbody>
</table>
Table 65  transform Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-outputschema</td>
<td>-outputschema schema</td>
</tr>
<tr>
<td></td>
<td>Use this option to specify an output schema. An error occurs if an</td>
</tr>
<tr>
<td></td>
<td>expression refers to an output field not in the output schema.</td>
</tr>
<tr>
<td></td>
<td>The -outputschema and -outputschemafile options are mutually</td>
</tr>
<tr>
<td></td>
<td>exclusive.</td>
</tr>
<tr>
<td></td>
<td>The -outputschema option is not required when you specify compile</td>
</tr>
<tr>
<td></td>
<td>And Run or run for the -flag option; however, when you specify</td>
</tr>
<tr>
<td></td>
<td>compile for the -flag option, you must include either the -outputschema</td>
</tr>
<tr>
<td></td>
<td>or the -outputschemafile option. See the -flag option description in</td>
</tr>
<tr>
<td></td>
<td>this table for information on the -compile suboption. For multiple</td>
</tr>
<tr>
<td></td>
<td>output data sets, repeat the -outputschema or -outputschemafile</td>
</tr>
<tr>
<td></td>
<td>option to specify the schema for all output data sets.</td>
</tr>
<tr>
<td>-outputschemafile</td>
<td>-outputschemafile schema_file</td>
</tr>
<tr>
<td></td>
<td>Use this option to specify an output schema. An error occurs if an</td>
</tr>
<tr>
<td></td>
<td>expression refers to an output field not in the output schema.</td>
</tr>
<tr>
<td></td>
<td>To use this option, the output schema must reside in a schema_file</td>
</tr>
<tr>
<td></td>
<td>which includes the name and path to the file. You can use an absolute</td>
</tr>
<tr>
<td></td>
<td>path, or by default the current UNIX directory.</td>
</tr>
<tr>
<td></td>
<td>The -outputschemafile and the -outputschema options are mutually</td>
</tr>
<tr>
<td></td>
<td>exclusive.</td>
</tr>
<tr>
<td></td>
<td>The -outputschemafile option is not required when you specify</td>
</tr>
<tr>
<td></td>
<td>compile And Run or run for the -flag option; however, when you specify</td>
</tr>
<tr>
<td></td>
<td>compile for the -flag option, you must include either the -outputschema</td>
</tr>
<tr>
<td></td>
<td>or the -outputschemafile option. See the -flag option description in</td>
</tr>
<tr>
<td></td>
<td>this table for information on the -compile suboption. For multiple</td>
</tr>
<tr>
<td></td>
<td>output data sets, repeat the -outputschema or -outputschemafile</td>
</tr>
<tr>
<td></td>
<td>option to specify the schema for all output data sets.</td>
</tr>
<tr>
<td>-part</td>
<td>-part {{input</td>
</tr>
<tr>
<td></td>
<td>[-param params]}</td>
</tr>
<tr>
<td></td>
<td>You can use this option 0 or more times. It indicates that the data</td>
</tr>
<tr>
<td></td>
<td>is hash partitioned. The required field_name is the name of a</td>
</tr>
<tr>
<td></td>
<td>partitioning key.</td>
</tr>
<tr>
<td></td>
<td>Exactly one of the suboptions -input and -output[ port ] must be</td>
</tr>
<tr>
<td></td>
<td>present. These suboptions determine whether partitioning occurs on</td>
</tr>
<tr>
<td></td>
<td>the input data or the output data. The default for port is 0.</td>
</tr>
<tr>
<td></td>
<td>If port is specified, it must be an integer which represents an</td>
</tr>
<tr>
<td></td>
<td>output data set where the data is partitioned.</td>
</tr>
<tr>
<td></td>
<td>The suboptions to the -key option are -ci for case-insensitive</td>
</tr>
<tr>
<td></td>
<td>partitioning, or -cs for a case-sensitive partitioning. The default</td>
</tr>
<tr>
<td></td>
<td>is case-sensitive. The -params suboption is to specify any property-</td>
</tr>
<tr>
<td></td>
<td>value pairs. Separate the pairs by commas (;).</td>
</tr>
</tbody>
</table>
You can transfer your input fields to your output fields using any one of the following methods:

- Set the value of the `-flag` option to `compileAndRun`. For example:

  ```
  osh "... | transform -expression expression -flag compileAndRun
         -dir dir_name -name file_name ..."
  ```

  See Table 65 for information on the `flag` option and the `compileAndRun` suboption.

- Use schema variables as part of the schema specification. A partial schema may be used for both the input and output schemas.

  This example shows a partial schema in the output:
osh "transform
  -expression expression
  -inputschema record(a:int32;b:string[5];c:time)
  -outputschema record(d:dfloat;outRec:*;)
  -flag compile ..."

where the schema for output 0 is:
  record(d:dfloat;a:int32;b:string[5];c:time)

This example shows partial schemas in the input and the output:
osh "transform
  -expression expression
  -inputschema record(a:int32;b:string[5];c:time;Inrec:*)
  -outputschema record(d:dfloat;outRec:*;)
  -flag compile ..."
osh "... | transform -flag run ... | ...

Output 0 contains the fields d, a, b, and c, plus any fields propagated from the up-stream operator.

- Use name matching between input and output fields in the schema specification. When input and output field names match and no assignment is made to the output field, the input field is transferred to the output data set unchanged. Any input field which doesn't have a corresponding output field is dropped. For example:

  osh "transform
    -expression expression
    -inputschema record(a:int32;b:string[5];c:time)
    -outputschema record(a:int32;)
    -outputschema record(a:int32;b:string[5];c:time)
    -flag compile ..."

Field a is transferred from input to output 0 and output 1. Fields b and c are dropped in output 0, but are transferred from input to output 1.

- Specify a reject data set. In the Transformation Language, it is generally illegal to use a null field in expressions except in the following cases:
  - In function calls to notnull(field_name) and null(field_name)
  - In an assignment statement of the form a=b where a and b are both nullable and b is null
  - In these expressions:

    if (null(a))
      b=a
    else
      b=a+1

    if (notnull(a))
      b=a+1
    else
      b=a
b = null(a)?a:a+1;
b = notnull(a)?a+1:a;

If a null field is used in an expression in other than these cases and a reject set is specified, the whole input record is transferred to the reject data set.

**The Transformation Language**

The Transformation Language is a subset of C, with extensions specific to dealing with records.

**General structure**

As in C, statements must be terminated by semi-colons and compound statements must be grouped by braces. Both C and C++ style comments are allowed.

**Names and Keywords**

Names of fields in records, local variable names, and language keywords can consist of alphanumeric characters plus the underscore character. They cannot begin with a numeric character. Names in the Transformation Language are case-sensitive but keywords are case-insensitive.

The available keywords fall into four groups:

- The keyword `extern` is used to declare global C functions. See “External Global C-Function Support” on page 21-14 below.
- The keywords `global`, `initialize`, `mainloop`, and `finish` mark blocks of code that are executed at different stages of record processing. An explanation of these keywords are in “Code Segmentation Keywords” on page 21-16.
- The keywords `droprecord`, `writerecord`, and `rejectrecord` control record processing. See “Record Processing Control” on page 21-17.
- The keywords `inputname` and `outputname` are used to declare data set aliases. See “Specifying Data Sets” on page 21-17.

**External Global C-Function Support**

Standard C functions are supported in the Transformation Language. Declare them in your expression file using the `extern` keyword and place them before your code segmentation keywords. The syntax for an external C function declaration is:

```
extern return_type function_name( [argument_type, argument_name...] );
```
Here is an expression file fragment that incorporates external C-function declarations:

```c
// externs this C function: int my_times(int x, int y) { ... }
extern int32 my_times(int32 x, int32 y);
// externs this C function: void my_print_message(char *msg) { ... }
extern void my_print_message(string msg);
inputname 0 in0;
outputname 0 out0;
mainloop
{ ... }
```

### C Function Schema Types and Associated C Types

The C function return and argument types can be any of the Orchestrate schema types listed below with their associated C types.

#### Table 66  C Function Schema Types and Associated C Types

<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Associated Native C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8</td>
<td>signed char</td>
</tr>
<tr>
<td>uint8</td>
<td>unsigned char</td>
</tr>
<tr>
<td>int16</td>
<td>short</td>
</tr>
<tr>
<td>uint16</td>
<td>unsigned short</td>
</tr>
<tr>
<td>int32</td>
<td>int</td>
</tr>
<tr>
<td>uint32</td>
<td>unsigned int</td>
</tr>
<tr>
<td>int64</td>
<td>long for Tru64</td>
</tr>
<tr>
<td></td>
<td>long long for Solaris and AIX</td>
</tr>
<tr>
<td>uint64</td>
<td>unsigned long for Tru64</td>
</tr>
<tr>
<td></td>
<td>unsigned long long for Solaris and AIX</td>
</tr>
<tr>
<td>sfloat</td>
<td>float</td>
</tr>
<tr>
<td>dfloat</td>
<td>double</td>
</tr>
<tr>
<td>string</td>
<td>char *</td>
</tr>
<tr>
<td>void</td>
<td>void</td>
</tr>
</tbody>
</table>

### Specifying the Location of Your C Libraries

To specify the locations of your static and dynamically-linked libraries, use the `-staticobj` and `-sharedobj` suboptions of the `-flag` option. These two suboptions take absolute path names as values. The file suffix is optional. The syntax is:

- `-staticobj` absolute_path_name
- `-sharedobj` absolute_path_name
An example static library specification is:
```
-flag compile
 -name generate_statistics
 -staticobj /external_functions/static/part_statistics.o
```
An example dynamic library specification is:
```
-flag compile
...
-sharedobj /external_functions/dynamic/generate
```
The shared object file name has `lib` prepended to it and has a platform-dependent object-file suffix: `.so` for Sun Solaris, Tru64, and Linux; `.sl` for HP-UX, and `.o` for AIX. The file must reside in this directory:
```
/external-functions/dynamic
```
For this example, the object filepath on Solaris is:
```
/external-functions/dynamic/libgenerate.so
```
Dynamically-linked libraries must be manually deployed to all running nodes. Add the library-file locations to your library search path.

See “Example 8: External C Function Calls” on page 21-84 for an example application that includes C header and source files, a Transformation Language expression file with calls to external C functions, and an `osh` script.

**Code Segmentation Keywords**

The Transformation Language provides keywords to specify when code is executed. Refer to “Example 1: Student-Score Distribution” on page 21-59 for an example of how to use of these keywords.

- **global { job_parameters }**
  
  Use this syntax to declare a set of global variables whose values are supplied by `osh` parameters. Values cannot be set with the Transformation Language. A warning message is issued if a value is missing.

- **initialize { statements }**
  
  Use this syntax to mark a section of code you want to be executed once before the main record loop starts. Global variables whose values are not given through `osh` parameters should be defined in this segment.

- **mainloop { statements }**
  
  Use this syntax to indicate the main record loop code. The `mainloop` segments is executed once for each input record.

- **finish { statements }**
  
  Use this syntax to mark a section of code you want to be executed once after the main record loop terminates.
Record Processing Control

The transform operator processes one input record at a time, generating zero or any number of output records and zero or one reject record for each input record, terminating when there are no more input records to process.

The transform operator automatically reads records by default. You do not need to specify this actions.

The Transformation Language lets you control the input and output of records with the following keywords.

- **write_record n;**
  Use this syntax to force an output record to be written to the specific data set whose port number is n.

- **drop_record;**
  Use this syntax to prevent the current input record from being written.

- **reject_record;**
  If you declare a reject data set, you can use this syntax to direct the current input record to it. You should only send a record to the reject data set if it is not going to another output data set.

**Note** Processing exceptions, such as null values for non-nullable fields, cause a record to be written to the reject data set if you have specified one. Otherwise the record is simply dropped.

Specifying Data Sets

By default, the **transform** operator supports a single input data set, one or more output data sets, and a single optional reject data set. There is no default correspondence between input and output. You must use **write_record port** to specify where you want your output sent. You can assign a name to each data set for unambiguous reference, using this syntax:

```
inputname 0 input-dataset-name;
outputname n output-dataset-name;
```

Because the **transform** operator accepts only a single input data set, the data set number for **inputname** is 0. You can specify 0 through (the number of output data sets - 1) for the **outputname** data set number. For example:

```
inputname 0 input-grades;
outputname 0 output-low-grades;
outputname 1 output-high-grades;
```
Data set numbers cannot be used to qualify field names. You must use the `inputname` and `outputname` data set names to qualify field names in your Transformation Language expressions. For example:

```plaintext
output-low-grades.field-a = input-grades.field-a + 10;
output-high-grades.field-a = output-low-grades.field-a - 10;
```

Field names that are not qualified by a data set name always default to output data set 0. It is good practice to use the `inputname` data set name to qualify input fields in expressions, and use the `outputname` dataset name to qualify output fields even though these fields have unique names among all data sets. The Transformation Language does not attempt to determine if an unqualified, but unique, name exists in another data set.

The `inputname` and `outputname` statements must appear first in your Transformation Language code. See the Transformation Language section of “Example 2: Student-Score Distribution With a Letter Grade Added to Example 1” on page 21-62 for an example.

**Data types and record fields**

The Transformation Language supports all legal Orchestrate schemas and all record types. Table 67 lists the simple field types. The complex field types follow that table.

Input and output fields can only be defined within the input/ output schemas. You must define them using the operator options, not through transformation expressions. Refer to “Syntax and Options” on page 21-5 for the details of the `transform` operator options.

You can reference input and output data set fields by name. Use the normal Orchestrate dot notation (for example, `s.field1`) for references to subrecord fields. Note that language keywords are not reserved, so field names can be the same as keywords if they are qualified by data set names, `in0.fielda`.

Fields may appear in expressions. Fields that appear on the left side of an assignment statement must be output fields. New values may not be assigned to input fields.
The fieldtype, or data type of a field, can be any legal Orchestrate data type. Fieldtypes can be simple or complex. Table 67 lists the simple field types. The complex field types follow.

Table 67  Simple Fieldtypes

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Forms</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>int8, int16, int32, int64</td>
<td>1, 2, 4, and 8-byte signed integers</td>
</tr>
<tr>
<td></td>
<td>uint8, uint16, uint32, uint64</td>
<td>1, 2, 4, and 8-byte unsigned integers</td>
</tr>
<tr>
<td>floating Point</td>
<td>sfloat</td>
<td>Single-precision floating point</td>
</tr>
<tr>
<td></td>
<td>dfloat</td>
<td>Double-precision floating point</td>
</tr>
<tr>
<td>string</td>
<td>string</td>
<td>Variable-length string</td>
</tr>
<tr>
<td></td>
<td>string[max=n_codepoint_units]</td>
<td>Variable-length string with upper bound on length</td>
</tr>
<tr>
<td></td>
<td>string[n_codepoint_units]</td>
<td>Fixed-length string</td>
</tr>
<tr>
<td>ustring</td>
<td>ustring</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ustring[max=n_codepoint_units]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ustring[n_codepoint_units]</td>
<td></td>
</tr>
<tr>
<td>decimal</td>
<td>decimal[p]</td>
<td>Decimal value with p (precision) digits. p must be between 1 and 255 inclusive.</td>
</tr>
<tr>
<td></td>
<td>decimal[p, s]</td>
<td>Decimal value with p digits and s (scale) digits to the right of the decimal point. p must be between 1 and 255 inclusive, and s must be between 0 and p inclusive.</td>
</tr>
<tr>
<td>date and time</td>
<td>date</td>
<td>Date with year, month, and day</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>Time with one second resolution</td>
</tr>
<tr>
<td></td>
<td>time[microseconds]</td>
<td>Time with one microsecond resolution</td>
</tr>
<tr>
<td></td>
<td>timestamp</td>
<td>Date/time with one second resolution</td>
</tr>
<tr>
<td></td>
<td>timestamp[microseconds]</td>
<td>Date/time with one microsecond resolution</td>
</tr>
</tbody>
</table>
Orchestrate supports the following complex field types:

- vector fields
- subrecord fields
- tagged fields.

**Note** Note that tagged fields cannot be used in expressions; they can only be transferred from input to output.

For descriptions of these field types see the Orchestrate 7.0 User Guide.

### Local Variables

Local variables are used for storage apart from input and output records. You must declare and initialize them before use within your transformation expressions.

The scope of local variables differs depending on which code segment defines them:

- Local variables defined within the `global` and `initialize` code segments can be accessed before, during, and after record processing.
- Local variables defined in the `mainloop` code segment are only accessible for the current record being processed.
- Local variables defined in the `finish` code segment are only accessible after all records have been processed.
Local variables can represent any of the simple value types:

- `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`
- `sfloat`, `dfloat`
- `decimal`
- `string`
- `date`, `time`, `timestamp`
- `raw`

Declarations are similar to C, as in the following examples:

- `int32 a[100];` declares `a` to be an array of 100 32-bit integers
- `dfloat b;` declares `b` to be a double-precision float
- `string c;` declares `c` to be a variable-length string
- `string[n] e;` declares `e` to be a string of length `n`
- `string[n] f[m];` declares `f` to be an array of `m` strings, each of length `n`
- `decimal[p] g;` declares `g` to be a decimal value with `p` (precision) digits
- `decimal[p, s] h;` declares `h` to be a decimal value with `p` (precision) digits and `s` (scale) digits to the right of the decimal

You cannot initialize variables as part of the declaration. They can only be initialized on a separate line. For example:

```c
int32 a;
a = 0;
```

The result is uncertain if a local variable is used without being initialized.

There are no local variable pointers or structures, but you can use arrays.

### Expressions

The Transformation Language supports standard C expressions, with the usual operator precedence and use of parentheses for grouping. It also supports field names as described in "Data types and record fields" on page 21-18, where the field name is specified in the schema for the data set.

### Language Elements

The Transformation Language supports the following elements:

- Integer, character, floating point, and string constants
- Local variables
- Field names
• Arithmetic operators
• Function calls
• Flow control
• Record processing control
• Code segmentation
• Data set name specification

Note that there are no date, time, or timestamp constants.

Operators

The Transformation Language supports several unary operators, which all apply only to simple value types.

Table 68  **Unary Operators**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Applies to</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>One’s complement</td>
<td>Integer</td>
<td>( a ) returns an integer with the value of each bit reversed</td>
</tr>
<tr>
<td>!</td>
<td>Complement</td>
<td>Integer</td>
<td>( ! a ) returns 1 if ( a ) is 0; otherwise returns 0</td>
</tr>
<tr>
<td>+</td>
<td>Unary plus</td>
<td>Numeric</td>
<td>( + a ) returns ( a )</td>
</tr>
<tr>
<td>-</td>
<td>Unary minus</td>
<td>Numeric</td>
<td>( - a ) returns the negative of ( a )</td>
</tr>
<tr>
<td>++</td>
<td>Incrementation operator</td>
<td>Integer</td>
<td>( a++ ) or ( ++a ) returns ( a + 1 )</td>
</tr>
<tr>
<td>--</td>
<td>Decrementation operator</td>
<td>Integer</td>
<td>( a-- ) or ( --a ) returns ( a - 1 ).</td>
</tr>
</tbody>
</table>

The Transformation Language supports a number of binary operators, and one ternary operator.

Table 69  **Binary and Ternary Operators**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Applies to</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>Numeric</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>Numeric</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Numeric</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Numeric</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>Modulo</td>
<td>Integers</td>
<td>( a % b ) returns the remainder when ( a ) is divided by ( b )</td>
</tr>
</tbody>
</table>

Orchestrate 7.0 Operators Reference
The transform Operator

Chapter 21

The Transformation Language

Chapter 21

The Transform Operator

The Transformation Language

21

23

Orchestrate 7.0 Operators Reference

Table 69  Binary and Ternary Operators (continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Applies to</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;</td>
<td>Left shift Integer</td>
<td>a &lt;&lt; b</td>
<td>returns a left-shifted b-bit positions</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Right shift Integer</td>
<td>a &gt;&gt; b</td>
<td>returns a right-shifted b-bit positions</td>
</tr>
<tr>
<td>==</td>
<td>Equals Any; a and b must be numeric or of the same data type</td>
<td>a == b</td>
<td>returns 1 (true) if a equals b and 0 (false) otherwise</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than Same as ==.</td>
<td>a &lt; b</td>
<td>returns 1 if a is less than b and 0 otherwise. (See the note below the table)</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than Same as ==</td>
<td>a &gt; b</td>
<td>returns 1 if a is greater than b and 0 otherwise. (See the note below the table)</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to Same as ==</td>
<td>a &lt;= b</td>
<td>returns 1 if a &lt; b or a == b, and 0 otherwise. (See the note below the table)</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to Same as ==</td>
<td>a &gt;= b</td>
<td>returns 1 if a &gt; b or a == b, and 0 otherwise. (See the note below the table)</td>
</tr>
<tr>
<td>!=</td>
<td>Not equals Same as ==</td>
<td>a != b</td>
<td>returns 1 if a is not equal to b, and 0 otherwise.</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise exclusive OR Integer</td>
<td>a ^ b</td>
<td>returns an integer with bit value 1 in each bit position where the bit values of a and b differ, and a bit value of 0 otherwise.</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND Integer</td>
<td>a &amp; b</td>
<td>returns an integer with bit value 0 in each bit position where the bit values of a and b are both 1, and a bit value of 0 otherwise.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise (inclusive) OR Integer</td>
<td>a</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND Any; a and b must be numeric or of the same data type</td>
<td>a &amp;&amp; b</td>
<td>returns 0 if either a == 0 or b == 0 (or both), and 1 otherwise.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Logical OR Any; a and b must be numeric or of the same data type</td>
</tr>
<tr>
<td>+</td>
<td>Concatenation String</td>
<td>a + b</td>
<td>returns the string consisting of substring a followed by substring b.</td>
</tr>
</tbody>
</table>

Orchestrate 7.0 Operators Reference
Table 69  Binary and Ternary Operators (continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Applies to</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>? :</td>
<td>Assignment</td>
<td>Any scalar; a and b must be numeric, numeric strings, or of the same data type</td>
<td>a = b places the value of b into a. Also, you can use &quot;=&quot; to do default conversions among integers, floats, decimals, and numeric strings.</td>
</tr>
</tbody>
</table>

**Note:** For the <, >, <=, and >= operators, if a and b are strings, lexicographic order is used. If a and b are date, time, or timestamp, temporal order is used.

The expression a * b * c evaluates as (a * b) * c. We describe this by saying that multiplication has left to right **associativity.** The expression a + b * c evaluates as a + (b * c). We describe this by saying multiplication has higher **precedence** than addition. **Table 70** describes the precedence and associativity of the Transformation Language operators. Operators listed in the same row of the table have the same precedence, and you use parentheses to force a particular order of evaluation. Operators in a higher row have a higher order of precedence than operators in a lower row.

Table 70  Precedence and Associativity of Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ) [ ]</td>
<td>left to right</td>
</tr>
<tr>
<td>! ~ ++ - -</td>
<td>right to left</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
</tr>
<tr>
<td>+ - (binary)</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td>^</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conditional Branching

The Transformation Language provides facilities for conditional branching. The following sections describe constructs available for conditional branching.

**if ... else**

\[
\text{if (expression) \ statement1 \ else \ statement2;}
\]

If expression evaluates to a non-zero value (true) then statement1 is executed. If expression evaluates to 0 (false) then statement2 is executed. Both statement1 and statement2 can be compound statements.

You can omit else statement2. In that case, if expression evaluates to 0 the if statement has no effect.

Sample usage:

\[
\text{if (a < b) \ abs_difference = b - a;}
\text{else \ abs_difference = a - b;}
\]

This code sets abs_difference to the absolute value of b - a.

**For Loop**

\[
\text{for (expression1; expression2; expression3) \ statement;}
\]

The order of execution is:

1. expression1. It is evaluated only once to initialize the loop variable.
2. expression2. If it evaluates to false, the loop terminates; otherwise, these expressions are executed in order:

   \[
   \text{statement}
   \text{expression3}
   \]

   Control then returns to 2.
A sample usage is:

```
sum = 0;
sum_squares = 0;
for (i = 1; i < n; i++)
{
    sum = sum + 1;
    sum_squares = sum_squares + i*i;
}
```

This code sets `sum` to the sum of the first `n` integers and `sum_squares` to the sum of the squares of the first `n` integers.

**While Loop**

```
while (expression) statement;
```

In a while loop, statement, which may be a compound statement, is executed repeatedly as long as expression evaluates to true. A sample usage is:

```
sum = 0;
i = 0;
while ((a[i] >= 0) && (i < n))
{
    sum = sum + a[i];
i++;
}
```

This evaluates the sum of the array elements `a[0]` through `a[n-1]`, or until a negative array element is encountered.

**Break**

The break command causes a for or while loop to exit immediately. For example, the following code does the same thing as the while loop shown immediately above:

```
sum = 0;
for (i = 0; i < n; i++)
{
    if (a[i] >= 0)
        sum = sum + a[i];
    else
        break;
}
```

**Continue**

The continue command is related to the break command, but used less often. It causes control to jump to the top of the loop. In the while loop, the test part is executed immediately. In a for loop, control passes to the increment step. If you want to sum all positive array entries in the array `a[n]`, you can use the continue statement as follows:

```
sum = 0;
for (i = 0; i < n; i++)
{
```
if (a[i] <= 0)  
    continue;  
    sum = sum + a[i];  
}

This example could easily be written using an else statement rather than a continue statement. The continue statement is most useful when the part of the loop that follows is complicated, to avoid nesting the program too deeply.

**Built-in Functions**

This section defines functions that are provided by the Transformation Language. It is presented in a series of tables that deal with data transformation functions of the following types:

- “Data Conversion Functions” on page 21-27
  - “date Field Functions” on page 21-27
  - “decimal and float Field Functions” on page 21-31
  - “raw Field Functions” on page 21-34
  - “time and timestamp Field Functions” on page 21-34
  - “Null Handling Functions” on page 21-38
- “Mathematical Functions” on page 21-39
- “string Field Functions” on page 21-41
- “ustring Field Functions” on page 21-48
- “Bit Manipulation Functions” on page 21-54
- “Job Monitoring Functions” on page 21-55
- “Miscellaneous Functions” on page 21-57

When a function generates an output value, it returns the result. For functions with optional arguments, simply omit the optional argument to accept the default value. Default conversions among integer, float, decimal, and numeric string types are supported in the input arguments and the return value of the function. All integers can be signed or unsigned.

The `transform` operator has default NULL handling at the record-level with individual field `overrides`. Options can be entered at the record level or the field level.

**Data Conversion Functions**

`date Field Functions`

Table 71 lists the transformation functions of date fields.
Orchestrate performs no automatic type conversion of date fields. Either an input data set must match the operator interface or you must effect a type conversion by means of the **transform** or **modify** operator.

A date conversion to or from a numeric field can be specified with any Orchestrate numeric data type. Orchestrate performs the necessary modifications and either translates a numeric field to the source data type or translates a conversion result to the numeric data type of the destination. For example, you can use the transformation function `month_day_from_date()` to convert a date to an `int8`, or to an `int16`, `int32`, `dfloat`, etc.

**date format**

The default format of the date contained in the string is `yyyy-mm-dd`. However, you can specify an optional format string that defines another format. The format string requires that you provide enough information for Orchestrate to determine a complete date (either day, month, and year, or year and day of year). The date components of a source string (date, month, and year) must be zero-padded to the character length specified by the format string. Orchestrate zero pads the date components of a destination string to the specified length.

The possible format components are:

- `%dd`: A two digit day.
- `%mm`: A two digit month.
- `%yy`: A two digit year derived from a year cutoff of 1900.
- `%year_cutoffyy`: A two digit year derived from `yy` and the specified year cutoff.
  
  (For example, if you specify the year format as `%2000yy`, two-digit values which represent years from 2000-2099 are imported and exported.)
- `%yyyy`: A four digit year.
- `%dd`: Day of year in three digit form (range of 1 - 366)

The default date format is as follows:

```
%yyyy-%mm-%dd
```

When you specify a date format string, prefix each component with the percent symbol (`%`). Separate the string's components with any character except the percent sign (`%`).

For example, the format string `%mm/ %dd/ %yyyy` specifies that slashes separate the string's date components; the format `%dd %yy` specifies that the string stores the date as a value from 1 to 366, derives the year from the current year cutoff of 1900, and separates the two components with a dash (`-`).
**uformat date Format**

The uformat date format provides support for international components in date fields. It's syntax is:

```
String%macroString%macroString%macroString
```

where `%macro` is a date formatting macro such as `%mmm` for a 3-character English month. Only the `String` components of `uformat` can include multi-byte Unicode characters.

**Important** Any argument that has to be double quoted cannot be a field name or a local variable. An argument must have the data format of its type.

### Table 71 date Field Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>date date_from_days_since</code></td>
<td>Returns <code>date</code> by adding the given integer to the baseline <code>date</code>. Converting an integer field into a date by adding the integer to the specified base date. The <code>date</code> must be in the format <code>yyyy-mm-dd</code> and must be either double quoted or a variable.</td>
</tr>
<tr>
<td><code>date date_from_julian_day(uint32)</code></td>
<td>Returns the <code>date</code> given a Julian day.</td>
</tr>
<tr>
<td><code>date date_from_string</code></td>
<td>Returns a <code>date</code> from the given string formatted in the optional format specification. By default the string format is <code>yyyy-mm-dd</code>. For format descriptions, see “date format” on page 21-28 and “uformat date Format” on page 21-29.</td>
</tr>
<tr>
<td><code>date date_from_ustring</code></td>
<td>Returns a <code>date</code> from the given <code>ustring</code> formatted in the optional format specification. By default the <code>ustring</code> format is <code>yyyy-mm-dd</code>. For format descriptions, see “date format” on page 21-28 and “uformat date Format” on page 21-29.</td>
</tr>
<tr>
<td><code>string string_from_date</code></td>
<td>Converts the <code>date</code> to a string representation using the given format specification. By default the <code>ustring</code> format is <code>yyyy-mm-dd</code>. For format descriptions, see “date format” on page 21-28 and “uformat date Format” on page 21-29.</td>
</tr>
</tbody>
</table>
### Table 71: date Field Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`ustring ustring_from_date (date, ‘date_format’</td>
<td>uformat)`</td>
</tr>
<tr>
<td><code>date date_from_timestamp(timestamp)</code></td>
<td>Returns the date from the given timestamp.</td>
</tr>
<tr>
<td>`int32 days_since_from_date (date, ‘source_date’</td>
<td>format_variable)`</td>
</tr>
<tr>
<td><code>uint32 julian_day_from_date(date)</code></td>
<td>Returns a Julian date given the date.</td>
</tr>
<tr>
<td><code>int8 month_day_from_date(date)</code></td>
<td>Returns the day of the month given the date. For example, the date 07-23-2001 returns 23.</td>
</tr>
<tr>
<td><code>int8 month_from_date(date)</code></td>
<td>Returns the month from the given date. For example, the date 07-23-2001 returns 7.</td>
</tr>
<tr>
<td>`date next_weekday_from_date (date, ‘day’</td>
<td>format_variable)`</td>
</tr>
<tr>
<td>`date previous_weekday_from_date (date, ‘day’</td>
<td>format_variable)`</td>
</tr>
</tbody>
</table>
Table 71  date Field Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int8 weekday_from_date</code> (date, &quot;origin_day&quot;</td>
<td>Returns the day of the week from date.</td>
</tr>
<tr>
<td></td>
<td>format_variable)</td>
</tr>
<tr>
<td><code>int16 year_day_from_date</code> (date)</td>
<td>Returns the day of the year (1-366) from date.</td>
</tr>
<tr>
<td><code>int16 year_from_date</code> (date)</td>
<td>Returns the year from date. For example, the date 07-23-2001 returns 2001.</td>
</tr>
<tr>
<td><code>int8 year_week_from_date</code> (date)</td>
<td>Returns the week of the year from date. For example, the date 07-23-2001 returns 30.</td>
</tr>
</tbody>
</table>

decimal and float Field Functions

Table 72 lists the transformation functions for decimal and float fields.

You can do the following transformations using the decimal and float field functions.

- Assign a decimal to an integer or float or numeric string, or compare a decimal to an integer or float or numeric string.

- Specify an optional `fix_zero` argument (int8) to cause a decimal field containing all zeros to be treated as a valid zero.

- Optionally specify a value for the rounding type (r_type) for many conversions. The values of r_type are:
  - **ceil**: Round the source field toward positive infinity. This mode corresponds to the IEEE 754 Round Up mode.
    
    **Examples**: 1.4 -> 2, -1.6 -> -1
  
  - **floor**: Round the source field toward negative infinity. This mode corresponds to the IEEE 754 Round Down mode.
    
    **Examples**: 1.6 -> 1, -1.4 -> -2
  
  - **round_inf**: Round or truncate the source field toward the nearest representable value, breaking ties by rounding positive values toward positive infinity and negative values toward negative infinity. This mode corresponds to the COBOL ROUNDED mode.
    
    **Examples**: 1.4 -> 1, 1.5 -> 2, -1.4 -> -1, -1.5 -> -2
- **trunc_zero (default):** Discard any fractional digits to the right of the right-most fractional digit supported in the destination, regardless of sign. For example, if the destination is an integer, all fractional digits are truncated. If the destination is another decimal with a smaller scale, round or truncate to the scale size of the destination decimal. This mode corresponds to the COBOL INTEGER-PART function.

**Examples:** 1.6 -> 1, -1.6 -> -1

### Table 72  decimal and float Field Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal decimal_from_decimal (decimal, &quot;r_type&quot;</td>
<td>Returns decimal in decimal representation, changing the precision and scale according to the returned type. The rounding type, r_type, may be <strong>ceil</strong>, <strong>floor</strong>, <strong>round_inf</strong>, or <strong>trunc_zero</strong> as described above this table. The default rtype is <strong>trunc_zero</strong>.</td>
</tr>
<tr>
<td>decimal decimal_from_dfloat (dfloat, &quot;r_type&quot;</td>
<td>Returns dfloat in decimal representation. The rounding type, r_type, may be <strong>ceil</strong>, <strong>floor</strong>, <strong>round_inf</strong>, or <strong>trunc_zero</strong> as described above this table. The default is <strong>trunc_zero</strong>.</td>
</tr>
<tr>
<td>decimal decimal_from_string (string, &quot;r_type&quot;</td>
<td>Returns string in decimal representation. The rounding type, r_type, may be <strong>ceil</strong>, <strong>floor</strong>, <strong>round_inf</strong>, or <strong>trunc_zero</strong> as described above this table. The default is <strong>trunc_zero</strong>.</td>
</tr>
<tr>
<td>decimal decimal_from_ustring (ustring, &quot;r_type&quot;</td>
<td>Returns ustring in decimal representation. The rounding type, r_type, may be <strong>ceil</strong>, <strong>floor</strong>, <strong>round_inf</strong>, or <strong>trunc_zero</strong> as described above this table. The default is <strong>trunc_zero</strong>.</td>
</tr>
<tr>
<td>dfloat dfloat_from_decimal (decimal, &quot;fix-zero&quot;</td>
<td>Returns decimal in dfloat representation.</td>
</tr>
<tr>
<td>int32 int32_from_decimal (decimal, &quot;r_type fix_zero&quot;)</td>
<td>Returns int32 in decimal representation. The rounding type, r_type, may be <strong>ceil</strong>, <strong>floor</strong>, <strong>round_inf</strong>, or <strong>trunc_zero</strong> as described above this table. The default is <strong>trunc_zero</strong>.</td>
</tr>
<tr>
<td>int64 int64_from_decimal (decimal, &quot;r_type fix_zero&quot;)</td>
<td>Returns int64 in decimal representation. The rounding type, r_type, may be <strong>ceil</strong>, <strong>floor</strong>, <strong>round_inf</strong>, or <strong>trunc_zero</strong> as described above this table. The default is <strong>trunc_zero</strong>.</td>
</tr>
<tr>
<td>uint64 uint64_from_decimal (decimal, &quot;r_type fix_zero&quot;)</td>
<td>Returns uint64 in decimal representation. The rounding type, r_type, may be <strong>ceil</strong>, <strong>floor</strong>, <strong>round_inf</strong>, or <strong>trunc_zero</strong> as described above this table. The default is <strong>trunc_zero</strong>.</td>
</tr>
</tbody>
</table>
### Table 72  
**Decimal and Float Field Functions (continued)**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string string_from_decimal</code> (decimal, &quot;fix_zero suppress_zero&quot;)</td>
<td>Returns string in decimal representation. fix_zero causes a decimal field containing all zeros to be treated as a valid zero. suppress_zero argument specifies that the returned ustring value will have no leading or trailing zeros. Examples: 000.100 -&gt; 0.1; 001.000 -&gt; 1; -001.100 -&gt; -1.1</td>
</tr>
<tr>
<td><code>ustring ustring_from_decimal</code> (decimal, &quot;fix_zero suppress_zero&quot;</td>
<td>Returns ustring in decimal representation. fix_zero causes a decimal field containing all zeros to be treated as a valid zero. suppress_zero argument specifies that the returned ustring value will have no leading or trailing zeros. Examples: 000.100 -&gt; 0.1; 001.000 -&gt; 1; -001.100 -&gt; -1.1</td>
</tr>
<tr>
<td><code>string string_from_decimal</code> (decimal, &quot;fix_zero suppress_zero&quot;</td>
<td>Returns string in decimal representation. fix_zero causes a decimal field containing all zeros to be treated as a valid zero. suppress_zero argument specifies that the returned ustring value will have no leading or trailing zeros. Examples: 000.100 -&gt; 0.1; 001.000 -&gt; 1; -001.100 -&gt; -1.1</td>
</tr>
<tr>
<td><code>dfloat mantissa_from_dfloat</code> (dfloat)</td>
<td>Returns the mantissa (the digits right of the decimal point) from dfloat.</td>
</tr>
<tr>
<td><code>dfloat mantissa_from_decimal</code> (decimal)</td>
<td>Returns the mantissa (the digits right of the decimal point) from decimal.</td>
</tr>
</tbody>
</table>
**raw Field Functions**

Use the raw field functions to transform a string into a raw data type and to determine the length of a raw value.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>raw raw_from_string(string)</code></td>
<td>Returns string in raw representation.</td>
</tr>
<tr>
<td><code>raw u_raw_from_string(ustring)</code></td>
<td>Returns ustring in raw representation.</td>
</tr>
<tr>
<td><code>int32 raw_length(raw)</code></td>
<td>Returns the length of the raw field.</td>
</tr>
</tbody>
</table>

**time and timestamp Field Functions**

Table 74 lists the transformation functions for time and timestamp fields.

Orchestrate performs no automatic conversions to or from the time and timestamp data types. You must use the `modify` or `transform` operator if you want to convert a source or destination field. Most field conversions extract a portion of the time, such as hours or minutes, and write it into a destination field.

Time conversion to a numeric field can be used with any Orchestrate numeric data type. Orchestrate performs the necessary modifications to translate a conversion result to the numeric data type of the destination. For example, you can use the transformation function `hours_from_time()` to convert a time to an `int8`, or to an `int16`, `int32`, `dfloat`, and so on.

The `string_from_time()` and `time_from_string()` conversion functions take a format as a parameter of the conversion. The default format of the time in the string is `hh:nn:ss`. However, you can specify an optional format string defining another time format. The format string must contain a specification for hours, minutes, and seconds.

**uformat time Format**

The `uformat` date format provides support for international components in time fields. It’s syntax is:

```
String%macroString%macroString%macroString
```

where `%macro` is a time formatting macro such as `%hh` for a two-digit hour. See “time Format” below for a description of the date format macros. Only the String components of `uformat` can include multi-byte Unicode characters.
**uformat timestamp Format**

This format is a concatenation of the uformat date and uformat time formats which are described in “date format” on page 21-28 and “uformat time Format” on page 21-34. The order of the formats does not matter, but the two formats cannot be mixed.

**time Format**

The time components of a source string (hours, minutes, and seconds) must be zero padded to the character length specified by the format string; Orchestrate zero pads the time components of a destination string to the specified length.

The components of the format string are:

- `%hh`: A two-digit hours component.
- `%nn`: A two-digit minutes component.
- `%ss`: A two-digit seconds component.
- `%ss.n`: A two-digit seconds component plus a fractional part, where `n` is the number of fractional digits with a maximum value of 6. If `n` is 0, no decimal point is printed as part of the seconds component. Trailing zeros are not suppressed. For example, a format string of `%hh:%nn:%ss.4` indicates that the string contains the seconds to four decimal places.

The default time format is as follows:

```
%hh:%nn:%ss
```

When you specify a time format string, prefix each component with the percent symbol (%). Separate the string’s components with any character except the percent sign (%).

**Table 74  time Field and timestamp Field Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8 hours_from_time(time)</td>
<td>Returns the hour portion of the given time.</td>
</tr>
<tr>
<td>int32 microseconds_from_time(time)</td>
<td>Returns the number of microseconds from the given time.</td>
</tr>
<tr>
<td>dfloat midnight_seconds_from_time(time)</td>
<td>Returns the number of seconds from midnight to time.</td>
</tr>
<tr>
<td>int8 minutes_from_time(time)</td>
<td>Returns the number of minutes from time.</td>
</tr>
<tr>
<td>dfloat seconds_from_time(time)</td>
<td>Returns the number of seconds from time.</td>
</tr>
</tbody>
</table>
Table 74  time Field and timestamp Field Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dfloat seconds_since_from_timestamp(timestamp, 'source_timestamp_string'</td>
<td>Returns the number of seconds from timestamp to the base timestamp, or optionally the second timestamp argument for the number of seconds between timestamps. The source_timestamp_string argument must be double quoted or be a variable.</td>
</tr>
<tr>
<td>time time_from_midnight_seconds(dfloat)</td>
<td>Returns the time given the number of seconds (dfloat) since midnight.</td>
</tr>
<tr>
<td>time time_from_string(string, 'time_format'</td>
<td>Returns a time representation of string using the optional time_format, uformat, or format_variable. By default, the time format is hh:mm:ss. For format descriptions, see “time Format” on page 21-35 and “uformat time Format” on page 21-34.</td>
</tr>
<tr>
<td>time time_from_ustring(ustring, 'time_format'</td>
<td>Returns a time representation of ustring using the optional time_format, uformat, or format_variable specification. By default, the time format is hh:mm:ss. For format descriptions, see “time Format” on page 21-35 and “uformat time Format” on page 21-34.</td>
</tr>
<tr>
<td>string string_from_time(time, &quot;time_format&quot;</td>
<td>Returns a string from time. The format argument is optional. The default time format is hh:mm:ss. For format descriptions, see “time Format” on page 21-35 and “uformat time Format” on page 21-34.</td>
</tr>
<tr>
<td>string string_from_time(time, &quot;time_format&quot;</td>
<td>Returns a ustring from time. The format argument is optional. The default time format is hh:mm:ss. For format descriptions, see “time Format” on page 21-35 and “uformat time Format” on page 21-34.</td>
</tr>
<tr>
<td>time time_from_timestamp(timestamp)</td>
<td>Returns the time from timestamp.</td>
</tr>
<tr>
<td>date date_from_timestamp(timestamp)</td>
<td>Returns the date from the given timestamp.</td>
</tr>
<tr>
<td>timestamp timestamp_from_date_time(date, time)</td>
<td>Returns a timestamp from date and time. The date specifies the date portion (yyyy:nn:dd) of the timestamp. The time argument specifies the time to be used when building the timestamp. The time argument must be in the hh:mm:ss format.</td>
</tr>
</tbody>
</table>
Table 74  time Field and timestamp Field Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp timestamp_from_seconds_since</td>
<td>Returns the timestamp from the number of seconds (dfloat) from the base timestamp or the original_timestamp_string argument. The original_timestamp_string must be double quoted or be a variable.</td>
</tr>
<tr>
<td>timestamp timestamp_from_string</td>
<td>Returns a timestamp from string, in the optional timestamp_format, uformat, or format_variable. The timestamp_format must be double quoted or be a variable. The default format is yyyy-mm-dd hh:mm:ss. uformat is described in “uformat timestamp Format” on page 21-35.</td>
</tr>
<tr>
<td>timestamp timestamp_from_ustring</td>
<td>Returns a timestamp from ustring, in the optional format specification. The timestamp_format must be a double quoted string, a uformat, or a variable. The default format is yyyy-mm-dd hh:mm:ss. uformat is described in “uformat timestamp Format” on page 21-35.</td>
</tr>
<tr>
<td>string string_from_timestamp</td>
<td>Returns a string from timestamp. The formatting specification is optional. The default format is yyyy-mm-dd hh:mm:ss.</td>
</tr>
<tr>
<td>ustring ustring_from_timestamp</td>
<td>Returns a ustring from timestamp. The formatting specification is optional. The default format is yyyy-mm-dd hh:mm:ss.</td>
</tr>
<tr>
<td>timestamp timestamp_from_time(time, &quot;time_format&quot;)</td>
<td>Returns a timestamp from time. For format descriptions, see “time Format” on page 21-35 and “uformat time Format” on page 21-34</td>
</tr>
<tr>
<td>date date_from_timestamp(timestamp)</td>
<td>Returns the date from the given timestamp.</td>
</tr>
<tr>
<td>timestamp timestamp_from_time(int32)</td>
<td>Returns a timestamp from the given UNIX time_t representation (int32).</td>
</tr>
<tr>
<td>int32 timet_from_timestamp(timestamp)</td>
<td>Returns the UNIX time_t representation of timestamp.</td>
</tr>
</tbody>
</table>
Null Handling Functions

Table 75 lists the transformation functions for NULL handling.

All Orchestrate data types support nulls. As part of processing a record, an operator can detect a null and take the appropriate action, for example, it can omit the null field from a calculation or signal an error condition.

Orchestrate represents nulls in two ways.

- It allocates a single bit to mark a field as null. This type of representation is called an out-of-band null.
- It designates a specific field value to indicate a null, for example a numeric field’s most negative possible value. This type of representation is called an in-band null. In-band null representation can be disadvantageous because you must reserve a field value for nulls and this value cannot be treated as valid data elsewhere.

The null-handling functions can change a null representation from an out-of-band null to an in-band null and from an in-band null to an out-of-band null.

Table 75  Null Handling Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>destination_field handle_null (source_field, value)</td>
<td>Change the source_field NULL representations from out-of-band representation to an in-band representation. The value field assigns the value that corresponds to NULL.</td>
</tr>
<tr>
<td>destination_field make_null (source_field, value)</td>
<td>Changes source_field NULL representation from in-band NULL representation to out-of-band. The value field allows multiple valid NULL values to be inputted as arguments.</td>
</tr>
<tr>
<td>int8 notnull(source_field)</td>
<td>Returns 1 if source_field is not NULL, otherwise returns 0.</td>
</tr>
<tr>
<td>int8 null(source_field)</td>
<td>Returns 1 if source_field is NULL, otherwise returns 0.</td>
</tr>
<tr>
<td>set_null()</td>
<td>This function is used with &quot;=&quot; to set the left side output field, when it is nullable, to null. For example: a_field = set_null();</td>
</tr>
<tr>
<td>int8 is_dfloat_inband_null (dfloat)</td>
<td>Returns 1 if dfloat is an inband null; otherwise it returns 0.</td>
</tr>
<tr>
<td>int8 is_int16_inband_null (int16)</td>
<td>Returns 1 if int16 is an inband null; otherwise it returns 0.</td>
</tr>
<tr>
<td>int8 is_int32_inband_null (int32)</td>
<td>Returns 1 if int32 is an inband null; otherwise it returns 0.</td>
</tr>
</tbody>
</table>
Table 75  **Null Handling Functions (continued)**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8 is_int64_inband_null (int64)</td>
<td>Returns 1 if int64 is an inband null; otherwise it returns 0.</td>
</tr>
<tr>
<td>int8 is_sfloat_inband_null (sfloat)</td>
<td>Returns 1 if sfloat is an inband null; otherwise it returns 0.</td>
</tr>
<tr>
<td>int8 is_string_inband_null (string)</td>
<td>Returns 1 if string is an inband null; otherwise it returns 0.</td>
</tr>
<tr>
<td>int8 u_is_string_inband_null (ustring)</td>
<td>Returns 1 if ustring is an inband null; otherwise it returns 0.</td>
</tr>
</tbody>
</table>

**Note**  Null-handling functions cannot be used for subrecord fields.

**Mathematical Functions**

Table 76  **Mathematical Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 abs(int32)</td>
<td>Returns the absolute value of int32.</td>
</tr>
<tr>
<td>dfloat acos(dfloat)</td>
<td>Returns the principal value of the arc cosine of dfloat.</td>
</tr>
<tr>
<td>dfloat asin(dfloat)</td>
<td>Returns the principal value of the arc sine of dfloat.</td>
</tr>
<tr>
<td>dfloat atan(dfloat)</td>
<td>Returns the principal value of the arc tangent of dfloat.</td>
</tr>
<tr>
<td>dfloat atan2(dfloat, dfloat)</td>
<td>Returns the principal value of the arc tangent of y/ x (where y is the first argument).</td>
</tr>
<tr>
<td>int ceil(decimal)</td>
<td>Returns the smallest integer value greater than or equal to decimal.</td>
</tr>
<tr>
<td>dfloat cos(dfloat)</td>
<td>Returns the cosine of the given angle (dfloat) expressed in radians.</td>
</tr>
<tr>
<td>dfloat cosh(dfloat)</td>
<td>Returns the hyperbolic cosine of dfloat.</td>
</tr>
<tr>
<td>dfloat exp(dfloat)</td>
<td>Returns the exponential of dfloat.</td>
</tr>
<tr>
<td>dfloat fabs(dfloat)</td>
<td>Returns the absolute value of dfloat.</td>
</tr>
<tr>
<td>int floor(decimal)</td>
<td>Returns the largest integer value less than or equal to decimal.</td>
</tr>
<tr>
<td>dfloat ldexp(dfloat, int32)</td>
<td>Reconstructs dfloat out of the mantissa and exponent of int32.</td>
</tr>
</tbody>
</table>
### Mathematical Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint64 llabs(int64)</code></td>
<td>Returns the absolute value of <code>int64</code>.</td>
</tr>
<tr>
<td><code>dfloat log(dfloat)</code></td>
<td>Returns the natural (base e) logarithm of <code>dfloat</code>.</td>
</tr>
<tr>
<td><code>dfloat log10(dfloat)</code></td>
<td>Returns the logarithm to the base 10 of <code>dfloat</code>.</td>
</tr>
<tr>
<td><code>int32 max(int32, int32)</code></td>
<td>Returns the larger of the two integers.</td>
</tr>
<tr>
<td><code>int32 min(int32, int32)</code></td>
<td>Returns the smaller of the two integers.</td>
</tr>
<tr>
<td><code>dfloat pow(dfloat, dfloat)</code></td>
<td>Returns the result of raising <code>x</code> (the first argument) to the power <code>y</code> (the second argument).</td>
</tr>
<tr>
<td><code>uint32 rand()</code></td>
<td>Returns a pseudo-random integer between 0 and $2^{32} - 1$. The function uses a multiplicative congruential random-number generator with period $2^{32}$. See the UNIX man page for <code>rand</code> for more details.</td>
</tr>
<tr>
<td><code>uint32 random()</code></td>
<td>Returns a random integer between 0 and $2^{31} - 1$. The function uses a nonlinear additive feedback random-number generator employing a default state array size of 31 long integers to return successive pseudo-random numbers. The period of this random-number generator is approximately $16 \times (2^{31} - 1)$. Compared with <code>rand</code>, <code>random</code> is slower but more random. See the UNIX man page for <code>random</code> for more details.</td>
</tr>
<tr>
<td><code>dfloat sin(dfloat)</code></td>
<td>Returns the sine of <code>dfloat</code> expressed in radians.</td>
</tr>
<tr>
<td><code>dfloat sinh(dfloat)</code></td>
<td>Returns the hyperbolic sine of <code>dfloat</code>.</td>
</tr>
<tr>
<td><code>dfloat sqrt(dfloat)</code></td>
<td>Returns the square root of <code>dfloat</code>.</td>
</tr>
<tr>
<td><code>int32 quotient_from_dfloat(dfloat1, dfloat2)</code></td>
<td>Returns the value of the quotient after <code>dfloat1</code> is divided by <code>dfloat2</code>.</td>
</tr>
<tr>
<td><code>srand(uint32)</code></td>
<td>Sets a new seed (<code>uint32</code>) for the <code>rand()</code> or <code>srand()</code> random number generator.</td>
</tr>
<tr>
<td><code>srandom(uint32)</code></td>
<td>Sets a random seed for the <code>random()</code> number generator. See the UNIX man page for <code>srandom</code> for more details.</td>
</tr>
<tr>
<td><code>dfloat tan(dfloat)</code></td>
<td>Returns the tangent of the given angle (<code>dfloat</code>) expressed in radians.</td>
</tr>
<tr>
<td><code>dfloat tanh(dfloat)</code></td>
<td>Returns the hyperbolic tangent of <code>dfloat</code>.</td>
</tr>
</tbody>
</table>
string Field Functions

Strings can be assigned (=), compared (==, <, >=, etc.), and concatenated (+) in the Transformation Language. In addition, the functions described in Table 78 below are available for string manipulations, and the functions described in Table 79 are available for ustring manipulations. When a long string is assigned to a short string, the long string is truncated to the length of the short string. The term white space refers to spaces, tabs, and any other blank space.

String Conversions and Lookup Tables

You can construct a string lookup table to use when default conversions do not yield satisfactory results. A string lookup table is a table of two columns and as many rows as are required to perform a conversion to or from a string as shown in Table 77

Table 77  A String Lookup Table

<table>
<thead>
<tr>
<th>Numeric Value</th>
<th>String or Ustring</th>
</tr>
</thead>
<tbody>
<tr>
<td>numVal1</td>
<td>string1</td>
</tr>
<tr>
<td>numVal2</td>
<td>string2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>numVal3</td>
<td>stringN</td>
</tr>
</tbody>
</table>

Each row of the lookup table specifies an association between a 16-bit integer or unsigned 32-bit integer value and a string or ustring. Orchestrate scans the Numeric Value or the String or Ustring column until it encounters the value or string to be translated. The output is the corresponding entry in the row.

The numeric value to be converted may be of the int16 or the uint32 data type. Orchestrate converts strings to values of the int16 or uint32 data type using the same table.

If the input contains a numeric value or string that is not listed in the table, Orchestrate operates as follows:

- If a numeric value is unknown, an empty string is returned by default. However, you can set a default string value to be returned by the string lookup table.
- If a string has no corresponding value, 0 is returned by default. However, you can set a default numeric value to be returned by the string lookup table.

A table definition defines the rows of a string or ustring lookup table and has the following form:

```
{propertyList} ['string' | 'ustring' = value 'string' | 'ustring' = value ...]
```
where:

- `propertyList` is one or more of the following options; the entire list is enclosed in braces and properties are separated by commas if there are more than one:
  - `case_sensitive`: perform a case-sensitive search for matching strings; the default is case-insensitive.
  - `default_value =defVal`: the default numeric value returned for a string that does not match any of the strings in the table.
  - `default_string =defString`: the default string returned for numeric values that do not match any numeric value in the table.
  - `string` or `ustring` specifies a comma-separated list of strings or ustrings associated with `value`; enclose each string or ustring in quotes.
  - `value` specifies a comma-separated list of 16-bit integer values associated with `string` or `ustring`.

Table 78  **string Field Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int8 is_alnum(string)</code></td>
<td>Returns 1 (true) if string consists entirely of alphanumeric characters.</td>
</tr>
<tr>
<td><code>int8 is_alpha(string)</code></td>
<td>Returns 1 (true) if string consists entirely of alphabetic characters.</td>
</tr>
<tr>
<td><code>int8 is_numeric(string)</code></td>
<td>Returns 1 (true) if string consists entirely of numeric characters, including decimal and sign.</td>
</tr>
</tbody>
</table>
int8 is_valid
("type_string", 'value_string')

Returns 1 (true) if value_string is valid according to type_string, including NULL. The type_string argument is required. It must specify an Orchestrate schema data type.

Integer types are checked to ensure the value_string is numeric (signed or unsigned), a whole number, and a valid value (for example, 1024 can not be assigned to an int8 type).

Decimal types are checked to ensure the value_string is numeric (signed or unsigned) and a valid value.

Float types are checked to ensure the value_string is numeric (signed or unsigned) and a valid value (exponent is valid).

String is always valid with the NULL exception below.

For all types, if the field cannot be set to NULL and the string is NULL, 0 (false) is returned.

Date, time, and timestamp types are checked to ensure they are correct, using the optional format argument, and valid values.

Raw cannot be checked since the input is a string.

Table 78  string Field Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int16 lookup_int16_from_string</td>
<td>Returns an integer corresponding to string using table_definition string or variable. See “String Conversions and Lookup Tables” on page 21-41 for more information.</td>
</tr>
<tr>
<td>string lookup_string_from_int16</td>
<td>Returns a string corresponding to int16 using table_definition string or variable. See “String Conversions and Lookup Tables” on page 21-41 for more information.</td>
</tr>
<tr>
<td>string lookup_string_from_uint32</td>
<td>Returns a string corresponding to uint32 using table_definition string or variable. See “String Conversions and Lookup Tables” on page 21-41 for more information.</td>
</tr>
<tr>
<td>uint32 lookup_uint32_from_string</td>
<td>Returns an unsigned integer from string using table_definition string or variable.</td>
</tr>
<tr>
<td>string lower_case(string)</td>
<td>Converts string to lowercase. Non-alphabetic characters are ignored in the transformation.</td>
</tr>
</tbody>
</table>
Table 78  string Field Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string string_from_date (date, &quot;date_format&quot;</td>
<td>Converts date to a string representation using the specified optional</td>
</tr>
<tr>
<td></td>
<td>format_variable</td>
</tr>
<tr>
<td>By default, the date format is yyyy-mm-dd.</td>
<td>For format descriptions, see &quot;date format&quot; on page 21-28 and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>string string_from_decimal (decimal,</td>
<td>Returns a string from decimal.</td>
</tr>
<tr>
<td>&quot;fix_zero suppress_zero&quot;</td>
<td>format_variable)</td>
</tr>
<tr>
<td>fix_zero causes a decimal field containing</td>
<td>suppress_zero argument specifies that the returned ustring will have no</td>
</tr>
<tr>
<td>all zeros to be treated as a valid zero.</td>
<td>leading or trailing zeros. Examples:</td>
</tr>
<tr>
<td>000.100 -&gt; 0.1; 001.000 -&gt; 1; -001.100-&gt;-1.1</td>
<td>The formatting specification is optional.</td>
</tr>
<tr>
<td>string string_from_time (time, &quot;time_format&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>format_variable</td>
</tr>
<tr>
<td>Returns a string from time. The format</td>
<td>optional. For format descriptions, see &quot;time Format&quot; on page 21-35 and</td>
</tr>
<tr>
<td>argument is optional. For format descriptions,</td>
<td>&quot;uformat time Format&quot; on page 21-34.</td>
</tr>
<tr>
<td>string string_from_timestamp (timestamp, &quot;</td>
<td>Returns a string from timestamp. The formatting</td>
</tr>
<tr>
<td>timestamp_format&quot;</td>
<td>format_variable)</td>
</tr>
<tr>
<td>specification is optional. The default format</td>
<td>is yyyy-mm-dd:hh:mm:ss.</td>
</tr>
<tr>
<td>string upper_case (string)</td>
<td>Converts string to uppercase. Non-alphabetic characters are ignored in the</td>
</tr>
<tr>
<td>Returns a string after reducing all</td>
<td>transformation.</td>
</tr>
<tr>
<td>consecutive white space in string to a single</td>
<td></td>
</tr>
<tr>
<td>space.</td>
<td></td>
</tr>
<tr>
<td>string compact_whitespace (string)</td>
<td>Returns a string after reducing all consecutive white space in string to</td>
</tr>
<tr>
<td>Returns a string after removing all leading</td>
<td>a single space.</td>
</tr>
<tr>
<td>and trailing white space.</td>
<td></td>
</tr>
<tr>
<td>string pad_string (string, pad_string,</td>
<td>Returns the string with the pad_string appended to the bounded length</td>
</tr>
<tr>
<td>pad_length)</td>
<td>string for pad_length number of characters. pad_length is an int16.</td>
</tr>
<tr>
<td>When the given string is a variable-length</td>
<td>If the given string is a fixed-length string, this function has no effect.</td>
</tr>
<tr>
<td>string, it defaults to a bounded-length of 1024 characters. If the given string is a fixed-length string, this function has no effect.</td>
<td></td>
</tr>
<tr>
<td>string strip_whitespace (string)</td>
<td>Returns string after stripping all white space in the string.</td>
</tr>
<tr>
<td>Returns string after removing all leading</td>
<td>Returns string after removing all leading and trailing white space.</td>
</tr>
<tr>
<td>and trailing white space.</td>
<td>Returns a string after removing all leading white space.</td>
</tr>
<tr>
<td>Returns a string after removing all trailing</td>
<td>Returns a string after removing all trailing white space.</td>
</tr>
<tr>
<td>white space.</td>
<td></td>
</tr>
</tbody>
</table>
Table 78  string Field Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| int32 string_order_compare (string1, string2, justification) | Returns a numeric value specifying the result of the comparison. The numeric values are:

-1: string1 is less than string2
0: string1 is equal to string2
1: string1 is greater than string2

The string justification argument is either 'L' or 'R'. It defaults to 'L' if not specified. 'L' means a standard character comparison, left to right. 'R' means that any numeric substrings within the strings starting at the same position are compared as numbers. For example an 'R' comparison of 'AB100' and 'AB99' indicates that AB100 is greater than AB99, since 100 is greater than 99. The comparisons are case sensitive. |
| string replace_substring (expression1, expression2, string) | Returns a string value that contains the given string, with any characters in expression1 replaced by their corresponding characters in expression2. For example:
```
replace_substring
("ABC:, "abZ", "AGDCBDA")
```
returns "aGDZbDa", where any 'A' gets replaced by 'a', any 'B' gets replaced by 'b' and any 'C' gets replaced by 'Z'.

If expression2 is longer than expression1, the extra characters are ignored.

If expression1 is longer than expression2, the extra characters in expression1 are deleted from the given string (the corresponding characters are removed.) For example:
```
replace_substring("ABC", "ab", "AGDCBDA")
```
returns "aGDbDa". |
| int32 count_substring (string, substring) | Returns the number of times that substring occurs in string. If substring is an empty string, the number of characters in string is returned. |
| int32 dcount_substring (string, delimiter) | Returns the number of fields in string delimited by delimiter, where delimiter is a string. For example,
```
dcount_substring("abcFdefFghi", "F")
```
returns 3.

If delimiter is an empty string, the number of characters in the string + 1 is returned. If delimiter is not empty, but does not exist in the given string, 1 is returned. |
Table 78  string Field Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string double_quote_string</code></td>
<td>Returns the given string expression enclosed in double quotes.</td>
</tr>
<tr>
<td><code>string substring_by_delimiter</code></td>
<td>The string and delimiter arguments are string values, and the occurrence</td>
</tr>
<tr>
<td></td>
<td>and numsubstr arguments are int32 values. This function returns numsubstr</td>
</tr>
<tr>
<td></td>
<td>substrings from string, delimited by delimiter and starting at substring</td>
</tr>
<tr>
<td></td>
<td>number occurrence. An example is:</td>
</tr>
<tr>
<td></td>
<td><code>substring_by_delimiter</code></td>
</tr>
<tr>
<td></td>
<td>(&quot;abcFdefFghiFjkl&quot;, &quot;F&quot;, 2, 2)</td>
</tr>
<tr>
<td></td>
<td>The string &quot;defghi&quot; is returned.</td>
</tr>
<tr>
<td></td>
<td>If occurrence is &lt; 1, then 1 is assumed. If occurrence does not point to</td>
</tr>
<tr>
<td></td>
<td>an existing field, the empty string is returned. If numsubstr is not</td>
</tr>
<tr>
<td></td>
<td>specified or is less than 1, it defaults to 1.</td>
</tr>
<tr>
<td><code>int32 index_of_substring</code></td>
<td>Returns the starting position of the nth occurrence of substring in string.</td>
</tr>
<tr>
<td></td>
<td>The occurrence argument is an integer indicating the nth occurrence.</td>
</tr>
<tr>
<td></td>
<td>If there is no nth occurrence or string doesn’t contain any substring, -1</td>
</tr>
<tr>
<td></td>
<td>is returned. If substring is an empty string, -2 is returned.</td>
</tr>
<tr>
<td><code>string left_substring</code></td>
<td>Returns the first length characters of string. If length is 0, it returns</td>
</tr>
<tr>
<td></td>
<td>the empty string. If length is greater than the length of the string, the</td>
</tr>
<tr>
<td></td>
<td>entire string is returned.</td>
</tr>
<tr>
<td><code>string right_substring</code></td>
<td>Returns the last length characters of string. If length is 0, it returns</td>
</tr>
<tr>
<td></td>
<td>the empty string. If length is greater than the length of string, the entire</td>
</tr>
<tr>
<td></td>
<td>string is returned.</td>
</tr>
<tr>
<td><code>string string_of_space</code></td>
<td>Returns a string containing count spaces. The empty string is returned for</td>
</tr>
<tr>
<td></td>
<td>a count of 0 or less.</td>
</tr>
<tr>
<td><code>string single_quote_string</code></td>
<td>Returns the expression string enclosed in single quotes.</td>
</tr>
<tr>
<td><code>string string_of_substring</code></td>
<td>Returns a string containing count occurrences of string. The empty string</td>
</tr>
<tr>
<td></td>
<td>is returned for a count of 0 or less.</td>
</tr>
</tbody>
</table>
**Table 78  string Field Functions (continued)**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| `string trimc_string`           | If only string is specified, all leading and trailing spaces and tabs are removed, and all multiple occurrences of spaces and tabs are reduced to a single space or tab.  

If string and character are specified, option defaults to 'R'.  
The available option values are:  
'A' remove all occurrences of character  
'B' remove both leading and trailing occurrences of character.  
'D' remove leading, trailing, and redundant white-space characters.  
'E' remove trailing white-space characters  
'F' remove leading white-space characters  
'L' remove all leading occurrences of character  
'R' remove all leading, trailing, and redundant occurrences of character  
'T' remove all trailing occurrences of character |
| `string system_time_date()`     | Returns the current system time in this 24-hour format:  

`hh:mm:ss dd:mmm:yyyy` |
| `int32 offset_of_substring`     | Searches for the substring in the string beginning at character number position, where position is an uint32.  

Returns the starting position of the substring. |
| `int8 string_case_compare`      | This is a case-insensitive version of `string_compare()` below. |
| `int8 string_compare`           | Compares two strings and returns the index (0 or 1) of the greater string. |
| `int8 string_num_case_compare`  | This is a case-insensitive version of `string_num_compare()` below. |
| `string string_num_concatenate` | Returns a string after appending uint16 characters from the second string onto the first string. |
| `int8 string_num_compare`       | Compares first uint16 characters of two given strings and returns the index (0 or 1) of the greater string. |
| `string string_num_copy`        | Returns the first uint16 characters from the given string. |
| `int32 string_length`           | Returns the length of the string. |
Orchestrate provides the `ustring` type for multi-byte Unicode-character strings. `ustrings` can be assigned (=), compared (==, <, >=, etc.), and concatenated (+) in the Transformation Language. In addition, the functions described in Table 79 are available for `ustring` manipulations. When a long string is assigned to a short string, the long string is truncated to the length of the short string. The term white space refers to spaces, tabs, and any other blank space.

### Table 78  string Field Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string substring(string, starting_position, length)</code></td>
<td>Copies parts of strings to shorter strings by string extraction. The <code>starting_position</code> specifies the starting location of the substring; <code>length</code> specifies the substring length. The arguments <code>starting_position</code> and <code>length</code> are <code>uint16</code> types and must be positive (&gt;= 0).</td>
</tr>
<tr>
<td><code>string char_from_num(int32)</code></td>
<td>Returns an ASCII character from the given <code>int32</code>. If given a value that is not associated with a character such as -1, the function returns a space. An example use is: <code>char_from_num(38)</code> which returns &quot;&amp;&quot;</td>
</tr>
<tr>
<td><code>int32 num_from_char(string)</code></td>
<td>Returns the numeric value of the ASCII-character in the string. When this function is given an empty string, it returns 0; and when it is given a multi-character string, it uses the first character in the string. An example use is: <code>num_from_char(&quot;&amp;&quot;)</code> which returns 38.</td>
</tr>
</tbody>
</table>

### ustring Field Functions

Orchestrate provides the `ustring` type for multi-byte Unicode-character strings. `ustrings` can be assigned (=), compared (==, <, >=, etc.), and concatenated (+) in the Transformation Language. In addition, the functions described in Table 79 are available for `ustring` manipulations. When a long string is assigned to a short string, the long string is truncated to the length of the short string. The term white space refers to spaces, tabs, and any other blank space.

### Table 79  ustring Field Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`ustring ustring_from_date(date, &quot;date_format&quot;</td>
<td><code>uformat</code></td>
</tr>
</tbody>
</table>
### ustring Field Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ustring ustring_from_decimal</code> (decimal, &quot;fix_zero suppress_zero&quot;</td>
<td>Returns a ustring from decimal. fix_zero causes a decimal field containing all zeros to be treated as a valid zero. suppress_zero argument specifies that the returned ustring value will have no leading or trailing zeros. Examples: 000.100 -&gt; 0.1; 001.000 -&gt; 1; -001.100 -&gt; -1.1 The format specification is optional.</td>
</tr>
<tr>
<td><code>ustring ustring_from_time</code> (time, &quot;time_format&quot;</td>
<td>Returns a ustring from time using an optional format specification. The default time format is hh:mm:ss. For format descriptions, see “time Format” on page 21-35 and “uformat time Format” on page 21-34.</td>
</tr>
<tr>
<td><code>ustring ustring_from_timestamp</code> (timestamp, &quot;timestamp_format&quot;</td>
<td>Returns a ustring from timestamp. The format specification is optional. The default format is yyyy-mm-dd hh:mm:ss.</td>
</tr>
<tr>
<td><code>int8 u_is_alnum(ustring)</code></td>
<td>Returns 1 (true) if ustring consists entirely of alphanumeric characters.</td>
</tr>
<tr>
<td><code>int8 u_is_alpha(ustring)</code></td>
<td>Returns 1 (true) if ustring consists entirely of alphabetic characters.</td>
</tr>
<tr>
<td><code>int8 u_is_numeric(ustring)</code></td>
<td>Returns 1 (true) if ustring consists entirely of numeric characters, including decimal and sign.</td>
</tr>
<tr>
<td><code>int16 lookup_int16_from_ustring</code> (ustring, &quot;table_definition&quot;</td>
<td>Returns an integer corresponding to ustring using table_definition string or variable. See “String Conversions and Lookup Tables” on page 21-41 for more information.</td>
</tr>
<tr>
<td><code>ustring lookup_ustring_from_int16</code> (int16, &quot;table_definition&quot;</td>
<td>Returns a ustring corresponding to int16 using table_definition string or variable. See “String Conversions and Lookup Tables” on page 21-41 for more information.</td>
</tr>
<tr>
<td><code>ustring lookup_ustring_from_uint32</code> (uint32, &quot;table_definition&quot;</td>
<td>Returns a ustring corresponding to uint32 using table_definition string or variable. See “String Conversions and Lookup Tables” on page 21-41 for more information.</td>
</tr>
<tr>
<td><code>uint32 lookup_uint32_from_ustring</code> (string, &quot;table_definition&quot;</td>
<td>Returns an unsigned integer from ustring using table_definition string or variable.</td>
</tr>
</tbody>
</table>
### int8 u_is_valid ('type_ustring', 'value_ustring')

Returns 1 (true) if value_ustring is valid according to type_ustring, including NULL. The type_ustring argument is required. It must specify an Orchestrate schema data type.

Integer types are checked to ensure the value_ustring is numeric (signed or unsigned), a whole number, and a valid value (for example, 1024 cannot be assigned to an int8 type).

Decimal types are checked to ensure the value_ustring is numeric (signed or unsigned) and a valid value.

Float types are checked to ensure the value_ustring is numeric (signed or unsigned) and a valid value (exponent is valid).

String is always valid with the NULL exception below.

For all types, if the field cannot be set to NULL and the string is NULL, 0 (false) is returned.

Date, time, and timestamp types are checked to ensure they are correct, using the optional format argument, and valid values.

Raw cannot be checked since the input is a string.

### ustring u_lower_case(ustring)

Converts ustring to lowercase. Non-alphabetic characters are ignored in the transformation.

### ustring u_upper_case(ustring)

Converts ustring to uppercase. Non-alphabetic characters are ignored in the transformation.

### ustring u_compact_whitespace(ustring)

Returns the ustring after reducing all consecutive white space in ustring to a single space.

### ustring u_pad_string(ustring, pad_ustring, pad_length)

Returns the ustring with pad_ustring appended to the bounded length string for pad_length number of characters. pad_length is an int16.

When the given ustring is a variable-length string, it defaults to a bounded-length of 1024 characters. If the given ustring is a fixed-length string, this function has no effect.

### ustring u_strip_whitespace(ustring)

Returns ustring after stripping all white space in the string.

### ustring u_trim_leading_trailing(ustring)

Returns ustring after removing all leading and trailing white space.
Table 79  

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ustring u_trim_leading( ustring )</td>
<td>Returns ustring after removing all leading white space.</td>
</tr>
<tr>
<td>ustring u_trim_trailing( ustring )</td>
<td>Returns a ustring after removing all trailing white space.</td>
</tr>
</tbody>
</table>
| int32 u_string_order_compare( ustring1, ustring2, justification) | Returns a numeric value specifying the result of the comparison. The numeric values are:  
  -1: ustring1 is less than ustring2  
  0: ustring1 is equal to ustring2  
  1: ustring1 is greater than ustring2  
  The string justification argument is either 'L' or 'R'. It defaults to 'L' if not specified. 'L' means a standard character comparison, left to right. 'R' means that any numeric substrings within the strings starting at the same position are compared as numbers. For example an 'R' comparison of 'AB100' and 'AB99' indicates that AB100 is greater than AB99, since 100 is greater than 99. The comparisons are case sensitive. |
| ustring u_replace_substring( expression1, expression2, ustring ) | Returns a ustring value that contains the given ustring, with any characters in expression1 replaced by their corresponding characters in expression2. For example:  
  u_replace_substring("ABC", "abZ", "AGDCBDA")  
  returns "aGDZbDa", where any "A" gets replaced by "a", any "B" gets replaced by "b" and any "C" gets replaced by "Z".  
  If expression2 is longer than expression1, the extra characters are ignored.  
  If expression1 is longer than expression2, the extra characters in expression1 are deleted from the given string (the corresponding characters are removed.) For example:  
  u_replace_substring("ABC", "ab", "AGDCBDA")  
  returns "aGDbDa". |
| int32 u_count_substring( ustring, sub_ustring ) | Returns the number of times that sub_ustring occurs in ustring. If sub_ustring is an empty string, the number of characters in ustring is returned. |
Table 79  ustring Field Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 u_dcount_substring (ustring, delimiter)</td>
<td>Returns the number of fields in ustring delimited by delimiter, where delimiter is a string. For example, dcount_substring(&quot;abcdefgghi&quot;, &quot;F&quot;) returns 3. If delimiter is an empty string, the number of characters in the string + 1 is returned. If delimiter is not empty, but does not exist in the given string, 1 is returned.</td>
</tr>
<tr>
<td>ustring u_double_quote_string (expression)</td>
<td>Returns the given ustring expression enclosed in double quotes.</td>
</tr>
<tr>
<td>ustring u_substring_by_delimiter (ustring, delimiter, occurrence, numsubstr)</td>
<td>The delimiter argument is a ustring value, and the occurrence and numsubstr arguments are int32 values. This function returns numsubstr substrings from ustring, delimited by delimiter and starting at substring number occurrence. An example is: substring_by_delimiter(&quot;abcdefgghiFjkl&quot;, &quot;F&quot;, 2, 2) The string &quot;defghi&quot; is returned. If occurrence is &lt; 1, then 1 is assumed. If occurrence does not point to an existing field, the empty string is returned. If numsubstr is not specified or is less than 1, it defaults to 1.</td>
</tr>
<tr>
<td>int32 u_index_of_substring (ustring, sub_ustring, occurrence)</td>
<td>Returns the starting position of the nth occurrence of sub_ustring in ustring. The occurrence argument is an integer indicating the nth occurrence. If there is no nth occurrence, 0 is returned; if sub_ustring is an empty string, -2 is returned; and if ustring doesn’t contain any sub_ustring, -1 is returned.</td>
</tr>
<tr>
<td>ustring u_left_substring (ustring, length)</td>
<td>Returns the first length characters of ustring. If length is 0, it returns the empty string. If length is greater than the length of the ustring, the entire ustring is returned.</td>
</tr>
<tr>
<td>ustring u_right_substring (ustring, length)</td>
<td>Returns the last length characters of ustring. If length is 0, it returns the empty string. If length is greater than the length of ustring, the entire ustring is returned.</td>
</tr>
<tr>
<td>ustring u_string_of_space (count)</td>
<td>Returns a ustring containing count spaces. The empty string is returned for a count of 0 or less.</td>
</tr>
<tr>
<td>ustring u_single_quote_string (expression)</td>
<td>Returns expression enclosed in single quotes.</td>
</tr>
</tbody>
</table>
Table 79  ustring Field Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ustring u_string_of_substring (ustring, count)</code></td>
<td>Returns a ustring containing count occurrences of ustring. The empty string is returned for a count of 0 or less.</td>
</tr>
<tr>
<td><code>ustring u_trimc_string (ustring [, character [, option]])</code></td>
<td>If only ustring is specified, all leading and trailing spaces and tabs are removed, and all multiple occurrences of spaces and tabs are reduced to a single space or tab. If ustring and character are specified, option defaults to 'R'. The available option values are: 'A' remove all occurrences of character 'B' remove both leading and trailing occurrences of character 'D' remove leading, trailing, and redundant white-space characters. 'E' remove trailing white-space characters 'F' remove leading white-space characters 'L' remove all leading occurrences of character 'R' remove all leading, trailing, and redundant occurrences of character 'T' remove all trailing occurrences of character</td>
</tr>
<tr>
<td><code>ustring u_system_time_date()</code></td>
<td>Returns the current system time in this 24-hour format: <code>hh:mm:ss dd:mmm:yyyy</code></td>
</tr>
<tr>
<td><code>int32 u_offset_of_substring (ustring, sub_ustring, position)</code></td>
<td>Searches for the sub_ustring in the ustring beginning at character number position, where position is an uint32. Returns the starting position of the substring.</td>
</tr>
<tr>
<td><code>int8 u_string_case_compare (ustring, ustring)</code></td>
<td>This is a case-insensitive version of <code>u_string_compare()</code> below.</td>
</tr>
<tr>
<td><code>int8 u_string_compare (ustring, ustring)</code></td>
<td>Compares two ustrings and returns the index (0 or 1) of the greater string.</td>
</tr>
<tr>
<td><code>int8 u_string_num_case_compare (ustring, ustring, uint16)</code></td>
<td>This is a case-insensitive version of <code>u_string_num_compare()</code> below.</td>
</tr>
<tr>
<td><code>ustring u_string_num_concatenate (ustring, ustring, uint16)</code></td>
<td>Returns a ustring after appending uint16 characters from the second ustring onto the first ustring.</td>
</tr>
<tr>
<td><code>int8 u_string_num_compare (ustring, ustring, uint16)</code></td>
<td>Compares first uint16 characters of two given ustrings and returns the index (0 or 1) of the greater ustring.</td>
</tr>
<tr>
<td><code>ustring u_string_num_copy (ustring, uint16)</code></td>
<td>Returns the first uint16 characters from the given ustring.</td>
</tr>
<tr>
<td><code>int32 u_string_length (ustring)</code></td>
<td>Returns the length of the ustring.</td>
</tr>
</tbody>
</table>
**Table 79** ustring Field Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ustring u_substring(ustring, starting_position, length)</code></td>
<td>Copies parts of ustrings to shorter strings by string extraction. The <code>starting_position</code> specifies the starting location of the substring; <code>length</code> specifies the substring length. The arguments <code>starting_position</code> and <code>length</code> are <code>uint16</code> types and must be positive (&gt;= 0).</td>
</tr>
<tr>
<td><code>ustring u_char_from_num(int32)</code></td>
<td>Returns a ustring character value from the given <code>int32</code>. If given a value that is not associated with a character such as -1, the function returns a space. An example use is: <code>u_char_from_num(38)</code> which returns &quot;&amp;&quot;</td>
</tr>
<tr>
<td><code>int32 u_num_from_char(ustring)</code></td>
<td>Returns the numeric value of the character in the ustring. When this function is given an empty string, it returns 0; and when it is given a multi-character string, it uses the first character in the string. An example use is: <code>u_num_from_char(&quot;&amp;&quot;)</code> which returns 38</td>
</tr>
</tbody>
</table>

**Bit Manipulation Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string bit_expand(uint64)</code></td>
<td>Expands the given <code>uint64</code> to a string containing the binary representation.</td>
</tr>
<tr>
<td><code>ustring u_bit_expand(uint64)</code></td>
<td>Expands the given <code>uint64</code> to a ustring containing the binary representation.</td>
</tr>
<tr>
<td><code>uint64 bit_compress(string)</code></td>
<td>Converts the string binary representation to an <code>uint64</code> field.</td>
</tr>
<tr>
<td><code>uint64 u_bit_compress(ustring)</code></td>
<td>Converts the ustring binary representation to an <code>uint64</code> field.</td>
</tr>
</tbody>
</table>
The Job Monitor reports on the current state of a job and supplies information about the operators in your data flow. By default, it continually gathers information about your jobs, but it does not send the information unless you request it from your user-written client.

The information it supplies falls into four categories: job status, metadata, monitor, and summary, and it is given in XML notation. If you do not have job monitoring disabled with the top-level \texttt{-nomonitor osh} option, you can also obtain custom report and summary information about the transform operator using the functions in the table below.

There are six functions: three for the \texttt{string} type and three for the \texttt{ustring} type. The \texttt{name_string} or \texttt{name_ustring} argument can be used to specify a name for the custom information, the \texttt{description_string} or \texttt{description_ustring} argument can be used to describe the type of information, and the \texttt{value_string} or \texttt{value_ustring} argument can be used to give the details of the information.

\begin{table}[h]
\centering
\begin{tabular}{|l|p{0.7\textwidth|}
\hline
Function & Description \\
\hline
\texttt{uint64 set\_bit} & Turns the uint64 bits that are listed by number in the string \texttt{list\_of\_bits} on or off, depending on whether the value of the \texttt{bit\_state} integer is 1 or 0. \texttt{bit\_state} is an optional argument, and has a default value of 1 which turns the list of bits on. \texttt{bit\_state} is an \texttt{optional argument, and has a default value of 1 which turns the list of bits on.} \\
& An example use is: \texttt{set\_bit(0, }'1,3,5,7'\texttt{)} which returns 85. \\
\hline
\texttt{uint64 u\_set\_bit} & This function is an internationalized version of \texttt{set\_bit()} above. \\
& \\
\end{tabular}
\caption{Bit Manipulation Functions (continued)}
\end{table}

Job Monitoring Functions

The Job Monitor reports on the current state of a job and supplies information about the operators in your data flow. By default, it continually gathers information about your jobs, but it does not send the information unless you request it from your user-written client.

The information it supplies falls into four categories: job status, metadata, monitor, and summary, and it is given in XML notation. If you do not have job monitoring disabled with the top-level \texttt{-nomonitor osh} option, you can also obtain custom report and summary information about the transform operator using the functions in the table below.

There are six functions: three for the \texttt{string} type and three for the \texttt{ustring} type. The \texttt{name_string} or \texttt{name_ustring} argument can be used to specify a name for the custom information, the \texttt{description_string} or \texttt{description_ustring} argument can be used to describe the type of information, and the \texttt{value_string} or \texttt{value_ustring} argument can be used to give the details of the information.
See the Orchestrate Job Monitoring chapter in the Orchestrate 7.0 User Guide for information about this facility.

Table 81  **Job Monitoring Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string set_custom_summary_info (name_string, description_string, value_string)</code></td>
<td>Call this function in the finish code segment.</td>
</tr>
<tr>
<td><code>ustring u_set_custom_summary_info (name_ustring, description_ustring, value_ustring)</code></td>
<td></td>
</tr>
<tr>
<td><code>string send_custom_report (name_string, description_string, value_string)</code></td>
<td>Call this function in the initialize code segment.</td>
</tr>
<tr>
<td><code>ustring u_send_custom_report (name_ustring, description_ustring, value_ustring)</code></td>
<td></td>
</tr>
<tr>
<td><code>string set_custom_instance_report (name_string, description_string, value_string)</code></td>
<td>Call this function in the mainloop code segment.</td>
</tr>
<tr>
<td><code>ustring u_set_custom_instance_report (name_ustring, description_ustring, value_ustring)</code></td>
<td></td>
</tr>
</tbody>
</table>
Miscellaneous Functions

Table 82 defines functions in the Transformation Language that do not fit into any of the above categories.

Table 82  Miscellaneous Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void force_error</td>
<td>Terminates the data flow when an error is detected, and prints error_message_string to stderr.</td>
</tr>
<tr>
<td>u_force_error</td>
<td>Terminates the data flow when an error is detected, and prints error_message_ustring to stderr.</td>
</tr>
<tr>
<td>string get_environment</td>
<td>Returns the current value of string, a UNIX environment variable. The functionality is the same as the C getenv function.</td>
</tr>
<tr>
<td>ustring u_get_environment</td>
<td>Returns the current value of ustring, a UNIX environment variable. The functionality is the same as the C getenv function.</td>
</tr>
<tr>
<td>int16 get_partition_num</td>
<td>Returns the current partition number.</td>
</tr>
<tr>
<td>int16 get_num_of_partitions</td>
<td>Returns the number of partitions.</td>
</tr>
<tr>
<td>void print_message</td>
<td>Prints message_string to stdout.</td>
</tr>
<tr>
<td>u_print_message</td>
<td>Prints message_ustring to stdout.</td>
</tr>
<tr>
<td>uint32 size_of</td>
<td>Returns the actual size value when it is stored, not the length.</td>
</tr>
</tbody>
</table>

The Transformation Language Versus C

The Transformation Language contains additions to C to handle record processing. Also, parts of C are not supported. The following sections indicated C language items that are not supported by the Transformation Language or that work differently in the Transformation Language.

Keywords

Keywords in the Transformation Language are not case sensitive.
Local Variables

You cannot initialize variables as part of the declaration. There are no local variable pointers or structures. Enums are not supported.

Operators

Several C operators are not part of the language:

• The comma operator
• Composite assignment operators such as +=.

Flow Control

The switch and case C keywords do not appear in the Transformation Language.

The if ... else if construct for multiple branching is not supported. You can accomplish the same effect by using multiple if statements in sequence together with complex boolean expressions. For example, where in C you could write:

```
if (size < 7)
tag = "S";
else if (size < 9)
tag = "M";
else
tag = "L";
```

in the Transformation Language you would write:

```
if (size < 7)
tag = "S";
if (size >= 7 && size < 9)
tag = "M";
if (size >= 9)
tag = "L";
```

Other Unsupported C Language Elements

The Transformation Language does not support:

• Casting
• Labeled statements
• Pre-Processor commands

References

Using the transform Operator

This section gives examples on how to construct data transformations using the transform operator.

Example 1: Student-Score Distribution

Application Logic

In this example, the transform operator is used to determine the student-score distribution for multiple classes. The input data set contains a record for each student in the classes. Each record contains a student_id field and a score field. The score field has the numerical score the student received for the class.

The input student records are distributed to one of three output data sets based on the value of the score field. Records with a score of under 75 (poor) are written to output 0; records with a score between 75 and 90 (fair) are written to output 1; and records with a score of over 90 (good) are written to output 2.

Data Flow Diagram

After the records are processed, a report is printed that shows the number of students in each score category.

Highlighted Transformation Language Components

This section points out the Transformation Language components this example illustrates. The Transformation Language code is given in the next section.

- **Code segmentation.** The use of the global{}, initialize{}, mainloop{}, and finish{} segments.
- **Global variables.** How to declare global variables and where to declare them. There is a single global variable: jobName.
- **Stage variables.** How to declare stage variables and where to declare them. Examples are `recNum0`, `recNum1`, and `recNum2`.
- **Record flow controls.** How to use the conditional statement with the `writerecord` command.
- **Local variables.** How to declare and initialize local variables. Examples from the code are `numOfPoorScores`, `numOfFairScores`, and `numOfGoodScores`.
- **Default conversions.** How to do default conversions using the assignment operator. An example that converts an `int32` to a `string` is:
  ```
  numOfGoodScore = recNum2
  ```
- **Message logging.** How to make function calls to `print_message()`.

**Transformation Language**

The record-processing logic can be expressed in the Transformation Language as follows. The expression file name for this example is `score_distr_expr`.

```transformation
global {
    // the global variable that contains the name for each job run
    string jobName;
}
initialize {
    // the number of records in output 0
    int32 recNum0;
    // the number of records in output 1
    int32 recNum1;
    // the number of records in output 2
    int32 recNum2;
    // initialization
    recNum0 = 0;
    recNum1 = 0;
    recNum2 = 0;
}
mainloop {
    // records in output 0
    if (score < 75)
    {
        recNum0++;
        writerecord 0;
    }
    // records of output 1
    if (score >= 75 && score <= 90)
    {
        RecNum1++;
    }
    // records in output 2
    if (score > 90)
    {
        recNum2++;
        writerecord 2;
    }
}
```
Chapter 21 The transform Operator

Using the transform Operator

```plaintext
writerecord 1;
}
// records of output2
if (score > 90)
{
    recNum2++;
    writerecord 2;
}
}
finish
{
    // define a string local variable to store the number of
    // students with poor scores
    string NumOfPoorScores;
    numOfPoorScores = recNum0;
    // define a string local variable to store the number of
    // students with fair scores
    string NumOfFairScores;
    numOfFairScores = recNum1;
    // define a string local variable to store the number of
    // students with good scores
    string NumOfGoodScores;
    numOfGoodScores = recNum2;
    // Print out the number of records in each output data set
    print_message(jobName+ " has finished running.");
    print_message("The number of students having poor scores are" +numOfPoorScores);
    print_message("The number of students having fair scores are" +numOfFairScores);
    print_message("The number of students having good scores are" +numOfGoodScores);
}

osh Command

An example osh command to run this job is:
osh -params "jobName=classA" -f score_distr

osh Script

The contents of score_distr are:

```plaintext
#compile the expression code
transform -inputschema record(student_id:string[10];score:int32;)
-outputschema record(student_id:string[10];score:int32;)
-outputschema record(student_id:string[10];score:int32;)
-outputschema record(student_id:string[10];score:int32;)
-expressionfile score_distr_expr -flag compile -name score_map;
#run the job
import -schema record(student_id:string[10];score:int32;)
-file [&jobName].txt | transform -flag run -name score_map
0> export -schema record(student_id:string[10];score:int32)
    -filename [&jobName]poor_score.out -overwrite
```
Example Input and Output

The input from classA.txt is:

A112387567 80
A218925316 95
A619846201 70
A731347820 75
A897382711 85
A327637289 82
A238950561 92
A238967521 87
A826381931 66
A763567100 89

The outputs are:

classApoor_score.out
A619846201 70
A826381931 66

classAfair_score.out
A112387567 80
A731347820 75
A897382711 85
A327637289 82
A238967521 87
A763567100 89

classAgood_score.out
A218925316 95
A238950561 92

The global variable jobName is initialized using the -params option. To determine the score distribution for class B, for example, assign jobName another value:

osh -params "jobName=classB" -f score_distr

Example 2: Student-Score Distribution With a Letter Grade Added to Example 1

Application Logic

The application logic in this example is similar to that of “Example 1: Student-Score Distribution” on page 21-59. In addition to the student_id and score fields, an additional field indicating the letter grade is added to each output record. The letter grade is based on the value of the score field: grade C for a score of under 75; grade B for a score between 75 and 90; and grade A for a score over 90.
Data Flow Diagram

Highlighted Transformation Language Components

This example demonstrates the use of the following components. The Transformation Language code is given in the next section.

- **Dataset alias.** How to define and use data set aliases.
- **Stage variables.** Declare stage variables in a row. An example is:
  ```
  int32 recNum0, recNum1, recNum2;
  ```
- **String constants.** How to assign a constant to a string variable. For example:
  ```
  out0.grade = "C"
  ```
- **Local variables.** How to declare local variables in a row. For example:
  ```
  int32 numOfCs, numOfBs, numOfAs
  ```

Transformation Language

The record-processing logic can be expressed in the Transformation Language as follows. The expression file name for this example is `score_grade_expr`.

```
```c
recNum1 = 0;
recNum2 = 0;
}
mainloop
{
    if (in0.score < 75)
        {  
            recNum0++;  
            out0.grade = "C";  
            writerecord 0;
        }
    if (in0.score >= 75 && in0.score <= 90)
        {  
            recNum1++;  
            out1.grade = "B";  
            writerecord 1;
        }
    if (in0.score > 90)
        {  
            recNum2++;  
            out2.grade = "A";  
            writerecord 2;
        }
}
finish
{
    // define string local variables to store the number of
    // students having different letter grades
    string numOfCs, numOfBs, numOfAs;
    // default conversions using assignments
    numOfCs = recNum0;
    numOfBs = recNum1;
    numOfAs = recNum2;
    // Print out the number of records in each output data set
    print_message("jobName+" has finished running.");
    print_message("The number of students getting C is "+numOfCs);
    print_message("The number of students getting B is "+numOfBs);
    print_message("The number of students getting A is "+numOfAs);
}

osh Command

An example osh command to run this job is:
    osh -params "jobName=classA" -f score_grade
```
osh Script

The contents of `score_grade` are:

```bash
# compile the expression code
transform -inputschema record(student_id:string[10];score:int32;)
 -outputschema record
  (student_id:string[10];score:int32;grade:string[1])
 -outputschema record
  (student_id:string[10];score:int32;grade:string[1])
 -outputschema record
  (student_id:string[10];score:int32;grade:string[1])
 -expressionfile score_grade_expr -flag compile -name score_map;
# run the job
import -schema record(student_id:string[10];score:int32;)
 -file [&jobName].txt | transform -flag run -name score_map
0> -export record(student_id:string[10];score:int32;grade:string[1])
  -file [&jobName]poor_scores.out -overwrite
1> -export record(student_id:string[10];score:int32;grade:string[1])
  -file [&jobName]fair_scores.out -overwrite
2> -export record(student_id:string[10];score:int32;grade:string[1])
  -file [&jobName]good_scores.out -overwrite
```

Example Input and Output

The input from `classA.txt` is the same as in “Example 1: Student-Score Distribution” on page 21-59.

The outputs are:

- `classApoor_scores.out`
  - A619846201 70 C
  - A826381931 66 C

- `classAfair_scores.out`
  - A112387567 80 B
  - A731347820 75 B
  - A697382711 85 B
  - A327637289 82 B
  - A238967521 87 B
  - A763567110 89 B

- `classAgood_scores.out`
  - A218925316 95 A
  - A238950561 92 A

Example 3: Student-Score Distribution with a Class Field Added to Example 2

Application Logic

The application logic in this example is similar to that in “Example 2: Student-Score Distribution With a Letter Grade Added to Example 1” on page 21-62. The difference is that another output field is added, named `class`.
The student class is determined by the first character of the student_id. If the first character is B, the student's class is Beginner; if the first character is I, the student's class is Intermediate; and if the first character is A, the student's class is Advanced. Records with the same class field are written to the same output.

The score field is not only transferred from input to output, but is also changed. If the score field is less than 75, the output score is:

\[
(\text{in0.score} + (200 - \text{in0.score}^2)/4)
\]

Otherwise the output score is:

\[
(\text{in0.score} + (200 - \text{in0.score}^2)/3)
\]

Data Flow Diagram

Highlighted Transformation Language Components

This example demonstrates the use of the following components. The Transformation Language code follows in the next section.

- **Binary and ternary operators.** How to use them in expressions. An example is: score_local.
- **String manipulations.** How to do string comparisons, string assignments, and function calls. Examples are: class_local and class_init.
- **Local variables.** How to use local variables in expressions. Examples are score_local, grade_local, and class_local.

Transformation Language

The record-processing logic can be expressed in the Transformation Language as follows. The expression file for this example is score_class_expr.

```plaintext
inputname 0 int0;
outputname 0 out0;
outputname 1 out1;
outputname 2 out2;
mainloop
{
  // define an int32 local variable to store the score
  ```
int32 score_local;
score_local=(in0.score < 75) ? (in0.score+(200-in0.score*2)/4):
    (in0.score+(200-in0.score*2)/3)
// define a string local variable to store the grade
string[1] grade_local;
if (score_local < 75) grade_local = "C";
if (score_local >= 75 && score_local <= 90) grade_local = "B";
if (score_local > 90) grade_local = "A";
// define string local variables to check the class level
string[max=15] class_local;
string[1] class_init;
class_init = substring(in0.student_id,0,1);
if (class_init == "B") class_local = "Beginner";
if (class_init == "I") class_local = "Intermediate";
if (class_init == "A") class_local = "Advanced";
// outputs
if (class_local == "Beginner")
{
    out0.score = score_local;
    out0.grade = grade_local;
    out0.class = class_local;
    writerecord 0;
}
if (class_local == "Intermediate")
{
    out1.score = score_local;
    out1.grade = grade_local;
    out1.class = class_local;
    writerecord 1;
}
if (class_local == "Advanced")
{
    out2.score = score_local;
    out2.grade = grade_local;
    out2.class = class_local;
    writerecord 2;
}

osh Command

The osh command to run this job is:
	osh -f score_class

osh Script

The contents of score_class are:

```
#compile the expression code
transform -inputschema record(student_id:string[10];score:int32;)
    -outputschema record
```
Chapter 21  The transform Operator

Using the transform Operator

```plaintext
(student_id:string[10];score:int32;grade:string[1];class:string[max=15])

-outputs schema record
  (student_id:string[10];score:int32;grade:string[1];class:string[max=15])
-outputs schema record
  (student_id:string[10];score:int32;grade:string[1];class:string[max=15])
-expression file score_class_expr -flag compile -name score_map;

#run the job
import -schema record(student_id:string[10];score:int32;) -file score.txt | transform -flag run -name score_map
0> -export record(student_id:string[10];score:int32;grade:string[1];class:string[max=15]) -file beginner.out -overwrite
1> -export record(student_id:string[10];score:int32;grade:string[1];class:string[max=15]) -file intermediate.out -overwrite
2> -export record(student_id:string[10];score:int32;grade:string[1];class:string[max=15]) -file advanced.out -overwrite
```

Example Input and Output

The input from `score.txt` is

```
B112387567 80
A218925316 95
A619846201 70
I731347820 75
B897382711 85
I327637289 82
A238950561 92
I238967521 87
B826381931 66
A763567100 89
```

The outputs are:

```
beginner.out
  B112387567 93 A Beginner
  B897382711 95 A Beginner
  B826381931 83 B Beginner

intermediate.out
  I731347820 91 A Intermediate
  I327637289 94 A Intermediate
  I238967521 95 A Intermediate

advanced.out
  A218925316 98 A Advanced
  A619846201 85 B Advanced
  A238950561 97 A Advanced
  A763567100 96 A Advanced
```
Example 4. Student Record Distribution With Null Score Values And a Reject Data Set

Application Logic

The application logic in this example is the same as in “Example 3: Student-Score Distribution with a Class Field Added to Example 2” on page 21-65. The difference is that the input records contain null score fields, and records with null score fields are transferred to a reject data set. This example shows you how to specify and use a reject data set.

The Transformation Language expression file is the same as that for Example 3.

Data Flow Diagram

osh Command

The osh command to run this job is:

osh -f score_reject

osh Script

The contents of score_reject are:

```bash
# compile the expression code
transform -inputschema record(student_id:string[10];
  score:nullable int32;)
-outputschema record
  (student_id:string[10];score:nullable int32;grade:string[1];
  class:string[max=15])
-outputschema record
  (student_id:string[10];score:nullable int32;grade:string[1];
  class:string[max=15])
```
- expressionfile score_class_expr -flag compile -name score_map
- reject;
# run the job
import -schema record(student_id:string[10]; score:nullable int32
{null_field='NULL'}) -file score_null.txt | transform -flag run
- name score_map
0> -export -schema record(student_id:string[10];
    score:nullable int32{null_field='NULL'}; grade:string[1];
    class:string[max=15]) -file beginner.out -overwrite
1> -export record(student_id:string[10];
    score:nullable int32{null_field='NULL'}; grade:string[1];
    class:string[max=15]) -file intermediate.out -overwrite
2> -export record(student_id:string[10];
    score:nullable int32{null_field='NULL'}; grade:string[1];
    class:string[max=15]) -file advanced.out -overwrite
3> -export record(student_id:string[10];
    score:nullable int32{null_field='NULL'}) -file reject.out
   -overwrite

Example Input and Output

The input from score_null.txt is

<table>
<thead>
<tr>
<th>student_id</th>
<th>score</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>B112387567</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>I218925316</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>A619846201</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>I731347820</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>B897382711</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>I327637289</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>A238950561</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>I238967521</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>B826381931</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>A763567100</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

The outputs are

beginner.out

<table>
<thead>
<tr>
<th>student_id</th>
<th>score</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>B897382711</td>
<td>95</td>
<td>A Beginner</td>
</tr>
<tr>
<td>B826381931</td>
<td>83</td>
<td>B Beginner</td>
</tr>
</tbody>
</table>

intermediate.out

<table>
<thead>
<tr>
<th>student_id</th>
<th>score</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>I218925316</td>
<td>98</td>
<td>A Intermediate</td>
</tr>
<tr>
<td>I731347820</td>
<td>91</td>
<td>A Intermediate</td>
</tr>
<tr>
<td>I238967521</td>
<td>95</td>
<td>A Intermediate</td>
</tr>
</tbody>
</table>

advanced.out

<table>
<thead>
<tr>
<th>student_id</th>
<th>score</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>A619846201</td>
<td>85</td>
<td>B Advanced</td>
</tr>
<tr>
<td>A238950561</td>
<td>97</td>
<td>A Advanced</td>
</tr>
</tbody>
</table>

reject.out

<table>
<thead>
<tr>
<th>student_id</th>
<th>score</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>B112387567</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>I327637289</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>A763567100</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>
Example 5. Student Record Distribution With Null Score Values Handled

Application Logic

The application logic in this example is the same as “Example 4. Student Record Distribution With Null Score Values And a Reject Data Set” on page 21-69. The difference is that null values are appropriately handled so that records with null score fields are not sent to a reject data set.

Data Flow Diagram

Highlighted Transformation Language Components

This example shows how to handle nulls in input fields, and how to set nulls on output fields. The Transformation Language is in the next section.

The input score field is assigned to the local variable score_local. Since, by default, the score field is nullable and score_local is non-nullable, any record containing a null score is either dropped or sent to the reject data set. To avoid this and to insure that all records go to their appropriate outputs, null handling is used.

There are three ways to handle nulls:

1. Use a function call to handle_null(). For example:
   
   ```
   score_local = handle_null(in0.score,"-100");
   ```

2. Use a function call to null() and the ternary operator. For example:

   ```
   score_local = null(in0.score)?-100:in0.score;
   ```

3. Use a function call to notnull() and the ternary operator. For example:

   ```
   score_local = notnull(in0.score)?in0.score:-100;
   ```
Setting a nullable field to null occurs when the record is written to output. In this example, after the statement `out0.score = score_local` executes, `out0.score` no longer contains null values, since the old null value has been replaced by -100. To reinstate the null field, the function `make_null()` is called to mark that any field containing -100 is a null field. When the score is -100, grade should be null; therefore, `set_null()` is called.

**Transformation Language**  
The record processing logic can be expressed in the Transformation Language as follows. The expression file name for this example is `score_null_handling_expr`.

```plaintext
inputname 0 in0;
outputname 0 out0;
outputname 1 out1;
outputname 2 out2;
mainloop
{
    // define an int32 local variable to store the score
    int32 score_local;
    // handle the null score
    score_local = handle_null(in0.score,"-100");
    // alternatives:
    // score_local = null(in0.score)?-100:in0.score;
    // score_local = notnull(in0.score)?in0.score:-100;
    // define a string local variable to store the grade
    string[1] grade_local;
    grade_local = "F";
    if ( score_local < 60 && score_local >= 0 )
        grade_local = "D";
    if ( score_local < 75 && score_local >= 60 )
        grade_local = "C";
    if ( score_local >= 75 && score_local <= 90 )
        grade_local = "B";
    if ( score_local > 90 )
        grade_local = "A";
    // define string local variables to check the class level
    string[max=15] class_local;
    string[1] class_init;
    class_init = substring(in0.student_id,0,1);
    if ( class_init == "B" )
        class_local = "Beginner";
    if ( class_init == "I" )
        class_local = "Intermediate";
    if ( class_init == "A" )
        class_local = "Advanced";
    // outputs
    if ( class_local == "Beginner" )
    {
```
out0.score = score_local;
out0.score = make_null(out0.score, "-100");
if ( grade_local == "F" )
    out0.grade = set_null();
else
    out0.grade = grade_local;
out0.class = class_local;
writerecord 0;
}
if ( class_local == "Intermediate" )
{
    out1.score = score_local;
    out1.score = make_null(out1.score, "-100");
    if ( grade_local == "F" )
        out1.grade = set_null();
    else
        out1.grade = grade_local;
    out1.class = class_local;
    writerecord 1;
}
if ( class_local == "Advanced" )
{
    out2.score = score_local;
    out2.score = make_null(out2.score, "-100");
    if ( grade_local == "F" )
        out2.grade = set_null();
    else
        out2.grade = grade_local;
    out2.class = class_local;
    writerecord 2;
}

osh Command

The osh script to run this job is:

osh -f score_null_handling

osh Script

The contents of score_null_handling are:

# compile the expression code
transform -inputschema record(student_id:string[10];score:nullable int32;)
    -outputschema record(student_id:string[10];score:nullable
        int32;grade:nullable string[1];class:string[max=15])
    -outputschema record(student_id:string[10];score:nullable
        int32;grade:nullable string[1];class:string[max=15])
    -outputschema record(student_id:string[10];score:nullable
        int32;grade:nullable string[1];class:string[max=15])
    -expressionfile score_null_handling_expr -flag compile
    -name score_map;
# run the job
import -schema record{student_id:string[10]; score:nullable int32{null_field='NULL'}} -file score_null.txt | transform -flag run -name score_map
0> -export record{student_id:string[10]; score:nullable int32{null_field='NULL'}; grade:nullable string[1]{null_field='FAILED'}; class:string[max=5];} -file beginner.out -overwrite
1> -export record{student_id:string[10]; score:nullable int32{null_field='NULL'}; grade:nullable string[1]{null_field='FAILED'}; class:string[max=5];} -file intermediate.out -overwrite
2> -export record{student_id:string[10]; score:nullable int32{null_field='NULL'}; grade:nullable string[1]{null_field='FAILED'}; class:string[max=5];} -file advanced.out -overwrite

Example Input and Output

The input from `score_null.txt` is:

B112387567 NULL
1218925316 95
A619846201 70
1731347820 75
B897382711 85
1327637289 NULL
A238950561 92
1238967521 87
B826381931 66
A763567100 NULL

The outputs are:

beginner.out
B112387567 NULL FAILED Beginner
B897382711 85 B Beginner
B826381931 66 C Beginner

intermediate.out
I218925316 95 A Intermediate
I731347820 75 B Intermediate
I327637289 NULL FAILED Intermediate
I238967521 87 B Intermediate

advanced.out
A619846201 70 C Advanced
A238950561 92 A Advanced
A763567100 NULL FAILED Advanced
Example 6. Student Record Distribution With Vector Manipulation

Application Logic

The application logic in this example is similar to that in “Example 5. Student Record Distribution With Null Score Values Handled” on page 21-71. As in Example 5, this example has two fields in its input schema: student_id and score. The score field, however, is not a single integer field as in Example 5 but is a vector of five elements which represent the class scores a student has received in a single semester. Only three out of five classes are required, therefore, the vector elements are specified to be nullable so that a NULL score can represent an elective course not taken by the student, rather than a failing grade as in Example 5.

In addition to the explicit input schema containing the score and student_id fields, the transform operator in this example receives a term string field from its upstream operator via a schema variable. The term field indicates whether the semester is a final or a midterm semester.

The output field grade_local is a vector of five elements, and a GPA float field is added to the output data set.

Data Flow Diagram

Highlighted Transformation Language Components

This example highlights the following components:

- **Vector variable definition.** Examples are:
  
  ```
  int32 score_local[5], and string[1] grade_local[5]
  ```

- **Vector variables with flow loops.** For example, while and for loops.
• **Using flow loops with controls.** For example, continue statements
• **Using vector variables in expressions.** For example, if statements and assignments.
• **Using a schema variable for an implicit transfer.** For example in -inputschema and -outputschema declarations, the field term is transferred from the up-stream operator to the transform operator and from the transform input to its output.

Transformation Language

The record-processing logic can be expressed in the Transformation Language as follows. The expression file name for this example is `score_vector_expr`.

```plaintext
inputname 0 in0;
outputname 0 out0;
outputname 1 out1;
outputname 2 out2;
mainloop
{
    // define an int32 local vector variable to store the scores
    int32 score_local[5];
    int32 vecLen;
    vecLen = 5;
    // handle null score
    int32 i;
    i = 0;
    while ( i < vecLen )
    {
        score_local[i] = handle_null(in0.score[i],"-100");
        // alternatives
        // score_local[i] = null(in0.score[i])?-100:in0.score[i];
        // score_local[i] = notnull(in0.score[i])?in0.score[i]:-100;
        i++;
    }
    // define a string local vector variable to store the grades
    string[1] grade_local[5];
    // define sfloat local variables to calculate GPA.
    sfloat tGPA_local, GPA_local;
    tGPA_local = 0.0;
    GPA_local = 0.0;
    // define an int8 to count the number of courses taken.
    int8 numOfScore;
    numOfScore = 0;
    for ( i = 0; i < vecLen; i++)
    {
        // Null score means the course is not taken,
        // and will not be counted.
        if ( score_local[i] == -100)
```

21 76
\[
\begin{align*}
\text{grade}_\text{local}[i] &= 'S'; \\
\text{continue;}
\end{align*}
\]

\text{numOfScore}++; \\
\text{if} \ ( \text{score}_\text{local}[i] < 60 \ \&\& \ \text{score}_\text{local}[i] >= 0 ) \\
\{ \\
\text{grade}_\text{local}[i] = 'D'; \\
\text{tGPA}_\text{local} = \text{tGPA}_\text{local} + 1.0;
\}
\text{if} \ ( \text{score}_\text{local}[i] < 75 \ \&\& \ \text{score}_\text{local}[i] >= 60 ) \\
\{ \\
\text{grade}_\text{local}[i] = 'C'; \\
\text{tGPA}_\text{local} = \text{tGPA}_\text{local} + 2.0;
\}
\text{if} \ ( \text{score}_\text{local}[i] >= 75 \ \&\& \ \text{score}_\text{local}[i] <= 90 ) \\
\{ \\
\text{grade}_\text{local}[i] = 'B'; \\
\text{tGPA}_\text{local} = \text{tGPA}_\text{local} + 3.0;
\}
\text{if} \ ( \text{score}_\text{local}[i] > 90 ) \\
\{ \\
\text{grade}_\text{local}[i] = 'A'; \\
\text{tGPA}_\text{local} = \text{tGPA}_\text{local} + 4.0;
\}
\]
\text{if} \ ( \text{numOfScore} > 0 ) \\
\text{GPA}_\text{local} = \text{tGPA}_\text{local} / \text{numOfScore};
\]

// define string local variables to check the class level 
\text{string}[\text{max}=15] \text{class}_\text{local}; 
\text{string}[1] \text{class}_\text{init}; 
\text{class}_\text{init} = \text{substring}(\text{i}0.\text{student_id}, 0, 1); 
\text{if} \ ( \text{class}_\text{init} == 'B' ) \\
\text{class}_\text{local} = 'Beginner'; 
\text{if} \ ( \text{class}_\text{init} == 'I' ) \\
\text{class}_\text{local} = 'Intermediate'; 
\text{if} \ ( \text{class}_\text{init} == 'A' ) \\
\text{class}_\text{local} = 'Advanced';

// outputs 
\text{if} \ ( \text{class}_\text{local} == 'Beginner' ) \\
\{ \\
\text{for} \ ( \ i = 0; \ i < \text{vecLen}; \ i++ ) \\
\{ \\
\text{out0.score}[i] = \text{score}_\text{local}[i]; \\
\text{out0.score}[i] = \text{make_null}(\text{out0.score}[i], '-100'); \\
\text{out0.grade}[i] = \text{grade}_\text{local}[i];
\}
\}
\text{out0.GPA} = \text{GPA}_\text{local}; 
\text{out0.class} = \text{class}_\text{local};
writerecord 0;
}
if ( class_local == "Intermediate" )
{
    for ( i = 0; i < vecLen; i++)
    {
        out1.score[i] = score_local[i];
        out1.score[i] = make_null(out1.score[i],"-100");
        out1.grade[i] = grade_local[i];
    }
    out1.GPA = GPA_local;
    out1.class = class_local;
    writerecord 1;
}
if ( class_local == "Advanced" )
{
    for ( i = 0; i < vecLen; i++)
    {
        out2.score[i] = score_local[i];
        out2.score[i] = make_null(out2.score[i],"-100");
        out2.grade[i] = grade_local[i];
    }
    out2.GPA = GPA_local;
    out2.class = class_local;
    writerecord 2;
}

osh Command

The osh script to run this job is:

osh -f score_vector

osh Script

The contents of score_vector are:

# compile the expression code
transform -inputschema record(student_id:string[10];score[5]:nullable int32;inRec:*) -outputschema record(student_id:string[10];score[5]:nullable int32;grade[5]:string[1];GPA:sfloat;class:string[max=15];outRec:*)

exportfile score_vector_expr -flag compile -name score_map;

# run the job
import -schema record(student_id:string[10];score[5]:nullable int32|null_field='NULL';term:string) -file score_vector.txt |
transform -flag run -name score_map
0> -export record(student_id:string[10];score[5]:nullable
Example Input and Output

The input from `score_vector.txt` is:

<table>
<thead>
<tr>
<th>Student ID</th>
<th>Score 1</th>
<th>Score 2</th>
<th>Score 3</th>
<th>Score 4</th>
<th>Score 5</th>
<th>Grade</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>B112387567</td>
<td>NULL</td>
<td>90</td>
<td>87</td>
<td>62</td>
<td>NULL</td>
<td>B</td>
<td>Final</td>
</tr>
<tr>
<td>I218925316</td>
<td>95</td>
<td>NULL</td>
<td>91</td>
<td>88</td>
<td>NULL</td>
<td>B</td>
<td>Midterm</td>
</tr>
<tr>
<td>A619846201</td>
<td>70</td>
<td>82</td>
<td>85</td>
<td>68</td>
<td>NULL</td>
<td>A</td>
<td>Final</td>
</tr>
<tr>
<td>I731347820</td>
<td>75</td>
<td>NULL</td>
<td>89</td>
<td>93</td>
<td>95</td>
<td>A</td>
<td>Final</td>
</tr>
<tr>
<td>B897382711</td>
<td>85</td>
<td>90</td>
<td>96</td>
<td>NULL</td>
<td>NULL</td>
<td>A</td>
<td>Midterm</td>
</tr>
<tr>
<td>I327637289</td>
<td>NULL</td>
<td>NULL</td>
<td>88</td>
<td>92</td>
<td>76</td>
<td>A</td>
<td>Final</td>
</tr>
<tr>
<td>A238950561</td>
<td>92</td>
<td>97</td>
<td>89</td>
<td>85</td>
<td>83</td>
<td>A</td>
<td>Final</td>
</tr>
<tr>
<td>I238967521</td>
<td>87</td>
<td>NULL</td>
<td>NULL</td>
<td>86</td>
<td>82</td>
<td>A</td>
<td>Midterm</td>
</tr>
<tr>
<td>B826381931</td>
<td>66</td>
<td>73</td>
<td>82</td>
<td>NULL</td>
<td>NULL</td>
<td>A</td>
<td>Final</td>
</tr>
<tr>
<td>A763567100</td>
<td>NULL</td>
<td>NULL</td>
<td>53</td>
<td>68</td>
<td>92</td>
<td>A</td>
<td>Final</td>
</tr>
</tbody>
</table>

The outputs are:

**beginner.out**

<table>
<thead>
<tr>
<th>Student ID</th>
<th>Score 1</th>
<th>Score 2</th>
<th>Score 3</th>
<th>Score 4</th>
<th>Score 5</th>
<th>Grade</th>
<th>Term</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>B112387567</td>
<td>NULL</td>
<td>90</td>
<td>87</td>
<td>62</td>
<td>NULL</td>
<td>B</td>
<td>B</td>
<td>2.7</td>
</tr>
<tr>
<td>B897382711</td>
<td>85</td>
<td>90</td>
<td>96</td>
<td>NULL</td>
<td>NULL</td>
<td>B</td>
<td>A</td>
<td>3.3</td>
</tr>
<tr>
<td>B826381931</td>
<td>66</td>
<td>73</td>
<td>82</td>
<td>NULL</td>
<td>NULL</td>
<td>C</td>
<td>B</td>
<td>2.3</td>
</tr>
</tbody>
</table>

**intermediate.out**

<table>
<thead>
<tr>
<th>Student ID</th>
<th>Score 1</th>
<th>Score 2</th>
<th>Score 3</th>
<th>Score 4</th>
<th>Score 5</th>
<th>Grade</th>
<th>Term</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>I218925316</td>
<td>95</td>
<td>NULL</td>
<td>91</td>
<td>88</td>
<td>NULL</td>
<td>A</td>
<td>B</td>
<td>3.7</td>
</tr>
<tr>
<td>I731347820</td>
<td>75</td>
<td>NULL</td>
<td>89</td>
<td>93</td>
<td>95</td>
<td>B</td>
<td>A</td>
<td>3.5</td>
</tr>
<tr>
<td>I327637289</td>
<td>NULL</td>
<td>NULL</td>
<td>88</td>
<td>92</td>
<td>76</td>
<td>B</td>
<td>B</td>
<td>3.3</td>
</tr>
<tr>
<td>I238967521</td>
<td>87</td>
<td>NULL</td>
<td>86</td>
<td>82</td>
<td>82</td>
<td>B</td>
<td>B</td>
<td>3.3</td>
</tr>
</tbody>
</table>

**advanced.out**

<table>
<thead>
<tr>
<th>Student ID</th>
<th>Score 1</th>
<th>Score 2</th>
<th>Score 3</th>
<th>Score 4</th>
<th>Score 5</th>
<th>Grade</th>
<th>Term</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A619846201</td>
<td>70</td>
<td>82</td>
<td>85</td>
<td>68</td>
<td>NULL</td>
<td>C</td>
<td>B</td>
<td>2.5</td>
</tr>
<tr>
<td>A238950561</td>
<td>92</td>
<td>97</td>
<td>89</td>
<td>85</td>
<td>83</td>
<td>A</td>
<td>B</td>
<td>3.4</td>
</tr>
<tr>
<td>A763567100</td>
<td>53</td>
<td>68</td>
<td>92</td>
<td>S</td>
<td>D</td>
<td>C</td>
<td>A</td>
<td>2.3</td>
</tr>
</tbody>
</table>

Example 7: Student Record Distribution Using Sub-record

**Application Logic**

The application logic in this example is similar to Example 6. The difference is that the input `score_report` field is a vector of 2-element sub-records. The schema of each sub-record is the same as the top-level record in Example 6 except that the subfield `score` is a vector of three elements. The sub-fields are `student_id`, `score`, and `term`. A new output sub-field, `GPA`, is added to each sub-record. A new output...
field, class, is added to the top-level record. No null-handling functions are used in this example because null-handling is not applicable to sub-records.

**Data Flow Diagram**

```
input data set

transform

output data sets

beginner  intermediate  advanced
```

**Highlighted Transformation Language Component**

This example shows you how to use sub-records in expressions.

**Transformation Language**

The record-processing logic can be expressed in the Transformation Language as follows. The expression file name for this example is `score_subrec_expr`.

```plaintext
input name 0 in0;
output name 0 out0;
output name 1 out1;
output name 2 out2;
mainloop {

// define an int32 local vector variable to store the scores
int32 score_local[3];

// define an index to store vector or subrec length
int32 vecLen, subrecLen;
vecLen = 3;
subrecLen = 2;

// index
int32 i, j;
i = 0;
j = 0;

// define a string local vector variable to store the grades
string[1] grade_local[3];

// define sfloat local variables to calculate GPA.
sfloat tGPA_local, GPA_local;
```
tGPA_local = 0.0;
GPA_local = 0.0;
// define string local variables to check the class level
string[1] class_init;
string[max=15] class_local;
class_init = substring(in0.student_id,0,1);
if ( class_init == "B" )
  class_local = "Beginner";
if ( class_init == "I" )
  class_local = "Intermediate";
if ( class_init == "A" )
  class_local = "Advanced";
// calculate grade and GPA
// The outer loop controls subrec
// The inner loop controls sub-fields
for ( j = 0; j < subrecLen; j++ )
{
  for ( i = 0; i < vecLen; i++ )
  {
    score_local[i] = in0.score_report[j].score[i];
    if ( score_local[i] < 60 && score_local[i] >= 0 )
    {
      grade_local[i] = "D";
      tGPA_local = tGPA_local + 1.0;
    }
    if ( score_local[i] < 75 && score_local[i] >= 60 )
    {
      grade_local[i] = "C";
      tGPA_local = tGPA_local + 2.0;
    }
    if ( score_local[i] >= 75 && score_local[i] <= 90 )
    {
      grade_local[i] = "B";
      tGPA_local = tGPA_local + 3.0;
    }
    if ( score_local[i] > 90 )
    {
      grade_local[i] = "A";
      tGPA_local = tGPA_local + 4.0;
    }
  }
  GPA_local = tGPA_local / vecLen;
// outputs
if ( class_local == "Beginner" )
{
  for ( i = 0; i < vecLen; i++ )
  {
    out0.score_report[j].score[i] = score_local[i];
    out0.score_report[j].grade[i] = grade_local[i];
  }
}
out0.score_report[j].GPA = GPA_local;
}
if (class_local == "Intermediate")
{
    for (i = 0; i < vecLen; i++)
    {
        out1.score_report[j].score[i] = score_local[i];
        out1.score_report[j].grade[i] = grade_local[i];
    }
    out1.score_report[j].GPA = GPA_local;
}
if (class_local == "Advanced")
{
    for (i = 0; i < vecLen; i++)
    {
        out2.score_report[j].score[i] = score_local[i];
        out2.score_report[j].grade[i] = grade_local[i];
    }
    out2.score_report[j].GPA = GPA_local;
}
// initialize these variables for next subrec
GPA_local = 0;
tGPA_local = 0;
}
// outputs
if (class_local == "Beginner")
{
    out0.class = class_local;
    writerecord 0;
}
if (class_local == "Intermediate")
{
    out1.class = class_local;
    writerecord 1;
}
if (class_local == "Advanced")
{
    out2.class = class_local;
    writerecord 2;
}

osh Command

osh -f score_subrec
osh Script

The contents of score_subrec are:

```bash
# compile the expression code
transform -inputschemafile score_subrec_input.schema
 -outputschemafile score_subrec_output.schema
 -expressionfile score_subrec_expr -flag compile -name score_map;
# run the job
import -schemafile score_subrec_input.schema -file score_subrec.txt |
transform -flag run -name score_map > a.v > b.v > c.v;
0> -export -schemafile score_subrec_output.schema -file beginner.out
 -overwrite < a.v;
1> -export -schemafile score_subrec_output.schema -file intermediate.out -overwrite < b.v;
2> -export -schemafile score_subrec_output.schema -file advanced.out
 -overwrite < c.v;
```

Input Schema

The contents of score_subrec_input.schema are:

```sql
record(
    student_id:string[10];
    score_report[2]:subrec(score[3]:nullable int32;term:string;)
)
```

Output Schema

The contents of score_subrec_output.schema are:

```sql
record(
    student_id:string[10];
    score_report[2]:subrec(score[3]:nullable int32;
        grade[3]:string[1];
        GPA:sfloat{out_format='%4.2g'};
        term:string;)
    class:string[max=15];
)
```

Example Input and Output

The input from score_subrec.txt is:

```
B112387567 90 87 62 Final 80 89 52 Midterm
L21895316 95 91 88 Midterm 92 81 78 Final
A619846201 70 82 85 Final 92 81 78 Midterm
L731347820 75 89 93 Final 85 79 92 Midterm
B897382711 85 90 96 Midterm 88 92 96 Final
L327637289 88 92 76 Final 82 96 86 Midterm
A238950561 92 97 89 Final 90 87 91 Midterm
L238967521 87 86 82 Midterm 97 96 92 Final
B826381931 66 73 82 Midterm 86 93 82 Final
A763567100 53 68 92 Final 48 78 92 Midterm
```
The outputs are:

beginner.out
B112387567 90 87 62 B B C 2.7 Final 80 89 52 B B D 2.3 Midterm
Beginner
B897382711 85 90 96 B B A 3.3 Midterm 88 92 96 B A A 3.7 Final
Beginner
B826381931 66 73 82 C C B 2.3 Midterm 86 93 82 B A B 3.3 Final
Beginner

intermediate.out
I218925316 95 91 88 A A B 3.7 Midterm 92 81 78 A B B 3.3 Final
Intermediate
I731347820 75 89 93 B B A 3.3 Final 85 79 92 B B A 3.3 Midterm
Intermediate
I327637289 88 92 76 B A B 3.3 Final 86 96 82 B A B 3.3 Midterm
Intermediate
I238967521 87 86 82 B B B 3.3 Final 97 96 92 A A A 4 Final
Intermediate

advanced.out
A619846201 70 82 85 C B B 2.7 Final 60 89 85 C B B 2.7 Midterm
Advanced
A238950561 92 97 89 A A B 3.7 Final 90 87 91 B B A 3.3 Midterm
Advanced
A763567100 53 68 92 D C A 2.3 Final 48 78 92 D B A 2.7 Midterm
Advanced

Example 8: External C Function Calls

This example demonstrates how external C functions can be included in the Transformation Language. See “External Global C-Function Support” on page 21-14 for the details of C-function support.

This example contains these components:

• C header file
• C source file
• Transformation Language expression file
• osh script to run the transform operator

C Header File: functions.h

int my_times( int x , int y );
unsigned int my_sum( unsigned int x , unsigned int y );
void my_print_message( char* msg);
#if defined (__alpha)
    long my_square( long x , long y );
#else
    long long my_square( long long x , long long y );
#endif
C Source File: functions.c

```c
#include <stdio.h>
#include "functions.h"
int my_times( int x , int y )
{
    int time;
    time = x * y;
    return time;
}

unsigned int my_sum( unsigned int x, unsigned int y )
{
    unsigned int sum;
    sum = x + y;
    return sum;
}

void my_print_message(char* msg)
{
    printf("%s\n",msg);
    return;
}
#if defined(__alpha)
    long my_square( long x, long y )
    {
        long square;
        square = x*x + y*y;
        return square ;
    }
#else
    long long my_square( long long x , long long y )
    {
        long long square;
        square = x*x + y*y;
        return square ;
    }
#endif
```

Transformation Language

The expression file name for this example is t_extern_func.

```plaintext
extern int32 my_times(int32 x, int32 y);
extern uint32 my_sum(uint32 x, uint32 y);
extern void my_print_message(string msg);
extern int64 my_square(int64 x, int64 y);
inputname 0 in0;
outputname 0 out0;
mainloop
{
    out0.times = my_times(in0.a1,in0.a2);
    out0.sum= my_sum(in0.a1,in0.a2);
    out0.square= my_square(in0.a1,in0.a2);
}
my_print_message("HELLO WORLD!");
write_record 0;
}

osh Script

transform
  - input_schema record(a1:int32; a2:int32)
  - output_schema record(times:int32; sum:uint32; square:int64;)
  - expressionfile tExtern_func
  - flag compile
  - name myExtern_func
  - staticobj /DIR/functions.o;
gen generator - schema record(a1:int32; a2:int32) | transform - flag run
  - name myExtern_func |
peek;
This part of the Orchestrate 7.0 Operators Reference describes the Orchestrate Import/Export and Cobol schema conversion utility operators. The chapters in this part are listed below.

Part 1 describes the Orchestrate General Library.

Part 3 describes the Orchestrate Join, Partitioning, Collection, Restructure, SAS, Sorting, and Statistics libraries, and the four RDBMS libraries that provide interfaces to DB2, INFORMIX, Oracle, and Teradata.

Chapter 22 Import/Export Introduction 22 1
Chapter 23 The import Operator 23 1
Chapter 24 The export Operator 24 1
Chapter 25 Import/Export Properties 25 1
Chapter 26 COBOL Schema Conversion 26 1
Import/Export Introduction

This chapter introduces the import and export facilities provided by Orchestrate. These facilities let you use Orchestrate with non-Orchestrate data.

Introduction 22 2

Record Schemas 22 3
  Example 1: Import Schema 22 3
  Example 2: Export Schema 22 4

Field and Record Properties 22 5

Complete and Partial Schemas 22 7
  Defining Partial Record Schemas 22 7
  Exporting with Partial Record Schemas 22 9
    Unmodified Intact Record 22 9
    Intact Record with Additional Fields 22 10

Implicit Import and Export 22 11
  Implicit Import/Export Operations with No Schemas Specified 22 11
  The Default Import Schema 22 12
  The Default Export Schema 22 13
  Overriding the Defaults 22 13

Error Handling during Import/Export 22 14
  Failure 22 14
  Warning 22 15

ASCII and EBCDIC Conversion Tables 22 15
  EBCDIC to ASCII 22 16
  ASCII to EBCDIC 22 20
  Conversion Table Irregularities 22 22
Generally, Orchestrate operators process only data formatted as and contained in Orchestrate data sets. The Orchestrate import operator imports non-Orchestrate data into Orchestrate and converts it to an Orchestrate data set. The Orchestrate export operator converts Orchestrate data sets to external representations.

The import/export utility consists of these operators:

- The import operator: imports one or more data files into a single data set (see Chapter 23, “The import Operator”).
- The export operator: exports a data set to one or more data files (see Chapter 24, “The export Operator”).

Introduction

Data must be formatted as an Orchestrate data set in order for Orchestrate to partition and process it. However, most Orchestrate applications process data that is represented in a format other than Orchestrate data sets including flat files, text or binary files that consist of data without metadata (a description of the data). The import and export operators are used to convert flat data files to data sets, and to convert data sets to flat data files.

A flat file must be a single file, not a parallel set of files or fileset.

Flat files processed by Orchestrate contain record-based data stored in a binary or text format. Although the layout of the data contained in these files is record-based, the files do not have a built-in schema describing the record layout.

To import and export flat files, you must describe their layout by means of Orchestrate record schemas. Record schemas define the name and data type of each field in a record. They can be partial or complete. (See “Record Schemas” below.) You can also optionally specify data properties. Properties supply information about the format of the imported or exported data. Properties can apply to both records and fields. (See Chapter 25, “Import/Export Properties.”).

In order for Orchestrate to process such data, it must first be imported into an Orchestrate data set. After data processing operations have been completed, data can be stored as an Orchestrate data set or exported to its original format or a different format.

The following data formats cannot be imported by the import operator:

- filesets, or parallel sets of files
• RDBMS tables
• SAS datasets
• COBOL data files

See “Orchestrated SAS Data Set Format” on page 32-9 for information on various types of SAS data sets.

For RDBMS tables and parallel SAS data sets, import and export are performed implicitly. In other words, you can work with these data sets as if they were Orchestrate data sets. For COBOL data files, you use the COBOL conversion utility to import and export the data. For other flat files, you must use the **import** and **export** operators.

---

**Important**

Do not use the **import** or **export** operators for RDBMS tables, SAS data sets, or COBOL data files. Only use **import** or **export** for flat files that are not COBOL data files.

---

**Record Schemas**

A record schema is an implicit or explicit description of the layout and properties of the record-oriented data contained in an Orchestrate data set. See the chapter on Orchestrate data sets in the Orchestrate 7.0 User Guide for information on schema contents and format. Implicit (default) record schemas are discussed in “The Default Import Schema” on page 22-12 and “The Default Export Schema” on page 22-13.

When you invoke either the **import** or the **export** operator, you must explicitly specify a schema, as in the following two examples.

---

**Example 1: Import Schema**

In this example, the **import** operator imports data and the **statistics** operator calculates statistics on the Income field of the imported data set before writing it to disk. Figure 38 shows the data flow for this example.
Here is the schema for importing the data into an Orchestrate data set as defined in osh.

```plaintext
record {
    Name: string;
    Age: int8;
    Gender: uint8;
    Income: sfloat;
}
```

**Example 2: Export Schema**

In this example, the same imported data set is again stored as a flat file after the `statistics` operator has processed it. As is sometimes the case, the same schema as in “Example 1: Import Schema” is used, and the data is returned to its original format.
Field and Record Properties

To define a record schema, you specify the following for each field:

- A field name
- A data type
- An optional length specification for string and raw fields
- An optional length specification for vector fields
- An optional nullability specification (by default, imported fields are not nullable)
- An optional set of one or more import/export properties, which can apply to the record as a whole or to a specific field. Properties are enclosed in braces ({}). Record properties appear after the keyword record and field properties appear before the final semicolon (;) in the definition of an individual field.

Here is a completely described data set record, where each byte of the record is contained in a field and all fields are accessible by Orchestrate because they have the required field definition information.

Figure 30 Data Flow for Example 2
The following **osh** code defines this record:

```bash
record {
    a: sfloat;
    b: int8;
    c: int8;
    d: dfloat;
    e: sfloat;
    f: decimal[1,0];
    g: decimal[1,0];
    h: int8;
}
```

If you wish to modify this example to specify that each record is to be delimited by a comma (,) you would add the record-level property `delim = ','`:

```bash
record {delim = ','} {
    a: sfloat;
    b: int8;
    c: int8;
    d: dfloat;
    e: sfloat;
    f: decimal[1,0];
    g: decimal[1,0];
    h: int8;
}
```

The same property `delim = ','` could be used as a field-level property, for example:

```bash
record {
    a: sfloat;
    b: int8 {delim = ','};
    c: int8;
    d: dfloat;
    e: sfloat;
    f: decimal[1,0];
    g: decimal[1,0];
    h: int8;
}
```

In this case, only field `b` would be delimited by a comma. For export, the other fields would be followed by the default delimiter, the ascii space (0x20).

These brief examples only scratch the surface of schema definition. See the chapter on Orchestrate data sets in the *Orchestrate 7.0 User Guide* for detailed information on schemas, and Chapter 25, “Import/Export Properties” for more information on record and field properties.
Complete and Partial Schemas

The schemas discussed so far are all examples of **complete** schemas, that is, a schema where you define each field. Orchestrate allows you to define another type of schema: a **partial** schema. To define a partial schema, define only the fields that will be acted on. For example, to process records with tens or hundreds of fields, you may have to access, and therefore define, only a few of them for sorting keys.

Here is a partial record schema where field information is defined for only two fields:

```
c:int8 f:decimal
```

The use of partial record schemas has advantages and disadvantages, which are summarized in **Table 83**.

<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>partial</td>
<td>Partial record schemas are simpler than complete record schemas, because you define only the fields of interest in the record. The <strong>import</strong> operator reads these records faster than records with a complete schema because it does not have to interpret as many field definitions.</td>
<td>Orchestrate treats the entire imported record as an atomic object; that is, Orchestrate cannot add, remove, or modify the data storage within the record. (Orchestrate can, however, add fields to the beginning or end of the record.)</td>
</tr>
<tr>
<td>complete</td>
<td>Orchestrate can add fields to the body of the record and remove or modify any existing field. Removing fields from a record allows you to minimize its storage requirement.</td>
<td>The <strong>import</strong> operator takes longer to read records with a complete schema than records with a partial schema.</td>
</tr>
</tbody>
</table>

Defining Partial Record Schemas

To define a partial record schema, you specify the **intact** property of the record schema. For information about this property, see “**intact**” on page 25-37. A record with no fields is the simplest kind to define and import, as in the following example:

```
record {intact, record_length=82, record_delim='\r\n'}()
```

The record schema defines an 82-byte record as specified by the **record_length=82** property. The 82 bytes includes 80 bytes of data plus two bytes for the newline.
and carriage-return characters delimiting the record as specified by
record_delim='\r\n'. No information is provided about any field. On import, the
two bytes for the carriage-return and line-feed characters are stripped from each
record of the data set and each record contains only the 80 bytes of data.

An imported record with no defined fields can be used only with keyless
partitioners, collectors, and operators. The term keyless means that no individual
fields of the record must be accessed in order for it to be processed. Orchestrate
supplies keyless operators to:

- Copy a data set
- Sample a data set to create one or more output data sets containing a random
  sample of records from the input data set
- Combine multiple input data sets into a single output data set
- Compress or decompress a data set.

In addition, you can use keyless partitioners, such as random or same, and
keyless collectors, such as round-robin, with this type of record.

An imported record with no defined fields cannot be used for field-based
processing such as:

- Sorting the data based on one or more key fields, such as a name or zip code
- Calculating statistics on one or more fields, such as income or credit card
  balance
- Creating analytic models to evaluate your data

A record with no field definitions may be passed to an operator that runs UNIX
code. Such code typically has a built-in record and field processing ability. See the
chapter on creating UNIX operators in the Orchestrate 7.0 User Guide for more
information on this type of operator.

You can define fields in partial schemas so that field-based processing can be
performed on them. To do so, define only the fields to be processed. For example,
the following partial record schema defines two fields of a record, Name and
Income, thereby making them available for use as inputs to partitioners, collectors,
and other operators.

record { intact, record_length=82, record_delim_string='\r\n' }
  { Name: string[20] { position=12, delim=none };
    Income: int32 { position=40, delim=',' text };
  }

In this schema, the defined fields occur at fixed offsets from the beginning of the
record. The Name field is a 20-byte string field starting at byte offset 12 and the
Income field is a 32-bit integer, represented by an ASCII text string, that starts at
byte position 40 and ends at the first comma encountered by the import operator.
For example, suppose that the record schema shown above is modified so that the
Name and Income fields follow variable-length fields (V1, V2, and V3). Instead of
fixed offsets, the Name and Income fields have relative offsets, based on the length
of preceding variable-length fields. In this case, you specify the position of the
Name and Income fields by defining the delimiters of the variable-length fields, as
in the following example:

```
record { intact, record_delim_string='\r\n' }
    { v1: string { delim=',' }; 
      v2: string { delim=',' }; 
      Name: string[20] { delim=None }; 
      v3: string { delim=',' }; 
      Income: int32 { delim='\'', text }; 
    }
```

### Exporting with Partial Record Schemas

The intact portion of a record is an atomic object; that is, Orchestrate cannot add,
remove, or modify the data storage contained in the intact portion. When a data
set containing an intact is exported, the entire intact portion is exported.

There are two types of data sets containing an intact definition.

- “Unmodified Intact Record”
- “Intact Record with Additional Fields”

### Unmodified Intact Record

If a schema being exported consists of a single intact record and nothing else, the
intact record is exported using its original import record formatting.

For example, consider a data set created using the following import schema:

```
record { intact, record_length=82, record_delim_string='\r\n' }
    { name: string[20] { position=12, delim=None }; 
      income: uint32 { position=40, delim='\'', text }; 
    }
```

In order to export this data set using the original record schema and properties,
you specify an empty record schema, as follows:

```
record {}
```

Because there are no properties, no braces ({} ) are needed.

This empty record schema causes the export operator to use the original record
schema and properties.
However, defining any record property in the export schema overrides all record-level properties specified at import. That is, all record-level properties are ignored by the `export` operator. For example, if you specify the following record schema to the `export` operator:

```plaintext
record { record_delim='\n' } {}
```
the intact record is exported with a newline character as the delimiter. The carriage-return character (`'\r'`) is dropped. Each exported record is 81 bytes long: 80 bytes of data and one byte for the newline character.

### Intact Record with Additional Fields

When a schema being exported consists of multiple intact records or one intact record together with other fields, consider each intact record to be a separate field of the exported record.

For example, consider a data set with the following record schema:

```plaintext
record {
    a: int32;
    b: string;
    x: record {intact, record_delim='\r\n', text, delim=',,,',}
    ( name: string[20] { position=0 }; address: string[30] { position=44 }; );
    y: record {intact, record_length=244, binary, delim=none }
    ( v1: int32 { position=84 }; v2: decimal(10,0) { position=13 }; );
}
```

The data set contains two intact fields, `x` and `y`, and two other fields, `a` and `b`. The intact fields retain all properties specified at the time of the import.

When this type of data set is exported, the record-level properties for each intact field (specified in braces after the keyword `record`) are ignored and you must define new record-level properties for the entire record.

For example, if you want to export records formatted as follows:

- a two-byte prefix is written at the beginning of each record
- the fields are ordered `x`, `y`, and `a`; field `b` is omitted
- a comma is added after `x`
- `y` has no delimiter
- `a` is formatted as binary.

then the export schema is:

```plaintext
record { record_prefix=2 }
    ( x: record { intact } () { delim=',' }; );
```
Implicit Import and Export

An implicit import or export is any import or export operation that Orchestrate performs without your invoking the corresponding operator. Orchestrate performs implicit import and export operations when it recognizes that an input or output file is not stored as an Orchestrate data set (its extension is not `.ds`).

Implicit Import/Export Operations with No Schemas Specified

Consider a program that takes a flat file as input and makes two copies of the file. The first copy is written to an Orchestrate data set. The second copy is compressed. No import operation is specified. Here is the `osh` command for the example:

```
$ osh "copy < user.dat > outDS1.ds | pcompress > compressed.ds"
```

No schema is specified, so Orchestrate uses the default schema (See “The Default Import Schema” on page 22-12 and “The Default Export Schema” on page 22-13).

Figure 31 shows the data flow model of this step. Orchestrate automatically performs the import operation required to read the flat file.
Implicit operator insertion also works for export operations. The following example shows the `tsort` operator writing its results to a data file, again with no schema specified:

```
$ osh "tsort -key a -hash -key b < inDS.ds > result.dat"
```

The output file does not end in the extension `.ds`, so Orchestrate inserts an export operator to write the output data set to the file.

The examples of implicit import/export insertion shown so far all use the default record schemas. However, you can also insert your own record schema to override the defaults.

The next sections discuss:

- “The Default Import Schema” on page 22-12
- “The Default Export Schema” on page 22-13
- “Overriding the Defaults” on page 22-13

**The Default Import Schema**

The default import schema specifies that imported records contain a single variable-length string field where the end of each record is delimited by a newline character. Here is the default import schema used by Orchestrate:
record {record_delim = '\n'}
  (rec:string;)

The following figure shows the record layout of the source data as defined by the
default import schema:

default record layout of source data

<table>
<thead>
<tr>
<th>record contents</th>
<th>&quot;\n&quot;</th>
</tr>
</thead>
</table>

After the import operation, the destination data set contains a single variable-
length string field named \textit{rec} corresponding to the entire imported record, as
follows:

variable-length string named rec

**The Default Export Schema**

The default export schema is:

\texttt{record()}

This record schema causes Orchestrate to export all fields as follows:

- Each field except the last field in the record is delimited by an ASCII space \texttt{(0x20)}
- The end of the record (and thus the end of the last field) is marked by the
  newline character \texttt{(\textbackslash n)}
- All fields are represented as text (numeric data is converted to a text
  representation).

For example, the following figure shows the layout of a data set record before and
after export when the default export schema is applied:

record in the source Orchestrate data set:

|-------|-----------|-----------|-------|

record after export as stored in a data file:

<table>
<thead>
<tr>
<th>int16 as text</th>
<th>0x20</th>
<th>string[8]</th>
<th>0x20</th>
<th>string[2]</th>
<th>0x20</th>
<th>int32 as text</th>
<th>&quot;\n&quot;</th>
</tr>
</thead>
</table>

**Overriding the Defaults**

To override the defaults of an implicit import or export operation, specify a
schema to replace the defaults.
Specify the schema as one of these:

- Part of an `osh` command, as in the following example:
  
  ```bash
  $ osh '[record_schema] ...'
  ```

- A text file and a reference to the file in the command line, as in the following example:
  
  ```bash
  $ osh 'copy < [record @'schema_file'] user.dat > outDS1.ds | pcompress > compressed.ds'
  ```

  where `schema_file` is the name of the text file containing the record schema definition. You can specify a schema anywhere on an `osh` command line by using the notation `[record @'schema_file']`.

- A schema variable defined upstream of a command, and subsequently input to the command, as in the following example:
  
  ```bash
  $ import_schema="record (Name:string; Age:int8 {default = 127}; Income:dfloat {skip = 2}; Phone:string;)
  
  $ osh 'copy < [$import_schema] user.dat > outDS1.ds | pcompress > compressed.ds"
  ```

  All implicit import/export operations in the command use that schema. However, you can override it for a particular data file by preceding the specification of a flat file with a different record schema.

To define formatting properties of records as a whole rather than as individual fields in an export operation, use this form:

```bash
record {record_properties}()
```

In this case, the `export` operator exports all the fields of the data set formatted according to the record properties specified in the braces.

## Error Handling during Import/Export

The `import` and `export` operators can return one of two types of error for each record: a failure or a warning.

### Failure

A failure means that both

- The `import` or `export` operator failed to read or write the record’s value correctly.
- The error makes it unlikely that the rest of the record can be interpreted.
An example of such a failure occurs when the import operator encounters an invalid length value of a variable-length field. On import, by default, an uninterpretable field is not written to the destination data set and processing continues to the next record. On export, by default, an uninterpretable field is not written to the destination file and processing continues to the next record.

However, you can configure the import and export operators to save records causing a failure in a reject data set. You can also configure the operators to terminate the application in the event of a failure. See Chapter 23, “The import Operator” and Chapter 24, “The export Operator”.

Orchestrate issues a message in the case of a failure.

- The message appears for up to five records in a row, for each partition of the imported or exported data set, when the same field causes the failure.
- After the fifth failure and message, messages no longer appear.
- After a record is successfully processed, the message counter is reset to zero and up to five more error messages per partition can be generated for the same field.

**Note**

No more than 25 failure messages can be output for the same failure condition during an import or export operation.

**Warning**

A warning means that the import operator or export operator failed to read or write the record’s value correctly but that the import or export of that record can continue. An example of a warning condition is a numeric field represented as ASCII text that contains all blanks. When such a condition occurs, the import or export operator does not issue a message and by default drops the record. To override this behavior, define a default value for the field that causes the warning.

If you have defined a default value for the record field that causes the warning:

- The import operator sets the field to its default value, writes the record to the destination data set, and continues with the next record.
- The export operator sets the field to its default value, exports the record, and continues with the next record.

**ASCII and EBCDIC Conversion Tables**

This section contains the ASCII to EBCDIC and EBCDIC to ASCII conversion tables used by the Orchestrate import/export operators.
If an input file uses a character set that is not the native character set of the host computer, the **import** operator must perform a conversion. For example, if ASCII is the native format for strings on your host computer, but the input data file represents strings using EBCDIC, you must convert EBCDIC to ASCII.

This section contains the lookup tables supplied by Orchestrate for converting between ASCII and EBCDIC. These conversion tables are as defined in DFSM S/M V S V 1 R2.0 U sing M agnetic Tapes, Document Number SC26-4923-01, by IBM.

**EBCDIC to ASCII**

Table 84 is an EBCDIC-to-ASCII conversion table that translates 8-bit EBCDIC characters to 7-bit ASCII characters. All EBCDIC characters that cannot be represented in 7 bits are represented by the ASCII character 0x1A. This translation is not bidirectional. Some EBCDIC characters cannot be translated to ASCII and some conversion irregularities exist in the table. See “Conversion Table Irregularities” on page 22-22 for more information.

<table>
<thead>
<tr>
<th>EBCDIC</th>
<th>ASCII</th>
<th>EBCDIC Meaning</th>
<th>ASCII</th>
<th>ASCII Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00</td>
<td>00 NUL</td>
<td>00 00 NUL</td>
<td>55 1A</td>
<td>AA 1A</td>
</tr>
<tr>
<td>01 01</td>
<td>01 SOH</td>
<td>01 01 SOH</td>
<td>56 1A</td>
<td>AB 1A</td>
</tr>
<tr>
<td>02 02</td>
<td>02 STX</td>
<td>02 02 STX</td>
<td>57 1A</td>
<td>AC 1A</td>
</tr>
<tr>
<td>03 03</td>
<td>03 ETX</td>
<td>03 03 ETX</td>
<td>58 1A</td>
<td>AD 1A</td>
</tr>
<tr>
<td>04 1A</td>
<td>09 SEL</td>
<td>04 1A SEL</td>
<td>59 1A</td>
<td>AE 1A</td>
</tr>
<tr>
<td>05 09</td>
<td>09 HT</td>
<td>05 09 HT</td>
<td>5A 5D</td>
<td>AF 1A</td>
</tr>
<tr>
<td>06 1A</td>
<td>RN L</td>
<td>06 1A RN L</td>
<td>5B 24</td>
<td>B0 1A</td>
</tr>
<tr>
<td>07 7F</td>
<td>DEL</td>
<td>07 7F DEL</td>
<td>5C 2A</td>
<td>B1 1A</td>
</tr>
<tr>
<td>08 1A</td>
<td>GE</td>
<td>08 1A GE</td>
<td>5D 29</td>
<td>B2 1A</td>
</tr>
<tr>
<td>09 1A</td>
<td>SPS</td>
<td>09 1A SPS</td>
<td>5E 3B</td>
<td>B3 1A</td>
</tr>
<tr>
<td>0A 1A</td>
<td>RPT</td>
<td>0A 1A RPT</td>
<td>5F 5E</td>
<td>B4 1A</td>
</tr>
<tr>
<td>0B 0B</td>
<td>VT</td>
<td>0B 0B VT</td>
<td>60 2D</td>
<td>B5 1A</td>
</tr>
<tr>
<td>0C 0C</td>
<td>FF</td>
<td>0C 0C FF</td>
<td>61 2F</td>
<td>B6 1A</td>
</tr>
</tbody>
</table>
Table 84  **EBCDIC to ASCII Conversion (continued)**

<table>
<thead>
<tr>
<th>EBCDIC</th>
<th>ASCII</th>
<th>EBCDIC Meaning</th>
<th>EBCDIC</th>
<th>ASCII</th>
<th>EBCDIC Meaning</th>
<th>EBCDIC</th>
<th>ASCII</th>
<th>EBCDIC Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0D</td>
<td>0D</td>
<td>CR</td>
<td>62</td>
<td>1A</td>
<td></td>
<td>B7</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>0E</td>
<td>0E</td>
<td>SO</td>
<td>63</td>
<td>1A</td>
<td></td>
<td>B8</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>0F</td>
<td>0F</td>
<td>SI</td>
<td>64</td>
<td>1A</td>
<td></td>
<td>B9</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>DLE</td>
<td>65</td>
<td>1A</td>
<td></td>
<td>BA</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>DC1</td>
<td>66</td>
<td>1A</td>
<td></td>
<td>BB</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>DC2</td>
<td>67</td>
<td>1A</td>
<td></td>
<td>BC</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>DC3</td>
<td>68</td>
<td>1A</td>
<td></td>
<td>BD</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1A</td>
<td>RES/ENP</td>
<td>69</td>
<td>1A</td>
<td></td>
<td>BE</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1A</td>
<td>NL</td>
<td>6A</td>
<td>7C</td>
<td></td>
<td>BF</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>08</td>
<td>BS</td>
<td>6B</td>
<td>2C</td>
<td>,</td>
<td>C0</td>
<td>7B</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1A</td>
<td>POC</td>
<td>6C</td>
<td>25</td>
<td></td>
<td>C1</td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>CAN</td>
<td>6D</td>
<td>5F</td>
<td>-</td>
<td>C2</td>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>EM</td>
<td>6E</td>
<td>3E</td>
<td>&gt;</td>
<td>C3</td>
<td>43</td>
<td>C</td>
</tr>
<tr>
<td>1A</td>
<td>1A</td>
<td>UBS</td>
<td>6F</td>
<td>3F</td>
<td>?</td>
<td>C4</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>1B</td>
<td>1A</td>
<td>CU1</td>
<td>70</td>
<td>1A</td>
<td></td>
<td>C5</td>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>1C</td>
<td>1C</td>
<td>IFS</td>
<td>71</td>
<td>1A</td>
<td></td>
<td>C6</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>1D</td>
<td>1D</td>
<td>IGS</td>
<td>72</td>
<td>1A</td>
<td></td>
<td>C7</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>1E</td>
<td>1E</td>
<td>IRS</td>
<td>73</td>
<td>1A</td>
<td></td>
<td>C8</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>1F</td>
<td>1F</td>
<td>ITB/ IUS</td>
<td>74</td>
<td>1A</td>
<td></td>
<td>C9</td>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>20</td>
<td>1A</td>
<td>DS</td>
<td>75</td>
<td>1A</td>
<td></td>
<td>CA</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>1A</td>
<td>SOS</td>
<td>76</td>
<td>1A</td>
<td></td>
<td>CB</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>1A</td>
<td>FS</td>
<td>77</td>
<td>1A</td>
<td></td>
<td>CC</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>1A</td>
<td>WUS</td>
<td>78</td>
<td>1A</td>
<td></td>
<td>CD</td>
<td>1A</td>
<td></td>
</tr>
</tbody>
</table>
### Table 84  **EBCDIC to ASCII Conversion (continued)**

<table>
<thead>
<tr>
<th>EBCDIC</th>
<th>ASCII</th>
<th>EBCDIC Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>1A</td>
<td>BYP/INP</td>
</tr>
<tr>
<td>25</td>
<td>0A</td>
<td>LF</td>
</tr>
<tr>
<td>26</td>
<td>17</td>
<td>ETB</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>ESC</td>
</tr>
<tr>
<td>28</td>
<td>1A</td>
<td>SA</td>
</tr>
<tr>
<td>29</td>
<td>1A</td>
<td>SFE</td>
</tr>
<tr>
<td>2A</td>
<td>1A</td>
<td>SM/SW</td>
</tr>
<tr>
<td>2B</td>
<td>1A</td>
<td>CSP</td>
</tr>
<tr>
<td>2C</td>
<td>1A</td>
<td>MFA</td>
</tr>
<tr>
<td>2D</td>
<td>05</td>
<td>ENQ</td>
</tr>
<tr>
<td>2E</td>
<td>06</td>
<td>ACK</td>
</tr>
<tr>
<td>2F</td>
<td>07</td>
<td>BEL</td>
</tr>
<tr>
<td>30</td>
<td>1A</td>
<td>BEL</td>
</tr>
<tr>
<td>31</td>
<td>1A</td>
<td>BEL</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
<td>SYN</td>
</tr>
<tr>
<td>33</td>
<td>1A</td>
<td>IR</td>
</tr>
<tr>
<td>34</td>
<td>1A</td>
<td>PP</td>
</tr>
<tr>
<td>35</td>
<td>1A</td>
<td>TRN</td>
</tr>
<tr>
<td>36</td>
<td>1A</td>
<td>NBS</td>
</tr>
<tr>
<td>37</td>
<td>04</td>
<td>EOT</td>
</tr>
<tr>
<td>38</td>
<td>1A</td>
<td>SBS</td>
</tr>
<tr>
<td>39</td>
<td>1A</td>
<td>IT</td>
</tr>
<tr>
<td>3A</td>
<td>1A</td>
<td>RFF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EBCDIC</th>
<th>ASCII</th>
<th>EBCDIC Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>60</td>
<td>CE 1A</td>
</tr>
<tr>
<td>7A</td>
<td>3A</td>
<td>: CF 1A</td>
</tr>
<tr>
<td>7B</td>
<td>23</td>
<td># D0 7D</td>
</tr>
<tr>
<td>7C</td>
<td>40</td>
<td>@ D1 4A J</td>
</tr>
<tr>
<td>7D</td>
<td>27</td>
<td>' D2 4B K</td>
</tr>
<tr>
<td>7E</td>
<td>3D</td>
<td>= D3 4C L</td>
</tr>
<tr>
<td>7F</td>
<td>22</td>
<td>&quot; D4 4D M</td>
</tr>
<tr>
<td>80</td>
<td>1A</td>
<td>D5 4E N</td>
</tr>
<tr>
<td>81</td>
<td>61</td>
<td>a D6 4F O</td>
</tr>
<tr>
<td>82</td>
<td>62</td>
<td>b D7 50 P</td>
</tr>
<tr>
<td>83</td>
<td>63</td>
<td>c D8 51 Q</td>
</tr>
<tr>
<td>84</td>
<td>64</td>
<td>d D9 52 R</td>
</tr>
<tr>
<td>85</td>
<td>65</td>
<td>e DA 1A</td>
</tr>
<tr>
<td>86</td>
<td>66</td>
<td>f DB 1A</td>
</tr>
<tr>
<td>87</td>
<td>67</td>
<td>g DC 1A</td>
</tr>
<tr>
<td>88</td>
<td>68</td>
<td>h DD 1A</td>
</tr>
<tr>
<td>89</td>
<td>69</td>
<td>i DE 1A</td>
</tr>
<tr>
<td>8A</td>
<td>1A</td>
<td>DF 1A</td>
</tr>
<tr>
<td>8B</td>
<td>1A</td>
<td>E0 5C \</td>
</tr>
<tr>
<td>8C</td>
<td>1A</td>
<td>E1 1A</td>
</tr>
<tr>
<td>8D</td>
<td>1A</td>
<td>E2 53 S</td>
</tr>
<tr>
<td>8E</td>
<td>1A</td>
<td>E3 54 T</td>
</tr>
<tr>
<td>8F</td>
<td>1A</td>
<td>E4 55 U</td>
</tr>
</tbody>
</table>
Table 84  **EBCDIC to ASCII Conversion (continued)**

<table>
<thead>
<tr>
<th>EBCDIC</th>
<th>ASCII</th>
<th>EBCDIC Meaning</th>
<th>EBCDIC</th>
<th>ASCII</th>
<th>EBCDIC Meaning</th>
<th>EBCDIC</th>
<th>ASCII</th>
<th>EBCDIC Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>3B</td>
<td>1A</td>
<td>CU3</td>
<td>90</td>
<td>1A</td>
<td></td>
<td>E5</td>
<td>56</td>
<td>V</td>
</tr>
<tr>
<td>3C</td>
<td>14</td>
<td>DC4</td>
<td>91</td>
<td>6A</td>
<td>j</td>
<td>E6</td>
<td>57</td>
<td>W</td>
</tr>
<tr>
<td>3D</td>
<td>15</td>
<td>NAK</td>
<td>92</td>
<td>6B</td>
<td>k</td>
<td>E7</td>
<td>58</td>
<td>X</td>
</tr>
<tr>
<td>3E</td>
<td>1A</td>
<td></td>
<td>93</td>
<td>6C</td>
<td>l</td>
<td>E8</td>
<td>59</td>
<td>Y</td>
</tr>
<tr>
<td>3F</td>
<td>1A</td>
<td>SUB</td>
<td>94</td>
<td>6D</td>
<td>m</td>
<td>E9</td>
<td>5A</td>
<td>Z</td>
</tr>
<tr>
<td>40</td>
<td>20</td>
<td>(space)</td>
<td>95</td>
<td>6E</td>
<td>n</td>
<td>EA</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>1A</td>
<td>RSP</td>
<td>96</td>
<td>6F</td>
<td>o</td>
<td>EB</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>1A</td>
<td></td>
<td>97</td>
<td>70</td>
<td>p</td>
<td>EC</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>1A</td>
<td></td>
<td>98</td>
<td>71</td>
<td>q</td>
<td>ED</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>1A</td>
<td></td>
<td>99</td>
<td>72</td>
<td>r</td>
<td>EE</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>1A</td>
<td></td>
<td>9A</td>
<td>1A</td>
<td></td>
<td>EF</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>1A</td>
<td></td>
<td>9B</td>
<td>1A</td>
<td></td>
<td>F0</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>47</td>
<td>1A</td>
<td></td>
<td>9C</td>
<td>1A</td>
<td></td>
<td>F1</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>48</td>
<td>1A</td>
<td></td>
<td>9D</td>
<td>1A</td>
<td></td>
<td>F2</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>49</td>
<td>1A</td>
<td></td>
<td>9E</td>
<td>1A</td>
<td></td>
<td>F3</td>
<td>33</td>
<td>3</td>
</tr>
<tr>
<td>4A</td>
<td>5B</td>
<td>¢</td>
<td>9F</td>
<td>1A</td>
<td></td>
<td>F4</td>
<td>34</td>
<td>4</td>
</tr>
<tr>
<td>4B</td>
<td>2E</td>
<td>.</td>
<td>A0</td>
<td>1A</td>
<td></td>
<td>F5</td>
<td>35</td>
<td>5</td>
</tr>
<tr>
<td>4C</td>
<td>3C</td>
<td>&lt;</td>
<td>A1</td>
<td>7E</td>
<td>(\infty)</td>
<td>F6</td>
<td>36</td>
<td>6</td>
</tr>
<tr>
<td>4D</td>
<td>28</td>
<td>(</td>
<td>A2</td>
<td>73</td>
<td>s</td>
<td>F7</td>
<td>37</td>
<td>7</td>
</tr>
<tr>
<td>4E</td>
<td>2B</td>
<td>+</td>
<td>A3</td>
<td>74</td>
<td>t</td>
<td>F8</td>
<td>38</td>
<td>8</td>
</tr>
<tr>
<td>4F</td>
<td>21</td>
<td></td>
<td></td>
<td>A4</td>
<td>75</td>
<td>u</td>
<td>F9</td>
<td>39</td>
</tr>
<tr>
<td>50</td>
<td>26</td>
<td>&amp;</td>
<td>A5</td>
<td>76</td>
<td>v</td>
<td>FA</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>1A</td>
<td></td>
<td>A6</td>
<td>77</td>
<td>w</td>
<td>FB</td>
<td>1A</td>
<td></td>
</tr>
</tbody>
</table>
Table 85 is an ASCII-to-EBCDIC conversion table that translates 7-bit ASCII characters to 8-bit EBCDIC characters. This translation is not bidirectional. Some EBCDIC characters cannot be translated to ASCII and some conversion irregularities exist in the table. See “Conversion Table Irregularities” on page 22-22 for more information.

Table 84  **EBCDIC to ASCII Conversion (continued)**

<table>
<thead>
<tr>
<th>EBCDIC</th>
<th>ASCII</th>
<th>EBCDIC Meaning</th>
<th>EBCDIC</th>
<th>ASCII</th>
<th>EBCDIC Meaning</th>
<th>EBCDIC</th>
<th>ASCII</th>
<th>EBCDIC Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>52 1A</td>
<td>A7 78</td>
<td>x</td>
<td>FC 1A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53 1A</td>
<td>A8 79</td>
<td>y</td>
<td>FD 1A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54 1A</td>
<td>A9 7A</td>
<td>z</td>
<td>FE 1A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ASCII to EBCDIC**

Table 85 is an ASCII-to-EBCDIC conversion table that translates 7-bit ASCII characters to 8-bit EBCDIC characters. This translation is not bidirectional. Some EBCDIC characters cannot be translated to ASCII and some conversion irregularities exist in the table. See “Conversion Table Irregularities” on page 22-22 for more information.

Table 85  **ASCII to EBCDIC Conversion**

<table>
<thead>
<tr>
<th>ASCII</th>
<th>EBCDIC</th>
<th>ASCII Meaning</th>
<th>ASCII</th>
<th>EBCDIC</th>
<th>ASCII Meaning</th>
<th>ASCII</th>
<th>EBCDIC</th>
<th>ASCII Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00</td>
<td>NUL</td>
<td></td>
<td>2B 4E</td>
<td>+</td>
<td></td>
<td>56 E5</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>01 01</td>
<td>SOH</td>
<td></td>
<td>2C 6B</td>
<td>,</td>
<td></td>
<td>57 E6</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>02 02</td>
<td>STX</td>
<td></td>
<td>2D 60</td>
<td>-</td>
<td></td>
<td>58 E7</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>03 03</td>
<td>ETX</td>
<td></td>
<td>2E 4B</td>
<td>.</td>
<td></td>
<td>59 E8</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>04 1A</td>
<td>SEL</td>
<td></td>
<td>2F 61</td>
<td>/</td>
<td></td>
<td>5A E9</td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>05 09</td>
<td>HT</td>
<td></td>
<td>30 F0</td>
<td>0</td>
<td></td>
<td>5B 4A</td>
<td>[</td>
<td></td>
</tr>
<tr>
<td>06 1A</td>
<td>RNL</td>
<td></td>
<td>31 F1</td>
<td>1</td>
<td></td>
<td>5C E0</td>
<td>\</td>
<td></td>
</tr>
<tr>
<td>07 7F</td>
<td>DEL</td>
<td></td>
<td>32 F2</td>
<td>2</td>
<td></td>
<td>5D 5A</td>
<td>]</td>
<td></td>
</tr>
<tr>
<td>08 1A</td>
<td>GE</td>
<td></td>
<td>33 F3</td>
<td>3</td>
<td></td>
<td>5E 5F</td>
<td>^</td>
<td></td>
</tr>
<tr>
<td>09 1A</td>
<td>SPS</td>
<td></td>
<td>34 F4</td>
<td>4</td>
<td></td>
<td>5F 6D</td>
<td>_</td>
<td></td>
</tr>
<tr>
<td>0A 25</td>
<td>LF</td>
<td></td>
<td>35 F5</td>
<td>5</td>
<td></td>
<td>60 79</td>
<td>'</td>
<td></td>
</tr>
<tr>
<td>0B 0B</td>
<td>VT</td>
<td></td>
<td>36 F6</td>
<td>6</td>
<td></td>
<td>61 81</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>
## Table 85  ASCII to EBCDIC Conversion (continued)

<table>
<thead>
<tr>
<th>ASCII</th>
<th>EBCDIC</th>
<th>ASCII Meaning</th>
<th>ASCII</th>
<th>EBCDIC</th>
<th>ASCII Meaning</th>
<th>ASCII</th>
<th>EBCDIC</th>
<th>ASCII Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0C</td>
<td>0C</td>
<td>FF</td>
<td>37</td>
<td>F7</td>
<td>7</td>
<td>62</td>
<td>82</td>
<td>b</td>
</tr>
<tr>
<td>0D</td>
<td>0D</td>
<td>CR</td>
<td>38</td>
<td>F8</td>
<td>8</td>
<td>63</td>
<td>83</td>
<td>c</td>
</tr>
<tr>
<td>0E</td>
<td>0E</td>
<td>SO</td>
<td>39</td>
<td>F9</td>
<td>9</td>
<td>64</td>
<td>84</td>
<td>d</td>
</tr>
<tr>
<td>0F</td>
<td>0F</td>
<td>SI</td>
<td>3A</td>
<td>7A</td>
<td></td>
<td>65</td>
<td>85</td>
<td>e</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>DLE</td>
<td>3B</td>
<td>5E</td>
<td>;</td>
<td>66</td>
<td>86</td>
<td>f</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>DC1</td>
<td>3C</td>
<td>4C</td>
<td>&lt;</td>
<td>67</td>
<td>87</td>
<td>g</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>DC2</td>
<td>3D</td>
<td>7E</td>
<td>=</td>
<td>68</td>
<td>88</td>
<td>h</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>DC3</td>
<td>3E</td>
<td>6E</td>
<td>&gt;</td>
<td>69</td>
<td>89</td>
<td>i</td>
</tr>
<tr>
<td>14</td>
<td>3C</td>
<td>DC4</td>
<td>3F</td>
<td>6F</td>
<td>?</td>
<td>6A</td>
<td>91</td>
<td>j</td>
</tr>
<tr>
<td>15</td>
<td>3D</td>
<td>NAK</td>
<td>40</td>
<td>7C</td>
<td>@</td>
<td>6B</td>
<td>92</td>
<td>k</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>SYN</td>
<td>41</td>
<td>C1</td>
<td>A</td>
<td>6C</td>
<td>93</td>
<td>l</td>
</tr>
<tr>
<td>17</td>
<td>26</td>
<td>ETB</td>
<td>42</td>
<td>C2</td>
<td>B</td>
<td>6D</td>
<td>94</td>
<td>m</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>CAN</td>
<td>43</td>
<td>C3</td>
<td>C</td>
<td>6E</td>
<td>95</td>
<td>n</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>EM</td>
<td>44</td>
<td>C4</td>
<td>D</td>
<td>6F</td>
<td>96</td>
<td>o</td>
</tr>
<tr>
<td>1A</td>
<td>3F</td>
<td>SUB</td>
<td>45</td>
<td>C5</td>
<td>E</td>
<td>70</td>
<td>97</td>
<td>p</td>
</tr>
<tr>
<td>1B</td>
<td>27</td>
<td>ESC</td>
<td>46</td>
<td>C6</td>
<td>F</td>
<td>71</td>
<td>98</td>
<td>q</td>
</tr>
<tr>
<td>1C</td>
<td>1C</td>
<td>FS</td>
<td>47</td>
<td>C7</td>
<td>G</td>
<td>72</td>
<td>99</td>
<td>r</td>
</tr>
<tr>
<td>1D</td>
<td>1D</td>
<td>GS</td>
<td>48</td>
<td>C8</td>
<td>H</td>
<td>73</td>
<td>A2</td>
<td>s</td>
</tr>
<tr>
<td>1E</td>
<td>1E</td>
<td>RS</td>
<td>49</td>
<td>C9</td>
<td>I</td>
<td>74</td>
<td>A3</td>
<td>t</td>
</tr>
<tr>
<td>1F</td>
<td>1F</td>
<td>US</td>
<td>4A</td>
<td>D1</td>
<td>J</td>
<td>75</td>
<td>A4</td>
<td>u</td>
</tr>
<tr>
<td>20</td>
<td>40</td>
<td>(space)</td>
<td>4B</td>
<td>D2</td>
<td>K</td>
<td>76</td>
<td>A5</td>
<td>v</td>
</tr>
<tr>
<td>21</td>
<td>4F</td>
<td>!</td>
<td>4C</td>
<td>D3</td>
<td>L</td>
<td>77</td>
<td>A6</td>
<td>w</td>
</tr>
<tr>
<td>22</td>
<td>7F</td>
<td>&quot;</td>
<td>4D</td>
<td>D4</td>
<td>M</td>
<td>78</td>
<td>A7</td>
<td>x</td>
</tr>
<tr>
<td>23</td>
<td>7B</td>
<td>#</td>
<td>4E</td>
<td>D5</td>
<td>N</td>
<td>79</td>
<td>A8</td>
<td>y</td>
</tr>
</tbody>
</table>
Conversion Table Irregularities

The EBCDIC-to-ASCII and ASCII-to-EBCDIC conversion tables previously shown are standard conversion tables. However, owing to the nature of EBCDIC-to-ASCII and ASCII-to-EBCDIC conversions, certain irregularities exist in the conversion tables. For example, an exclamation point is defined in EBCDIC as 0x5A. In ASCII 7-bit and 8-bit codes, an exclamation point is defined as 0x21. 

Table 86 shows the conversion irregularities.

Table 86 Conversion Table Irregularities

<table>
<thead>
<tr>
<th>EBCDIC Code</th>
<th>8-Bit ASCII Code</th>
<th>7-Bit ASCII Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphic Hex</td>
<td>Graphic Hex</td>
<td>Graphic Hex</td>
</tr>
<tr>
<td>¢</td>
<td>4A</td>
<td>[</td>
</tr>
<tr>
<td>!</td>
<td>5A</td>
<td>]</td>
</tr>
<tr>
<td>[</td>
<td>AD</td>
<td>(n/ a)</td>
</tr>
<tr>
<td>]</td>
<td>BD</td>
<td>(n/ a)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4F</td>
</tr>
<tr>
<td>_</td>
<td>6A</td>
<td>_</td>
</tr>
</tbody>
</table>
The import operator reads one or more non-Orchestrate source files and converts the source data to a destination Orchestrate data set.

Example 1: Importing from a Single Data File
Example 2: Importing from Multiple Data Files
Data Flow Diagram

![Data Flow Diagram for the import operator]

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>0</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1, or 2 if you specify optional data sets</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>none</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>output data set 0: user defined reject data set: reject:raw;</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>none</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel or sequential</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set(s)</td>
<td>clear by default</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>
The `import` operator:

- Takes one or more files or named pipes as input. You can import data from one or more sources, provided all the imported data has the same record layout.
- Writes its results to a single output data set.
- Allows you to specify an optional output data set to hold records that are not successfully imported.
- Has an output interface schema corresponding to the import schema of the data files read by the operator.

The `import` operator can import source files containing several types of data:

- Data in variable-length blocked/spanned records: Imported records have a variable-length format. This format is equivalent to IBM format-V, format-VS, format-VB, and format-VBS files.
- Data in fixed-length records: This format is equivalent to IBM format-F and format-FB files.
- Data in prefixed records: Records fields are prefixed by a length indicator. Records can be of either fixed or variable length.
- Delimited records: Each record in the input file is separated by one or more delimiter characters (such as the ASCII newline character `
`). Records can be of either fixed or variable length.
- Implicit: Data with no explicit record boundaries.

Orchestrate accepts a variety of record layouts. For example, a data file may represent integers as ASCII strings, contain variable-length records, or represent multi-byte data types (for example, 32-bit integer) in big-endian or little-endian format. See Chapter 25, “Import/Export Properties” for more information on data representation.

When the `import` operator imports a fixed-length `string` or `ustring` that is less-than or greater-than the declared schema length specification, an error is generated.

**Syntax and Options**

Terms in italic typeface are option strings you supply. When your option string contains a space or a tab character, you must enclose it in single quotes.

```
import
  { -file filename | -fileset filename | -filepattern pattern }
  {[ -schema schema ] | [ -schemafile filename ] }
  [ -source progname [ args ] ] [ -sourcelist filename ] [ -checkpoint n ]
  [ -dontUseOffsetsWithSources ] [ -filter command ] [ -keepPartitions ]
```
There are two types of required options.

- You must include one of the following options to specify the imported files: `-file`, `-filepattern`, `-fileset`, `-source`, `-sourcelist`.

- You must include either `-schema` or `-schemafile` to define the layout of the imported data unless you specify `-fileset` and it contains a schema. You can select only one of these three options.

You can optionally include other arguments.

### Table 88  import Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-checkpoint</code></td>
<td><code>-checkpoint n</code></td>
</tr>
<tr>
<td></td>
<td>Import n records per segment to the output data set. (A data segment contains all records written to a data set by a single Orchestrate step.)</td>
</tr>
<tr>
<td></td>
<td>By default, the value of n is 0, that is, the entire input is imported for each iteration of the step. If the step containing the import operator performs multiple iterations, the output data set of the import operator contains multiple copies of the input data.</td>
</tr>
<tr>
<td></td>
<td>The source files of the import operation must all be data files or source programs, that is, no pipes or other input devices can be used. In addition, you cannot specify a filter on the input.</td>
</tr>
<tr>
<td></td>
<td>An import operator that creates a segmented output data set must be contained in a checkpointed step.</td>
</tr>
<tr>
<td><code>-dontUseOffsetsWithSources</code></td>
<td><code>-dontUseOffsetsWithSources</code></td>
</tr>
<tr>
<td></td>
<td>Start the source program's output stream at 0. Do not use this option with checkpointing.</td>
</tr>
</tbody>
</table>
Table 88  import Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-file</td>
<td>file nodeName file0 ... file nodeName filen</td>
</tr>
<tr>
<td></td>
<td>Specifies an input file or pipe.</td>
</tr>
<tr>
<td></td>
<td>You can include multiple -file options to specify multiple input files. The file names may contain multi-byte Unicode characters. Note: You can specify a hyphen to signal that import takes its input from the stdin of osh.nodeName optionally specifies the name of the node on which the file is located. If you omit nodeName, the importer assumes that the file is located on the node on which the application is invoked. You cannot use -file with -filepattern, -fileset, -source, or -sourcelist.</td>
</tr>
<tr>
<td>-filepattern</td>
<td>filepattern pattern</td>
</tr>
<tr>
<td></td>
<td>Specifies a group of files to import.</td>
</tr>
<tr>
<td></td>
<td>For example, you can use the statement:</td>
</tr>
<tr>
<td></td>
<td>filepattern data*.txt</td>
</tr>
<tr>
<td></td>
<td>You cannot use -filepattern with the -file, -fileset, -source, -sourcelist or -readers options.</td>
</tr>
<tr>
<td>-fileset</td>
<td>file_set fs</td>
</tr>
<tr>
<td></td>
<td>Specifies file_set.fs, a file containing a list of files, one per line, used as input by the operator. The suffix .fs identifies the file to Orchestrate as a file set. See &quot;File Sets&quot; on page 23-10. file_set may contain multi-byte Unicode characters.</td>
</tr>
<tr>
<td></td>
<td>To delete a file set and the data files to which it points, invoke the Orchestrate data set administration utility, orchadmin. See the Orchestrate 7.0 Installation and Administration Manual for details. You cannot use -fileset with -file, -filepattern, -source, or -sourcelist.</td>
</tr>
</tbody>
</table>
-filter

Specifies a UNIX command to process input files as the data is read from a file and before it is converted into the destination data set. command may contain multi-byte Unicode characters.

For example, you could specify the following filter:

`-filter 'grep 1997'`

to filter out all records that do not contain the string "1997". Note that the source data to the import operator for this example must be newline-delimited for grep to function properly.

Orchestrate checks the return value of the -filter process, and errors out when an exit status is not OK.

You cannot use -filter with -source.

-keepPartitions

Partitions the imported data set according to the organization of the input file(s).

By default, record ordering is not preserved, because the number of partitions of the imported data set is determined by the configuration file and any constraints applied to the data set.

However, if you specify -keepPartitions, record ordering is preserved, because the number of partitions of the imported data set equals the number of input files and the preserve-partitioning flag is set in the destination data set.

-missingFile

Determines how the importer handles a missing input file.

By default, the importer fails and the step terminates if an input file is missing (corresponding to -missingFile error). Specify -missingFile okay to override the default.

However, if the input file name has a node name prefix of "*", missing input files are ignored and the import operation continues. (This corresponds to -missingFile okay.)
Chapter 23  The import Operator

23

Orchestrate 7.0 Operators Reference

Table 88  import Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-readers</td>
<td>Specifies the number of instances of the import operator on each processing node. The default is one operator per node per input data file. If numReaders is greater than one, each instance of the import operator reads a contiguous range of records from the input file. The starting record location in the file for each operator (or seek location) is determined by the data file size, the record length, and the number of instances of the operator, as specified by numReaders, which must be greater than zero. All instances of the import operator for a data file run on the processing node specified by the file name. The output data set contains one partition per instance of the import operator, as determined by numReaders. The imported data file(s) must contain fixed-length records (as defined by the record_length=fixed property). The data source must be a file or files. No other devices are allowed. This option is mutually exclusive with the filepattern option.</td>
</tr>
<tr>
<td>-rejects</td>
<td>Configures operator behavior if a record is rejected. The default behavior is to continue. Rejected records are counted but discarded. The number of rejected records is printed as a log message at the end of the step. However, you can configure the operator to either fail and terminate the application or save, that is, create output data set 1 to hold reject records. If -rejects fail is specified and a record is not successfully imported, the import operator issues an error and the step terminates. If -rejects fail is not specified and a record is not successfully imported, the import operator issues a warning and the step does not terminate.</td>
</tr>
<tr>
<td>-reportProgress</td>
<td>By default (yes) the operator displays a progress report at each 10% interval when the importer can ascertain file size. Reporting occurs only if: import reads a file as opposed to a named pipe, the file is greater than 100 KB, records are fixed length, and there is no filter on the file. Disable this reporting by specifying -reportProgress no.</td>
</tr>
</tbody>
</table>
Table 88  import Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-schema</td>
<td>-schema record_schema</td>
</tr>
<tr>
<td></td>
<td>Specifies the import record schema. The value to this option may contain</td>
</tr>
<tr>
<td></td>
<td>multi-byte Unicode characters.</td>
</tr>
<tr>
<td></td>
<td>You can also specify a file containing the record schema using the syntax:</td>
</tr>
<tr>
<td></td>
<td>-schema record @'file_name'</td>
</tr>
<tr>
<td></td>
<td>where file_name is the path name of the file containing the record schema.</td>
</tr>
<tr>
<td></td>
<td>You cannot use -schema with -schemafile.</td>
</tr>
<tr>
<td>-schemafile</td>
<td>-schemafile file_name</td>
</tr>
<tr>
<td></td>
<td>Specifies the name of a file containing the import record schema. It may</td>
</tr>
<tr>
<td></td>
<td>contain multi-byte Unicode characters.</td>
</tr>
<tr>
<td></td>
<td>You cannot use -schemafile with -schema.</td>
</tr>
<tr>
<td>-source</td>
<td>-source prog_name args</td>
</tr>
<tr>
<td></td>
<td>Specifies the name of a program providing the source data to the import</td>
</tr>
<tr>
<td></td>
<td>operator. Orchestrate calls prog_name and passes to it any arguments</td>
</tr>
<tr>
<td></td>
<td>specified. prog_name and args may contain multi-byte Unicode characters.</td>
</tr>
<tr>
<td></td>
<td>You can specify multiple -source arguments to the operator. Orchestrate</td>
</tr>
<tr>
<td></td>
<td>creates a pipe to read data from each specified prog_name.</td>
</tr>
<tr>
<td></td>
<td>You can prefix prog_name with either a node name to explicitly specify the</td>
</tr>
<tr>
<td></td>
<td>processing node that executes prog_name or a node name prefix of &quot;**, which</td>
</tr>
<tr>
<td></td>
<td>causes Orchestrate to run prog_name on all processing nodes executing the</td>
</tr>
<tr>
<td></td>
<td>operator.</td>
</tr>
<tr>
<td></td>
<td>If this import operator runs as part of a checkpointed step, as defined by</td>
</tr>
<tr>
<td></td>
<td>-checkpoint, Orchestrate calls prog_name once for each iteration of the step.</td>
</tr>
<tr>
<td></td>
<td>Orchestrate always appends three arguments to prog_name as shown below:</td>
</tr>
<tr>
<td></td>
<td>prog_name args -s H L</td>
</tr>
<tr>
<td></td>
<td>where H and L are 32-bit integers. H and L are set to 0 for the first step</td>
</tr>
<tr>
<td></td>
<td>iteration or if the step is not checkpointed.</td>
</tr>
<tr>
<td></td>
<td>For each subsequent iteration of a checkpointed step, H and L specify the (64-</td>
</tr>
<tr>
<td></td>
<td>bit) byte offset (H = upper 32 bits, L = lower 32 bits) at which the source</td>
</tr>
<tr>
<td></td>
<td>program's output stream should be restarted.</td>
</tr>
<tr>
<td></td>
<td>At the end of each step iteration, prog_name receives a signal indicating a</td>
</tr>
<tr>
<td></td>
<td>broken pipe. The prog_name can recognize this signal and perform any</td>
</tr>
<tr>
<td></td>
<td>cleanup before exiting.</td>
</tr>
<tr>
<td></td>
<td>Orchestrate checks the return value of the -source process, and errors out</td>
</tr>
<tr>
<td></td>
<td>when an exit status is not OK.</td>
</tr>
<tr>
<td></td>
<td>You cannot use -source with -filter, -file, -filepattern, or -fileset.</td>
</tr>
</tbody>
</table>
How to Import Data

To import data:

1. Follow the procedure described in “Specify the Layout of the Imported Data.”
2. Follow the procedure described in “Specify the Source of the Imported Data.”

Specify the Layout of the Imported Data

Define a schema describing the layout of the data file to be imported. Refer to the chapter on Orchestrate data sets in the Orchestrate 7.0 User Guide for a general discussion of record schemas. Refer to the these sections for a discussion of record schemas as they pertain to import/export operations: “Record Schemas” on page 22-3 and “Complete and Partial Schemas” on page 22-7.

A schema definition can also define record and field properties. Refer to Chapter 25, “Import/Export Properties” to learn about setting up record and field properties.

The Orchestrate data set that is output by the import operator is formatted according to the schema you have defined. However, this output schema does not define properties that were set up in the original schema definition.

Note

If you do not provide a schema, Orchestrate assumes that imported data is formatted according to the default schema discussed in “The Default Import Schema” on page 22-12.

Specify the Source of the Imported Data

The source of the data to be imported can be one of these:

- A file or named pipe (see “Files and Named Pipes” on page 23-10)
• A file set (see “File Sets” on page 23-10)
• A file pattern (see “File Patterns” on page 23-11)
• A source program’s output (see “Source Program’s Output” on page 23-11)
• The output of several source programs defined in a source list (see “List of Source Programs” on page 23-11)

You can also explicitly specify the nodes and directories from which the operator imports data (see “Nodes and Directories” on page 23-11).

Files and Named Pipes

Specify the -file option and the path name of the file or named pipe, which may optionally be preceded by the node name. You may specify more than one file or named pipe. For each one, repeat the operation.

---

**Note**

For external sequential data files (as opposed to Orchestrate file sets) that are to be processed by a UNIX custom operator that functions in parallel, first import the data sequentially and then export it in parallel using the file *file_name* option. The wild card ensures that the operator writes to the file on every node on which it executes. (See Chapter 24, “The export Operator.”)

---

File Sets

A file set is a text file containing a list of source files to import. The file set must contain one file name per line. The name of the file has the form file_name.fs, where fs identifies the file to Orchestrate as a file set.

Specify the -fileset option and the path name of the file set.

Shown below is a sample file set:

```plaintext
--Orchestrate File Set v1
--LFile
node0:/home/user1/files/file0
node0:/home/user1/files/file1
node0:/home/user1/files/file2
--LFile
node1:/home/user1/files/file0
node1:/home/user1/files/file1
--Schema
record {record_delim="\n"}
  a:int32;
  b:int32;
  c:int16;
  d:sfloat;
  e:string[10];
}
```

The first line of the file set must be specified exactly as shown above.
The list of all files on each processing node must be preceded by the line `--LFile` and each file name must be prefixed by its node name.

A file set can optionally contain the record schema of the source files as the last section, beginning with `--Schema`. If you omit this part of the file set, you must specify a schema to the operator by means of either the `-schema` or `-schemafile` option.

### File Patterns

A file pattern specifies a group of similarly named files from which to import data. The wild card character allows for variations in naming. For example, the file pattern `inFile*.data` imports files that begin with `inFile` and end with `.data`. The file names can contain multi-byte Unicode characters.

For example, to import the data from the files that match the file pattern `state*.txt`, use the statement:

```
-filepattern state*.txt
```

Specify the `-filepattern` option and the pattern.

### Source Program’s Output

You can specify the name of a program providing the source data to the `import` operator.

Specify the `-source` option and the program name and program arguments, if any.

### List of Source Programs

You can specify a file containing the names of multiple programs that provide source data to the `import` operator.

Specify the `-sourcelist` option and the file name.

### Nodes and Directories

Indicate the source to be imported by specifying:

```
[ nodeName ] path_name
```

The operator assumes that `path_name` is relative to the working directory from which the application was invoked.

You can also indicate one of these:

- A specific processing node on which the operator runs. Do this by specifying the node’s name. When you do, Orchestrate creates an instance of the operator on that node to read the `path_name`. The `nodeName` must correspond to a `node` or `fastname` parameter of the Orchestrate configuration file. See the discussion of configuration files in the Orchestrate 7.0 Installation and Administration Manual.
• All processing nodes on which the operator runs. Do this by specifying the asterisk wild card character (*). When you do, Orchestrate reads path_name on every node on which the operator is running.

For example, you can supply the following specification as the file pattern on the osh command line:

```
*:inFile*.data
```

This imports all files of the form `inFile*.data` residing on all processing nodes of the default node pool. You can include a relative or absolute path as part of `inFile`. If you do not supply an absolute path, `import` searches for the files on all nodes using the same current working directory as the one from which you invoked the application.

### Example 1: Importing from a Single Data File

In this example, a single file is imported from disk. The file contains fixed-length records and no field or record delimiters. The step contains a single instance of the `import` operator, which:

• Reads a file from disk
• Converts the file to a single data set
• Saves the output in a persistent data set
• Saves records that cause a failure in a reject data set

**Figure 41** shows the data flow for this example.
Example 2: Importing from Multiple Data Files

In this example two files are imported from disk. Each one contains fixed-length records. Figure 32 shows the data-flow diagram for this example.

Here is the `osh` schema declaration for this example:

```bash
$ example1_schema="record {record_length = fixed, delim = none, binary} {  
  Name:string[80];  
  Age:int8 {default = 127};  
  Income:dfloat {text, width = 8};  
  Address:subrec (    
    Street:string[64];  
    City:string[24];  
    State:string[4];  
  );  
  Zip:string[12];  
  Phone:string[12];  
}"
```
The format of the source file and the import schema are the same as those in “Example 1: Importing from a Single Data File” on page 23-12 and the step similarly contains a single instance of the import operator. However, this example differs from the first one in that it imports data from two flat files instead of one and does not save records that cause an error into a reject data set.

Specify the data flow with the following osh code:

```bash
$ osh "import -file inFile0.data -file inFile1.data -schema $example1_schema > outDS.ds"
```
The export Operator

The export operator exports an Orchestrate data set to one or more UNIX data files or named pipes located on one or more disk drives.

- Data Flow Diagram 24 2
- Properties 24 3
- Syntax and Options 24 5
- How to Export Data 24 10
  - Specify Record and Field Layout of Exported Data 24 10
  - Specifying Destination Files or Named Pipes for Exported Data 24 12
    - Files and Named Pipes 24 12
    - File Sets 24 12
    - Destination Program’s Input 24 14
    - List of Destination Programs 24 14
    - Nodes and Directories 24 14
- Example 1: Data Set Export to a Single File 24 15
- Example 2: Data Set Export to Multiple Files 24 17

The **export** operator exports an Orchestrate data set to one or more UNIX data files or named pipes located on one or more disk drives.
Data Flow Diagram

- **input data set**
- **list of file names (fileset option)**
- **exported files**
- **export**
- **outRec:*;**
- **reject data set (optional)**
Properties

Table 89 export Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>0 or 1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>(\text{inRec:}*); plus user-supplied export schema</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>output files: none</td>
</tr>
<tr>
<td></td>
<td>reject data set: (\text{outRec:}*);</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>(\text{inRec -&gt; outRec}) if you specify a reject data set</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>any</td>
</tr>
<tr>
<td>Collection method</td>
<td>any</td>
</tr>
<tr>
<td>Preserve-partitioning flag in</td>
<td>clear</td>
</tr>
<tr>
<td>output data set(s)</td>
<td></td>
</tr>
<tr>
<td>Composite operator</td>
<td>yes</td>
</tr>
</tbody>
</table>

The **export** operator:

- Takes a single Orchestrate data set as input.
- Writes its results to one or more output files.
- Allows you to specify an optional output data set to hold records that cause a failure.
- Takes as its input interface schema the schema of the exported data.
- Generates an error and terminates if you try to write data to a file that is not empty. (This behavior can be overridden.)
- Deletes files already written if the step of which it is a part fails. (This behavior can be overridden.)

The source data set may be persistent (stored on disk) or virtual (not stored on disk).

The **export** operator writes several formats of exported data, including:

- Fixed-length records: a fixed-length format equivalent to IBM format-F and format-FB files.
• Data in prefixed records: Record fields are prefixed by a length indicator. Records can be of either fixed or variable length.

• Delimited records: A delimiter such as the ASCII line feed character is inserted after every record. Records can be of either fixed or variable length.

• Implicit: No explicit boundaries delimit the record.

Note: The export operator does not support variable-length blocked records (IBM format-VB or format-VBS).

Orchestrate accepts a variety of record layouts for the exported file. For example, a data file may represent integers as ASCII strings, contain variable-length records, or represent multi-byte data types in big-endian or little-endian format. See Chapter 25, “Import/Export Properties” for more information.

Note: When your character setting is UTF-16, the export operator appends and prepends the byte-order mark \x{0fe}\x{0ff}\x{00} to every column value.
Syntax and Options

Terms in italic typeface are option strings you supply. When your option string contains a space or a tab character, you must enclose it in single quotes.

```plaintext
export
{ -file filename | -fileset listName | -destination programe[ args] | destinationlist filename }
{ -schema schema | -schemafile schemafie }
{-add_bom { utf16be | utf16le | utf8 }
{-append}
{-create | -replace | -discard_records | -discard_schema_and_records}
{-diskpool diskpool}
{-dontUseOffsetsWithDestinations}
{-filter command}
{-maxFileSize numMB] [-nocleanup]
{-overwrite}
{-prefix prefix]
{-rejects continue | fail | save}
{-single[FilePerPartition]}
{-suffix suffix]
{-writeSchema | -omitSchema]
```

**Note**
The following option values can contain multi-byte Unicode characters:
- the `-file` and `-destinationlist` file names and the base name of the file for the `-fileset` option
- the program name and arguments to the `-destination` option
- the `-schema` option schema and the schema filename given to the `-schemafile` option
- the prefix value given to the `-prefix` option, and the suffix value given to the `-suffix` option
- the `-filter` option command
- the value given to the `-diskpool` option

There are two types of required options:
- You must include exactly one of the following to specify the exported files:
  `-file`, `-fileset`, `-destination`, or `-destinationlist`. 
- You must include exactly one of `-schema` or `-schemafile` to define the layout of the exported data.

Table 90  **export Operator Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-add_bom</code></td>
<td>`.add_bom { utf16be</td>
</tr>
<tr>
<td></td>
<td>With this option you can add a BOM to your exported file.</td>
</tr>
<tr>
<td></td>
<td>The <code>utf16be</code> value specifies <code>FE FF</code>, <code>utf16le</code> specifies <code>FF FE</code>, and <code>utf8</code> specifies <code>EF BB BF</code>.</td>
</tr>
<tr>
<td><code>-append</code></td>
<td><code>-append</code></td>
</tr>
<tr>
<td></td>
<td>Append exported data to an existing file. By default the step terminates if you attempt to export data to a file that is not empty. This option overrides the default behavior.</td>
</tr>
<tr>
<td></td>
<td>You cannot use this option with <code>-overwrite</code>.</td>
</tr>
<tr>
<td><code>-destination</code></td>
<td><code>-destination prog_name [ args]</code></td>
</tr>
<tr>
<td></td>
<td>In single quotation marks specify the name of a program that reads the data generated by the <code>export</code> operator. Specify the program's arguments, if any. Orchestrate calls <code>prog_name</code> and passes to it any specified arguments.</td>
</tr>
<tr>
<td></td>
<td>You can specify multiple <code>-destination</code> options to the operator: for each <code>-destination</code>, specify the option and supply the <code>prog_name</code> and <code>args</code> (if any). The <code>prog_name</code> and <code>args</code> values may contain multi-byte Unicode values.</td>
</tr>
<tr>
<td></td>
<td>If this <code>export</code> operator runs as part of a checkpointed step, Orchestrate calls <code>prog_name</code> once for each iteration of the step.</td>
</tr>
<tr>
<td></td>
<td>Orchestrate always appends three additional arguments to <code>prog_name</code>: <code>prog_name [ args] -s H L</code></td>
</tr>
<tr>
<td></td>
<td>where <code>H</code> and <code>L</code> are 32-bit integers. For the first step iteration, or if the step is not checkpointed, <code>H</code> and <code>L</code> are set to 0.</td>
</tr>
<tr>
<td></td>
<td>For each subsequent iteration of a checkpointed step, <code>H</code> and <code>L</code> specify the (64-bit) byte offset (<code>H</code> = upper 32 bits, <code>L</code> = lower 32 bits) of the exported data in the total export stream from the operator.</td>
</tr>
<tr>
<td></td>
<td>After all data has been written to the program, <code>prog_name</code> is called once more with an appended switch of <code>-e</code> (corresponding to end of file) and is not passed the <code>-s</code> switch. On last call <code>prog_name</code> can perform any final operation, for example, write a trailing label to a tape.</td>
</tr>
<tr>
<td></td>
<td>If the export operation fails, Orchestrate calls <code>prog_name</code> once with the appended switch <code>-c</code> (cleanup) and no <code>-s</code> switch. This gives the program an opportunity to clean up.</td>
</tr>
<tr>
<td></td>
<td>You cannot use this option with <code>-filter</code>.</td>
</tr>
</tbody>
</table>
### Table 90  export Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>-destinationlist</strong></td>
<td>Specifies in single quotation marks <code>file_name</code>, the name of a file containing the names of multiple destination programs, where each command line is listed on a separate line of the file. <code>file_name</code> may contain multi-byte Unicode characters. Orchestrate calls the programs as if you specified multiple <code>-destination</code> options. See the description of <code>-destination</code> for more information.</td>
</tr>
<tr>
<td><strong>-dontUseOffsets</strong></td>
<td>Do not supply the <code>-s, H, L</code> arguments to destination programs. This means that the byte offset is always 0. See the <code>-destination</code> option for more information.</td>
</tr>
<tr>
<td><strong>-file</strong></td>
<td>Supply the name of an output file or pipe. The file or pipe must be empty unless you specify either the <code>-append</code> or <code>-overwrite</code> option. You can include multiple <code>-file</code> options to specify multiple input files. For each one, specify <code>-file</code> and supply the file name. The file name can contain multi-byte Unicode characters. <strong>Note:</strong> You can specify a hyphen to signal that <code>export</code> writes its output to the <code>stdout</code> for <code>osh</code>. You cannot use this option with <code>-filesset</code>.</td>
</tr>
</tbody>
</table>
The suboptions are:

- **-create**  Create the file set. If it already exists, this option generates an error.
- **-replace** Remove the existing fileset and replace it with a new one.
- **-discard_records** Keep the existing files and schema listed in \texttt{filesetName.fs} but discard the records; create the file set if it does not exist.
- **-discard_schema_and_records** Keep existing files listed in \texttt{filesetName.fs} but discard the schema and records; create the file set if it does not exist.

The previous suboptions are mutually exclusive with each other and also with the **-append** option.

- **-diskpool**  \texttt{diskpool} Specify the name of the disk pool into which to write the file set. \texttt{diskpool} can contain multi-byte Unicode characters.
- **-maxFileSize** \texttt{numMB} Specify the maximum file size in MB. Supply integers. The value of \texttt{numMB} must be equal to or greater than 1.
- **-omitSchema** Omit the schema from \texttt{filesetName.fs}. The default is for the schema to be written to the file set.

---

Table 90  **export Operator Options (continued)**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-fileset \texttt{filesetName.fs} { create</td>
<td>replace</td>
</tr>
</tbody>
</table>

Specifies the name of the file set, \texttt{filesetName}, a file into which the operator writes the names of all data files that it creates. The suffix \texttt{.fs} identifies the file to Orchestrate as a file set. \texttt{filesetName} can contain multi-byte Unicode characters.

The name of each export file generated by the operator is written to \texttt{filesetName.fs}, one name per line.

---

continued
Table 90  export Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-fileset</td>
<td>Specify the prefix of the name of the file set components. It can contain multi-byte Unicode characters. If you do not specify a prefix, the system writes the following: export.username, where username is your login.</td>
</tr>
<tr>
<td>-prefix</td>
<td>Remove the existing file set and create a new one.</td>
</tr>
<tr>
<td>-replace</td>
<td>Create one file per partition. The default is to create many files per partition. This can be shortened to -single.</td>
</tr>
<tr>
<td>-singleFilePerPartition</td>
<td>Specifying the suffix of the name of the file set components. It can contain multi-byte Unicode characters. The operator omits the suffix by default.</td>
</tr>
<tr>
<td>-suffix</td>
<td>Only with -fileset. Write the schema to the file set. This is the default.</td>
</tr>
<tr>
<td>-filter</td>
<td>Specifies a UNIX command to process all exported data after the data set is exported but before the data is written to a file. command can contain multi-byte Unicode characters.</td>
</tr>
<tr>
<td>-nocleanup</td>
<td>Configures the operator to skip the normal data file deletion if the step fails. By default, the operator attempts to delete partial data files and perform other cleanup operations on step failure.</td>
</tr>
<tr>
<td>-overwrite</td>
<td>The default action of the operator is to issue an error if you attempt to export data to a file that is not empty. Select -overwrite to override the default behavior and overwrite the file.</td>
</tr>
<tr>
<td>-rejects</td>
<td>Configures operator behavior if a record is rejected. The default behavior is to continue. Rejected records are counted but discarded. The number of rejected records is printed as a log message at the end of the step.</td>
</tr>
<tr>
<td></td>
<td>However, you can configure the operator to either fail and terminate the application or save, that is, create output data set 0 to hold reject records.</td>
</tr>
<tr>
<td></td>
<td>If you use -rejects fail, osh generates an error upon encountering a record that cannot be successfully exported; otherwise osh generates a warning upon encountering a record that cannot be successfully exported.</td>
</tr>
</tbody>
</table>
How to Export Data

Specify Record and Field Layout of Exported Data

Provide the export operator with a schema that defines the record format of the exported data. The schema can define:

- The properties of the exported record, that is, the destination record
- The fields of the source data set to be exported
- The properties of exported fields
- The order in which they are exported

The export operator writes only those fields specified by the export schema and ignores other fields, as in the following example, where the schema of the source Orchestrate data set differs from that of the exported data:

Table 90  export Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-schema</td>
<td>-schema record_schema</td>
</tr>
<tr>
<td></td>
<td>Specifies in single quotation marks the export record schema. You can also specify a file containing the record schema using the syntax:</td>
</tr>
<tr>
<td></td>
<td>-schema record @file_name</td>
</tr>
<tr>
<td></td>
<td>where file_name is the path name of the file containing the record schema. The file_name and record_schema can contain multi-byte Unicode characters. You cannot use this option with -schemafile.</td>
</tr>
<tr>
<td>-schemafile</td>
<td>-schemafile schema_file</td>
</tr>
<tr>
<td></td>
<td>Specifies in single quotation marks the name of a file containing the export record schema. The file name can contain multi-byte Unicode characters. This is equivalent to:</td>
</tr>
<tr>
<td></td>
<td>-schema record @schema_file</td>
</tr>
<tr>
<td></td>
<td>You cannot use this option with -schema.</td>
</tr>
</tbody>
</table>
In the example shown above, the export schema drops the fields Name, Address, and Phone and moves the field Gender to the beginning of the record.

Table 91  Example: Data Set Schema Versus Export Schema

<table>
<thead>
<tr>
<th>Source Orchestrate Data Set Schema</th>
<th>Export Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>record (</td>
<td>record (</td>
</tr>
<tr>
<td>Name: string;</td>
<td>Gender: string[1];</td>
</tr>
<tr>
<td>Address: string;</td>
<td>State: string[2];</td>
</tr>
<tr>
<td>State: string[2];</td>
<td>Age: int8;</td>
</tr>
<tr>
<td>Age: int8;</td>
<td>Income: dfloat;</td>
</tr>
<tr>
<td>Gender: string[1];</td>
<td>)</td>
</tr>
<tr>
<td>Income: dfloat;</td>
<td></td>
</tr>
<tr>
<td>Phone: string;</td>
<td></td>
</tr>
</tbody>
</table>

In the example shown above, the export schema drops the fields Name, Address, and Phone and moves the field Gender to the beginning of the record.

Note: If you do not provide a schema, Orchestrate assumes that exported data is formatted according to the default schema discussed in “The Default Export Schema” on page 22-13.

Here is how you set up export schemas:

- To export all fields of the source record and format them according to the default export schema, specify the following:

  record ( )


- To export all fields of the source record but override one or more default record properties, add new properties, or do both, specify:

  record {record_properties} ()

  Refer to Chapter 25, “Import/Export Properties” to learn about setting up record and field properties.

- To export selected fields of the source record, define them as part of the schema definition, as follows:

  record ( field_definition0; ... field_definitionN; )

  where field_definition0; ... field_definitionN are the fields chosen for export. No record-level properties have been defined and the default export schema is applied.

- You can define properties for records and for fields, as in the following example:

  record {delim = none, binary} { 
    Name: string {delim = ',,'}; 
    Age: int8 {default = 127}; 
    Address: subrec { 

Refer to Chapter 25, “Import/Export Properties” to learn about setting up record and field properties.

See the chapter on Orchestrate data sets in the Orchestrate 7.0 User Guide for a general discussion of schemas. Refer to the following sections for a discussion of schemas as they pertain to import/export operations: “Record Schemas” on page 22-3 and “Complete and Partial Schemas” on page 22-7.

### Specifying Destination Files or Named Pipes for Exported Data

The destination of the data to be exported can be one of the following:

- One or more files or named pipes (see “Files and Named Pipes” on page 24-11)
- A file set (see “File Sets” on page 24-11)
- A program’s input (see “Destination Program’s Input” on page 24-13)
- The input to several programs defined in a destination list (see “List of Destination Programs” on page 24-13).

You can also explicitly specify the nodes and directories to which the operator exports data (see “Nodes and Directories” on page 24-13).

### Files and Named Pipes

Specify the -file option and the path name of the file or named pipe, which may optionally be preceded by the node name. You may specify more than one file or named pipe. For each one, repeat the operation.

| Important | For external sequential data files (as opposed to Orchestrate file sets) that are to be processed by a UNIX custom operator that functions in parallel, you must first import the data sequentially. See Chapter 23, “The import Operator”. You then export the data in parallel by means of the file *file_name option. The wildcard (*) ensures that the operator writes to the file on every node on which it executes. |

### File Sets

The export operator can generate and name exported files, write them to their destination, and list the files it has generated in a file whose extension is .fs. The data files and the file that lists them are called a file set. They are established by
means of the `fileset` option. This option is especially useful because some operating systems impose a 2 GB limit on the size of a file and you must distribute the exported files among nodes to prevent overruns.

When you choose `-fileset`, the `export` operator runs on nodes that contain a disk in the `export` disk pool. If there is no `export` disk pool, the operator runs in the default disk pool, generating a warning when it does. However, if you have specified a disk pool other than `export` (by means of the `-diskpool` option), the operator does not fall back and the export operation fails.

The `export` operator writes its results according to the same mechanism. However, you can override this behavior by means of the `-diskpool` option. (Refer to the Orchestrate 7.0 Installation and Administration Manual chapter on configuration files for information on disk pools.)

The amount of data that may be stored in each destination data file is limited (typically to 2 GB) by the characteristics of the file system and the amount of free disk space available. The number of files created by a file set depends on:

- The number of processing nodes in the default node pool
- The number of disks in the `export` or default disk pool connected to each processing node in the default node pool
- The size of the partitions of the data set

You name the file set and can define some of its characteristics. The name of the file set has the form `file_name.fs`, where `.fs` identifies the file as a file set to Orchestrate. Specify the `-fileset` option and the path name of the file set.

| Note | When you choose `-fileset`, the `export` operator names the files it generates and writes the name of each one to the file set. By contrast, you have to name the files when you choose the `-file` option. |

The names of the exported files created by the operator have the following form:

```
nodeName:dirName/prefixPXXXXX_FYYYYsuffix
```

where:

- `nodeName` is the name of the node on which the file is stored and is written automatically by Orchestrate.
- `dirName` is the directory path and is written automatically by Orchestrate.
- `prefix` is by default `export` `userName`, where `userName` is your login name, and is written automatically by Orchestrate. However, you can either define your own prefix or suppress the writing of one by specifying `""` as the prefix.
- `XXXXX` is a 6-digit hexadecimal string preceded by `P` specifying the partition number of the file written by `export`. The first partition is partition
000000. The partition numbers increase for every partition of the data set but might not be consecutive.

- YYYY is a 4-digit hexadecimal string preceded by F specifying the number of the data file in a partition. The first file is 0000. The file numbers increase for every file but might not be consecutive.

Orchestrate creates one file on every disk connected to each node used to store an exported partition.

- suffix is a user-defined file-name suffix. By default, it is omitted.

For example, if you specify a prefix of “file” and a suffix of “_exported”, the third file in the fourth partition would be named:

```
node1: dir_name/fileP000003_F0002_exported
```

Some data sets, such as sorted data sets, have a well-defined partitioning scheme. Because the files created by the export operator are numbered by partition and by data file within a partition, you can sort the exported files by partition number and file number to reconstruct your sorted data.

**Destination Program’s Input**

You can specify the name of a program receiving the exported data. Such a program is called a destination program.

Specify the `-destination` option and the program name and program arguments, if any.

Orchestrate creates a pipe to write data to each specified program.

**List of Destination Programs**

You can specify a file containing the names of multiple programs that receive data from the export operator. Specify the `-destinationlist` option and the file name.

The number of entries in `-destinationlist` determines the number of partitions on which the operator runs, regardless of the configuration file’s contents. Incoming data is repartitioned as necessary.

**Nodes and Directories**

Indicate the destination of exported data by specifying:

```
[ nodeName: ] path_name
```

If you omit the optional node name, the operator exports files or named pipes only to the processing node on which the application was invoked.

If you specify only the file name the operator assumes that path_name is relative to the working directory from which the application was invoked.
You can also indicate one of these:

- A specific processing node to which the output is written. Do this by specifying the node’s name. The name must correspond to a `nodeName` or `fastname` parameter of the Orchestrate configuration file. (Refer to the discussion of configuration files in the Orchestrate 7.0 Installation and Administration Manual.)

- All processing nodes on which the operator runs. Do this using the asterisk wild card character (*). When you do, Orchestrate writes one file to every node on which the operator is running.

For example, you can supply the following as the exported file name on the `osh` command line:

```
*:outFile.data
```

This exports data to files called `outFile.data`, which are created on all processing nodes on which the operator runs. The name `outFile.data` is a relative path name and the exporter writes files using the same current working directory as the one from which you invoked the application.

**Example 1: Data Set Export to a Single File**

In this example, the `export` operator:

- Runs on a single processing node because it writes to only one file.
- Takes a persistent data set as its source.
- Writes records to a reject data set, if they cannot be successfully exported.
- The destination files created by the `export` operator are laid out according to the schema supplied in an argument to the operator.
Figure 43 shows the data flow for Example 1.

![Data Flow Diagram](image)

The following osh code specifies the layout of the destination data file:

```bash
exp_example_1="record {delim = none, binary} {
    Name: string {delim = ','};
    Age: int8 {default = 127};
    Address: subrec {
        Street: string[64];
        City: string[24];
        State: string[4];
    };
    Zip: string[12];
    Phone: string[12];
}"
```

In this schema, the `export` operator automatically inserts a newline character at the end of each record according to the default. However, numeric data is exported in binary mode and no field delimiters are present in the exported data file, except for the comma delimiting the Name field. Both properties are non-default (see "The Default Export Schema" on page 22-13). They have been established through the definition of record and field properties. The record properties are `delim = none` and `binary`. The field Name is explicitly delimited by a comma (`delim = ','`) and this field property overrides the record-level property of `delim = none`. See Chapter 25, "Import/Export Properties" for a full discussion of this subject.
The following `osh` code uses the specified schema to export the data:

```
$ osh "export -file outFile.dat
    -schema $exp_example_1
    -rejects save < inDS.ds > errDS.ds"
```

Example 2: Data Set Export to Multiple Files

In this example the operator exports a data set to multiple data files by means of the `-fileset` option and saves records to a reject data set if they cannot be successfully exported.

Figure 33 shows the data flow of this example:

The following `osh` code specifies the layout of the destination data files:

```
$ exp_example_2="record {record_length = fixed, delim = none, binary} {
   Name:string[64];
   Age:int8;
   Income:dfloat;
   Zip:string[12];
   Phone:string[12];
}"
```

In this example, all fields are of specified length and the schema defines the fixed record length property.
The following `osh` code uses the specified schema to export the data:

```bash
$ osh "export -fileset listFile.fs -schema $exp_example_2 -rejects save < inDS.ds > errDS.ds"
```

Upon completion of the step, `listFile.fs` contains a list of every destination data file created by the operator, one file name per line, and contains the record schema used for the export.

Here are the contents of `listFile.fs`:

```bash
--Orchestrate File Set v1
--LFile
  node0:/home/user1/sfiles/node0/export.user1.P000000_F0000
  node0:/home/user1/sfiles/node0/export.user1.P000000_F0001
--LFile
  node1:/home/user1/sfiles/node1/export.user1.P000001_F0000
  node1:/home/user1/sfiles/node1/export.user1.P000001_F0001
  node1:/home/user1/sfiles/node1/export.user1.P000001_F0002
--Schema
  record {record_length = fixed, delim = none, binary} {
    Name:string[64];
    Age:int8;
    Income:dfloat;
    Zip:string[12];
    Phone:string[12];
  }
```

For more information on file sets, see “File Sets” on page 24-11.
Import/Export Properties

Import/Export properties allow you to define the layout of imported and exported data, including the representation of numbers, strings, times, and dates.

Setting Properties 25 3
Properties 25 5
  Record-Level Properties 25 5
  Field Properties 25 6
Properties: Reference Listing 25 14
  actual_length 25 17
  ascii 25 18
  big_endian 25 19
  binary 25 19
  c_format 25 21
  charset 25 22
  check_intact 25 22
  date_format 25 23
  days_since 25 24
  decimal_separator 25 25
  default 25 25
  default_date_format 25 26
  default_time_format 25 26
  delim 25 27
  delim_string 25 28
  drop 25 29
  ebcdic 25 30
  export_ebcdic_as_ascii 25 31
  fill 25 31
  final_delim 25 32
  final_delim_string 25 33
  fix_zero 25 34
  generate 25 34
  import_ascii_as_ebcdic 25 35
in_format  25 36
intact   25 37
julian   25 37
link     25 38
little_endian  25 39
max_width  25 39
midnight_seconds  25 40
native_endian  25 41
nofix_zero  25 41
null_field  25 42
null_length  25 43
out_format  25 44
overpunch  25 45
packed   25 45
padchar  25 46
position  25 47
precision  25 48
prefix   25 49
print_field  25 51
quote    25 52
record_delim  25 53
record_delim_string  25 53
record_format  25 53
record_length  25 54
record_prefix  25 55
reference  25 55
round    25 55
scale    25 56
separate  25 58
skip     25 59
tagcase  25 59
text     25 60
time_format  25 61
timestamp_format  25 62
vector_prefix  25 63
width    25 64
zoned    25 65
You can add import/export properties to a schema when you import and export flat files. Properties define the layout of imported and exported data, including such things as how numbers, strings, times, and dates are represented. Properties can apply to records or to fields, and there are numerous default properties that you do not need to specify.

The property determines how the data is represented in the file from which the data is imported or to which the data is exported.

**Note**
Properties apply only when you import or export data. They are not part of the internal Orchestrate representation of the data, which only needs to know the type of the data.

Orchestrate assumes that data to import or export is formatted according to default properties if:

- You do not specify either record or field properties for imported or exported data
- Orchestrate performs an unmodified implicit import or export operation (see “Implicit Import/Export Operations with No Schemas Specified” on page 22-11)

The defaults are discussed in “The Default Import Schema” on page 22-12 and “The Default Export Schema” on page 22-13. You explicitly define properties to override these defaults.

### Setting Properties

You establish the properties of imported and exported data as part of defining the data’s schema. Define record-level properties (which describe the format of the entire record) as follows:

```
record { prop1 [ = arg1 ], prop2 [ = arg2 ], ... propN [ = argN ] }  
(field_definitions ...) 
```

- Define record properties after the word `record` and before the enumeration of field definitions.
- Enclose the definition in braces `{ }`.
- Define properties in a comma-separated list if there is more than one property.
- Attribute values to the properties with the attribution sign `=`. If the attributed values are strings, enclose them in single quotes.
- Specify special characters by starting with a backslash escape character. For example, `\t` represents an ASCII tab delimiter character.
For example, the following defines a record in which all fields are delimited by a comma except the final one, which is delimited by a comma followed by an ASCII space character:

```plaintext
record {delim_string = ',', final_delim_string = ', '} {
    a:int32;
    b:string;
    c:int8;
    d:raw;
}
```

Define field-level properties (which describe the format of a single field) as follows:

```plaintext
field_definition { prop1 [ = arg1 ], prop2 [ = arg2 ], ... propN [ = argN ] };
```

where `field_definition` is the name and data type of the field.

- Define field properties after the `field_definition` and before its final semi-colon.
- Enclose the definition of properties in braces ( {} ).
- Define properties in a comma-separated list if there is more than one property.
- Attribute values to the properties with the attribution sign ( = ). If the attributed values are strings, enclose them in single quotes.
- Specify special characters by starting with a backslash escape character. For example, \t represents an ASCII tab delimiter character.

For example, the following specifies that the width of a string field is 40 bytes and that the ASCII space pad character is used when it is exported:

```plaintext
record (a: string { width = 40, padchar = ' ' }; )
```

You can specify many default values for properties at the record level. If you do, the defined property applies to all fields of that data type in the record, except where locally overridden. For example, if numeric data of imported or exported records are represented in binary format, you can define the binary property at record level, as in the following example:

```plaintext
record {binary, delim = none} (a:int32; b:int16; c:int8;)
```

With one exception, properties set for an individual field can override the default properties set at record level. The exception is the fill property, which is defined at record level but cannot be overridden by an individual field or fields.

For example, the following schema sets the record length as fixed with no field delimiters and the layout of all fields as binary. However, the definition of the properties of the purposeOfLoan field are as follows: the field is formatted as text, its length is 5 bytes (and not the 4 bytes of the int32 data type), and it is delimited by a comma:

```plaintext
record {
    record_length = fixed, delim = none, binary
    purposeOfLoan: string { width = 5, delim = ', ' };
}
```
Properties

Certain properties are associated with entire records, and other properties with fields of specific data types, but most can be used with fields of all data types and the records that contain them. This section contains the following topics:

- “Record-Level Properties” on page 25-5
- “Numeric Field Properties” on page 25-7
- “String Field Properties” on page 25-8
- “Ustring Field Properties” on page 25-9
- “Decimal Field Properties” on page 25-9
- “Date Field Properties” on page 25-10
- “Time Field Properties” on page 25-10
- “Timestamp Field Properties” on page 25-11
- “Raw Field Properties” on page 25-11
- “Vector Properties” on page 25-11
- “Nullable Field Properties” on page 25-12
- “Tagged Subrecord Field Properties” on page 25-13

Each property that is described in the following section is discussed in detail in “Properties: Reference Listing” on page 25-14.

Record-Level Properties

Some properties apply only to entire records. These properties, which establish the record layout of the imported or exported data or define partial schemas, can be specified only at the record level.

If you include no properties, default import and export properties are used; see “The Default Import Schema” on page 22-12 and “The Default Export Schema” on page 22-13.
Table 92 lists the record-level properties that cannot be set at the field level.

Table 92  Setting Record Level Properties

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Use</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>intact</td>
<td>Defines a partial schema and optionally verifies it</td>
<td>25-37</td>
</tr>
<tr>
<td>check_intact</td>
<td>and</td>
<td>25-23</td>
</tr>
<tr>
<td>record_length</td>
<td>Defines the fixed length of a record</td>
<td>25-54</td>
</tr>
<tr>
<td>record_prefix</td>
<td>Defines field’s length prefix as being 1, 2, or 4 bytes long</td>
<td>25-55</td>
</tr>
<tr>
<td>record_format</td>
<td>With (\text{type} = \text{implicit}), field length is determined by field content.</td>
<td>25-54</td>
</tr>
<tr>
<td></td>
<td>With (\text{type} = \text{varying}), defines IBM blocked or spanned format.</td>
<td></td>
</tr>
<tr>
<td>record_delim</td>
<td>Defines character(s) delimiting a record</td>
<td>25-53</td>
</tr>
<tr>
<td>record_delim_string</td>
<td>and</td>
<td>25-53</td>
</tr>
<tr>
<td>fill</td>
<td>Defines the value used to fill gaps between fields of an exported record. This property applies to export only and cannot be set at the field level.</td>
<td>25-32</td>
</tr>
</tbody>
</table>

Field Properties

You can define properties of imported and exported fields. This section lists field properties according to the data types that they can be used with, under the following categories.

- “Numeric Field Properties” on page 25-7
- “String Field Properties.” on page 25-8
- “Decimal Field Properties” on page 25-9
- “Date Field Properties” on page 25-10
- “Time Field Properties” on page 25-10
- “Timestamp Field Properties” on page 25-11
- “Raw Field Properties” on page 25-11
- “Vector Properties” on page 25-11
There is some overlap in these tables. For example, a field might be both decimal and nullable, and so field properties in both the decimal and nullable categories would apply to the field. Furthermore many properties such as `delim` apply to fields of all (or many) data types, and to avoid tedious repetition are not listed in any of the categories. However, they all appear in “Properties: Reference Listing” on page 25-14.

**Numeric Field Properties**

Numeric data types can be signed or unsigned integers of 8, 16, 32, or 64 bits:

- `int8, int16, int32, int64`
- `uint8, uint16, uint32, uint64`

Numeric data types can also be single- or double-precision floating-point numbers:

- `sfloat`
- `dfloat`

By default, the import and export operators assume that numeric fields are represented as text. The import operator invokes the C functions `strtol()`, `strtoul()`, or `strtod()` to convert the text representation to a numeric format. The export operator invokes the C function `sprintf()` to convert the numeric representation to text.

When exported as text, numeric fields take up a varying number of bytes, based on the size of the actual field values. The maximum number of bytes is as follows:

- 8-bit signed or unsigned integers: 4 bytes
- 16-bit signed or unsigned integers: 6 bytes
- 32-bit signed or unsigned integers: 11 bytes
- 64-bit signed or unsigned integers: 21 bytes
- Single-precision float: 14 bytes (sign, digit, decimal point, 7 fraction, "E", sign, 2 exponent)
- Double-precision float: 24 bytes (sign, digit, decimal point, 16 fraction, "E", sign, 3 exponent)
**Numeric Field Properties**

**Table 93  Numeric Field Properties**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Use</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>big_endian</td>
<td>Specifies the byte ordering as big-endian</td>
<td>25-19</td>
</tr>
<tr>
<td>c_format</td>
<td>Non-default format of text-numeric-text translation</td>
<td>25-21</td>
</tr>
<tr>
<td>in_format</td>
<td>Format of text translation to numeric field</td>
<td>25-36</td>
</tr>
<tr>
<td>little_endian</td>
<td>Specifies the byte ordering as little-endian</td>
<td>25-39</td>
</tr>
<tr>
<td>max_width</td>
<td>Defines the maximum width of the destination field</td>
<td>25-40</td>
</tr>
<tr>
<td>native_endian</td>
<td>Specifies the byte ordering as native-endian</td>
<td>25-41</td>
</tr>
<tr>
<td>out_format</td>
<td>Format of numeric translation to text field</td>
<td>25-44</td>
</tr>
<tr>
<td>padchar</td>
<td>Pad character of exported strings or numeric values</td>
<td>25-46</td>
</tr>
<tr>
<td>width</td>
<td>Defines the exact width of the destination field</td>
<td>25-64</td>
</tr>
</tbody>
</table>

**String Field Properties.**

**Table 94  String Field Properties**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Use</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_width</td>
<td>Defines the maximum width of the destination field</td>
<td>25-40</td>
</tr>
<tr>
<td>padchar</td>
<td>Defines the pad character of exported strings or numeric values</td>
<td>25-46</td>
</tr>
<tr>
<td>width</td>
<td>Defines the exact width of the destination field</td>
<td>25-64</td>
</tr>
</tbody>
</table>
Ustring Field Properties

Table 95  ustring Field Properties

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Use</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>charset</td>
<td>At the field level, it defines the character set to be used for ustring fields; at the record level it applies to the other import/export properties that support multi-byte Unicode character data.</td>
<td>25-22</td>
</tr>
<tr>
<td>max_width</td>
<td>Defines the maximum width of the destination field</td>
<td>25-40</td>
</tr>
<tr>
<td>padchar</td>
<td>Defines the pad character of exported strings or numeric values</td>
<td>25-46</td>
</tr>
<tr>
<td>width</td>
<td>Defines the exact width of the destination field</td>
<td>25-64</td>
</tr>
</tbody>
</table>

Decimal Field Properties

Table 96  Decimal Field Properties

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Use</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal_separator</td>
<td>Specifies an ASCII character to separate the integer and fraction components of a decimal</td>
<td>25-25</td>
</tr>
<tr>
<td>fix_zero</td>
<td>Treat a packed decimal field containing all zeros (normally illegal) as a valid representation of zero</td>
<td>25-34</td>
</tr>
<tr>
<td>max_width</td>
<td>Defines the maximum width of the destination field</td>
<td>25-40</td>
</tr>
<tr>
<td>packed</td>
<td>Defines the field as containing a packed decimal</td>
<td>25-45</td>
</tr>
<tr>
<td>precision</td>
<td>Defines the precision of a decimal</td>
<td>25-49</td>
</tr>
<tr>
<td>round</td>
<td>Defines the rounding mode</td>
<td>25-56</td>
</tr>
<tr>
<td>scale</td>
<td>Defines the scale of a decimal</td>
<td>25-57</td>
</tr>
<tr>
<td>separate</td>
<td>Defines the field as containing an unpacked decimal with a separate sign byte</td>
<td>25-58</td>
</tr>
<tr>
<td>width</td>
<td>Defines the exact width of the destination field</td>
<td>25-64</td>
</tr>
<tr>
<td>zoned</td>
<td>Defines the field as containing an unpacked decimal in text format</td>
<td>25-65</td>
</tr>
</tbody>
</table>
**Date Field Properties**

Table 97  Date Field Properties

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Use</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>big_endian</td>
<td>Specifies the byte ordering as big-endian</td>
<td>25-19</td>
</tr>
<tr>
<td>binary</td>
<td>In this context, a synonym for the julian property</td>
<td>25-19</td>
</tr>
<tr>
<td>charset</td>
<td>Specifies the character set for the date</td>
<td>25-22</td>
</tr>
<tr>
<td>date_format</td>
<td>Defines a text-string date format or uformat format other than the default; uformat can contain multi-byte Unicode characters</td>
<td>25-23</td>
</tr>
<tr>
<td>days_since</td>
<td>The field stores the date as a signed integer containing the number of days since date_in_ISO_format or uformat.</td>
<td>25-25</td>
</tr>
<tr>
<td>default_date_format</td>
<td>Provides support for international date components</td>
<td>25-26</td>
</tr>
<tr>
<td>julian</td>
<td>Defines the date as a binary numeric value containing the Julian day</td>
<td>25-38</td>
</tr>
<tr>
<td>little_endian</td>
<td>Specifies the byte ordering as little-endian</td>
<td>25-39</td>
</tr>
<tr>
<td>native_endian</td>
<td>Specifies the byte ordering as native-endian</td>
<td>25-41</td>
</tr>
<tr>
<td>text</td>
<td>Specifies text as the data representation</td>
<td>25-60</td>
</tr>
</tbody>
</table>

**Time Field Properties**

Table 98  Time Field Properties

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Use</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>big_endian</td>
<td>Specifies the byte ordering as big-endian</td>
<td>25-19</td>
</tr>
<tr>
<td>binary</td>
<td>In this context, a synonym for the julian property</td>
<td>25-19</td>
</tr>
<tr>
<td>charset</td>
<td>Specifies the character set for the time field.</td>
<td>25-22</td>
</tr>
<tr>
<td>default_time_format</td>
<td>Provides support for international time components</td>
<td>25-27</td>
</tr>
<tr>
<td>little_endian</td>
<td>Specifies the byte ordering as little_endian</td>
<td>25-39</td>
</tr>
<tr>
<td>native_endian</td>
<td>Specifies the byte ordering as native_endian</td>
<td>25-41</td>
</tr>
<tr>
<td>midnight_seconds</td>
<td>Represent the time field as a binary 32-bit integer containing the number of seconds elapsed from the previous midnight.</td>
<td>25-40</td>
</tr>
</tbody>
</table>
Table 98  Time Field Properties (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Use</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>time_format</code></td>
<td>Defines a text-string time format or uformat format other than the default</td>
<td>25-61</td>
</tr>
<tr>
<td><code>midnight_seconds</code></td>
<td>Defines the field as a binary 32-bit integer containing the number of seconds elapsed from the previous midnight; see “midnight_seconds” on page 25-40.</td>
<td>25-40</td>
</tr>
<tr>
<td><code>native_endian</code></td>
<td>Specifies the byte ordering as native-endian</td>
<td>25-41</td>
</tr>
<tr>
<td><code>text</code></td>
<td>Specifies text as the data representation</td>
<td>25-60</td>
</tr>
<tr>
<td><code>timestamp_format</code></td>
<td>Defines a text-string date format or uformat other than the default; uformat can contain multi-byte Unicode characters</td>
<td>25-61</td>
</tr>
</tbody>
</table>

**Timestamp Field Properties**

Table 99  Timestamp Field Properties

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Use</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>big_endian</code></td>
<td>Specifies the byte ordering as big-endian.</td>
<td>25-19</td>
</tr>
<tr>
<td><code>binary</code></td>
<td>In this context, a synonym for the <code>julian</code> property</td>
<td>25-19</td>
</tr>
<tr>
<td><code>charset</code></td>
<td>Specifies the character set for the timestamp field</td>
<td>25-22</td>
</tr>
<tr>
<td><code>little_endian</code></td>
<td>Specifies the byte ordering as little_endian</td>
<td>25-39</td>
</tr>
<tr>
<td><code>native_endian</code></td>
<td>Specifies the byte order as native_endian</td>
<td>25-41</td>
</tr>
<tr>
<td><code>text</code></td>
<td>Specifies text as the data representation</td>
<td>25-60</td>
</tr>
<tr>
<td><code>timestamp_format</code></td>
<td>Defines a text-string date format or uformat other than the default; uformat can contain multi-byte Unicode characters</td>
<td>25-62</td>
</tr>
</tbody>
</table>

**Raw Field Properties**

There are no type-specific properties for raw fields.

**Vector Properties**

Orchestrate supports vector fields of any data type except subrecord and tagged subrecord. Orchestrate vectors can be of either fixed or variable length.
No special properties are required for fixed-length vectors. The import and export operators perform the number of import/export iterations defined by the number of elements in the vector.

However, variable-length vectors can have the properties described in Table 100.

Table 100  Variable-Length Vector Field Properties

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Use</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>link</strong></td>
<td>This field holds the number of elements in a variable-length vector</td>
<td></td>
</tr>
<tr>
<td><strong>prefix</strong></td>
<td>Defines the length of prefix data, which specifies the length in bytes</td>
<td></td>
</tr>
<tr>
<td><strong>vector_prefix</strong></td>
<td>Defines the length of vector prefix data, which specifies the number of elements</td>
<td></td>
</tr>
</tbody>
</table>

Nullable Field Properties

All Orchestrate data types support null values. Fields may be declared to be nullable or not.

Table 101 describes the null field properties.

Table 101  Nullable Field Properties

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Use</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>actual_length</strong></td>
<td>Defines the number of bytes to skip in an imported record if the field contains a null; and the number of bytes to fill with null-field or specified pad-character, if the exported field contains a null</td>
<td></td>
</tr>
<tr>
<td><strong>null_field</strong></td>
<td>Specifies the value representing null.</td>
<td></td>
</tr>
<tr>
<td><strong>null_length</strong></td>
<td>Specifies the value of the length prefix of a variable-length field that contains a null</td>
<td></td>
</tr>
</tbody>
</table>

Subrecord Field Properties

The following properties apply to subrecords. These properties establish the subrecord layout of the imported or exported data. They cannot be specified at the field level.
If you include no properties, default import and export properties are used; see “The Default Import Schema” on page 22-12 and “The Default Export Schema” on page 22-13.

Table 102 Setting Record Level Properties

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Use</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>Defines the value used to fill gaps between fields of an exported record. This property applies to export only and cannot be set at the field level</td>
<td>25-32</td>
</tr>
<tr>
<td>record_delim</td>
<td>Specifies a single ASCII or multi-byte Unicode character to delimit a record</td>
<td>25-53</td>
</tr>
<tr>
<td>record_delim_string</td>
<td>Specifies an string or ustring to delimit a record</td>
<td>25-53</td>
</tr>
<tr>
<td>record_format</td>
<td>With type = implicit, field length is determined by field content.</td>
<td>25-54</td>
</tr>
<tr>
<td></td>
<td>With type = varying, defines IBM blocked or spanned format.</td>
<td></td>
</tr>
<tr>
<td>record_length</td>
<td>Defines the fixed length of a record</td>
<td>25-54</td>
</tr>
<tr>
<td>record_prefix</td>
<td>Defines field’s length prefix as being 1, 2, or 4 bytes long</td>
<td>25-55</td>
</tr>
<tr>
<td>record_format</td>
<td>With type = implicit, field length is determined by field content.</td>
<td>25-54</td>
</tr>
<tr>
<td></td>
<td>With type = varying, defines IBM blocked or spanned format.</td>
<td></td>
</tr>
</tbody>
</table>

Tagged Subrecord Field Properties

A tagged subrecord is a field whose type can vary. The subfields of the tagged subrecord are the possible types. The tag value of the tagged subrecord selects which of those types is used to interpret the field’s value for the record. In memory, the tag is stored with the field. On import or export, it must be broken out as a separate field.

A tagged subrecord field in an imported or exported record must be preceded by an uint32 tag field whose value identifies which of the tagged aggregate’s components is active. The tag field can be one of these:

- A prefix of the tagged aggregate
- A separate field in the record (that is, at the same level as the tagged subrecord field)
Table 103 describes the tagged *subrecord* properties.

Table 103  Tagged Subrecord Properties

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Use</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefix</td>
<td>Prefix of a tagged subrecord containing the value of the tag</td>
<td>25-50</td>
</tr>
<tr>
<td>reference</td>
<td>Holds the name of the field containing the tag value</td>
<td>25-55</td>
</tr>
<tr>
<td>tagcase</td>
<td>Holds the tag value for a tag case field in a subrecord</td>
<td>25-59</td>
</tr>
</tbody>
</table>

Properties: Reference Listing

Table 104 lists all record and field properties in alphabetical order.

Table 104  Record and Field Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Specifies</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual_length</td>
<td>Bytes to skip in imported record if the field contains a null; bytes to fill with null-field or specified pad character, if exported field contains a null.</td>
<td>25-17</td>
</tr>
<tr>
<td>ascii</td>
<td>Character set of data is in ASCII format.</td>
<td>25-18</td>
</tr>
<tr>
<td>big_endian</td>
<td>Multi-byte data types are formatted as big endian.</td>
<td>25-19</td>
</tr>
<tr>
<td>binary</td>
<td>Field value represented in binary format; decimal represented in packed decimal format; julian day format; time represented as number of seconds from midnight; timestamp formatted as two 32-bit integers</td>
<td>25-19</td>
</tr>
<tr>
<td>c_format</td>
<td>Format of text-numeric translation format to and from numeric fields</td>
<td>25-21</td>
</tr>
<tr>
<td>charset</td>
<td>Specifies a character set</td>
<td>25-22</td>
</tr>
<tr>
<td>check_intact</td>
<td>Error checking of imported records with partial record schema (a suboption of intact)</td>
<td>25-23</td>
</tr>
<tr>
<td>date_format</td>
<td>Defines a text-string date format or uformat format other than the default; uformat can contain multi-byte Unicode characters</td>
<td>25-23</td>
</tr>
<tr>
<td>days_since</td>
<td>The imported or exported field stores the date as a signed integer containing the number of days since date_in_ISO_format or uformat.</td>
<td>25-25</td>
</tr>
</tbody>
</table>
Table 104  **Record and Field Properties (continued)**

<table>
<thead>
<tr>
<th>Property</th>
<th>Specifies</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal_separator</td>
<td>Specifies an ASCII character to separate the integer and fraction components of a decimal</td>
<td>25-25</td>
</tr>
<tr>
<td>default</td>
<td>Default value for a field that causes an error</td>
<td>25-26</td>
</tr>
<tr>
<td>default_date_format</td>
<td>Provides support for international date components in date fields</td>
<td>25-26</td>
</tr>
<tr>
<td>default_time_format</td>
<td>Provides support for international time components in time fields</td>
<td>25-27</td>
</tr>
<tr>
<td>delim</td>
<td>Trailing delimiter of all fields</td>
<td>25-27</td>
</tr>
<tr>
<td>delim_string</td>
<td>One or more ASCII characters forming trailing delimiter of all fields</td>
<td>25-28</td>
</tr>
<tr>
<td>drop</td>
<td>Field dropped on import</td>
<td>25-30</td>
</tr>
<tr>
<td>ebc dic</td>
<td>Character set of data is in EBCDIC format</td>
<td>25-31</td>
</tr>
<tr>
<td>export_ebcdic_as_ascii</td>
<td>Exported field converted from EBCDIC to ASCII</td>
<td>25-31</td>
</tr>
<tr>
<td>fill</td>
<td>Byte value to fill in gaps in exported record</td>
<td>25-32</td>
</tr>
<tr>
<td>final_delim</td>
<td>Delimiter character trailing last field of record</td>
<td>25-32</td>
</tr>
<tr>
<td>final_delim_string</td>
<td>Delimiter string trailing last field of record</td>
<td>25-33</td>
</tr>
<tr>
<td>fix_zero</td>
<td>A packed decimal field containing all zeros (normally illegal) is treated as a valid representation of zero</td>
<td>25-34</td>
</tr>
<tr>
<td>generate</td>
<td>Creation of exported field</td>
<td>25-34</td>
</tr>
<tr>
<td>import_ascii_as_ebcdic</td>
<td>Translation of imported string field from ASCII to EBCDIC</td>
<td>25-35</td>
</tr>
<tr>
<td>in_format</td>
<td>Format of text translation to numeric field</td>
<td>25-36</td>
</tr>
<tr>
<td>intact</td>
<td>The record definition defines a partial record schema</td>
<td>25-38</td>
</tr>
<tr>
<td>julian</td>
<td>The imported or exported field represents the date as a numeric value containing Julian day</td>
<td>25-38</td>
</tr>
<tr>
<td>link</td>
<td>A field holds the length of another, variable-length field of the record; field may be a vector</td>
<td>25-38</td>
</tr>
<tr>
<td>little_endian</td>
<td>Multi-byte data types are formatted as little endian</td>
<td>25-39</td>
</tr>
<tr>
<td>max_width</td>
<td>Maximum width of the destination field.</td>
<td>25-40</td>
</tr>
<tr>
<td>Property</td>
<td>Specifies</td>
<td>See Page</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>midnight_seconds</td>
<td>The field represents the time as a binary 32-bit integer containing the number of seconds elapsed from the previous midnight</td>
<td>25-40</td>
</tr>
<tr>
<td>native_endian</td>
<td>Multi-byte data types are formatted as defined by the native format of the machine; this is the default for import/export operations</td>
<td>25-41</td>
</tr>
<tr>
<td>nofix_zero</td>
<td>A packed decimal field containing all zeros generates an error (default)</td>
<td>25-41</td>
</tr>
<tr>
<td>null_field</td>
<td>Value of null field in file to be imported; value to write to exported file if source field is null</td>
<td>25-42</td>
</tr>
<tr>
<td>null_length</td>
<td>Imported length meaning a null value; exported field contains a null</td>
<td>25-43</td>
</tr>
<tr>
<td>overpunch</td>
<td>The field has a leading or ending byte that contains a character which specifies both the numeric value of that byte and whether the number as a whole is negatively or positively signed</td>
<td>25-45</td>
</tr>
<tr>
<td>out_format</td>
<td>Format of numeric translation to text field</td>
<td>25-44</td>
</tr>
<tr>
<td>packed</td>
<td>The imported or exported field contains a packed decimal</td>
<td>25-45</td>
</tr>
<tr>
<td>padchar</td>
<td>Pad character of exported strings or numeric values</td>
<td>25-46</td>
</tr>
<tr>
<td>position</td>
<td>The byte offset of the field in the record</td>
<td>25-47</td>
</tr>
<tr>
<td>precision</td>
<td>The precision of the packed decimal</td>
<td>25-49</td>
</tr>
<tr>
<td>prefix</td>
<td>Prefix of all fields of record; also length of prefix holding the length in bytes of a vector</td>
<td>25-50</td>
</tr>
<tr>
<td>print_field</td>
<td>Each imported field generates message</td>
<td>25-51</td>
</tr>
<tr>
<td>quote</td>
<td>Field is enclosed in quotes or another ASCII character; useful for variable-length fields</td>
<td>25-52</td>
</tr>
<tr>
<td>record_delim</td>
<td>Record delimited by a single ASCII character</td>
<td>25-53</td>
</tr>
<tr>
<td>record_delim_string</td>
<td>Record delimited by one or more ASCII characters</td>
<td>25-53</td>
</tr>
<tr>
<td>record_format</td>
<td>Variable-length blocked records or implicit records</td>
<td>25-54</td>
</tr>
<tr>
<td>record_length</td>
<td>Fixed length records</td>
<td>25-54</td>
</tr>
</tbody>
</table>
Table 104  Record and Field Properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Specifies</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>record_prefix</td>
<td>Records with a length prefix stored as binary data</td>
<td>25-55</td>
</tr>
<tr>
<td>reference</td>
<td>Name of field containing field length</td>
<td>25-55</td>
</tr>
<tr>
<td>round</td>
<td>Rounding mode of source decimal</td>
<td>25-56</td>
</tr>
<tr>
<td>scale</td>
<td>The scale of the decimal</td>
<td>25-57</td>
</tr>
<tr>
<td>separate</td>
<td>The imported or exported field contains an unpacked decimal with a separate sign byte</td>
<td>25-58</td>
</tr>
<tr>
<td>skip</td>
<td>Number of bytes skipped from the end of the previous field to the beginning of this one</td>
<td>25-59</td>
</tr>
<tr>
<td>tagcase</td>
<td>Defines the active field in a tagged subrecord</td>
<td>25-59</td>
</tr>
<tr>
<td>text</td>
<td>Field represented as text-based data; decimal represented in string format; date, time, and timestamp representation is text-based</td>
<td>25-60</td>
</tr>
<tr>
<td>time_format</td>
<td>The format of an imported or exported field representing a time as a string</td>
<td>25-61</td>
</tr>
<tr>
<td>timestamp_format</td>
<td>The format of an imported or exported field representing a timestamp as a string</td>
<td>25-62</td>
</tr>
<tr>
<td>vector_prefix</td>
<td>Prefix length for element count</td>
<td>25-63</td>
</tr>
<tr>
<td>width</td>
<td>The exact width of the destination field</td>
<td>25-64</td>
</tr>
<tr>
<td>zoned</td>
<td>The imported or exported field contains an unpacked decimal represented by either ASCII or EBCDIC text</td>
<td>25-65</td>
</tr>
</tbody>
</table>

The remainder of this chapter consists of an alphabetic listing of all properties. In presenting the syntax, field_definition is the name and data type of the field whose properties are defined.

**actual_length**

Used with null_length (25-43).

- On import, specifies the actual number of bytes to skip if the field’s length equals the null_length
- On export, specifies the number of bytes to fill with the null-field or specified pad character if the exported field contains a null.
Applies To
Nullable fields of all data types; cannot apply to record, subrec, or tagged.

Syntax
field_definition { actual_length = length }; 
where length is the number of bytes skipped by the import operator or filled with zeros by the export operator when a field has been identified as null.

See Also
“null_length” on page 25-43.

Example
In this example:
• On import, the import operator skips the next ten bytes of the imported data if the imported field has a length prefix of 255
• On export, the length prefix is set to 255 if source field a contains a null and the export operator fills the next ten bytes with one of these: zeros, the null-field specification, the pad-character specification for the field.

record { prefix = 2 } ( 
  a: nullable string { null_length = 255, actual_length = 10 }; )

ascii
Specifies that the ASCII character set is used by text-format fields of imported or exported data. This property is equivalent to specifying the US_ASCII character set and is the default.

Applies To
All data types except raw, ustring; record, subrec, or tagged containing at least one non-raw field. For ustring, the same functionality is available using the charset property.

Syntax
record { ascii }
field_definition { ascii }; 
This property is mutually exclusive with ebc dic.

See Also
Example
The following specification overrides the record-level property setting for field b:

```plaintext
record {ebcdic} {
    a:int32;
    b:string {ascii};
}
```

**big_endian**

Specifies the byte ordering of multi-byte data types in imported or exported data as big-endian.

**Applies To**

Fields of the integer, date, time, or timestamp data type; record, subrec, or tagged tagged if it contains at least one field of this type.

**Syntax**

```plaintext
record {big_endian}

field_definition {big_endian};
```

**Restrictions**

Orchestrate ignores the endian property of a data type that is not formatted as binary.

This property is mutually exclusive with little_endian and native_endian.

**Examples**

The following specification defines a record schema using a big-endian representation:

```plaintext
record {big_endian, binary, delim = none} {
    a:int32;
    b:int16;
    c:int8;
}
```

The following specification overrides the record-level property in a definition of field b.

```plaintext
record {big_endian, binary, delim = none} {
    a:int32;
    b:int16 {little_endian};
    c:int8;
}
```

**binary**

Specifies the data representation format of a field in imported or exported data as binary.
Applies To
Fields of all data types except string, ustring, and raw; record, subrec or tagged containing at least one field that is neither string nor raw.

Syntax
```plaintext
record { binary }
field_definition { binary };
```
This option specifies binary data; data is formatted as text by default (see 25-60).

Meanings
The `binary` property has different meanings when applied to different data types:
- For decimals, `binary` means packed (see “packed” on page 25-45).
- For other numerical data types, `binary` means “not text”.
- For dates, `binary` is equivalent to specifying the `julian` property for the `date` field (see “julian” on page 25-38).
- For time, `binary` is equivalent to `midnight_seconds` (see 25-40).
- For timestamp, `binary` specifies that the first integer contains a Julian day count for the date portion of the timestamp and the second integer specifies the time portion of the timestamp as the number of seconds from midnight; on export, `binary` specifies to export a timestamp to two 32-bit integers.

Restrictions
If you specify `binary` as a property of a numeric field, the data type of an imported or exported field must be the same as the corresponding field defined in a record schema. No type conversions are performed among the different numeric data types (as would be the case if `text` was specified instead).

This property is mutually exclusive with `text`, `c_format`, `in_format`, and `out_format`.

Examples
For example, the following defines a schema using binary representation for the imported or exported numeric fields with no delimiter between fields:
```plaintext
record { binary, delim = none } ( a:int32; b:int16; c:int8; )
```
The following statement overrides the record-level setting of field `b` as `text`:
```plaintext
record {text} ( a:int32; b:int16; c:int8; )
```
c_format

Perform non-default conversion either of imported data from string to integer or floating-point data or of exported data from integer or floating-point data to a string.

This property specifies a C-language format string used for both import and export. To specify separate import/ export strings, see “in_format” on page 25-36 and “out_format” on page 25-44.

Applies To

Imported or exported data of the integer or floating-point type; record, subrec, or tagged if it contains at least one field of this type.

Syntax

    field_definition { c_format = 'sscanf_or_sprintf_string' };

sscanf_or_sprintf_string is a control string in single quotation marks containing formatting specification for both scanf() to convert string data to floating-point or integer data and sprintf() to convert floating-point or integer data to strings. See the appropriate C language documentation for details of this formatting string.

Discussion

A UNIX data file can represent numeric values as strings. By default, on import Orchestrate invokes the C strtol(), strtoll(), strtoul(), strtoull(), or strtod() function to convert a string to a numeric field.

An Orchestrate data file can represent numeric data in integer and floating-point data types. By default, on export Orchestrate invokes the C sprintf() function to convert a numeric field formatted as either integer or floating point data to a string.

If these functions do not output data in a satisfactory format, you can use the c_format property to specify a format that is used both by the scanf() function to convert string data to integer or floating point data and by the sprintf() function to convert integral or floating point data to string data.

The associated imported/ exported field must represent numeric data as text.

Example

The meaning of the expression

    record ( a:int32 { c_format = '%x', width = 8 } );
varies:

- On import, the expression uses the `c_format` property to ensure that string data in the imported file is formatted in the Orchestrate record as field `a`, a 32-bit integer formatted as an 8-byte hexadecimal string.

- On export, the same expression ensures that the field `a`, consisting of a 32-bit integer, is exported from the Orchestrate record as an 8-byte hexadecimal string.

**charset**

Specifies a character set defined by the International Components for Unicode (ICU). Refer to Chapter 7: National Language Support in the Orchestrate 7.0 User Guide for information on character sets.

**Applies To**

At the field level, this option applies only to `ustrings`. At the record level, this option applies to the fields that do not specify a character set and to these properties that support multi-byte Unicode character data:

- `delim`
- `delim_string`
- `record_delim`
- `record_delim_string`
- `final_delim`
- `final_delim_string`
- `quote`
- `default`
- `padchar`
- `null_field`
- `date_format`
- `time_format`
- `timestamp_format`

**Syntax**

```
record { charset = charset }
field_definition { charset = charset };
```

**Example**

```
record { charset=charset1 }
  {a: ustring { charset=ISO-8859-15 } {delim = xxx},
   b: date { charset=ISO-8859-15 },
   c: ustring}
```

Where the user defined record `charset`, `charset1`, applies to field `c` and to the `delim` specification for field `a`. `ISO-8859-15` applies to field `a` and `b`. Notice that the field character setting for field `a` does not apply to its `delim` property.
**check_intact**

Used only with the `intact` property, performs error checking when records are imported with a partial record schema (see the section “Partial Schemas and the intact Record Property” in Chapter 5 of the Orchestrate 7.0 User Guide).

**Applies To**

`record`, if it is qualified by the `intact` record property. (See “intact” on page 25-37).

**Syntax**

```plaintext
record { intact, check_intact }
```

By default, when Orchestrate imports records with a partial schema, it does not perform error checking in order to maximize the import speed. This property overrides the default. Error-checking in this case verifies, during the import operation, that the record contents conform to the field description. In addition, downstream of this verification, the operator that acts on the input fields verifies their value.

**Example**

For example, the following statement uses `intact` to define a partial record schema and uses `check_intact` to direct Orchestrate to perform error checking:

```plaintext
record { intact=rName, check_intact, record_length=82,
         record_delim_string='\r\n' } ()
```

**date_format**

Specifies the format of an imported or exported text-format date.

**Applies To**

Field of date data type; `record`, `subrec`, or `tagged` if it contains at least one field of this type.

**Syntax**

```plaintext
record { date_format = format_string | uformat }
```

`uformat` is described in “default_date_format” on page 25-26.

`format_string` can contain one or a combination of the following elements:

- `%dd`: A two-digit day.
- `%mm`: A two-digit month.
- `%year_cutoffYY`: A two-digit year derived from `yy` and the specified four-digit year cutoff, for example `%1970YY`.  

Orchestrate 7.0 Operators Reference
• \%yy: A two-digit year derived from a year cutoff of 1900.
• \%yyyy: A four-digit year.
• \%ddd: Day of year in three-digit form (range of 1-366).
• \%mmm: Three-character month abbreviation.

The format_string is subject to the following restrictions:
• It cannot have more than one element of the same type, for example it cannot contain two \%dd elements.
• It cannot have both \%dd and \%ddd.
• It cannot have both \%yy and \%yyyy.
• It cannot have both \%mm and \%ddd.
• It cannot have both \%mmm and \%ddd.
• It cannot have both \%mm and \%mmm.
• If it has \%dd, it must have \%mm or \%mmm.
• It must have exactly one of \%yy or \%yyyy.

When you specify a date format string, prefix each component with the percent symbol (%). Separate the string’s components with any character except the percent sign (%).

The default date_format is \%yyyy-%mm-%dd.

If this format string does not include a day, it is set to the first of the month in the destination field. If the format string does not include the month and day, they default to January 1. Note that the format string must contain a month if it also contains a day; that is, you cannot omit only the month.

The year_cutoff is the year defining the beginning of the century in which all two-digit years fall. By default, the year cutoff is 1900; therefore, a two-digit year of 97 represents 1997.

You can specify any four-digit year as the year cutoff. All two-digit years then specify the next possible year ending in the specified two digits that is the same or greater than the cutoff. For example, if you set the year cutoff to 1930, the two-digit year 30 corresponds to 1930, and the two-digit year 29 corresponds to 2029.

On import and export, the year_cutoff is the base year.

For a source string, the date components (date, month, and year) must be zero padded to the character length specified by the format string. For a destination string, Orchestrate zero pad the date components to the specified length.

The format as \%ddd-%yy specifies that the string stores the date as a value from 1 to 366 and the year is derived from the current year cutoff of 1900.
This property is mutually exclusive with `days_since`, `text`, and `julian`.

**See Also**

See the chapter on Orchestrate data types in the Orchestrate 7.0 User Guide for more information on date formats.

**days_since**

The imported or exported field stores the date as a signed integer containing the number of days since `date_in_ISO_format` or `uformat`.

**Applies To**

Fields of the date data type; `record`, `subrec`, or `tagged` if it contains at least one field of this type.

**Syntax**

```plaintext
record { days_since = date_in_ISO_format | uformat }
```

where `date_in_ISO_format` is in the form `%yyyy-%mm-%dd`; and `uformat` is the default date format as described in "default_date_format" on page 25-26. The imported or exported field is always stored as binary data.

This property is mutually exclusive with `date_format`, `julian`, and `text`.

**decimal_separator**

Specifies an ASCII character to separate the integer and fraction components of a decimal.

**Applies To**

`record` and decimal fields.

**Syntax**

```plaintext
record { decimal_separator = ASCII_character }
field_definition { decimal_separator = ASCII_character };
```

**Example**

```plaintext
record {decimal_separator = ','}(a:decimal, b:decimal
    {decimal_separator = '.'})
```

where the decimal separator for field a is ',' and is '.' for field b.
**default**

Sets the default value of an imported or exported field:
- On import if the generate property is set
- On both import and export if an error occurs

**Applies To**

Fields of integer, float, string, ustring, and decimal data types. Cannot apply to `record`, `subrecord`, or `tagged`.

**Syntax**

```plaintext
field_definition { default = field_value }
```

where `field_value` is the default value of an imported or exported field. For ustring fields, `field_value` may contain multi-byte Unicode characters.

On import, if you specify a default value, the import operator sets the destination field in the data set to the default value instead of rejecting an erroneous record. On export, the export operator sets the field in the destination data file to the default value instead of rejecting an erroneous record.

The property is mutually exclusive with `link`.

**Example**

The following defines the default value of field `c` as 127:

```plaintext
record (a:int32; b:int16; c:int8 {default = 127};)
```

**default_date_format**

This property provides support for international date components in date fields.

**Applies To**

`record` and date fields.

**Syntax**

```plaintext
record {default_date_format = String%macroString%macroString%macroString}
```

where `%macro` is a formatting macro such as `%mmm` for a 3-character English month. The section “date_format” on page 25-23 lists the date formatting macros. The String components can be strings or ustrings.

**Example**

```plaintext
record { { default_date_format = jeudi%ddaoût%yyyy };
```
**default_time_format**

This property provides support for international time components in time fields.

**Applies To**

*record* and *time fields*.

**Syntax**

```plaintext
record {default_time_format = String%macroString%macroString%macroString}
```

where `%macro` is a formatting macro such as `%hh` for a two-digit hour. The section “time_format” on page 25-61 lists the time formatting macros. The `String` components can be `strings` or `ustrings`.

**Example**

```plaintext
record ( { default_time_format = %hh&nnA.M.};)
```

**delim**

Specifies the trailing delimiter of all fields in the record.

**Applies To**

*record*; any *field* of the record.

**Syntax**

```plaintext
record { delim = delim_char }

field_definition { delim = delim_char };
```

where `delim_char` can be one of the following:

- `ws` to have the import operator skip all standard whitespace characters (space, tab, and newline) trailing after a field.

- `end` to specify that the last field in the record is composed of all remaining bytes until the end of the record.

- `none` to specify that fields have no delimiter.

- `null` to specify that the delimiter is the null character.

- You can specify an ASCII or multi-byte Unicode character enclosed in single quotation marks. To specify multiple characters, use `delim_string` (see “delim_string” on page 25-28).

**Discussion**

By default, the import operator assumes that all fields but the last are whitespace delimited.
By default, the export operator inserts a space after every exported field except the last. On export, you can use `delim` to specify the trailing delimiter that the export operator writes after each field in the destination data file. Specify:

- `ws` to have the export operator insert an ASCII space (0x20) after each field
- `none` to write a field with no delimiter and no length prefix.

You can use a backslash (\) as an escape character to specify special characters. For example, \t represents an ASCII tab delimiter character.

You can use `final_delim` (25-32) to specify a different delimiter for the last field of the record.

Mutually exclusive with `prefix`, `delim_string`, and `reference`.

**Examples**

The following statement specifies that all fields have no delimiter:

```plaintext
record {delim = none} {
  a:int32;
  b:string;
  c:int8;
  d:raw;
}
```

The following statement specifies a comma as a delimiter at record-level but overrides this setting for field `d`, which is composed entirely of bytes until the end of the record:

```plaintext
record {delim = ','} {
  a:int32;
  b:string;
  c:int8;
  d:raw{delim = end};
}
```

Note that in this example, the record uses the default record delimiter of a newline character. This means that:

- On import, field `d` contains all the bytes to the newline character.
- On export, a newline character is inserted at the end of each record.

**delim_string**

Like `delim`, but specifies a string of one or more ASCII or multi-byte Unicode characters forming a trailing delimiter.

**Applies To**

`record`; any field of the record.

**Syntax**

```plaintext
record {delim_string = 'ASCII_string' | 'multi_byte_Unicode_string'}
```
field_definition { delim_string = 'ASCII_string' | 'multi_byte_Unicode_string' };

You can use a backslash (\) as an escape character to specify special characters within a string. For example, \t represents an ASCII tab delimiter character. Enclose the string in single quotation marks.

This property is mutually exclusive with prefix, delim, and reference.

Note

Even if you have specified the character set as EBCDIC, ASCII_string is always interpreted as ASCII character(s).

Import and Export Behavior

Import behavior and export behavior differ:

• On import, the import operator skips exactly one instance of a trailing ASCII_string.

• On export, you use delim_string to specify the trailing delimiter that the export operator writes after each field in the destination data file. If you specify none, a field is written with no delimiter.

Examples

The following statement specifies that all fields are delimited by a comma followed by a space:

record { delim_string = ', ' } (a:int32; b:string; c:int8; d:raw; )

In the following example, the delimiter setting of one comma is overridden for field b, which will be delimited by a comma followed by an ASCII space character, and for field d, which will be delimited by the end of the record:

record { delim = ', ' } (a:int32; b:string {delim_string = ', '}; c:int8; d:raw {delim = end}; )

Note that in this example, the record uses the default record delimiter of a newline character. This means that:

• On import field d contains all the bytes to the newline character.

• On export a newline character is inserted at the end of each record.
drop

Specifies a field to be dropped on import and not stored in the Orchestrate data set.

Applies To
Fields of all data types. Cannot apply to record, subrec, or tagged.

Syntax

    field_definition { drop };

You can use this property when you must fully define the layout of your imported data in order to specify field locations and boundaries but do not want to use all the data in the Orchestrate application.

Restrictions
This property is valid for import only and is ignored on export.
This property is mutually exclusive with link.

See Also
“padchar” on page 25-46.

Example
In this example, the variable-length string field, b, is skipped as the record is imported.

    record (a:int32; b:string {drop}; c:int16;)

In the following example, all fields are written as strings to the Orchestrate data set on import and are delimited by a comma except the last field; records are delimited by the default newline character. The last four fields are dropped from every record on import, and generated on export. This technique is useful when fields of the source data are not be processed in an Orchestrate application but place holders for those fields must be retained in the resultant file containing the exported results. In addition, all bytes of the generated fields in the exported data file will be filled with ASCII spaces, as defined by the padchar property:

    record { delim =',' }{
        first_name : string[20];
        middle_init : string[1];
        last_name : string[40];
        street : string[40];
        apt_num : string[4];
        city : string[30];
        state : string[2];
        prev_street : string[40] {drop, generate, padchar=' '};
    }
prev_apt_num : string[4] { drop, generate, padchar=' ' };
prev_city : string[30] { drop, generate, padchar=' ' };
prev_state : string[2] { delim = end, drop, generate, padchar=' ' };

**ebcdic**

Specifies that the EBCDIC character set is used by text-format fields of imported or exported data. Setting this property is equivalent to specifying the EBCDIC equivalent of the US_ASCII character set.

** Applies To **

All data types except raw and ustring; record, subrec, or tagged if it contains at least one field of this type. For ustring, the same functionality is available through the charset property.

** Syntax **

```plaintext
record { ebcdic }

field_definition { ebcdic };
```

This property is mutually exclusive with ascii.

Orchestrate's default character set is ASCII-formatted. (Orchestrate supplies lookup tables for converting between ASCII and EBCDIC. See “ASCII and EBCDIC Conversion Tables” on page 22-15 for more information.)

** See Also **

“ascii” on page 25-18.

**export_ebcdic_as_ascii**

Export a string formatted in the EBCDIC character set as an ASCII string.

** Applies To **

Fields of the string data type and records; record, subrec, or tagged if it contains at least one field of this type. This property does not apply to ustring; you can obtain the same functionality using the charset property.

** Syntax **

```plaintext
record { export_ebcdic_as_ascii }

field_definition { export_ebcdic_as_ascii };
```

This property cannot be used on import.
fill

Specifies the byte value used to fill the gaps between fields of an exported record caused by field positioning.

Applies To
record; cannot be specified for individual fields.

Syntax
record {fill [ = fill_value ] }

where fill_value specifies a byte value that fills in gaps. By default the fill value is 0. The fill_value can also be one of these:
• a character or string in single quotation marks
• an integer between 0 and 255.

Restriction
You cannot override the record-level fill property with an individual field definition.

Examples
In the following example, the two-byte gap between fields a and b are to be filled with ASCII spaces:
record {fill = ' '} {a:int32; b:int16 {skip = 2};}

In the following example, the gaps are to be filled with the value 127:
record { fill = 127} {a:int32; b:int16 {skip = 2};}

final_delim

Specifies a delimiter for the last field. If specified, final_delim precedes the record_delim if both are defined.

Applies To
The last field of a record.

Syntax
record {final_delim = delim_value}

where delim_value is one of the following:
• ws (a white space)
• end (end of record, the default)
• **none** (no delimiter, field length is used)
• **null** (0x00 character)
• 'a' (specified ASCII or multi-byte Unicode character, enclosed in single quotation marks)

### Example

In this example, commas delimit all fields except the last. Since `end` is specified as the **final_delim**, the record delimiter serves as the delimiter of the final field. This is the newline character. Note that it is not specified as the **record_delim**, because newline-delimited records are the default.

```plaintext
record {delim = ',', final_delim = end}
 {
  checkingAccountStatus : int32;
  durationOfAccount     : sfloat;
  creditHistory         : int32;
  purposeOfLoan         : int32;
  creditAmount          : sfloat;
  savingsAccountAmount  : int32;
  yearsPresentlyEmployed: int32;
  installmentRate       : sfloat;
 }
```

#### final_delim_string

Like **final_delim**, but specifies a string of one or more ASCII or multi-byte Unicode characters that are the delimiter string of the last field of the record. The **final_delim_string** property precedes the **record_delim** property, if both are used.

#### Applies To

The last field of a record.

#### Syntax

```plaintext
record {final_delim_string = 'ASCII_string' | 'multi_byte_Unicode_string'}
```

where string characters are enclosed in single quotation marks.

### Import and Export

Behavior on import and export varies:

• On import, the import operator skips the delimiter in the source data file. The import operator skips exactly one instance of a trailing ASCII_string during import.
• On export, `final_delim_string` specifies the trailing delimiter that the export operator writes after the last field of each record.

Even if you have specified the character set as EBCDIC, `'ASCII_string'` is always interpreted as ASCII characters.

Example

For example, the following statement specifies that all fields are delimited by a comma, except the final field, which is delimited by a comma followed by an ASCII space character:

```orcl
    record {delim_string = ',', final_delim_string = ', '} (  
        a:int32;  
        b:string;  
        c:int8;  
        d:raw;  
    )
```

`fix_zero`

Treat a packed decimal field containing all zeros (normally illegal) as a valid representation of zero.

Applies To

Fields of the packed decimal data type on import; all decimal fields on export (exported decimals are always packed); `record`, `subrec`, or `tagged` if it contains at least one field of these types.

Syntax

```orcl
    field_definition { fix_zero };
```

Caution

Omitting `fix_zero` causes the application to generate an error if it encounters a decimal containing all zeros.

This property overrides the `nocheck` option to `packed`.

See Also

“`packed`” on page 25-45 and “`nofix_zero`” on page 25-41. The latter option restores default behavior.

generate

On export, creates a field and sets it to the default value.
Applies To
Fields of all data types. Cannot apply to record, subrec, or tagged.

Syntax
field_definition { generate, default = default_value };

where:
• default is the ordinary default property (see 25-26) and not a sub-option of generate
• default_value is the optional default value of the field that is generated; if you do not specify a default_value, Orchestrate writes the default value associated with that data type.

Discussion
You can specify both the drop property and the generate property in the same property list. When the schema is used for import it drops the field, and when it is used for export, it generates the field.

This type of statement is useful if the field is not important when the data set is processed but you must maintain a place holder for the dropped field in the exported data.

Restriction
This property is valid for export only and is ignored on import.

Examples
The following statement creates a new field in an exported data set:
record ( a:int32; b:int16 {generate, default=0}; c:int8; )

The following statement causes field b to be:
• dropped on import
• generated as a 16-bit integer field, which is initialized to 0, on export.
record ( a:int32; b:int16 {drop, generate, default =0}; c:int8; )

import_ascii_as_ebcdic
Translate a string field from ASCII to EBCDIC as part of an import operation. This property makes it possible to import EBCDIC data into Orchestrate from ASCII text.
**Applies To**

Fields of the string data type; record, subrec, or tagged if it contains at least one field of this type. This property does not apply to ustring; you can obtain the same functionality using the charset property.

**Syntax**

```plaintext
field_definition { import_ascii_as_ebcdic };
```

**Restriction**

Use only in import operations. On export, you can use `export_ebcdic_as_ascii` to export the EBCDIC field back to ASCII. See “export_ebcdic_as_ascii” on page 25-31.

**Example**

In the following example, the string field `x` contains an ASCII string that is converted to EBCDIC on import:

```plaintext
record { x:string[20] { import_ascii_as_ebcdic };
```

**in_format**

On import, perform non-default conversion of an input string to an integer or floating-point data.

This property specifies a C-language format string used for import. See also “out_format” on page 25-44.

**Applies To**

Imported data of integer or floating-point type; record, subrec, or tagged if it contains at least one field of this type.

**Syntax**

```plaintext
field_definition { in_format = 'sscanf_string' };
```

where `sscanf_string` is a control string in single quotation marks interpreted by the C `scanf()` function to convert character data to numeric data. See the appropriate C language documentation for details of this formatting string.

**Discussion**

A UNIX data file can represent numeric values as strings. By default, on import Orchestrate invokes the C `strtol()`, `strtoll()`, `strtoul()`, `strtoull()`, or `strtol()` function to convert a string to a numeric field. If these functions do not output data in a satisfactory format, you can specify the `out_format` property.
When you do, Orchestrate invokes the `sprintf()` function to convert the string to numeric data. You pass formatting arguments to the function.

When strings are converted to 8-, 16-, or 32-bit signed integers, the `sscanf_string` must specify options as if the generated field were a 32-bit signed integer. Orchestrate converts the 32-bit integer to the proper destination format.

**Restrictions**

The export operator ignores this property; use it only for import operations.

This property is mutually exclusive with `binary`.

**Example**

The following statement assumes that the data to be converted is represented as a string in the input flat file. The data is imported as field `a`, a 32-bit integer formatted as an 8-byte hexadecimal string:

```plaintext
record (a:int32 {in_format = '%x', width=8};)
```

**intact**

Define a partial record schema (see the section “Partial Schemas and the intact Record Property ” in Chapter 5 of the Orchestrate 7.0 User Guide).

**Applies To**

`record`.

**Syntax**

```plaintext
record {in tact [=rName] ... } ( field_definitions;)
```

where `rName` is an optional identifier of the intact schema.

**See also**

“check_intact” on page 25-23.

**Example**

For example, the following statement uses `intact` to define a partial record schema:

```plaintext
record {in tact =rName, record_length=82, record_delim_string='
'} ()
```

In this example, the record schema defines an 82-byte record (80 bytes of data, plus one byte each for the carriage return and line feed characters delimiting the record). On import, the two bytes for the carriage return and line feed characters are removed from the record and not saved; therefore, each record of the data set contains only the 80 bytes of data.
julian

Specifies that the imported or exported field represents the date as a binary numeric value denoting the Julian day.

Applies To

Fields of the date data type, record, subrec, or tagged if it contains at least one field of this type.

Syntax

record { binary, julian }
field_definition { binary, julian };

Note

The imported or exported value is always represented as binary data.

A Julian day specifies the date as the number of days from 4713 BCE January 1, 12:00 hours (noon) GMT. For example, January 1, 1998 is Julian day count 2,450,815. In this context, binary has the same meaning as julian.

This property is mutually exclusive with days_since, date_format, and text.

link

Specifies that a field holds either the length of another (variable-length) field in the record or the length of the tag value of a tagged.

Applies To

Numeric or string fields; cannot apply to a record, subrec, or tagged.

Syntax

field_definition { link };

Discussion

A variable-length field must specify the number of elements it contains by means either of a prefix of the vector (see “vector_prefix” on page 25-63) or a link to another field.

- On import, the link field is dropped and does not appear in the schema of the imported data set.
- On export, the link field is not exported from the data set, but is generated from the length of the field referencing it and then exported.

The data type of a field defined by the link import/export property must:
• Be a uint32 or data type that can be converted to uint32; see Chapter 13, “The modify Operator” for information on data type conversions.

• Have a fixed-length external format on export.

• Not be a vector.

This property is mutually exclusive with drop.

Examples

In this example, field a contains the element length of field c, a variable-length vector field.

record (a:uint32 {link}; b:int16; c[]):int16 {reference = a};

In this example, field a contains the length of field c, a variable-length string field.

record (a:uint32 {link}; b:int16; c:string {reference = a};)

little_endian

Specify the byte ordering of multi-byte data types as little-endian. The default mode is native-endian.

Applies To

Fields of the integer, date, time, or timestamp data type; record, subrec, or tagged if it contains at least one field of this type.

Syntax

record { little_endian }
field_definition { little_endian };

This property is mutually exclusive with big_endian and native_endian.

Examples

The following specification defines a schema using a little-endian representation:

record {little_endian, binary, delim = none} {
  a:int32;
  b:int16;
  c:int8;
}

The following declaration overrides the big_endian record-level property in a definition of field b:

record {big_endian, binary, delim = none} {
  a:int32;
  b:int16 {little_endian};
  c:int8; }
max_width

Specifies the maximum number of 8-bit bytes of an imported or exported text-format field. Base your width specification on the value of your -impexp_charset option setting. If it's a fixed-width charset, you can calculate the maximum number of bytes exactly. If it's a variable length encoding, calculate an adequate maximum width for your fields.

Applies To

Fields of all data types except date, time, timestamp, and raw; record, subrec, or tagged if it contains at least one field of this type.

Syntax

```plaintext
record { max_width = n }
field_definition { max_width = n };
```

where \( n \) is the maximum number of bytes in the field; you can specify a maximum width of 255 bytes.

This property is useful for a numeric field stored in the source or destination file in a text representation.

If you specify neither width nor max_width, numeric fields exported as text have the following number of bytes as their maximum width:

- 8-bit signed or unsigned integers: 4 bytes
- 16-bit signed or unsigned integers: 6 bytes
- 32-bit signed or unsigned integers: 11 bytes
- 64-bit signed or unsigned integers: 21 bytes
- single-precision float: 14 bytes (sign, digit, decimal point, 7 fraction, "E", sign, 2 exponent)
- double-precision float: 24 bytes (sign, digit, decimal point, 16 fraction, "E", sign, 3 exponent)

Restriction

On export, if you specify the max_width property with a dfloat field, the max_width must be at least eight characters long.

midnight_seconds

Represent the imported or exported time field as a binary 32-bit integer containing the number of seconds elapsed from the previous midnight.
Applies To
The time data type; record, subrec, or tagged if it contains at least one field of this type.

Syntax
```
record { binary, midnight_seconds };
field_definition { binary, midnight_seconds };
```

Note
The imported or exported integer is always stored as binary data. This property is mutually exclusive with time_format and text.

native_endian
Specify the byte ordering of multi-byte data types as native-endian. This is the default mode for import/export operations.

Applies To
Fields of the integer, date, time, or timestamp data type; record, subrec, or tagged if it contains at least one field of this type.

Syntax
```
record { native_endian };
field_definition { native_endian };
```

where native_endian specifies that all multi-byte data types are formatted as defined by the native format of the machine.

This property is mutually exclusive with big_endian and little_endian.

Note
Orchestrate ignores the endian property of a data type that is not formatted as binary.

Examples
The following defines a schema using a native-endian representation:
```
record {native_endian, binary, delim = none} {
    a:int32;
    b:int16;
    c:int8;
}
```

nofix_zero
Generate an error when a packed decimal field containing all zeros is encountered. This is the default behavior of import and export.
Applies To
Import behavior and export behavior differ:
• Fields of the packed decimal data type on import
• All decimal fields on export (exported decimals are always packed); record, subrec, or tagged if it contains at least one field of this type.

Syntax
    field_definition { nofix_zero };

null_field
Specifies the value representing null.

Applies To
Fields whose data type is nullable; cannot apply to record, subrec, or tagged.

Syntax
    field_definition { null_field = 'byte_value' | 'multi_byte_Unicode_value' }; 
where byte_value or multi_byte_Unicode_value is:
• On import, the value given to a field containing a null;
• On export, the value given to an exported field if the source field is set to null.
The byte_value or multi_byte_Unicode_value can take one of these forms:
• A number or string that defines the value to be written if the field contains a null. Enclose the string or ustring in single quotation marks.
• A standard C-style string literal escape character. For example, you can represent a byte value by \ooo, where each o is an octal digit 0 - 7 and the first o is < 4, or by \xhh, where each h is a hexadecimal digit 0 - F. You must use this form to encode non-printable byte values.

This property is mutually exclusive with null_length and actual_length.

The null_field parameter can be used for imported or exported data. For a fixed-width data representation, you can use padchar to specify a repeated trailing character if byte_value is shorter than the fixed width of the field.

Restriction
Specifying null_field with a non-nullable Orchestrate field generates an error.

See Also
“padchar” on page 25-46.
Example

For example, the following schema defines two nullable fields:

```plaintext
record (
  a: nullable int8 {null_field = '-127'};
  b: nullable string[4] {null_field = '0000'};
)
```

The import operator interprets a source-field value of -127 in field `a` as null and a value of all zeros in field `b` as null and sets the corresponding destination fields to null. The export operator sets the destination field to -127 for `a` and 0000 for `b` if the corresponding source data field is null.

null_length

Specifies the value of the length prefix of a variable-length field that contains a null.

Applies To

Fields whose data type is nullable; cannot apply to `record`, `subrec`, or `tagged`.

Syntax

```plaintext
field_definition { null_length= length }
```

where `length` is the length in bytes of a variable-length field that contains a null.

When a variable-length field is imported, a length of `null_length` in the source field indicates that it contains a null. When a variable-length field is exported, the export operator writes a length value of `null_length` if it contains a null.

This property is mutually exclusive with `null_field`.

Restriction

Specifying `null_length` with a non-nullable Orchestrate field generates an error.

See Also

“actual_length” on page 25-17

Example

For example, the following schema defines a nullable, variable-length string field, prefixed by a two-byte length:

```plaintext
record {prefix = 2} (
  a: nullable string {null_length = 255};
)
```

Import and export results differ:

- On import, the imported field is assumed to be zero length and to contain a null if the length prefix contains 255; field `a` of the imported data is set to null.
On export, the length prefix is set to 255 and the length of the actual destination is zero if field a contains null.

**out_format**

On export, perform non-default conversion of an integer or floating-point data type to a string.

This property specifies a C-language format string used for export of a text field. See also “in_format” on page 25-36.

**Applies To**

Integer or floating-point data to be exported to numeric strings.

**Syntax**

```
field_definition { out_format = 'sprintf_string'; }
```

where `sprintf_string` is a control string in single quotation marks containing formatting specification for `sprintf()` to convert floating-point or integer data to strings. See the appropriate C language documentation for details of this formatting string.

When 8-, 16-, or 32-bit signed integers are converted to strings, the `sprintf_string` must specify options as if the source field were a 32-bit integer. Orchestrate converts the source field to a 32-bit signed integer before passing the value to `sprintf()`. For 8-, 16-, and 32-bit unsigned integers, specify options as if the source field were a 32-bit unsigned integer.

**Discussion**

An Orchestrate data file can represent numeric data in integer and floating-point data types. By default, on export Orchestrate invokes the `C sprintf()` function to convert a numeric field formatted as either integer or floating point data to a string. If this function does not output data in a satisfactory format, you can specify the `out_format` property. When you do, Orchestrate invokes the `sprintf()` function to convert the numeric data to a string. You pass formatting arguments to the function.

**Restrictions**

The import operator ignores this property; use it only for export operations.

The property is mutually exclusive with `binary`.

**Example**

The following statement defines an exported record containing two integer values written to the exported data file as a string:
record {
    a:int32 {out_format = '%x', width = 8};
    b:int16 {out_format = '%x', width = 4};
}

overpunch

Specifies that the imported or exported decimal field has a leading or ending byte that contains a character which specifies both the numeric value of that byte and whether the number as a whole is negatively or positively signed. This representation eliminated the need to add a minus or plus sign to the beginning or end of a number. All the digits besides the overpunched byte represent normal numbers.

Use one of these formats:

- To indicate that the overpunched value is in the leading byte, use this syntax:
  `{overpunch}

  For example, in the overpunched number 8567, 8 indicates that the leading byte has a value of 2 and that the decimal is positively signed. It is imported as 2567.

- To indicate that the overpunched value is in the last byte, use this syntax:
  `{trailing, overpunch}

  For example, in the overpunched number 567K, K indicates that the last byte has a value of 2 and that the number is negatively signed. It is imported as -5672.

Applies To

Decimal fields.

Syntax

record { {overpunch | trailing, overpunch} option_1 .... option_n }
field_definition { {overpunch | trailing, overpunch} option_1,...option_n};

packed

Specifies that the imported or exported field contains a packed decimal.

Applies To

Fields of the decimal, string, and ustring data types; record, subrec, or tagged if it contains at least one field of this type.
Syntax

```plaintext
record { packed option_1 .... option_n}
field_definition { packed option_1,...option_n};
```

where option_1 ... option_n is a list of one or more options; separate options with a comma if there are more than one. The options are:

- **check** (default) perform the Orchestrate data verification process on import/ export. Note that this property is ignored if you specify **fix_zero**.
- **nocheck** bypass the Orchestrate data verification process on import/ export. Note that this property is ignored if you specify **fix_zero**.

The options **check** and **nocheck** are mutually exclusive.

- **signed** (default) use the sign of the source decimal on import or export.
- **unsigned** generate a sign nibble of 0xf, meaning a positive value, regardless of the imported or exported field’s actual sign. This format corresponds to the COBOL PICTURE 999 format (as opposed to S999).

The options **signed** and **unsigned** are mutually exclusive.

Discussion

The precision and scale of either the source decimal on import or the destination decimal on export defaults to the precision and scale of the Orchestrate decimal field. Use the **precision** and **scale** properties to override these defaults.

For example, the following schema specifies the Orchestrate decimal field has a precision of 5 and a scale of 2:

```plaintext
record {packed} ( a:decimal[5,2]; )
```

Import and export results differ:

- On import, field a is imported from a packed decimal representation three bytes long.
- On export, the field is written to three bytes in the destination.

**padchar**

Specifies a pad character used when Orchestrate strings or numeric values are exported to an external string representation.

**Applies To**

string, ustring, and numeric data types on export; record, subrec, or tagged if it contains at least one field of this type.
Syntax

```plaintext
record { padchar = 'char(s)' }
field_definition { padchar = 'char(s)' }
```

where `char(s)` is one or more pad characters, single byte for string fields and multi-byte Unicode for ustring fields. You can specify `null` to set the pad character to 0x00; the default pad character is 0x20 (ASCII space). Enclose the pad character in single quotation marks if it is not the value `null`.

The pad character is used when the external string representation is larger than required to hold the exported field. In this case, the external string is filled with the pad character to its full length.

Restrictions

This property is ignored on import.

Example

In the following example, the destination string will be padded with spaces to the full length, if the string field `a` is less than 40 bytes long.

```plaintext
record (a:string {width = 40, padchar = ' '});
```

Notes

Orchestrate fixed-length strings also have a `padchar` property, which is part of a schema rather than an import/export property, as in the following example. Here the exported fixed-length string is also padded with spaces to the length of the external field:

```plaintext
record (a:string[20, padchar = ' ' {width = 40});
```

You can globally override the default pad character using the Orchestrate environment variable `APT_STRING_PADCHAR`. The Option and Syntax of this environment variable is shown below:

```plaintext
export APT_STRING_PADCHAR='character' # ksh
setenv APT_STRING_PADCHAR 'character' # csh
```

where `character` is the default pad character enclosed in single quotation marks.

position

Specifies the starting position of a field in the imported source record or exported destination record. The starting position can be either an absolute byte offset from the first record position (0) or the starting position of another field.

Applies To

Fields of all data types; cannot apply to `record`, `subrec`, or `tagged`.
Syntax

```plaintext
field_definition { position = byte_offset | field_name };
```

where:

- `byte_offset` is an integer value that is greater than or equal to 0 and less than the record length and that indicates the absolute byte offset of the field from position 0 in the record.
- `field_name` specifies the name of another field in the record at whose beginning boundary the defined field also starts, as in the example below.

Discussion

Specifies the byte offset of a field in the imported source record or exported destination record, where the offset of the first byte is 0. If you omit this property, a field starts immediately after the preceding field. Note that a field can start at a position preceding the end of the previous field.

Examples

For example, the following defines a schema using this property:

```plaintext
record {binary, delim = none} (a:int32; b:int16 {position = 6};)
```

Import and export results differ:

- On import, field `b` starts at absolute byte offset 6 in the source record, that is, there is a 2-byte gap between fields `a` and `b`. Note that numeric values are represented in a binary format in this example. The default numeric format is text.
- On export, the export operator skips two bytes after field `a`, then writes field `b`. By default on export, any skipped bytes are set to zero in the destination record. You can use the record-level `fill` property (25-32) to specify a value for the skipped bytes.

You can specify a `field_name` as a position, as in the following example:

```plaintext
record {binary, delim = none} (a:string {delim = ws};
b:int16;
c:raw[2] {position = b};)
```

Field `c` starts at field `b`. Import behavior and export behavior differ:

- On import, the schema creates two fields from field `b` of the imported record, interpreting the same external value using two different field types.
- On export, the schema first writes the contents of field `b` to destination field `b` and then to destination field `c`.
**precision**

Import behavior and export behavior differ:

- On import, specifies the precision of a source packed decimal.
- On export, specifies the precision of a destination string when a decimal is exported in text format.

**Applies To**

Imported strings representing packed decimals; exported packed decimals to be written as strings; record, subrec, or tagged if it contains at least one field of this type.

**Syntax**

```plaintext
record { precision=p } field_definition { precision=p };
```

where \( p \) is the precision of the source packed decimal on import and the precision of the destination string on export; \( p \) has no limit.

**Discussion**

When a source decimal is exported to a string representation, the export operator uses the precision and scale defined for the source decimal field to determine the length of the destination string. The precision and scale properties override this default. When they are defined, the export operator truncates or pads the source decimal to fit the size of the destination string. If you include the width property, the export operator truncates or pads the source decimal to fit the size specified by width.

**Restriction**

The precision property is ignored on export if you also specify text.

**Example**

The following example shows a schema used to import a source field with the same precision and scale as the destination decimal:

```plaintext
record ( a:decimal[6,2]; )
```

Orchestrate imports the source field to a decimal representation with a 6-digit precision.

The following example shows a schema that overrides the default to import a source field with a 4-digit precision:

```plaintext
record ( a:decimal[6,2] {precision = 4}; )
```
prefix
Specifies that each imported/exported field in the data file is prefixed by 1, 2, or 4 bytes containing, as a binary value, either the field’s length or the tag value for a tagged subrecord.

Applies To
All fields and record.

Syntax

code
```plaintext
record { prefix = prefix }
field_definition { prefix = prefix };
```

where prefix is the integer 1, 2, or 4, which denotes a 1-, 2-, or 4-byte prefix containing the field length or a character enclosed in single quotes.

Discussion
You can use this option with variable-length fields. Variable-length fields can be either delimited by a character or preceded by a 1-, 2-, or 4-byte prefix containing the field length.

Import behavior and export behavior differ:

- On import, the import operator reads the length prefix but does not include the prefix as a separate field in the imported data set.
- On export, the export operator inserts the prefix before each field. You can use this option with variable-length fields. Variable-length fields can be either delimited by a character or preceded by a 1-, 2-, or 4-byte prefix containing the field length.

This property is mutually exclusive with delim, delim_string, quote, final_delim, and reference.

Example
In the following example, fields a and b are both variable-length string fields preceded by a 2-byte string length:

code
```plaintext
record {prefix = 2} {  
a:string;  
b:string;  
c:int32; }
```

In the following example, the 2-byte prefix is overridden for field b, where the prefix is a single byte:

code
```plaintext
record {prefix = 2} {  
a:string;  
b:string {prefix = 1}; }
```
For tagged subrecords, the tag field may be either a prefix of the tagged aggregate or another field in the record associated with the tagged aggregate by the link property. Shown below is an example in which the tagged aggregate is preceded by a one-byte unsigned integer containing the tag:

```plaintext
record {
  .
  tagField: tagged {prefix=1} (aField:string; bField:int32; cField:sfloat;);
}
```

**print_field**

For debugging purposes only; causes the import operator to display each value imported for the field.

**Applies To**

Fields of all data types and record.

**Syntax**

```plaintext
record { print_field }
field_definition { print_field };
```

**Discussion**

This property causes import to write out a message for either selected imported fields or all imported fields, in the form:

Importing N: D

where:

- N is the field name.
- D is the imported data of the field. Non-printable characters contained in D are prefixed with an escape character and written as C string literals; if the field contains binary data, it is output in octal format.

**Restrictions**

This property is ignored on export.

**Example**

For example, the following schema uses print_field:

```plaintext
record {print_field} (a:string; b:int32;)
```
By default, imported numeric fields represent data as text. In this case, the import operator issues the following message:

Importing a: "the string"
Importing b: "4660"

The following schema specifies that the numeric data is represented in binary form:

```plaintext
record {binary, print_field} (a:string; b:int32;)
```

In this case, the import operator prints the binary data in an octal format as shown below:

Importing a: "a string"
Importing b: "\000\000\022\064"

**quote**

Specifies that a field is enclosed in single quotes, double quotes, or another ASCII or multi-byte Unicode character or pair of ASCII or multi-byte Unicode characters.

**Applies To**

Fields of any data type; record, subrec, or tagged if it contains at least one field of this type.

**Syntax**

```plaintext
record { quote = 'quotechar' | 'quotechars' }
field_definition { quote = 'quotechar' | 'quotechars' ];
```

where `quotechar` is one of the following: **single**, **double**, an ASCII or Unicode character, and `quotechars` is a pair of ASCII or multi-byte Unicode characters, for example, ['']. Enclose `quotechar` or `quotechars` in single quotation marks.

This property is mutually exclusive with `prefix` and `reference`.

**Discussion**

This property is useful for variable-length fields contained either in quotes or in other characters marking the start and end of a field.

- On import, the leading quote character is ignored and all bytes up to but not including the trailing quote character are imported.
- On export, the export operator inserts the leading quote character, the data, and a trailing quote character.

Quote characters are not counted as part of a field’s length.
Example

The following example specifies that the data imported into the variable-length string field b is contained in double quotes:

```plaintext
record {
  a: int32;
  b: string {quote = double};
  c: int8;
  d: raw;
}
```

**record_delim**

Specifies a single ASCII or multi-byte Unicode character to delimit a record.

**Applies To**

- `record`; cannot be a field property.

**Syntax**

```plaintext
record {record_delim \[= 'delim_char'\]}
```

where `delim_char` can be a newline character, a null, or one ASCII or multi-byte Unicode character. If no argument is specified, the default is a newline character.

This property is mutually exclusive with `record_delim_string`, `record_prefix`, and `record_format`.

**record_delim_string**

Specifies an ASCII or multi-byte Unicode string to delimit a record.

**Applies To**

- `record`; cannot be a field property.

**Syntax**

```plaintext
record {record_delim_string = 'ASCII_string' | 'multi_byte_Unicode_string'}
```

where `ASCII_string` or `multi_byte_Unicode_string` is the string that delimits the record.

**Restrictions**

You cannot specify special characters by starting with a backslash escape character. For example, specifying \t, which represents an ASCII tab delimiter character, generates an error.
This property is mutually exclusive with `record_delim`, `record_prefix`, and `record_format`.

**record_format**

Specifies that data consists of variable-length blocked records or implicit records.

**Applies To**

`record`; cannot be a field property.

**Syntax**

```
record { record_format = { type = type, format = format }}
```

where `type` is either `implicit` or `varying`.

If you choose the `implicit` property, data is imported or exported as a stream with no explicit record boundaries. You may not use the property `delim = end` with this format.

On import, the import operator converts the input into records using schema information passed to it. Field boundaries are implied by the record schema passed to the operator. You cannot save rejected records with this record format.

On export, the records are written with no length specifications or delimiters marking record boundaries. The end of the record is inferred when all of the fields defined by the schema have been parsed.

The `varying` property is allows you to specify one of the following IBM blocked or spanned formats: `V`, `VB`, `VS`, `VBS`, or `VR`. Data is imported using that format.

This property is mutually exclusive with `record_length`, `record_delim`, `record_delim_string`, and `record_prefix`.

**record_length**

Import or export fixed-length records.

**Applies To**

`record`; cannot be a field property.

**Syntax**

```
record { record_length = fixed | nbytes }
```

where:

- `fixed` specifies fixed-length records; the record schema must contain only fixed-length elements so that Orchestrate can calculate the record length.
• nbytes explicitly specifies the record length in bytes if the record contains variable-length elements.

On export, the export operator pads the records to the specified length with either zeros or the fill character if one has been specified.

This property is mutually exclusive with record_format.

**record_prefix**

Specifies that a variable-length record is prefixed by a 1-, 2-, or 4-byte length prefix.

**Applies To**

record; cannot apply to fields.

**Syntax**

```
record {record_prefix [ = prefix ]}
```

where prefix is 1, 2, or 4. If you do not specify a value for prefix, the variable defaults to 1.

This property is mutually exclusive with record_delim, record_delim_string, and record_format.

**reference**

Points to a link field containing the length of an imported/ exported field.

**Applies To**

Variable-length vectors of all data types; cannot apply to record, subrec, or tagged.

**Syntax**

```
field_definition { reference = link_field ;}
```

where link_field is the name of a field of the same record that holds the length of the field defined by field_definition.

Variable-length fields can specify the number of elements they contain by means of a link to another field that contains their length or the tag of a tagged subrecord.

This property is mutually exclusive with prefix, delim_string, quote, and delim.
Example
The following statement specifies that the link field a contains the length of the variable-length string field c:

```plaintext
record {delim = none, binary}
  a:int32 {link}; b:int16; c:string {reference = a};
```

round
Round decimals on import or export.

Applies To
Fields of the decimal data type; record, subrec, or tagged if it contains at least one field of this type.

Syntax
```plaintext
record { round = rounding_type }
field_definition { round = rounding_type }
```

where rounding_type can be one of the following:

- **ceil**: Round the source field toward positive infinity. This mode corresponds to the IEEE 754 Round Up mode.
  
  Examples: 1.4 -> 2, -1.6 -> -1

- **floor**: Round the source field toward negative infinity. This mode corresponds to the IEEE 754 Round Down mode.
  
  Examples: 1.6 -> 1, -1.4 -> -2

- **round_inf**: Round the source field toward the nearest representable value, breaking ties by rounding toward positive infinity or negative infinity. This mode corresponds to the COBOL ROUNDED mode.
  
  Examples: 1.4 -> 1, 1.5 -> 2, -1.4 -> -1, -1.5 -> -2

- **trunc_zero** (default): Truncate the source field toward zero. Discard fractional digits to the right of the right-most fractional digit supported in the destination, regardless of sign. For example, if the destination is an integer, all fractional digits are truncated. If the destination is another decimal with a smaller scale, truncate to the scale size of the destination decimal. This mode corresponds to the COBOL INTEGER-PART function.
  
  Examples: 1.6 -> 1, -1.6 -> -1

Import and Export Behavior
Import behavior and export behavior differ:
• On import, this property specifies how Orchestrate rounds the source field to fit into the destination decimal when the source field is imported to a decimal.

• On export, this property specifies how to round a source decimal when its precision and scale greater than those of the destination.

**scale**

Specifies the scale of a packed decimal.

**Applies To**

Imported strings representing packed decimals; exported packed decimals to be written as strings; record, subrec, or tagged if it contains at least one field of this type.

**Syntax**

```plaintext
record { scale = s }
field_definition { scale = s ; }
```

where `s` is the scale.

**Discussion**

By default, the import operator uses the scale defined for the Orchestrate decimal field to import the source field. You can change this. On import, the `scale` property specifies the scale of the source packed decimal.

By default, when the export operator exports a source decimal to a string representation, it uses the precision and scale defined for the source decimal field to determine the length of the destination string. You can override the default by means of the `precision` and `scale` properties. When you do, the export operator truncates or pads the source decimal to fit the size of the destination string. If you include the `width` property, the export operator truncates or pads the source decimal to fit the size specified by `width` (25-64).

**Restrictions**

The `scale` property is ignored on export if you also specify `text`.

The value of `scale` must be less than the precision and greater than 0. The precision is specified by the `precision` property. See “precision” on page 25-49.

**Example**

The following example is a schema used to import a source field with the same precision and scale as the destination decimal:

```plaintext
record ( a:decimal[6,2]; )
```
Orchestrate imports the source field to a decimal representation with a 2-digit scale.

The following schema overrides this default to import a source field with a 4-digit precision and a 1-digit scale:
```plaintext
record { a:decimal[6,2] {precision = 4, scale = 1}; }
```

**separate**

Specifies that the imported or exported field contains an unpacked decimal with a separate sign byte.

**Applies To**

Fields of the decimal data type; record, subrec, or tagged if it contains at least one field of this type.

**Syntax**

```plaintext
field_definition { separate[, option] };
```

where **option** can be one of these:

- **leading** (default)—the sign is contained in the first byte
- **trailing**—the sign is contained in the last byte

**Discussion**

By default, the sign of an unpacked decimal is contained in the first byte of the imported string. The following table defines the legal values for the sign byte for both ASCII and EBCDIC:

<table>
<thead>
<tr>
<th>Sign</th>
<th>ASCII</th>
<th>EBCDIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>positive</td>
<td>0x2B (ASCII “+”)</td>
<td>0x43 (EBCDIC “+”)</td>
</tr>
<tr>
<td>negative</td>
<td>0x2D (ASCII “-”)</td>
<td>0x60 (EBCDIC “-”)</td>
</tr>
</tbody>
</table>

**Example**

For example, the following schema specifies that the Orchestrate decimal field contains a leading sign and has a precision of 5 and a scale of 2:
```plaintext
record { a:decimal[5,2] {separate}; }
```

Import and export results differ:
• On import, field \( a \) is imported from a decimal representation six bytes long with the sign in the first byte.

• On export, the field is written to six bytes in the destination: the five contained by the decimal and one byte to contain the sign.

**skip**

Skip a number of bytes from the end of the previous imported/ exported field to the beginning of the field.

**Applies To**

Fields of all data types. Cannot apply to record, subrec, or tagged.

**Syntax**

```plaintext
field_definition { skip = nbytes);
```

where `nbytes` is the number of bytes to skip after the previous record. The value of `nbytes` can be negative but the absolute record offset computed from this and the previous field position must always be greater than or equal to 0.

On export, any skipped bytes are set to zero by default. The record-level fill property specifies an explicit value for the skipped bytes.

**Example**

For example, the following statement defines a record schema for the import or export operator:

```plaintext
record { a:int32 {position = 4}; b:int16 {skip = 2}; }
```

Import and export results differ:

• On import, this schema creates each record from an input data file by importing a 32-bit integer, beginning at byte 4 in each input record, skipping the next 2 bytes of the data file, and importing the next two bytes as a 16-bit integer.

• On export, the export operator fills in the first four bytes with zeros, writes out the 32-bit integer, fills the next two bytes with zeroes, and writes the 16-bit integer.

**tagcase**

Explicitly specifies the tag value corresponding to a subfield in a tagged. By default the fields are numbered 0 to \( N-1 \), where \( N \) is the number of fields.
Applies To
Fields within tagged.

Syntax
   field_definition { tagcase = n }
where n is an integer denoting the tag value.

text
Specifies the data representation type of a field as being text rather than binary. Data is formatted as text by default.

Applies To
Fields of all data types except ustring; record. For ustring, the same functionality is available through the charset property.

Syntax
   record { text }
   field_definition { text };
This property is mutually exclusive with binary.

Discussion
Data is formatted as text by default, as follows:

• For the date data type, text specifies that the source data on import or the destination data on export, contains a text-based date in the form %yyyy-%mm-%dd or uformat. See "default_date_format" on page 25-26 for a description of uformat.

• For the decimal data type: an imported or exported field represents a decimal in a string format with a leading space or '-' followed by decimal digits with an embedded decimal point if the scale is not zero.

   For import, the source string format is:
   [+ | -]ddd[.ddd]

   For export, the destination string format is:
   [+ | -]ddd.[ddd]

   Any precision and scale arguments are ignored on export if you specify text.

• For numeric fields (int8, int16, int32, uint8, uint16, uint32, sfloat, and dfloat): the import and export operators assume by default that numeric fields are represented as text; the import operator converts the text representation to a
numeric format by means of C functions to. (See “c_format” on page 25-21, “in_format” on page 25-36, and “out_format” on page 25-44.)

- For the time data type: text specifies that the imported or exported field represents time in the text-based form %hh%n%n%s or uformat. See “default_time_format” on page 25-27 for a description of uformat.

- For the timestamp data type: text specifies a text-based timestamp in the form %yyyy-%mm-%dd %hh%n%n%s or uformat, which is default_date_format and default_time_format concatenated. Refer to “default_date_format” on page 25-26 and “default_time_format” on page 25-27.

**Example**

If you specify the following record schema:

```plaintext
record {text} (a:decimal[5,2];)
```

import and export results are as follows:

- On import, the source decimal is read from a 7-byte string (five bytes for the precision, one for the sign and one for the decimal point).

- On export, the field is written out as a 7-byte string.

**time_format**

Specifies the format of an imported or exported field representing a time as a string or ustring.

**Applies To**

Fields of the time data type; record, subrec, or tagged if it contains at least one field of this type.

**Syntax**

```plaintext
field_definition { time_format = time_format | uformat; }
```

uformat is described in “default_time_format” on page 25-27.

The possible components of the time_format string are:

- %hh: A two-digit hours component.

- %nn: A two-digit minute component (nn represents minutes because mm is used for the month of a date).

- %ss: A two-digit seconds component.

- %ss.n: A two-digit seconds plus fractional part, where n is the number of fractional digits with a maximum value of 6. If n is 0, no decimal point is printed as part of the seconds component. Trailing zeros are not suppressed.
You must prefix each component of the format string with the percent symbol. Separate the string’s components with any character except the percent sign (%). Enclose time_format in quotation marks.

**Discussion**

By default, the format of the time contained in the string is `%hh:%nn:%ss`. However, you can specify a format string defining the format of the string field. The format string must specify hours, minutes, and seconds.

- On import, the time components of the source string must be zero-padded to the character length specified by the format string.
- On export, Orchestrate zero-pads the date components of the destination string to the specified length.

**Restrictions**

You cannot specify time_format with either of the following: midnight_seconds and text. That is, you can specify only one of these options.

**Example**

For example, you define a format string as `%hh:%nn:%ss.3` to specify that the string contains the seconds to three decimal places, that is, to milliseconds.

**timestamp_format**

Specifies the format of an imported or exported field representing a timestamp as a string.

**Applies To**

Fields of the timestamp data type; record, subrec, or tagged if it contains at least one field of this type.

**Syntax**

```plaintext
record ( {timestamp_format = timestamp_format | uformat} )
field_definition { timestamp_format = timestamp_format | uformat };
```

uformat is default_date_format and default_time_format concatenated. The two formats can be in any order but their elements cannot be mixed. The uformat formats are described in “default_date_format” on page 25-26 and “default_time_format” on page 25-27.

The timestamp_format is as follows:

For the date:
• %dd: A two-digit day.
• %mm: A two-digit month.
• %<year_cutoff>yy: A two-digit year derived from yy and the specified four-digit year cutoff.
• %yy: A two-digit year derived from a year cutoff of 1900.
• %yyyy: A four-digit year.
• %ddd: Day of year in three-digit form (range of 1 - 366)

For the time:
• %hh: A two-digit hours component.
• %nn: A two-digit minute component (nn represents minutes because mm is used for the month of a date).
• %ss: A two-digit seconds component.
• %ss.n: A two-digit seconds plus fractional part, where n is the number of fractional digits with a maximum value of 6. If n is 0, no decimal point is printed as part of the seconds component. Trailing zeros are not suppressed. See “Example” on page 25-62 under time_format.

You must prefix each component of the format string with the percent symbol (%). Enclose timestamp_format in single quotation marks.

Default
If you do not specify the format of the timestamp it defaults to the string %yyyy-%mm-%dd %hh:%nn:%ss.

vector_prefix
Specifies 1-, 2-, or 4-byte prefix containing the number of elements in the vector.

Applies To
Fields that are variable-length vectors, which are formatted accordingly.

Syntax

```c
record { vector_prefix [ = n ] }
field_definition { vector_prefix [ = n ] };
```

where n is the optional byte size of the prefix containing the number of elements in the vector; n can be 1 (the default), 2, or 4.

If a vector_prefix is defined for the entire record, you can override the definition for individual vectors.
Discussion

Variable-length vectors must use either a prefix on the vector or a link to another field in order to specify the number of elements in the vector. If the variable-length vector has a prefix, you use the property `vector_prefix` to indicate the prefix length. By default, the prefix length is assumed to be one byte. Behavior on import differs from that on export:

- On import, the source data file must contain a prefix of each vector containing the element count. The import operator reads the length prefix but does not include the prefix as a separate field in the imported data set.

- On export, the export operator inserts the element count as a prefix of each variable-length vector field.

For multi-byte prefixes, the byte ordering is determined by the setting of the `little_endian` (25-23), `big_endian` (25-19), or `native_endian` (25-45) property.

Examples

The following schema specifies that all variable-length vectors are prefixed by a one-byte element count:

```plaintext
record {vector_prefix} (a[]:int32; b[]:int32; )
```

In the following record schema, the `vector_prefix` of the record (1 byte long by default) is overridden for field b, whose `vector_prefix` is two bytes long:

```plaintext
record {vector_prefix} (a[]:int32; b[]:int32 {vector_prefix = 2} )
```

The schema shown below specifies that the variable-length vector a is prefixed by a one-byte element count, and vector b is prefixed by a two-byte element count:

```plaintext
record (a[]:int32 {vector_prefix}; b[]:int32 {vector_prefix = 2};)
```

Import and export results differ:

- On import, the source data file must contain a prefix of each vector containing the element count.

- On export, the export operator inserts the element count as a prefix of each vector.

width

Specifies the number of 8-bit bytes of an imported or exported text-format field. Base your width specification on the value of your `-imexp_charset` option setting. If it's a fixed-width charset, you can calculate the number of bytes exactly. If it's a variable length encoding, base your calculation on the width and frequency of your variable-width characters.
Applies To

Fields of all data types except **date**, **time**, **timestamp**, and **raw**; **record**, **subrec**, or **tagged** if it contains at least one field of this type.

Syntax

```plaintext
record { width = n }
field_definition { width = n };
```

where \( n \) is the number of bytes in the field; you can specify a maximum width of 255 bytes.

Discussion

This property is useful for numeric fields stored in the source or destination file in a text representation.

If no width is specified and you do not use **max_width** to specify a maximum width, numeric fields exported as text have the following number of bytes of maximum width:

- 8-bit signed or unsigned integers: 4 bytes
- 16-bit signed or unsigned integers: 6 bytes
- 32-bit signed or unsigned integers: 11 bytes
- 64-bit signed or unsigned integers: 21 bytes
- single-precision float: 14 bytes (sign, digit, decimal point, 7 fraction, "E", sign, 2 exponent)
- double-precision float: 24 bytes (sign, digit, decimal point, 16 fraction, "E", sign, 3 exponent)

Restriction

On export, if you specify the **width** property with a dfloat field, the **width** must be at least eight bytes long.

**zoned**

Specifies that the field contains an unpacked decimal using either ASCII or EBCDIC text.

Applies To

Fields of the decimal data type; **record**, **subrec**, or **tagged** if it contains at least one field of this type.
Syntax

```plaintext
record { zoned[, option] }
field_definition { zoned[, option ]};
```

where `option` can be either **trailing** or **leading**:

- **trailing** (default) specifies that the sign nibble is in the last byte
- **leading** specifies that the sign nibble is in the first byte

Discussion

Import and export behavior differ:

- On import, the file is read from a zoned representation of the same length, with zoning as defined by the property.
- On export, the file is written to the destination.

The following table defines how the sign is represented in both the ASCII and EBCDIC formats:

<table>
<thead>
<tr>
<th>Sign</th>
<th>ASCII</th>
<th>EBCDIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>positive</td>
<td>Indicated by representing the sign digit normally.</td>
<td>Upper nibble equal to: 0xA, 0xC, 0xE, 0xF</td>
</tr>
<tr>
<td>negative</td>
<td>Indicated by setting the 0x40 bit in the sign digit’s byte. This turns “0” through “9” into “p” through “y”.</td>
<td>Upper nibble equal to: 0xB, 0xD</td>
</tr>
</tbody>
</table>

Example

For example, the following schema specifies that the Orchestrate decimal field has a precision of 5 and a scale of 2 and that its sign nibble is found in the last byte of the field:

```plaintext
record { a:decimal[5,2]{zoned}; }
```

The precision and scale of the source decimal on import, or the destination decimal on export, defaults to the precision and scale of the Orchestrate decimal field. You can use the **precision** (25-49) and **scale** (25-57) properties to override these defaults.
You may want to import data from or export data to COBOL data files. Orchestrate includes a utility, `readcobol`, that converts a COBOL File Description (FD) into an Orchestrate-compatible schema. You can then use the generated schema to import or export COBOL data. This chapter describes how to use this utility.

### Using the COBOL Schema Conversion Utility

The `readcobol` utility takes as input a COBOL program, program excerpt, or data definition file and creates an Orchestrate record schema from each FD (File Description) section or 01 Level statement in the input. You can then use the record schema to import from a COBOL data file, or export to a COBOL data file.
The following COBOL standards are supported:

- ANSI X3.23-1974
- ANSI X3.23-1985
- IBM OSVS
- IBM VSC2
- IBM COB/370
- X/Open CAE Specification, COBOL language (XPG-4)
- Micro Focus COBOL Version 4.0 - May 1995

For example, shown below is a COBOL program excerpt:

01 XYZ-RECORD.
   05 XYZ-REC-TYPE PIC X(2).
   88 HEADER-OR-TRAILER VALUE 'CH' 'CT'.
   88 XYZ-N VALUE 'N3' 'N4'.
   05 XYZ-ACT-NO PIC S9(17) COMP-3.
   05 XYZ-SEGMENT-INDICATORS COMP-3.
   10 XYZ-A-OCCURS PIC S9(01).
   10 XYZ-B-OCCURS PIC S9(01).
   05 XYZ-DATA1Y PIC S9(03).

The Orchestrate record schema generated by readcobol from this excerpt is stored in the file XYZ-RECORD.schema (the name of the 01 Level statement with the suffix .schema). This schema is shown below:

record { record_format = { type = implicit }, big_endian, ascii, binary, delim = none } (  
   XYZ_REC_TYPE:string[2]; // picture is: X(2)  
   XYZ_ACT_NO:decimal[17]; // picture is: S9(17)  
   XYZ_SEGMENT_INDICATORS:subrec (  
      XYZ_A_OCCURS:decimal[1,0]; // picture is: S9(01)  
      XYZ_A_BUCKETS_OCCURS:decimal[1,0]; // picture is: S9(01)  
      XYZ_B_OCCURS:decimal[1,0]; // picture is: S9(01)  
   );  
   XYZ_DATA1Y:string[3]; // picture is: S9(03)  
)

Orchestrate field names do not support the dash character; therefore, all dashes in the COBOL names are converted to underscores in the Orchestrate schema.

In addition to the field definitions shown above in the Orchestrate record schema, readcobol also includes the following record formatting information:

record { record_format = { type = implicit }, big_endian, ascii, binary, delim = none } ( ... )

where:

- record_format = { type = implicit } specifies to import/export data as a stream with no explicit record boundaries. This format means there is no explicit
Chapter 26  COBOL Schema Conversion

Using the Generated Schema

The `readcobol` utility writes the generated schema to a file. Once you have the record schema in a file, you can use that schema to import or export data. Both the Orchestrate `import` and `export` operators have a `-schemafile` option that you use to specify the name of the file.

For example, the previous section created a file named `XYZ-RECORD.schema` containing a record schema. In order to use this file with the `import` operator you could use the following command:

```
$ osh "import -schemafile XYZ-RECORD.schema ... 
```

Supported COBOL Data Storage Formats

This section describes the storage formats supported by `readcobol`, and how those formats are translated to Orchestrate field definitions.

Any unsupported storage formats occurring in an FD section or a program excerpt cause `readcobol` to issue an error message and abort schema generation. However, unsupported storage formats may appear in any other section of a
COBOL program, such as SD or WD, or in a REDEFINES section if that REDEFINES is not used by readcobol for record schema generation.

The following COBOL storage formats are supported:

- COMP, COMP-4, and BINARY (equivalent formats)
- COMP-1 (single precision floating point)
- COMP-2 (double precision floating point)
- COMP-3 and PACKED DECIMAL (equivalent formats)
- DISPLAY (implicit or explicit)

The readcobol utility also supports:

- OCCURS N TIMES where N is an integer value. The field is repeated N times in the generated schema, with a numerical suffix of 1, 2, 3, etc. appended to the end of each field name.
- REDEFINES (see “Handling COBOL Redefines” on page 26-7 for more information on how to configure readcobol for REDEFINES)
- SIGN clause for numeric strings in DISPLAY format.
  - If the SIGN clause includes the SEPARATE option, the sign is treated as explicitly visible and the string length in the generated Orchestrate record schema is incremented by one to include the sign.
- SYNC and SYNCHRONIZED

The following options for PIC or PICTURE are also recognized:

- A, B, X, E, V, S, and 9

Note that the following COBOL storage formats are not supported:

- COMP-5 and COMP-X (Micro Focus types only)
- INDEX
- POINTER
- PROCEDURE POINTER

Table 107 shows how COBOL storage formats are converted to Orchestrate data types.
You invoke `readcobol` from the UNIX command line. Its syntax is:

```
readcobol [options ...] input_file_name
```

The input file name is required.

The input file to `readcobol` must be in ASCII format. The `readcobol` utility has the following guidelines for the input file:

- No maximum line length is enforced
- No distinction is made between area A and area B of each COBOL line
- Any case, including mixed, is accepted
- COBOL keywords recognized by `readcobol` are treated as reserved words
- The period at the end of statements is treated as optional.

The `readcobol` utility has two output options:

1. By default, `readcobol` generates one ASCII output file for each FD section or 01 Level statement in the input file. The name of a generated file is the

<table>
<thead>
<tr>
<th>COBOL Storage Format</th>
<th>PIC</th>
<th>Corresponding Orchestrate Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISPLAY</td>
<td>alpha,</td>
<td>string[n] where n is the byte length of the DISPLAY field</td>
</tr>
<tr>
<td></td>
<td>alphanumeric, numeric</td>
<td></td>
</tr>
<tr>
<td>DISPLAY (specify <code>-c</code> on <code>readcobol</code> command line)</td>
<td>numeric</td>
<td>decimal[p,s] The precision and scale of the decimal are determined from the COBOL PIC format definition.</td>
</tr>
<tr>
<td>COMP, COMP-4, BINARY</td>
<td>numeric</td>
<td>1, 2, and 4 byte COMP numbers are translated to Orchestrate int8, int16, or int32, respectively</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3, and 5-8 byte COMP numbers are translated to raw fields of the same length.</td>
</tr>
<tr>
<td>COMP-3, PACKED DECIMAL</td>
<td>numeric</td>
<td>decimal[p,s] The precision and scale of the decimal are determined from the COBOL PIC format definition.</td>
</tr>
<tr>
<td>COMP-1</td>
<td>not applicable</td>
<td>sfloat</td>
</tr>
<tr>
<td>COMP-2</td>
<td>not applicable</td>
<td>dfloat</td>
</tr>
</tbody>
</table>
corresponding name from the FD section or 01 Level statement, with the extension .schema.

For example, shown below is COBOL fragment containing an 01 Level statement:

```cobol
01 XYZ-RECORD.
   05 XYZ-DATA1 PIC S9(11)V99 COMP-3.
   05 XYZ-DATA1X PIC X(25).
   05 XYZ-DATA1Y PIC S9(03).
```

The file name created by `readcobol` from this fragment is `XYZ-RECORD.schema`.

If you specify the `-s` option on the `readcobol` command line, `readcobol` output is written to the command's `stdout`. This allows you to pipe the output of `readcobol` to another utility, possibly as part of a shell script.

If a problem is encountered, an error message is emitted describing the error. If the problem is with the COBOL syntax, or because the input file uses unsupported COBOL features, the line causing the error is also displayed along with its line number.

Table 108 describes the command line arguments for `readcobol`.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-a</code></td>
<td>Specifies that the imported or exported COBOL data file uses the ASCII character set. This is the default.</td>
</tr>
<tr>
<td><code>-b</code></td>
<td>Specifies that the that the imported or exported COBOL data file uses the big endian binary number format. Big endian is the default format.</td>
</tr>
<tr>
<td><code>-c</code></td>
<td>Specifies that numeric data in DISPLAY format is converted to the Orchestrate decimal data type. The precision and scale of the decimal are determined from the COBOL PIC format definition.</td>
</tr>
<tr>
<td><code>-e</code></td>
<td>Specifies that the imported or exported COBOL data file uses the EBCDIC character set. ASCII is the default character set.</td>
</tr>
<tr>
<td><code>-f</code></td>
<td>The first six columns of a COBOL file often contain line numbers or comments. By default, <code>readcobol</code> discards the first six columns of each line before parsing the line. Using this option specifies that the input file is free-format, and configures <code>readcobol</code> to parse the first six columns of each line when generating the record schema.</td>
</tr>
<tr>
<td><code>-l</code></td>
<td>Specifies that the that the imported or exported COBOL data file uses the little endian binary number format. Big endian is the default format.</td>
</tr>
</tbody>
</table>
Chapter 26  COBOL Schema Conversion

Table 108  Arguments for readcobol (continued)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-n</td>
<td>Specifies that the that the imported or exported COBOL data file uses the native endian format of the machine hosting the file. Big endian is the default format.</td>
</tr>
<tr>
<td>-p redefine_uses_path [-p redefine_uses_path ... ]</td>
<td>Specifies a path of colon separated element names indicating the COBOL field definition to use in the case where a field definition is redefined in the input file. You include a -p for each redefines that you want specify.</td>
</tr>
<tr>
<td>-s</td>
<td>Specifies to write the output of readcobol to stdout, rather than to files.</td>
</tr>
<tr>
<td>-x n</td>
<td>Specifies the alignment size for machine on which the COBOL file is located. The default value is 1. Accepted values for n are 1, 2, 4, and 8.</td>
</tr>
</tbody>
</table>

Handling COBOL Redefines

A COBOL REDEFINES statement specifies an alternative data storage definition for a field. This section describes how to configure readcobol to handle COBOL REDEFINES statements.

The following example COBOL file contains a REDEFINES statement for the field XYZ-DATA1:

```
01 XYZ-RECORD.
   05 XYZ-DATA1 PIC S9(11)V99 COMP-3.
   05 XYZ-DATA1X redefines xyz-data1 PIC x(25).
   05 XYZ-DATA1Y
      Redefines
      XYZ-Data1 PIC S9(03).
```

In order to specify the field definition that you want to use to generate the Orchestrate record schema for XYZ-DATA1, you use the -p argument in the readcobol command line as shown below:
$ readcobol -f -p XYZ-RECORD:XYZ-DATA1Y input_file_name

As you can see, you specify both the 01 Level statement and the REDEFINES field specifying the definition for XYZ-DATA1.
PART 3

The Operator Libraries

This part of the Orchestrate 7.0 Operators Reference describes the Orchestrate Join, Partitioning, Collection, Restructure, SAS, Sorting, and Statistics libraries, and the four RDBMS libraries that provide interfaces to DB2, INFORMIX, Oracle, and Teradata. The chapters in this part are listed below.

Part 1 describes the Orchestrate general library.

Part 2 describes the Orchestrate Import/Export and Cobol schema conversion utility operators.

Chapter 27 The Partitioning Library 27 1
Chapter 28 The Join Library 28 1
Chapter 29 The Collection Library 29 1
Chapter 30 The Restructure Library 30 1
Chapter 31 The Sorting Library 31 1
Chapter 32 The SAS Interface Library 32 1
Chapter 33 The Statistics Library 33 1
Chapter 34 The DB2 Interface Library 34 1
Chapter 35 The Informix Interface Library 35 1
Chapter 36 The Oracle Interface Library 36 1
Chapter 37 The Teradata Interface Library 37 1
The Partitioning Library

Describes the operators in the Partitioning Library which allow you to specify how your data sets are partitioned.

Overview of Partitioners 27 2
The entire Partitioner 27 3
  Using the Partitioner 27 4
  Data Flow Diagram 27 5
  Properties 27 5
  Syntax 27 5
The hash Partitioner 27 5
  Specifying Hash Keys 27 7
  Example 27 7
  Using the Partitioner 27 8
  Data Flow Diagram 27 8
  Properties 27 8
  Syntax and Option 27 9
The modulus Partitioner 27 10
  Data Flow Diagram 27 11
  Properties 27 11
  Syntax and Option 27 11
  Example 27 12
The random Partitioner 27 13
  Using the Partitioner 27 14
  Data Flow Diagram 27 14
  Properties 27 14
  Syntax 27 15
The range Partitioner 27 15
  Considerations When Using Range Partitioning 27 16
  The Range Partitioning Algorithm 27 16
  Specifying Partitioning Keys 27 16
  Creating a Range Map 27 17
Overview of Partitioners

Orchestrate allows you to use **partitioners**, sometimes called **partitioning operators**, to explicitly set the partitioning method of a data set.

By default, Orchestrate inserts partition and sort components in your data flow to meet the partitioning and sorting needs of your use of Orchestrate's predefined operators. See the section “Orchestrate-Inserted Partition and Sort Components” on page xxxvi of this Reference for information on this facility.

Use the partitioners described in this chapter when you want to explicitly control the partitioning and sorting behavior of an operator. You can also create a custom partitioner using the C++ API documented in the Orchestrate 7.0 Developer Guide.
The partitioning library contains seven partitioners. They are:

- The **entire** partitioner. Every instance of an operator on every processing node receives the complete data set as input. It is useful when you want the benefits of parallel execution but every instance of the operator needs access to the entire input data set. This partitioner is described on page 3.

- The **hash** partitioner. Records are partitioned based on a function of one or more fields (the hash partitioning keys) in each record. This partitioner is described on page 5.

- The **modulus** partitioner. This partitioner assigns each record of an input data set to a partition of its output data set as determined by the value of a specified key field modulo the number of partitions. This partitioner is described on page 10.

- The **random** partitioner. Records are randomly distributed across all processing nodes. Like **roundrobin**, random partitioning can rebalance the partitions of an input data set to guarantee that each processing node receives an approximately equal-sized partition. This partitioner is described on page 13.

- The **range** partitioner. Divides a data set into approximately equal size partitions based on one or more partitioning keys. This partitioner is described on page 15. It is used with the help of one of the following:
  - The **writerangemap** operator. This operator takes an input data set produced by sampling and partition sorting a data set and writes it to a file in a form usable by the range partitioner. The range partitioner uses the sampled and sorted data set to determine partition boundaries. The writerangemap operator is described on page 23.
  - The **makerangemap** utility, which determines the approximate range of a data set by sampling the set. The makerangemap utility is described on page 26.

- The **roundrobin** partitioner. The first record goes to the first processing node, the second to the second processing node, and so on. When Orchestrate reaches the last processing node in the system, it starts over. This method is useful for resizing partitions of an input data set that are not equal in size. This partitioner is described on page 28.

- The **same** partitioner. No repartitioning is done. With this partitioning method, records stay on the same processing node. This partitioner is described on page 31.

### The entire Partitioner

In entire partitioning, every instance of an operator on every processing node receives the complete data set as input. This partitioning method is useful when
you want the benefits of parallel execution but every instance of the operator needs access to the entire input data set. For example, you can use this partitioning method to propagate an entire lookup table to each processing node.

When you use the **entire** partitioner, the output data set of the partitioner must be either:

1. A virtual data set connected to the input of a parallel operator using the **any** partitioning method
   
   A virtual data set output by a partitioner overrides the partitioning method of an operator using the **any** partitioning method.

2. A persistent data set

For example, Figure 45 shows an operator that uses the **any** partitioning method.

![Figure 45 Using the entire Partitioner](image)

The **osh** command for this example is:

```
$ osh "... | entire | op ..."
```

### Using the Partitioner

The **entire** partitioner takes a single data set as input and repartitions it to create a single output data set, each partition of which contains a complete copy of the input data set.
**Data Flow Diagram**

```
input data set

entire

output data set
```

**Properties**

Table 109  *entire Partitioner Properties*

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema:</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec to outRec without modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>entire</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output set</td>
<td>set</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

**Syntax**

The *entire* partitioner has no options. Its syntax is simply:

```
entire
```

**The hash Partitioner**

The *hash* partitioner examines one or more fields of each input record, called *hash key* fields. Records with the same values for all hash key fields are assigned to the same processing node. This type of partitioning method is useful when grouping data to perform a processing operation.
When you remove duplicates, you can hash partition records so that records with the same partitioning key values are on the same node. You can then sort the records on each node using the hash key fields as sorting key fields, then remove duplicates, again using the same keys. Although the data is distributed across partitions, the hash partitioner ensures that records with identical keys are in the same partition, allowing duplicates to be found.

The **hash** partitioner guarantees to assign all records with the same hash keys to the same partition, but it does not control the size of each partition. For example, if you hash partition a data set based on a zip code field, where a large percentage of your records are from one or two zip codes, you can end up with a few partitions containing most of your records. This behavior can lead to bottlenecks because some nodes are required to process more records than other nodes.

For example, Figure 46 shows the possible results of hash partitioning a data set using the field age as the partitioning key. Each record with a given age is assigned to the same partition, so for example records with age 36, 40, or 22 are assigned to partition 0. The height of each bar represents the number of records in the partition.

![Figure 46 Hash Partitioning Example](image)

As you can see in Figure 46, the key values are randomly distributed among the different partitions. The partition sizes resulting from a hash partitioner are dependent on the distribution of records in the data set so even though there are three keys per partition, the number of records per partition varies widely, because the distribution of ages in the population is non-uniform.

When hash partitioning, you should select hashing keys that create a large number of partitions. For example, hashing by the first two digits of a zip code produces a maximum of 100 partitions. This is not a large number for a parallel processing system. Instead, you could hash by five digits of the zip code to create up to 10,000 partitions. You also could combine a zip code hash with an age hash (assuming a maximum age of 190), to yield 1,500,000 possible partitions.
Fields that can only assume two values, such as yes/no, true/false, male/female, are particularly poor choices as hash keys.

**Specifying Hash Keys**

Hash keys specify the criteria used to determine the partition into which the hash partitioner assigns a record. The hash partitioner guarantees to assign all records with identical hash keys to the same partition.

The hash partitioner lets you set a primary key and multiple secondary keys. You must define a single primary key, and you have the option of defining as many secondary keys as required by your application. Note, however, that each record field can be used only once as a key. Therefore, the total number of primary and secondary keys must be less than or equal to the total number of fields in the record.

The data type of a partitioning key may be any Orchestrate data type except raw, subrecord, tagged aggregate, or vector.

By default, the hash partitioner uses a case sensitive hash function for strings. You can override this default to perform case insensitive hashing on string fields. In this case, records containing string keys which differ only in case are assigned to the same partition.

**Example**

Figure 47 shows a step using the hash partitioner:

```bash
myDS data set
schema:
a:int32; b:int32; c:int16; d:sfloat; e:string;

hash

partitioning method = any

op
```

Figure 47 Using the hash Partitioner
In this example, fields a and b are specified as partitioning keys. Shown below is the osh command:

```
$ osh "... | hash -key a -key b | op ...
```

By default, the hash partitioner uses a case-sensitive hashing algorithm. You can override this by using the -ci option to the partitioner, which is applied to the string field e in the following example:

```
$ osh "... | hash -key e -ci | op ...
```

To prevent the output of the hash partitioner from being repartitioned, the hash partitioner sets the preserve-partitioning flag in its output.

**Using the Partitioner**

The hash partitioner takes a single data set as input and repartitions the input data set to create a single output data set. Each partition of the output data set contains only a subset of the records from the input data set. You must specify at least one key field to the partitioner.

**Data Flow Diagram**

![Data Flow Diagram](image)

**Properties**

Table 110  hash Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec to outRec without modification</td>
</tr>
</tbody>
</table>
Syntax and Option

Here is the syntax for the `hash` partitioner in an `osh` command:

```
hash
  -key field [-ci | -cs] [-param params]
[-key field [-ci | -cs] [-param params] ...]
  [-collation_sequence locale | collation_file_pathname | OFF]
```

There is one required option, `-key`. You can specify it multiple times.

**Table 111 hash Partitioner Option**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-key</code></td>
<td>`-key field [-ci</td>
</tr>
</tbody>
</table>

Specifies that `field` is a partitioning key field for the `hash` partitioner. You can designate multiple key fields where the order is unimportant.

The key field must be a field of the data set using the partitioner.

The data type of all partitioning key may be any Orchestrate data type including nullable data types.

By default, the `hash` partitioner uses a case sensitive algorithm for hashing. This means that uppercase strings are distinct from lowercase strings. You can override this default to perform case insensitive hashing, by using the `-ci` option after the field name.

The `-param` suboption allows you to specify extra parameters for a field. Specify parameters using `property=value` pairs separated by commas.
The modulus Partitioner

In data mining, data is often arranged in buckets, that is, each record has a tag containing its bucket number. You can use the modulus partitioner to partition the records according to this number. The modulus partitioner assigns each record of an input data set to a partition of its output data set as determined by a specified key field in the input data set. This field can be the tag field.

The partition number of each record is calculated as follows:

\[
\text{partition\_number} = \text{fieldname} \mod \text{number\_of\_partitions}
\]

where:

- fieldname is a numeric field of the input data set.
- number\_of\_partitions is the number of processing nodes on which the partitioner executes. If an partitioner is executed on three processing nodes it has three partitions. Orchestrate automatically passes the number of partitions to partitioners, so you need not supply this information.
Data Flow Diagram

Properties

Table 112 modulus Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema:</td>
<td>inRec:*;</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec to outRec without modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>modulus</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output set</td>
<td>set</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

Syntax and Option

The syntax for the modulus operation in an osh command is shown below:

```
modulus -key filename
```

There is one option. It is required, and you can specify it only once.
Example

In this example, the **modulus** partitioner partitions a data set containing ten records. Four processing nodes run the partitioner, and the **modulus** partitioner divides the data among four partitions.

The input interface schema is as follows:

```plaintext
a:uint32; date:date;
```

Field `a`, of type `uint32`, is specified as the key field, on which the modulus operation is calculated.

Here is the input data set. Each line represents a record:

```
64123 1960-03-30
61821 1960-06-27
44919 1961-06-18
22677 1960-09-24
90746 1961-09-15
21870 1960-01-01
87702 1960-12-22
4705 1961-12-13
47330 1961-03-21
88193 1962-03-12
```

The following table shows the output data set divided among four partitions by the **modulus** partitioner.

<table>
<thead>
<tr>
<th>Partition 0</th>
<th>Partition 1</th>
<th>Partition 2</th>
<th>Partition 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>22677 1960-09-24</td>
<td>87702 1960-12-22</td>
<td>44919 1961-06-18</td>
<td></td>
</tr>
<tr>
<td>4705 1961-12-13</td>
<td>47330 1961-03-21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>88193 1962-03-12</td>
<td>90746 1961-09-15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here are three sample modulus operations, corresponding to the values of the three key fields shown above with double underscores:

Table 113  **modulus** Partitioner Option

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-key</td>
<td>- key field name</td>
</tr>
</tbody>
</table>

Supply the name of key field on whose value the modulus is calculated. The key field must be a numeric field, which is converted to an `uint64` internally.
The random Partitioner

In random partitioning, records are randomly distributed across all processing nodes. Like round robin, random partitioning can rebalance the partitions of an input data set to guarantee that each processing node receives an approximately equal-sized part of the data. The random partitioning method has a slightly higher overhead than roundrobin because of the extra processing required to calculate a random value for each record.

When you use the random partitioner, the output data set of the partitioner must be either:

1. A virtual data set connected to the input of a parallel operator using the any partitioning method
   - A virtual data set output by a partitioner overrides the partitioning method of an operator using the any partitioning method.
2. A persistent data set

For example, Figure 48 shows an operator that uses the any partitioning method.

![Diagram](random_partitioner.png)

Figure 48 Using the random Partitioner

To override the any partitioning method of op and replace it by the random partitioning method, you place the random partitioner into the step.
Here is the `osh` command for this example:

```
$ osh "... | random | op ..."
```

**Using the Partitioner**

The `random` partitioner takes a single data set as input and repartitions the input data set to create a single output data set, each partition of which contains a random subset of the records in the input data set.

**Data Flow Diagram**

![Data Flow Diagram]

**Properties**

Table 114  *random Operator Properties*

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema:</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec to outRec without modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>random</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output set</td>
<td>cleared</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>
Syntax

The syntax for the random partitioner in an osh command is:

```
random
```

The random partitioner has no options.

The range Partitioner

A range partitioner divides a data set into approximately equal size partitions based on one or more partitioning keys. Range partitioning is often a preprocessing step to performing a total sort on a data set.

This chapter describes the range partitioner, the partitioner that implements range partitioning. It also describes the writerangemap operator, which you use to construct the range map file required for range partitioning, and the stand-alone makerangemap utility.

The range partitioner guarantees that all records with the same partitioning key values are assigned to the same partition and that the partitions are approximately equal in size so all nodes perform an equal amount of work when processing the data set.

Figure 49 shows an example of the results of a range partition. The partitioning is based on the age key, and the age range for each partition is indicated by the numbers in each bar. The height of the bar shows the size of the partition.

![Range Partitioning Example](image)

**Figure 49  Range Partitioning Example**

All partitions are of approximately the same size. In an ideal distribution, every partition would be exactly the same size. However, you typically observe small differences in partition size.

In order to size the partitions, the range partitioner orders the partitioning keys. The range partitioner then calculates partition boundaries based on the partitioning keys in order to evenly distribute records to the partitions. As shown above, the distribution of partitioning keys is often not even; that is, some partitions contain many partitioning keys, and others contain relatively few.
However, based on the calculated partition boundaries, the number of records in each partition is approximately the same.

Range partitioning is not the only partitioning method that guarantees equivalent-sized partitions. The random and roundrobin partitioning methods also guarantee that the partitions of a data set are equivalent in size. However, these partitioning methods are keyless; that is, they do not allow you to control how records of a data set are grouped together within a partition.

Considerations When Using Range Partitioning

The range partitioner creates an output data set with approximately equal size partitions where the partitions are ordered by the partitioning keys. This type of partitioning is often useful when you want to perform a total sort of a data set so that the individual records of a partition are sorted and the partitions themselves are sorted. See “The tsort Operator” on page 31-2 for more information on performing a total sort.

In order to perform range partitioning your application requires two steps: one step to calculate the range partition boundaries, and a second step to actually use the partitioner. Thus the range partitioner adds processing overhead to your application.

If you only want to perform load balancing so that your partitions are approximately equal in size, you should use the random or roundrobin partitioners. These partitioners add little overhead in comparison to range partitioning. You should use the range partitioner only when you need ordered partitions, not as a general-purpose load-balancing partitioner.

The Range Partitioning Algorithm

The range partitioner uses a probabilistic splitting technique to range partition a data set. This technique is described in Parallel Sorting on a Shared-Nothing Architecture Using Probabilistic Splitting by DeWitt, Naughton, and Schneider in Query Processing in Parallel Relational Database Systems by Lu, Ooi, and Tan, IEEE Computer Society Press, 1994.

Specifying Partitioning Keys

The range partitioner uses the partitioning keys to determine the partition boundaries of a data set. You must define at least one partitioning key, and you have the option of defining as many keys as required by your application. However, each record field can be used only once as a key. Therefore, the total number of keys must be less than or equal to the total number of fields in the record.
The data type of a key can be any one of the defined Orchestrate data types except raw, subrecord, tagged aggregate, or vector. Specifying a partitioning key of one of these data types causes the range partitioner to issue an error and abort execution.

By default, the range partitioner does case-sensitive comparison. This means that uppercase strings appear before lowercase strings in a partitioned data set. You can override this default if you want to perform case-insensitive partitioning on string fields.

By default, the range partitioner uses an ascending order, so that smaller values appear before larger values in the partitioned data set. You can specify a descending order as well, so that larger values appear before smaller values in the partitioned data set.

### Creating a Range Map

This section describes the procedure for creating and configuring a range map. Once you have created the range map, you can supply it to the range partitioner to partition a data set. See “Example: Configuring and Using range Partitioner” on page 27-20 for the code required to configure and use the range partitioner.

To perform range partitioning, the range partitioner must determine the partition boundaries of a data set as determined by the partitioning keys. For example, if you specify age as the partitioning key, the range partitioner must be able to determine the low and high age values that are the boundaries for each partition in order to generate equally sized partitions. In this case, all records with the same age value between the first set of boundaries is assigned to partition 0; records within the next set of boundaries is assigned to partition 1, etc.

In order for the range partitioner to determine the partition boundaries, you pass the range partitioner a sorted sample of the data set to be range partitioned. From this sample, the range partitioner can determine the appropriate partition boundaries for the entire data set.

| Note | When you use the range partitioner in preparation for a total sort, it is important that the keys for the range partitioning and the sorting be specified the same way, including such attributes as case sensitivity and sort order. |

- **To use a range partitioner:**
  1. Create a random sample of records from the data set to be partitioned using the sample partitioner. Your sample should contain at least 100 records per processing node in order to accurately determine the partition boundaries.

     See Chapter 17, “The sample Operator” for more information on the sample partitioner.
2 Use the `tsort` partitioner, in sequential mode, to perform a complete sort of the sampled records using the partitioning keys as sorting keys. Since your sample size should typically be less than 25,600 (assuming a maximum of 256 nodes in your system), the sequential-mode sort is quick.

You must sort the sampled data set using the same fields as sorting keys, and in the same order, as you specify the partitioning keys to the `range` partitioner. Also, the sorting keys must have the same characteristics for case sensitivity and ascending or descending ordering as you specified for the `range` partitioner.

3 Use the `writerangemap` operator to store the sorted, sampled data set to disk as a file. This file is called a range map. See “The `writerangemap` Operator” on page 27-23 for more information on this operator.

4 Configure the `range` partitioner using the sorted, sampled data file. The `range` partitioner determines the partition boundaries for the entire data set based on this sample.

5 Use the `-key` argument to specify the partitioning keys to the `range` partitioner. Note that you must specify the same fields as the partitioning keys, and in the same order, as you specified as sorting keys above in Step 2. Also, the partitioning keys must have the same characteristics for case sensitivity and ascending or descending ordering as you specified for the sort.
Figure 50 shows a data flow where the second step begins with a range partitioner:

Note in Figure 50 that you sample and sort the data set used to configure the range partitioner in one Orchestrate step, and use the range partitioner in a second step. This is because all the processing of the sorted sample is not complete until the first step ends.

This example shows a range partitioner configured from the same data set that you want to partition. However, you may have multiple data sets whose record distribution can be accurately modeled using a range partitioner configured from a single data set. In this case, you can use the same range partitioner for all the data sets.
Example: Configuring and Using range Partitioner

This section gives an example of configuring a range partitioner. This example contains two steps: one to create the sorted, sampled data set used by the range partitioner and a second step containing an operator that uses the partitioner.

The input data set for this example has the following schema:

```
record {
    a: int32;
    b: int8;
    c: string[5];
    d: int16;
    e: string;
}
```

You decide to create a range partition using fields a and c as the range partitioning keys. Your system contains 16 processing nodes.

Here are the UNIX shell and osh commands for these two steps:

```
$ numSampled=1600 # Line 1
$ numRecs=`dsrecords inDS.ds | cut -f1 -d' '` # Line 2
$ percent=`echo '10 k $numSampled $numRecs / 100 * p q' | dc` # Line 3

$ osh "sample $percent < inDS.ds | tsort -key a -key c [seq] | writerangemap
    -rangemap sampledData -overwrite
    -interface 'record(a:int32; c:string[5];)'"

$ osh "range -sample sampledData -key a -key c < inDS.ds | op1
..."
```

The sample size required by the range partitioner is at least 100 records per processing node. Since there are 16 processing nodes, you specify the sample size as 1600 records on Line 1.

On Lines 2 and 3 you calculate the sample size as a percentage of the total number of records in the data set. This calculation is necessary because the sample operator requires the sample size to be expressed as a percentage.

In order to calculate the percentage, you use the dsrecords utility to obtain the number of records in the input data set. The return value of dsrecords has the form "# records" where # is the number of records. Line 2 returns the record count and strips off the word "record" from the value.

Line 3 then calculates a floating point value for the sample percentage from the 1600 records required by the sample and the number of records in the data set. This example uses the UNIX dc command to calculate the percentage. In this command, the term 10 k specifies that the result has 10 digits to the left of the decimal point. See the man page on dc for more information.
The \texttt{range} partitioner in this example partitions an input data set based on fields $a$ and $c$. Therefore, the \texttt{writerangemap} operator only writes fields $a$ and $c$ to the output file used to generate the \texttt{range} partitioner.

**Using the Partitioner**

The \texttt{range} partitioner takes a single data set as input and repartitions the input data set to create a single output data set, each partition of which contains a subset of the records from the input data set.

**Data Flow Diagram**

![Data Flow Diagram]

**Properties**

Table 115 \textit{range} Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema:</td>
<td>\texttt{inRec:*}</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>\texttt{outRec:*}</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>\texttt{inRec to outRec without modification}</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>range</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output set</td>
<td>set</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>
Syntax and Options

The syntax for the range partitioner is shown below. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose the value in single quotes. At least one -key option is required.

```
range
  -key fieldname [-ci | -cs] [-asc | desc] [-nulls first | last] [-ebcdic]
    [-params params]
  [-key fieldname [-ci | -cs] [-asc | desc] [-nulls first | last] [-ebcdic]
    [-params params] ...]
  [-collation_sequence locale| collation_file_pathname| OFF]
  [-sample sorted_sampled_data_set]
```

Table 116  range Partitioner Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-key</td>
<td>-key fieldname [-ci</td>
</tr>
</tbody>
</table>

Specifies that fieldname is a partitioning key field for the range partitioner. You can designate multiple partitioning key fields. You must specify the same fields, in the same order and with the same characteristics, for ascending/ descending and case sensitive/ insensitive order, as used to sort the sampled data set.

The field name fieldname must be a field of the data set using the partitioner or a field created using an input field adapter on the data set.

By default the range partitioner uses a case-sensitive algorithm. You can perform case-insensitive partitioning by using the -ci option after the field name.

-ascending specifies ascending order sort; records with smaller values for fieldname are assigned to lower number partitions. This is the default.

-descending specifies descending order sort; records with smaller values for fieldname are assigned to lower number partitions.

-nulls first | last) specifies whether nulls appear first or last in the sorted partition. The default is first.

-ebcdic specifies that the EBCDIC collating sequence is used.

The -param suboption allows you to specify extra parameters for a field. Specify parameters using property=value pairs separated by commas.
The writerangemap Operator

The writerangemap operator takes an input data set produced by sampling and partition sorting a data set and writes it to a file in a form usable by the range partitioner. The range partitioner uses the sampled and sorted data set to determine partition boundaries.

Data Flow Diagram

```
input data set

writerangemap

The operator takes a single data set as input. You specify the input interface schema of the operator using the -interface option. Only the fields of the input data set specified by -interface are copied to the output file.

```

---

Table 116  range Partitioner Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-collation_sequence</td>
<td>-collation_sequence locale</td>
</tr>
<tr>
<td></td>
<td>This option determines how your string data is sorted. You can:</td>
</tr>
<tr>
<td></td>
<td>• Specify a predefined IBM ICU locale</td>
</tr>
<tr>
<td></td>
<td>• Write your own collation sequence using ICU syntax, and supply its collation_file_pathname</td>
</tr>
<tr>
<td></td>
<td>• Specify OFF so that string comparisons are made using Unicode code-point value order, independent of any locale or custom sequence.</td>
</tr>
<tr>
<td></td>
<td>By default, Orchestrate sorts strings using byte-wise comparisons.</td>
</tr>
<tr>
<td></td>
<td>For more information, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:</td>
</tr>
<tr>
<td></td>
<td><a href="http://oss.software.ibm.com/icu/userguide/Collate_Intro.htm">http://oss.software.ibm.com/icu/userguide/Collate_Intro.htm</a></td>
</tr>
</tbody>
</table>

| -sample | specifies the file containing the sorted, sampled data set used to configure the range partitioner. |
|         | Specified file name is sorted_sampled_data_set |

---
Properties

Table 117  writerangemap Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>0 (produces a data file as output)</td>
</tr>
<tr>
<td>Input interface schema:</td>
<td>specified by the interface arguments</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>none</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec to outRec without modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>sequential only</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>range</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output set</td>
<td>set</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

Syntax and Options

The syntax for the `writerangemap` operator is shown below. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose the value in single quotes.

```
writerangemap
  [·key fieldname[·key fieldname...]] | [·interface schema]
  ·collation_sequence locale|collation_file_pathname|OFF
  [·overwrite]
  ·rangemap filename
```
### Table 118 | writerangemap Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-key</code></td>
<td><code>-key</code> fieldname</td>
</tr>
<tr>
<td></td>
<td>Specifies an input field copied to the output file. Only information about the specified field is written to the output file. You only need to specify those fields that you use to range partition a data set. You can specify multiple <code>-key</code> options to define multiple fields. This option is mutually exclusive with <code>-interface</code>. You must specify either <code>-key</code> or <code>-interface</code>, but not both.</td>
</tr>
<tr>
<td><code>-collation_sequence</code></td>
<td>`-collation_sequence locale</td>
</tr>
<tr>
<td></td>
<td>This option determines how your string data is sorted. You can: • Specify a predefined IBM ICU locale • Write your own collation sequence using ICU syntax, and supply its <code>collation_file_pathname</code> • Specify <code>OFF</code> so that string comparisons are made using Unicode code-point value order, independent of any locale or custom sequence. By default, Orchestrate sorts strings using byte-wise comparisons. For more information, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site: <a href="http://oss.software.ibm.com/icu/us...">http://oss.software.ibm.com/icu/us...</a>.</td>
</tr>
<tr>
<td><code>-interface</code></td>
<td><code>-interface</code> schema</td>
</tr>
<tr>
<td></td>
<td>Specifies the input fields copied to the output file. Only information about the specified fields is written to the output file. You only need to specify those fields that you use to range partition a data set. For a description of schema format, see “Record Schemas” on page 22-3. This option is mutually exclusive with <code>-key</code>; that is, you can specify <code>-key</code> or <code>-interface</code>, but not both.</td>
</tr>
<tr>
<td><code>-overwrite</code></td>
<td><code>-overwrite</code></td>
</tr>
<tr>
<td></td>
<td>Tells the operator to overwrite the output file, if it exists. By default, the operator does not overwrite the output file. Instead it generates an error and aborts the application if the file already exists.</td>
</tr>
<tr>
<td><code>-rangemap</code></td>
<td><code>-rangemap</code> filename</td>
</tr>
<tr>
<td></td>
<td>Specifies the pathname of the output file which will contain the sampled and sorted data.</td>
</tr>
</tbody>
</table>
Using the writerange Operator

For an example showing the use of the writerange operator, see “Example: Configuring and Using range Partitioner” on page 27-20.

The makerangemap Utility

Orchestrate supplies the `makerangemap` utility to generate the sorted, sampled data file, or range map, used to configure a range partitioner. The `makerangemap` utility determines the sample size based on the number of processing nodes in your system as defined by the Orchestrate configuration file, or you can explicitly specify the sample size as a number of records or as a percentage of the input data set.

Syntax and Options

The syntax for `makerangemap` is below. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose the value in single quotes.

```
makerangemap -rangemap filename
    [-f]
    -key fieldname [ci | cs | -asc | -desc | -ebcdic]
    [-key fieldname ...]
    -collation_sequence locale|collation_file_pathname| OFF
    [-percentage percent]
    [-size samplesize]
```

where:

- `filename` specifies the name of the file containing the range map, a file containing the sorted, sampled records used to configure a range partitioner.
- `fieldname` specifies the field(s) of the input data used as sorting key fields. You can specify one or more key fields. Note that you must sort the sampled data set using the same fields as sorting keys, and in the same order, as you specify as partitioning keys. Also, the sorting keys must have the same ascending/descending and case sensitive/insensitive properties.
## Table 119  **makerangemap Utility Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>-collation_sequence</strong></td>
<td>`-collation_sequence locale</td>
</tr>
<tr>
<td></td>
<td>This option determines how your string data is sorted. You can:</td>
</tr>
<tr>
<td></td>
<td>• Specify a predefined IBM ICU locale</td>
</tr>
<tr>
<td></td>
<td>• Write your own collation sequence using ICU syntax, and supply its collation_file_pathname</td>
</tr>
<tr>
<td></td>
<td>• Specify OFF so that string comparisons are made using Unicode code-point value order, independent of any locale or custom sequence.</td>
</tr>
<tr>
<td></td>
<td>By default, Orchestrate sorts strings using byte-wise comparisons.</td>
</tr>
<tr>
<td></td>
<td>For more information, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:</td>
</tr>
<tr>
<td></td>
<td><a href="http://oss.software.ibm.com/icu/userguide/Collate_Intro.htm">http://oss.software.ibm.com/icu/userguide/Collate_Intro.htm</a></td>
</tr>
<tr>
<td><strong>-f</strong></td>
<td><code>-f</code></td>
</tr>
<tr>
<td></td>
<td>Specifies to overwrite the output file, if it exists. By default, the utility does not overwrite the output file. Instead, it generates an error and aborts if the file already exists.</td>
</tr>
<tr>
<td><strong>-key</strong></td>
<td>`-key fieldname [-ci</td>
</tr>
<tr>
<td></td>
<td>Specifies that fieldname is a sorting key field. You can designate multiple sorting key fields. You must specify the same fields, in the same order and with the same characteristics for ascending/descending and case sensitive/insensitive order, as used by the range partitioner.</td>
</tr>
<tr>
<td></td>
<td>The field name fieldname must be a field of the input data set.</td>
</tr>
<tr>
<td></td>
<td>By default, the sort uses a case-sensitive algorithm. This means that uppercase strings come before lowercase strings. You can override this default to perform case-insensitive sorting, where:</td>
</tr>
<tr>
<td></td>
<td>-cs specifies case sensitive (default)</td>
</tr>
<tr>
<td></td>
<td>-ci specifies case insensitive</td>
</tr>
<tr>
<td></td>
<td>-ebcdic specifies that fieldname be sorted using the EBCDIC collating sequence.</td>
</tr>
<tr>
<td></td>
<td>By default, the sort uses ascending order, so that records with smaller values for fieldname come before records with larger values. You can specify descending sorting order, so that records with larger values come first, where:</td>
</tr>
<tr>
<td></td>
<td>-asc specifies ascending, the default</td>
</tr>
<tr>
<td></td>
<td>-desc specifies descending</td>
</tr>
</tbody>
</table>
Using the makerangemap Utility

As described in “Example: Configuring and Using range Partitioner” on page 27-20, you need to write two Orchestrate steps to configure and use a range partitioner. The first step configures the sorted, sampled file, or range map, and the second step includes the range partitioner that uses the range map.

Orchestrate supplies a UNIX command line utility, makerangemap, that you can also use to create a range map. Using this utility eliminates the Orchestrate step used to create the range map from your application.

The makerangemap utility determines the sample size equal to 100 times the number of processing nodes in your system in the default node pool as defined by the Orchestrate configuration file. For example, if your configuration file contains 32 nodes in the default node pool, makerangemap creates a sample 3200 records in size.

You can explicitly specify the sample size as a total number of records for the entire sample or as a percentage of the input data set. This method may be useful when the operator using the range partitioner executes on a subset of processing nodes.

Table 119 makerangemap Utility Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-percentage or -p</td>
<td>-percentage percent</td>
</tr>
<tr>
<td></td>
<td>Specifies the sample size of the input data set as a percentage.</td>
</tr>
<tr>
<td></td>
<td>The sample size defaults to 100 records per processing node in the default node pool as defined by the Orchestrate configuration file.</td>
</tr>
<tr>
<td></td>
<td>If specified, percent should be large enough to create a sample of 100 records for each processing node executing the operator using the range partitioner.</td>
</tr>
<tr>
<td></td>
<td>The options -size and -percentage are mutually exclusive.</td>
</tr>
<tr>
<td>-rangemap or -rm</td>
<td>-rangemap filename</td>
</tr>
<tr>
<td></td>
<td>Specifies the pathname of the output file containing the sampled and sorted data.</td>
</tr>
<tr>
<td>-size or -s</td>
<td>-size samplesize</td>
</tr>
<tr>
<td></td>
<td>Specifies the size of the sample taken from the input data set. The size defaults to 100 records per processing node in the default node pool as defined by the Orchestrate configuration file.</td>
</tr>
<tr>
<td></td>
<td>If specified, samplesize should be set to at least 100 multiplied by the number of processing nodes executing the operator using the range partitioner.</td>
</tr>
<tr>
<td></td>
<td>The options -size and -percentage are mutually exclusive.</td>
</tr>
</tbody>
</table>
nodes, not on all nodes in your system. For example, if you execute the operator on only eight processing nodes, your sample size need only be 800 records. In this case, you can explicitly specify the sample size in terms of a record count, or as a percentage of the input data set.

As an example of the use of `makerangemap`, consider “Example: Configuring and Using range Partitioner” on page 27-20. In that example, two `osh` commands are used to configure and use the range partitioner. Instead, you could use the `makerangemap` utility and one `osh` command, as follows:

```
$ makerangemap -rangemap sampledData -key a -key c inDS.ds
$ osh "range -sample sampledData -key a -key c < inDS.ds | op1 ...
```

The roundrobin partitioner

In **round robin** partitioning, the first record of an input data set goes to the first processing node, the second to the second processing node, and so on. When you reach the last processing node in the system, start over. This method is useful for resizing the partitions of an input data set that are not equal in size. Round robin partitioning always creates approximately equal-sized partitions.

When you use the **roundrobin** partitioner, the output data set of the partitioner must be either:

1. A virtual data set connected to the input of a parallel operator using the any partitioning method
   
   A virtual data set output by a partitioner overrides the partitioning method of an operator using the any partitioning method.

2. A persistent data set

For example, Figure 51 shows an operator that uses the any partitioning method.

```
Figure 51  Using the random Partitioner
```
To override the any partitioning method of op and replace it by the random partitioning method, you place the roundrobin partitioner into the step as shown.

Shown below is the osh command for this example:

```bash
$ osh "... | roundrobin | op ..."
```

**Using the Partitioner**

The roundrobin partitioner takes a single data set as input and repartitions the input data set to create a single output data set.

**Data Flow Diagram**

![Data Flow Diagram](image)

**Properties**

Table 120  **roundrobin Operator Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec to outRec without modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>roundrobin</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output set</td>
<td>cleared</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>
Syntax

The syntax for the `roundrobin` partitioner in an `osh` command is shown below. There are no options.

```
roundrobin
```

The same Partitioner

In `same` partitioning, the operator using the data set as input performs no repartitioning. Records stay in the same partition.

When you use the `same` partitioner, the output data set of the partitioner must be either:

1. A virtual data set connected to the input of a parallel operator using the `any` partitioning method
   - A virtual data set output by a partitioner overrides the partitioning method of an operator using the `any` partitioning method.
2. A persistent data set

For example, **Figure 52** shows an operator that uses the `any` partitioning method.

![Figure 52 Using the random Partitioner](image)

To override the `any` partitioning method of `op` and replace it by the same partitioning method, you place the `same` partitioner into the step as shown.

Shown below is the `osh` command for this example:

```
$ osh "... | same | op ..."
```
Using the Partitioner

The \texttt{same} partitioner takes a single data set as input and repartitions the input data set to create a single output data set.

Data Flow Diagram

![Data Flow Diagram](image)

Properties

Table 121 \texttt{same} Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec to outRec without modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>same</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output set</td>
<td>propagated</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

Syntax

The syntax for the \texttt{same} partitioner in an \texttt{osh} command is shown below. There are no options.

\texttt{same}
The Join Library

Describes the operators in the Join Library which transfer records and values from left and right input data sets based on the values of key fields.

### Overview
28 2

### Data Flow Diagrams
28 3
- innerjoin, leftouterjoin, and rightouterjoin 28 3
- fullouterjoin 28 4

### Properties
28 4
- Transfer Behavior 28 4

### Input Data Set Requirements
28 5
- Memory Use 28 5

### Comparison with Other Operators
28 5
- Input Data Used in the Examples 28 6

#### The innerjoin Operator
28 7
- Syntax and Options 28 7
- Example 28 9

#### The leftouterjoin Operator
28 9
- Syntax and Options 28 10
- Example 28 11

#### The rightouterjoin Operator
28 12
- Syntax and Options 28 12
- Example 28 13

#### The fullouterjoin Operator
28 14
- Syntax and Options 28 15
- Example 28 16
Overview

This chapter of the Orchestrate 7.0 Operators Reference describes how to use the join operators. To understand the information in this chapter you should be familiar with the operator concepts defined in the Orchestrate 7.0 User Guide.

The join library contains four operators. The `innerjoin`, `leftouterjoin`, and `rightouterjoin` operators accept two or more input data sets and perform cascading joins on them. The `fulloutjoin` operator accepts exactly two data sets. All four operators output a single data set.

In this chapter, the first input data set and the data set resulting from an intermediate join are called the left data set and the data set joining them is called the right data set.

Here is a brief description of each join operator:

- The `innerjoin` operator outputs the records from two or more input data sets whose key fields contain equal values. Records whose key fields do not contain equal values are dropped. This operator is described on page 7.

- The `leftouterjoin` operator outputs all values from the left data set and outputs values from the right data set only where key fields match. The operator drops the key field from the right data set. Otherwise, the operator writes default values. This operator is described on page 9.

- The `rightouterjoin` operator outputs all values from the right data set and outputs values from the left data set only where key fields match. The operator drops the key field from the left data set. Otherwise, the operator writes default values. This operator is described on page 12.

- The `fullouterjoin` operator transfers records in which the contents of the key fields are equal from both input data sets to the output data set. It also transfers records whose key fields contain unequal values from both input data sets to the output data set. This operator is described on page 14.
Data Flow Diagrams

innerjoin, leftouterjoin, and rightouterjoin

The innerjoin, leftouterjoin, and rightouterjoin operators accept two or more input data sets and perform cascading joins on them. The data set that results from joining the first and second datasets is joined with the third dataset, and the result of that join is joined with the fourth dataset, and so on, until the last data set is joined.

```
key1a:type1a; ... key1n:type1n;
input0Rec:*;            key1a:type1a; ...
                        key1n:type1n;
                        input1Rec:*;
                        (with modification)
                        key1a:type1a; ...
                        key1n:type1n;
                        input2Rec:*;
                        (with modification)
                        key1a:type1a; ...
                        key1n:type1n;
                        inputnRec:*;
```

```
input1           input2                ...         inputn
```

```
join operator
```

```
output join data set
```
**fullouterjoin**

```
input0

key1a:type1a; ... key1n:type1n;
input0Rec:*;

key1a:type1a; ... key1n:type1n;
input1Rec:*;

fullouterjoin

input0Rec:*; input1Rec:*;
(with modification)

fullouterjoin

output join data set
```

## Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>2 or more for the <strong>innerjoin</strong>, <strong>leftouterjoin</strong>, <strong>rightouterjoin</strong> operators and exactly 2 for the <strong>fullouterjoin</strong> operator</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema:</td>
<td>key1a:type1a; ... key1n:type1n; input0Rec:*;</td>
</tr>
<tr>
<td></td>
<td>key1a:type1a; ... key1n:type1n; input1Rec:*;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>leftRec:<em>; rightRec:</em>;</td>
</tr>
<tr>
<td>Transfer behavior from source to output</td>
<td>leftRec; -&gt;leftRec; rightRec; -&gt;rightRec; with modifications</td>
</tr>
<tr>
<td>Composite operator</td>
<td>yes</td>
</tr>
<tr>
<td>Input partitioning style</td>
<td>keys in same partition</td>
</tr>
</tbody>
</table>

## Transfer Behavior

The join operators:

- Transfer the schemas of the input data sets to the output and catenates them to produce a 'join' data set.
• Transfer the schema variables of the input data sets to the corresponding output schema variables. The effect is a catenation of the fields of the schema variables of the input data sets. For the fullouterjoin operator, duplicate field names are copied from the left and right data sets as leftRec_field and rightRec_field, where field is the name of the duplicated key field.

• Transfer duplicate records to the output, when appropriate.

• The innerjoin and leftouterjoin operators drop the key fields from the right data set.

• The rightouterjoin operator drops the key fields from the left data set.

Input Data Set Requirements

These are as follows:

• The innerjoin, leftouterjoin, and rightouterjoin operators accept two or more input data sets, perform cascading joins on them, and produce one output data set. The fullouterjoin operator acts on exactly two input data sets to produce one output data set.

• Each record of the left data set and the right data set must have one or more key fields with the same names and compatible data types. If they do not, you can change unmatched names and incompatible data types by means of the modify operator. See Chapter 13, “The modify Operator” for a description of the modify operator.

  Compatible data types are those that Orchestrate converts by default, translating a value in a source field to the data type of a destination field.

• Key fields can contain nulls. The join operators treat nulls as distinct values for the purposes of key comparison. Null handling is automatic.

Memory Use

For the right data set, for each value of the key group, the collection of records with that key value must fit comfortably in memory to prevent paging from slowing performance.

Comparison with Other Operators

The join operators are similar in some ways to the lookup and merge operators in that each are used to combine data from two or more sources. Each operator takes a primary input on input port 0 and one or more secondary inputs on input ports 1 and possibly 2, 3, 4, etc. and produces an output on output port 0 and one or more reject outputs on output ports 1 and possibly 2, 3, 4, etc. The differences are summarized in Table 123.
The next sections discuss the four join operators individually and give example joins for each. To simplify the examples, only two input data sets are used. However, the **innerjoin**, **leftouterjoin**, and **rightouterjoin** operators accept two or more input data sets and perform cascading joins on them, but the process is similar for each join. 

### Table 123  Comparison of Joins, Lookup, and Merge

<table>
<thead>
<tr>
<th>Description</th>
<th>Joins</th>
<th>Lookup</th>
<th>Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>RDBMS-style relational tables</td>
<td>Source and lookup table in RAM</td>
<td>Master table and one or more update tables</td>
</tr>
<tr>
<td><strong>Memory usage</strong></td>
<td>Light</td>
<td>Heavy</td>
<td>Light</td>
</tr>
<tr>
<td><strong>Number and names of inputs</strong></td>
<td>2 or more inputs</td>
<td>1 source and N lookup tables</td>
<td>1 master table and N update tables</td>
</tr>
<tr>
<td><strong>Handling of duplicates in primary input</strong></td>
<td>OK, produces a cross-product</td>
<td>OK</td>
<td>Warning given. Duplicate will be an unmatched primary.</td>
</tr>
<tr>
<td><strong>Handling of duplicates in secondary input</strong></td>
<td>OK, produces a cross-product</td>
<td>Warning given. The second lookup table entry is ignored.</td>
<td>OK only when N = 1</td>
</tr>
<tr>
<td><strong>Options on unmatched primary</strong></td>
<td>NONE</td>
<td>Fail, continue, drop, or reject. Fail is the default.</td>
<td>Keep or drop. Keep is the default.</td>
</tr>
<tr>
<td><strong>Options on unmatched secondary</strong></td>
<td>NONE</td>
<td>NONE</td>
<td>Capture in reject sets</td>
</tr>
<tr>
<td><strong>On match, secondary entries are reusable</strong></td>
<td>reusable</td>
<td>reusable</td>
<td>reusable</td>
</tr>
<tr>
<td><strong>Number of outputs</strong></td>
<td>1</td>
<td>1 output and optionally 1 reject</td>
<td>1 output and 1 reject for each update table</td>
</tr>
<tr>
<td><strong>Captured in reject sets</strong></td>
<td>Does not apply</td>
<td>Unmatched primary entries</td>
<td>Unmatched secondary entries</td>
</tr>
</tbody>
</table>

**Input Data Used in the Examples**

The next sections discuss the four join operators individually and give example joins for each. To simplify the examples, only two input data sets are used. However, the **innerjoin**, **leftouterjoin**, and **rightouterjoin** operators accept two or more input data sets and perform cascading joins on them, but the process is similar for each join.
Each operator section shows the following input data sets as acted upon by the operator in question. The data sets, which are presented in tabular form, track real estate purchases. Equal values in the price field of each data set are shown with double underscores.

<table>
<thead>
<tr>
<th>left data set</th>
<th>right data set</th>
</tr>
</thead>
<tbody>
<tr>
<td>status field</td>
<td>price field</td>
</tr>
<tr>
<td>Sold</td>
<td>125</td>
</tr>
<tr>
<td>Sold</td>
<td>213</td>
</tr>
<tr>
<td>Offered</td>
<td>378</td>
</tr>
<tr>
<td>Pending</td>
<td>575</td>
</tr>
<tr>
<td>Pending</td>
<td>649</td>
</tr>
<tr>
<td>Offered</td>
<td>777</td>
</tr>
<tr>
<td>Offered</td>
<td>908</td>
</tr>
<tr>
<td>Pending</td>
<td>908</td>
</tr>
<tr>
<td></td>
<td>price field</td>
</tr>
<tr>
<td></td>
<td>id field</td>
</tr>
<tr>
<td></td>
<td>113</td>
</tr>
<tr>
<td></td>
<td>BR9658</td>
</tr>
<tr>
<td></td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>C22538</td>
</tr>
<tr>
<td></td>
<td>285</td>
</tr>
<tr>
<td></td>
<td>RU5713</td>
</tr>
<tr>
<td></td>
<td>628</td>
</tr>
<tr>
<td></td>
<td>SA5680</td>
</tr>
<tr>
<td></td>
<td>668</td>
</tr>
<tr>
<td></td>
<td>JA1081</td>
</tr>
<tr>
<td></td>
<td>777</td>
</tr>
<tr>
<td></td>
<td>DE1911</td>
</tr>
<tr>
<td></td>
<td>908</td>
</tr>
<tr>
<td></td>
<td>FR2081</td>
</tr>
</tbody>
</table>

The innerjoin Operator

The `innerjoin` operator transfers records from both input data sets whose key fields contain equal values to the output data set. Records whose key fields do not contain equal values are dropped.

Syntax and Options

The syntax for the `innerjoin` operator is:

```
innerjoin
- key field_name[ -cs | -ci ] [ -param params ]
[ - key field_name[ -cs | -ci ] [ -param params ] ... ]
[ : collation_sequence locale | collation_file_pathname | OFF ]
```

There is one required option, `-key`. You can specify it multiple times. Only top-level, non-vector fields can be keys.
Table 124 innerjoin Operator option

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-key</td>
<td>-key field_name[·cs</td>
</tr>
<tr>
<td>-collation_sequence</td>
<td>-collation_sequence locale</td>
</tr>
</tbody>
</table>
Example

In this example, the **innerjoin** operation is performed on two input data sets using the `price` field as the key field. Equal values in the `price` field of the left and right data sets are shown with double underscores.

Here are the results of the **innerjoin** operation:

<table>
<thead>
<tr>
<th>left data set</th>
<th>right data set</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>status</strong></td>
<td><strong>field</strong></td>
</tr>
<tr>
<td>Sold</td>
<td>125</td>
</tr>
<tr>
<td>Sold</td>
<td>213</td>
</tr>
<tr>
<td>Offered</td>
<td>378</td>
</tr>
<tr>
<td>Pending</td>
<td>575</td>
</tr>
<tr>
<td>Pending</td>
<td>649</td>
</tr>
<tr>
<td>Offered</td>
<td>777</td>
</tr>
<tr>
<td>Offered</td>
<td>908</td>
</tr>
<tr>
<td>Pending</td>
<td>908</td>
</tr>
</tbody>
</table>

The osh syntax for the above example is:

```
$ osh "... innerjoin -key price ..."
```

The leftouterjoin Operator

The **leftouterjoin** operator transfers all values from the left data set and transfers values from the right data set only where key fields match. The operator drops the key field from the right data set. Otherwise, the operator writes default values.
Syntax and Options

The syntax for the `leftouterjoin` operator is:

```
leftouterjoin
  -key field_name[·cs | ·ci] [·param params]
  [·key field_name[·cs | ·ci] [·param params] ...]
  [·collation_sequence locale|collation_file_pathname| OFF]
```

There is one required option, `-key`. You can specify it multiple times. Only top-level, non-vector fields can be keys.

### Table 125  leftouterjoin Operator Option

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-key</code></td>
<td><code>-key field_name[·ci or·cs] [·param params]</code></td>
</tr>
<tr>
<td></td>
<td>Specify the name of the key field or fields.</td>
</tr>
<tr>
<td></td>
<td>You can specify multiple keys. For each one, specify the <code>-key</code> option and supply the key's name.</td>
</tr>
<tr>
<td></td>
<td>By default, Orchestrate interprets the value of key fields in a case-sensitive manner. Specify <code>-ci</code> to override this default. Do so for each key you choose, for example:</td>
</tr>
<tr>
<td></td>
<td><code>-key A ·ci ·key B ·ci</code></td>
</tr>
<tr>
<td><code>-collation_sequence</code></td>
<td>`-collation_sequence locale</td>
</tr>
<tr>
<td></td>
<td>This option determines how your string data is sorted. You can:</td>
</tr>
<tr>
<td></td>
<td>• Specify a predefined IBM ICU locale</td>
</tr>
<tr>
<td></td>
<td>• Write your own collation sequence using ICU syntax, and supply its collation_filepathname</td>
</tr>
<tr>
<td></td>
<td>• Specify <code>OFF</code> so that string comparisons are made using Unicode code-point value order, independent of any locale or custom sequence.</td>
</tr>
<tr>
<td></td>
<td>By default, Orchestrate sorts strings using byte-wise comparisons.</td>
</tr>
<tr>
<td></td>
<td>For more information, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:</td>
</tr>
</tbody>
</table>
Example

In this example, the `leftouterjoin` operation is performed on two data sets using the `price` field as the key field. Equal values in the `price` field of each data set are shown with double underscores.

Here are the input data sets:

<table>
<thead>
<tr>
<th>left data set</th>
<th>right data set</th>
</tr>
</thead>
<tbody>
<tr>
<td>status Field</td>
<td>price Field</td>
</tr>
<tr>
<td>Sold</td>
<td>125</td>
</tr>
<tr>
<td>Sold</td>
<td>213</td>
</tr>
<tr>
<td>Offered</td>
<td>378</td>
</tr>
<tr>
<td>Pending</td>
<td>575</td>
</tr>
<tr>
<td>Pending</td>
<td>649</td>
</tr>
<tr>
<td>Offered</td>
<td>777</td>
</tr>
<tr>
<td>Offered</td>
<td>908</td>
</tr>
<tr>
<td>Pending</td>
<td>908</td>
</tr>
</tbody>
</table>

Here are the results of the `leftouterjoin` operation on the left and right data sets.

<table>
<thead>
<tr>
<th>status Field</th>
<th>price Field</th>
<th>id Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sold</td>
<td>125</td>
<td>BR9658</td>
</tr>
<tr>
<td>Sold</td>
<td>213</td>
<td></td>
</tr>
<tr>
<td>Offered</td>
<td>378</td>
<td></td>
</tr>
<tr>
<td>Pending</td>
<td>575</td>
<td></td>
</tr>
<tr>
<td>Pending</td>
<td>649</td>
<td></td>
</tr>
<tr>
<td>Offered</td>
<td>777</td>
<td>JA1081</td>
</tr>
<tr>
<td>Offered</td>
<td>908</td>
<td>DE1911</td>
</tr>
<tr>
<td>Offered</td>
<td>908</td>
<td>FR2081</td>
</tr>
<tr>
<td>Pending</td>
<td>908</td>
<td>DE1911</td>
</tr>
<tr>
<td>Pending</td>
<td>908</td>
<td>FR2081</td>
</tr>
</tbody>
</table>

Here is the syntax for the example shown above in an `osh` command:

```bash
$ osh "... leftouterjoin -key price ..."
```
The rightouterjoin Operator

The **rightouterjoin** operator transfers all values from the right data set and transfers values from the left data set only where key fields match. The operator drops the key field from the left data set. Otherwise, the operator writes default values.

**Syntax and Options**

The syntax for the **rightouterjoin** operator is shown below:

```
rightouterjoin
  · key field_name [· cs | · ci] [· param params]
  [· key field_name [· cs | · ci] [· param params] ...]
  [· collation_sequence locale| collation_file_pathname| OFF]
```

There is one required option, **-key**. You can specify it multiple times. Only top-level, non-vector fields can be keys.

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>-key</strong></td>
<td>· key field_name [· ci or · cs] [· param params]</td>
</tr>
<tr>
<td></td>
<td>Specify the name of the key field or fields.</td>
</tr>
<tr>
<td></td>
<td>You can specify multiple keys. For each one, specify the <strong>-key</strong> option and supply the key’s name.</td>
</tr>
<tr>
<td></td>
<td>By default, Orchestrate interprets the value of key fields in a case-sensitive manner. Specify <strong>-ci</strong> to override this default. Do so for each key you choose, for example:</td>
</tr>
<tr>
<td></td>
<td>· key A · ci · key B · ci</td>
</tr>
<tr>
<td></td>
<td>The <strong>param</strong> suboption allows you to specify extra parameters for a field. Specify parameters using property=value pairs separated by commas.</td>
</tr>
</tbody>
</table>
Table 126  rightouterjoin Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-collation_sequence</td>
<td>-collation_sequence locale</td>
</tr>
</tbody>
</table>

This option determines how your string data is sorted. You can:

- Specify a predefined IBM ICU locale
- Write your own collation sequence using ICU syntax, and supply its collation_file_pathname
- Specify OFF so that string comparisons are made using Unicode code-point value order, independent of any locale or custom sequence.

By default, Orchestrate sorts strings using byte-wise comparisons.

For more information, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:


Example

In this example, the rightouterjoin operation is performed on two data sets using the price field as the key field. Equal values in the price field of each data set are shown with double underscores.
Here are the input data sets:

<table>
<thead>
<tr>
<th>left data set</th>
<th>price Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>status Field</td>
<td>price Field</td>
</tr>
<tr>
<td>Sold</td>
<td>125</td>
</tr>
<tr>
<td>Sold</td>
<td>213</td>
</tr>
<tr>
<td>Offered</td>
<td>378</td>
</tr>
<tr>
<td>Pending</td>
<td>575</td>
</tr>
<tr>
<td>Pending</td>
<td>649</td>
</tr>
<tr>
<td>Offered</td>
<td>777</td>
</tr>
<tr>
<td>Offered</td>
<td>908</td>
</tr>
<tr>
<td>Pending</td>
<td>908</td>
</tr>
</tbody>
</table>

| right data set | price Field | id Field |
|----------------|-------------|
| price Field    | id Field    |
| 113            | NI6325      |
| 125            | BR9658      |
| 285            | CZ2538      |
| 628            | RU5713      |
| 668            | SA5680      |
| 777            | JA1081      |
| 908            | DE1911      |
| 908            | FR2081      |

Here are the results of the rightouterjoin operation on the left and right data sets.

| status Field | price Field | id Field |
|--------------|-------------|
|              | 113         | NI6325   |
| Sold         | 125         | BR9658   |
|              | 285         | CZ2538   |
|              | 628         | RU5713   |
|              | 668         | SA5680   |
| Offered      | 777         | JA1081   |
| Offered      | 908         | DE1911   |
| Offered      | 908         | FR2081   |
| Pending      | 908         | DE1911   |
| Pending      | 908         | FR2081   |

Here is the syntax for the example shown above in an osh command:

```
$ osh "... rightouterjoin -key price ..."
```

The fullouterjoin Operator

The fullouterjoin operator transfers records whose key fields are equal in both input data sets to the output data set. It also transfers records whose key fields contain unequal values from both input data sets to the output data set.
The output data set:

- Contains all input records, except where records match; in this case it contains the cross-product of each set of records with an equal key.
- Contains all input fields.
- Renames identical field names of the input data sets as follows: \texttt{leftRec\_field} (left data set) and \texttt{rightRec\_field} (right data set), where field is the field name.
- Supplies default values to the output data set, where values are not equal.

**Syntax and Options**

The syntax for the \texttt{fullouterjoin} operator is shown below:

```
fullouterjoin
 -key field\_name [-cs | -ci] [-param params]
 [-key field\_name [-cs | -ci] [-param params] ...]
 [-collation\_sequence locale|collation\_file\_pathname| OFF]
```

There is one required option, \texttt{-key}. You can specify it multiple times. Only top-level, non-vector fields can be keys.
Example

In this example, the `rightouterjoin` operation is performed using the `price` field as the key field. Equal values in the `price` field of each data set are shown with double underscores.
Here are the input data sets:

**left data set**

<table>
<thead>
<tr>
<th>status Field</th>
<th>price Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sold</td>
<td>125</td>
</tr>
<tr>
<td>Sold</td>
<td>213</td>
</tr>
<tr>
<td>Offered</td>
<td>378</td>
</tr>
<tr>
<td>Pending</td>
<td>575</td>
</tr>
<tr>
<td>Pending</td>
<td>649</td>
</tr>
<tr>
<td>Offered</td>
<td>777</td>
</tr>
<tr>
<td>Offered</td>
<td>908</td>
</tr>
<tr>
<td>Pending</td>
<td>908</td>
</tr>
</tbody>
</table>

**right data set**

<table>
<thead>
<tr>
<th>price Field</th>
<th>id Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>113</td>
<td>NI6325</td>
</tr>
<tr>
<td>125</td>
<td>BR9658</td>
</tr>
<tr>
<td>285</td>
<td>CZ2538</td>
</tr>
<tr>
<td>628</td>
<td>RU5713</td>
</tr>
<tr>
<td>668</td>
<td>SA5680</td>
</tr>
<tr>
<td>777</td>
<td>JA1081</td>
</tr>
<tr>
<td>908</td>
<td>DE1911</td>
</tr>
<tr>
<td>908</td>
<td>FR2081</td>
</tr>
</tbody>
</table>

Here are the results of the **fullouterjoin** operation on the left and right data sets.

<table>
<thead>
<tr>
<th>status Field</th>
<th>leftRec_Price Field</th>
<th>rightRec_price Field</th>
<th>id Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sold</td>
<td></td>
<td></td>
<td>113 NI6325</td>
</tr>
<tr>
<td>Sold</td>
<td>125</td>
<td>125</td>
<td>BR9658</td>
</tr>
<tr>
<td>Sold</td>
<td>213</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sold</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offered</td>
<td>378</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pending</td>
<td>575</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pending</td>
<td>649</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offered</td>
<td>777</td>
<td>777</td>
<td>JA1081</td>
</tr>
<tr>
<td>Offered</td>
<td>908</td>
<td>908</td>
<td>DE1911</td>
</tr>
<tr>
<td>Offered</td>
<td>908</td>
<td>908</td>
<td>FR2081</td>
</tr>
<tr>
<td>Pending</td>
<td>908</td>
<td>908</td>
<td></td>
</tr>
<tr>
<td>Pending</td>
<td>908</td>
<td>908</td>
<td></td>
</tr>
</tbody>
</table>

The syntax for the example shown above in an **osh** command is:

```
$ osh "... fullouterjoin -key price ..."
```
The Collection Library

Describes the order in which the three collectors read your records.

Overview of the Collectors 29 1
The ordered Collector 29 2
Ordered Collecting 29 2
Properties 29 3
Syntax 29 4
The roundrobin Collector 29 4
Round Robin Collecting 29 4
Properties 29 5
Syntax 29 5
The sortmerge Collector 29 5
Understanding the sortmerge Collector 29 5
Data Flow Diagram 29 6
Specifying Collecting Keys 29 6
Properties 29 8
Syntax and Options 29 8

Overview of the Collectors

This chapter of the Orchestr8te 7.0 Operators Reference describes how to use the collectors, sometimes called collection operators. To understand the information in this chapter you should be familiar with the collection concepts defined in the Orchestr8te 7.0 User Guide.
The collection library contains three collectors:

- The **ordered** collector. Reads all records from the first partition, then all records from the second partition, and so on. This collection method preserves the sorted order of an input data set that has been totally sorted. In a totally sorted data set, the records in each partition of the data set, as well as the partitions themselves, are ordered. This collector is described on page 2.

- The **roundrobin** collector. Reads a record from the first input partition, then from the second partition, and so on. After reaching the last partition, the collector starts over. After reaching the final record in any partition, the collector skips that partition. This collector is described on page 4.

- The **sortmerge** collector. Reads records in an order based on one or more fields of the record. The fields used to define record order are called **collecting keys**. This chapter is described on page 5.

## The ordered Collector

Orchestrate allows you to use collectors to explicitly set the collection method of a data set. This chapter describes how to use the **ordered** collector.

### Ordered Collecting

In **ordered** collection, the collector reads all records from the first input partition, then all records from the second input partition, and so on, to create the output data set. This collection method preserves the record order of each partition of the input data set and may be useful as a preprocessing action before exporting a sorted data set to a single data file.

When you use the **ordered** collector, the output data set of the collector must be one of these:

- A virtual data set connected to the input of a sequential collector using the **any** collection method. A virtual data set output by a collector overrides the collection method of a collector using the **any** collection method.

- A persistent data set. If the data set exists, it must contain only a single partition unless a full overwrite of the data set is being performed.
For example, Figure 53 shows an operator using the any collection method, preceded by an ordered collector.

![Diagram of ordered collector](image)

Figure 53  Using the ordered Collector

The any collection method of 0 p is overridden by inserting an ordered collector into the step.

The ordered collector takes a single partitioned data set as input and collects the input data set to create a single sequential output data set with one partition.

**Properties**

Table 128  ordered Collector Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec to outRec without modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>sequential</td>
</tr>
<tr>
<td>Collection method</td>
<td>ordered</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>
**Syntax**

The syntax for the `ordered` collector in an `osh` command is:

```bash
$ osh ' ... | ordered | ... '
```

The `ordered` collector has no options.

**The roundrobin Collector**

Orchestrate allows you to use collectors to explicitly set the collection method of a data set. This chapter describes how to use the `roundrobin` collector.

**Round Robin Collecting**

In `roundrobin` collection, the collector reads one record from the first input partition, then one from the second partition, and so on. After reaching the last partition, the collector starts over. After the final record in any partition has been read, that partition is skipped.

When you use the `roundrobin` collector, the output data set of the collector must be either:

- A virtual data set connected to the input of a sequential operator using the `any` collection method. A virtual data set output by a collector overrides the collection method of an operator using the `any` collection method.

- A persistent data set. If the data set exists, it must contain only a single partition unless a full overwrite is being done.

For example, Figure 54 shows an operator using the `any` collection method, preceded by a `roundrobin` collector.
The any collection method of Op is overridden by inserting a roundrobin collector into the step.

The roundrobin collector takes a single data set as input and collects the input data set to create a single sequential output data set with one partition.

Properties

Table 129  roundrobin Collector Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec to outRec without modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>sequential</td>
</tr>
<tr>
<td>Collection method</td>
<td>round robin</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

Syntax

The syntax for the roundrobin collector in an osh command is:

osh " ... | roundrobin | ... "

The roundrobin collector has no options.

The sortmerge Collector

The sortmerge collector reads records in an order based on one or more fields of the record. The fields used to define record order are called collecting keys. You use the sortmerge collector to implement the sorted merge collection method. This section describes the sortmerge collector.

Understanding the sortmerge Collector

The sortmerge collector implements the sorted merge collection method for an input data set. The sortmerge collector determines the order of input records by
examining one or more collecting key fields in the current record of each partition of an input data set. Records are collected base on the sort order of these keys.

Typically, you use the sortmerge collector with a partition-sorted data set as created by the psort or tsort operator. In this case, you specify as the collecting key fields those fields you specified as sorting key fields to the sorting operator.

When you use the sortmerge collector, the output data set of the collector must be one of the following:

- A virtual data set connected to the input of a sequential operator using the any collection method. A virtual data set output by a collector overrides the collection method of an operator using the any collection method.
- A persistent data set. If the data set exists, it must contain only a single partition unless a full overwrite is being done.

The sortmerge collector takes a partitioned data set as input and collects the input data set to create a single sequential output data set with one partition.

Data Flow Diagram

![Data Flow Diagram](image)

Specifying Collecting Keys

Collecting keys specify the criteria for determining the order of records read by the sortmerge collector. The sortmerge collector allows you to set one primary collecting key and multiple secondary collecting keys. The sortmerge collector first examines the primary collecting key in each input record. For multiple records with the same primary key value, the sortmerge collector then examines any secondary keys to determine the order of records input by the collector.

For example, Figure 55 shows the current record in each of three partitions of an input data set to the collector:
In this example, the records consist of three fields. The first-name and last-name fields are strings, and the age field is an integer.

Figure 56 shows the order of the three records read by the `sortmerge` collector, based on different combinations of collecting keys.

You must define a single primary collecting key for the `sortmerge` collector, and you may define as many secondary keys as are required by your application. Note, however, that each record field can be used only once as a collecting key. Therefore, the total number of primary and secondary collecting keys must be less than or equal to the total number of fields in the record.

The data type of a collecting key can be any Orchestrate type except `raw`, `subrec`, `tagged`, or `vector`. Specifying a collecting key of these types causes the `sortmerge` collector to issue an error and abort execution.

By default, the `sortmerge` collector uses ascending sort order and case-sensitive comparisons. Ascending order means that records with smaller values for a collecting field are processed before records with larger values. You also can specify descending sorting order, so records with larger values are processed first.

With a case-sensitive algorithm, records with uppercase strings are processed before records with lowercase strings. You can override this default to perform case-insensitive comparisons of string fields.
Properties

Table 130  sortmerge Collector Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec to outRec without modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>sequential</td>
</tr>
<tr>
<td>Collection method</td>
<td>sortmerge</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

Syntax and Options

The sortmerge collector in osh uses the following syntax:

```
sortmerge
  -key field_name [-ci | -cs] [-asc | -desc] [-nulls first | last]
  [-ebcdic] [-param params]
```

You must specify at least one -key field to the sortmerge collector.
### Table 131  sortmerge Collection Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-key</code></td>
<td>`-key field_name [-ci</td>
</tr>
<tr>
<td></td>
<td>`[ -nulls first</td>
</tr>
<tr>
<td></td>
<td><code>[ -param params]</code></td>
</tr>
</tbody>
</table>

Specifies that `field_name` is a collecting key field for the `sortmerge` collector. You can designate multiple key fields. The first key you list is the primary key.

`field_name` must be a field of the data set using the collector or a field created using an input field adapter on the data set.

By default, the `sortmerge` collector does case-sensitive comparisons. This means that records with uppercase strings are processed before records with lowercase strings. You can optionally override this default to perform case-insensitive collecting by using the `-ci` option after the field name.

By default, the `sortmerge` collector uses ascending order, so that records with smaller values for `field_name` are processed before records with larger values. You can optionally specify descending sorting order, so that records with larger values are processed first, by using `-desc` after the field name.

By default, the `sortmerge` collector sorts fields with null keys first. If you wish to have them sorted last, specify `-nulls last` after the field name.

By default, data is represented in the ASCII character set. To represent data in the EBCDIC character set, specify the `-ebcdic` option.

The `-param` suboption allows you to specify extra parameters for a field. Specify parameters using `property=value` pairs separated by commas.

<table>
<thead>
<tr>
<th><code>-collation_sequence</code></th>
<th>`-collation_sequence locale</th>
<th>collation_file_pathname</th>
<th>OFF</th>
</tr>
</thead>
</table>

This option determines how your string data is sorted. You can:

- Specify a predefined IBM ICU locale
- Write your own collation sequence using ICU syntax, and supply its `collation_file_pathname`
- Specify `OFF` so that string comparisons are made using Unicode code-point value order, independent of any locale or custom sequence.

By default, Orchestrate sorts strings using byte-wise comparisons.

For more information, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:

http://oss.software.ibm.com/icu/userguide/Collate_Intro.htm
The Restructure Library

Describes the operators that modify the record schemas of input data sets and change the level of fields within the records.

Overview of the Restructure Operators  30 3
The aggtorec Operator  30 3
   Output Formats  30 4
   Properties  30 4
   Syntax and Options  30 5
   Example 1: The aggtorec Operator without the toplevelkeys option  30 6
   Example 2: The aggtorec Operator with Multiple Key Options  30 7
   Example 3: The aggtorec Operator with the toplevelkeys Option  30 8
The field_export Operator  30 9
   Data Flow Diagram  30 10
   Properties  30 10
   Syntax and Options  30 10
   Example  30 11
The field_import Operator  30 13
   Data Flow Diagram  30 13
   Properties  30 14
   Syntax and Options  30 14
   Example  30 15
The makesubrec Operator  30 17
   Data Flow Diagram  30 17
   Properties  30 18
   Transfer Behavior  30 18
   Subrecord Length  30 19
   Syntax and Options  30 19
   Example  30 20
The makevect Operator  30 21
   Data Flow Diagram  30 22
   Properties  30 22
Transfer Behavior  30 22
Non-Consecutive Fields  30 23
Syntax and Option  30 23
Example 1: The makevect Operator  30 24
Example 2: The makevect Operator with Missing Input Fields  30 24

The promotesubrec Operator  30 25
Data Flow Diagram  30 26
Properties  30 26
Syntax and Option  30 26
Example  30 27

The splitsubrec Operator  30 27
Data Flow Diagram  30 28
Properties  30 28
Syntax and Option  30 29
Example  30 29

The splitvect Operator  30 30
Data Flow Diagram  30 31
Properties  30 31
Syntax and Option  30 32
Example  30 32

The tagbatch Operator  30 33
Tagged Fields and Operator Limitations  30 33
Operator Action and Transfer Behavior  30 34
Data Flow Diagram  30 35
Properties  30 35
Added, Missing, and Duplicate Fields  30 35
Input Data Set Requirements  30 35
Syntax and Options  30 36
Example 1: The tagbatch Operator, Simple Flattening of Tag Cases  30 38
Example 2: The tagbatch Operator, Missing and Duplicate Cases  30 40
Example 3: The tagbatch Operator with Multiple Keys  30 41

The tagswitch Operator  30 43
Data Flow Diagram  30 43
Properties  30 43
Input and Output Interface Schemas  30 44
The case Option  30 44
Using the Operator  30 45
Syntax and Options  30 45
Example 1: The tagswitch Operator, Default Behavior  30 46
Example 2: The tagswitch Operator, One Case Chosen  30 48
Overview of the Restructure Operators

Orchestrate features ten operators that modify the record schema of the input data set and the level of fields within records.

Table 132  Restructure Operators Listing

<table>
<thead>
<tr>
<th>Operator</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggtorec</td>
<td>groups records that have the same key-field values into an output record.</td>
</tr>
<tr>
<td>field_export</td>
<td>combines the input fields specified in your output schema into a string- or raw-valued field.</td>
</tr>
<tr>
<td>field_import</td>
<td>exports an input string or raw field to the output fields specified in your import schema.</td>
</tr>
<tr>
<td>makesubrec</td>
<td>combines specified vector fields into a vector of subrecords.</td>
</tr>
<tr>
<td>makevect</td>
<td>combines specified fields into a vector of fields of the same type.</td>
</tr>
<tr>
<td>promotesubrec</td>
<td>converts input subrecord fields to output top-level fields.</td>
</tr>
<tr>
<td>splitsubrec</td>
<td>separates input subrecords into sets of output top-level vector fields.</td>
</tr>
<tr>
<td>splitvect</td>
<td>promotes the elements of a fixed-length vector to a set of similarly-named top-level fields.</td>
</tr>
<tr>
<td>tagbatch</td>
<td>converts tagged fields into output records whose schema supports all the possible fields of the tag cases.</td>
</tr>
<tr>
<td>tagswitch</td>
<td>The contents of tagged aggregates are converted to Orchestrate-compatible records.</td>
</tr>
</tbody>
</table>

The aggtorec Operator

The aggtorec operator takes input records that have been sorted on one or more key fields and groups same-valued key-field records into top-level output records containing subrecords. The number of top-level output records corresponds to the number of unique key-field values in the input data set.

It is important to first sort the data set on the same key fields that you use for the aggtorec operator.
The `promotesubrec` operator often performs the inverse operation. It is documented in “The promotesubrec Operator” on page 30-25.

Output Formats

The format of a top-level output record depends on whether you specify the `-toplevel` option. The following simplified example demonstrates the two output formats. In both cases, the `-key` option value is `field-b`, and the `-subrecname` option value is `sub`.

This is the input data set:

```
field-a:3 field-b:1
field-a:4 field-b:2
field-a:5 field-b:2
```

The two forms of output are:

- When the `-toplevel` option is not specified, each output top-level record consists entirely of subrecords, where each subrecord has exactly the same fields as its corresponding input record, and all subrecords in a top-level output record have the same values in their key fields.

  ```
  sub: [0: (sub. field-a:3 sub.field-b:1)]
  sub: [0: (sub. field-a:4 sub. field-b:2)
  1: (sub. field-a:5 sub. field-b:2)]
  ```

- When the `-toplevel` option is specified, the input key field or fields remain top-level fields in the output top-level record, and the non-key fields are placed in a subrecord.

  ```
  field-b:1 sub: [0: (sub. field-a:3)]
  field-b:2 sub: [0: (sub. field-a:4)
  1: (sub. field-a:5)]
  ```

Properties

Table 133  **aggtorec Operator Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema:</td>
<td><code>inRec:*</code></td>
</tr>
<tr>
<td>Output interface schema:</td>
<td>See the section “Output Formats” on page 30-4.</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>See the section “Output Formats” on page 30-4.</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
</tbody>
</table>
Syntax and Options

The syntax of the `aggtorec` operator is as follows. You must specify at least one `-key` option and the `subrecname` option.

```
aggtorec
   -key key_field [-ci|-cs] [-param params]
   [ -key key_field [-ci|-cs] [-param params] ... ]
   [ -collation_sequence locale|collation_file_pathname|OFF ]
   -subrecname subrecname
   [ -toplevelkeys ]
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-key</code></td>
<td>`-key key_field [-ci</td>
</tr>
<tr>
<td></td>
<td>This option is required. Specify one or more fields.</td>
</tr>
<tr>
<td></td>
<td>If you do not specify <code>-toplevelkeys</code>, all records whose key fields contain identical values are gathered into the same record as subrecords. Each field becomes the element of a subrecord.</td>
</tr>
<tr>
<td></td>
<td>If you specify the <code>-toplevelkeys</code> option, the key field appears as a top-level field in the output record. All non-key fields belonging to input records with that key field appear as elements of a subrecord in that key field’s output record.</td>
</tr>
<tr>
<td></td>
<td>You can specify multiple keys. For each one, specify the <code>-key</code> option and supply the key’s name.</td>
</tr>
<tr>
<td></td>
<td>By default, Orchestrate interprets the value of key fields in a case-sensitive manner if the values are strings. Specify <code>-ci</code> to override this default. Do so for each key you choose. For example:</td>
</tr>
<tr>
<td></td>
<td><code>-key A -ci -key B -ci</code></td>
</tr>
<tr>
<td></td>
<td>The <code>-param</code> suboption allows you to specify extra parameters for a field. Specify parameters using <code>property=value</code> pairs separated by commas.</td>
</tr>
</tbody>
</table>
Example 1: The aggtorec Operator without the toplevelkeys option

This example shows an input data set and its processing by the aggtorec operator. In the example, the operation runs sequentially, and:

- The key is d, and the input data set has been sorted on d.
- The subrecord name is sub.
- The -toplevelkeys option has not been specified.

The osh command is:

```
$ osh "... aggtorec -key d -subrecname sub ..."
```

The input data set is:

```
a:1 b:00:11:01 c:1960-01-02 d:A
a:3 b:08:45:54 c:1946-09-15 d:A
a:1 b:12:59:01 c:1955-12-22 d:B
a:2 b:07:33:04 c:1950-03-10 d:B
a:2 b:12:00:00 c:1967-02-06 d:B
a:2 b:07:37:04 c:1950-03-10 d:B
a:3 b:07:56:03 c:1977-04-14 d:B
a:3 b:09:58:02 c:1960-05-18 d:B
```
In the output, records with an identical value in field \(d\) have been gathered as subrecords in the same record, as shown here:

```
sub: [0: (sub.a:1 sub.b:00:11:01 sub.c:1960-01-02 sub.d:A)
  1: (sub.a:3 sub.b:08:45:54 sub.c:1946-09-15 sub.d:A)]
sub: [0: (sub.a:1 sub.b:12:59:01 sub.c:1955-12-22 sub.d:B)
  1: (sub.a:2 sub.b:07:33:04 sub.c:1950-03-10 sub.d:B)
  2: (sub.a:2 sub.b:12:00:00 sub.c:1967-02-06 sub.d:B)
  3: (sub.a:2 sub.b:07:37:04 sub.c:1950-03-10 sub.d:B)
  4: (sub.a:3 sub.b:07:56:03 sub.c:1977-04-14 sub.d:B)
  5: (sub.a:3 sub.b:09:58:02 sub.c:1960-05-18 sub.d:B)]
sub: [0: (sub.a:1 sub.b:11:43:02 sub.c:1980-06-03 sub.d:C)
  1: (sub.a:2 sub.b:01:30:01 sub.c:1985-07-07 sub.d:C)
  2: (sub.a:2 sub.b:11:30:01 sub.c:1985-07-07 sub.d:C)
  3: (sub.a:3 sub.b:10:28:02 sub.c:1992-11-23 sub.d:C)
  4: (sub.a:3 sub.b:12:27:00 sub.c:1929-08-11 sub.d:C)
  5: (sub.a:3 sub.b:06:33:03 sub.c:1999-10-19 sub.d:C)
  6: (sub.a:3 sub.b:11:18:22 sub.c:1992-11-23 sub.d:C)]
```

Example 2: The aggtorec Operator with Multiple Key Options

This example takes the same input data set as in “Example 1: The aggtorec Operator without the toplevelkeys option” on page 30-6. A gain, the `toplevelkeys` option is not included; however, there are two key fields, \(a\) and \(d\), instead of a single field, \(d\).

The operation runs sequentially. The example outputs multiple records with \(a=1\) and \(a=2\) because the data is not sorted on \(a\).

Here is the `osh` command:

```
$ osh "... aggtorec -key d -key a -subrecname sub ..."
```

The input data set is:

```
a:1 b:00:11:01 c:1960-01-02 d:A
a:3 b:08:45:54 c:1946-09-15 d:A
a:1 b:12:59:01 c:1955-12-22 d:B
a:2 b:07:33:04 c:1950-03-10 d:B
a:2 b:12:00:00 c:1967-02-06 d:B
a:2 b:07:37:04 c:1950-03-10 d:B
a:3 b:10:28:02 c:1992-11-23 d:C
a:3 b:12:27:00 c:1929-08-11 d:C
a:3 b:06:33:03 c:1999-10-19 d:C
a:3 b:11:18:22 c:1992-11-23 d:C
```
In the output, records with identical values in both field d and field a have been gathered as subrecords in the same record:

Example 3: The aggtorec Operator with the toplevelkeys Option

This example shows the same input record as in the previous two examples. The specifications for this example are:

- The -toplevelkeys option is specified.
- The keys are a and d.
- The input schema is as follows:
  a:uint8; b:time; c:date; d:string[1];
- The subrecord name is sub.

Here is the osh command:

```
$ osh "... aggtorec -key d -key a -subrecname sub -toplevelkeys ..."
```

In the output:

- Fields a and d are written as the top-level fields of a record only once when they have the same value, although they can occur multiple times in the input data set.
- Fields b and c are written as subrecords of the same record in which the key fields contain identical values.

Here is a small example of input records:
The field_export Operator

The \texttt{field_export} operator exports an input field or fields to a string or raw valued output field, and transfers the remaining input fields to output unchanged. The fields incorporated in the string or raw output field are dropped.

You supply an export schema to guide the creation of the string or raw field using the \texttt{-schema} or \texttt{-schemafile} option.
The export schema syntax is:

```plaintext
record{text, delim='delimiter', final_delim='delimiter'}
  (field1:type;field2:type;...fieldn:type;)
```

The specified fields are combined into a single raw output field. You use the `-field` option to specify the name of the output field, and specify `-type string` to export to a string field.

You can optionally save rejected records in a separate reject data set. The default behavior is to continue processing and report a count of failures.

**Data Flow Diagram**

```
input data set

inRec:*  

field_export

r:string | raw and outRec:* with the exported fields dropped

output data set  reject data set
```

**Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1, and optionally a reject data set</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>r:string OR raw; outRec:* without the exported fields</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec -&gt; outRec (exported fields are dropped)</td>
</tr>
</tbody>
</table>

**Syntax and Options**

The `-field` option is required. You must also specify either the `-schema` or `-schemafile` option. Option values you supply are shown in italic typeface. When
your value contains a space or a tab character, you must enclose the value in single quotes.

```plaintext
field_export
- field field_name
- schema schema_name | - schemaf ile file_name
[ - saveRejects]
[ -type string | raw]
```

### Table 136  **field_export Operator Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-field</td>
<td>- field field_name</td>
</tr>
<tr>
<td></td>
<td>Specifies the name of the string- or raw-type field to which the input field or fields are exported.</td>
</tr>
<tr>
<td>-saveRejects</td>
<td>- saveRejects</td>
</tr>
<tr>
<td></td>
<td>Specifies that the operator continues when it encounters a reject record and writes the record to an output reject data set. The default action of the operator is to continue and report a count of the failures to the message stream. If you specify this suboption, you must attach a second output data set to the operator.</td>
</tr>
<tr>
<td>-schema</td>
<td>- schema schema_name</td>
</tr>
<tr>
<td></td>
<td>Specifies the name of the export schema. The schema specifies how input fields are packed into the exported string- or raw-type field.</td>
</tr>
<tr>
<td></td>
<td>You must specify either -schema or -schemaf ile.</td>
</tr>
<tr>
<td>-schemaf ile</td>
<td>- schemaf ile file_name</td>
</tr>
<tr>
<td></td>
<td>Specifies the name of a file containing the export schema. The schema specifies how input fields are packed into the exported string- or raw-type field.</td>
</tr>
<tr>
<td></td>
<td>You must specify either -schemaf ile or -schema.</td>
</tr>
<tr>
<td>-type</td>
<td>- type string</td>
</tr>
<tr>
<td></td>
<td>Specifies the data type of the output field; this is the type to which the operator converts input data. The operator converts the input data to raw-type data by default.</td>
</tr>
</tbody>
</table>

### Example

The **field_export** operator exports an input data set whose record schema is:

```plaintext
record(value:decimal[5,2];SN:int16;time:time)
```

to an output data set whose record schema is:
record(exported:string;time:time;)

The operator exports the first two input fields to a single string output field in which the two items are displayed as text separated by a comma. Here is the export schema that guides this operation:

\[ record\{text,delim='\,',final_delim=none\}\{SN:int16;value:decimal[5,2]\}\]

The osh commands for this example are:

\[
\text{\$ expsch='record\{text,delim='','\,final_delim=none\}\{SN:int16;value:decimal[5,2]\}' } \\
\text{\$ osh '... field_export -field exported -type string -schema $expsch ...'}
\]

Here is the input data set:

<table>
<thead>
<tr>
<th>value: 657.65</th>
<th>SN: 153</th>
<th>time: 00:08:36</th>
</tr>
</thead>
<tbody>
<tr>
<td>value: 652.16</td>
<td>SN: 13</td>
<td>time: 00:08:20</td>
</tr>
<tr>
<td>value: 306.65</td>
<td>SN: 391</td>
<td>time: 00:08:52</td>
</tr>
<tr>
<td>value: 292.65</td>
<td>SN: 299</td>
<td>time: 00:09:24</td>
</tr>
<tr>
<td>value: 365.65</td>
<td>SN: 493</td>
<td>time: 00:08:28</td>
</tr>
<tr>
<td>value: 449.46</td>
<td>SN: 580</td>
<td>time: 00:09:00</td>
</tr>
<tr>
<td>value: 271.79</td>
<td>SN: 303</td>
<td>time: 00:09:32</td>
</tr>
<tr>
<td>value: 098.15</td>
<td>SN: 216</td>
<td>time: 00:09:08</td>
</tr>
<tr>
<td>value: 404.54</td>
<td>SN: 678</td>
<td>time: 00:08:44</td>
</tr>
<tr>
<td>value: 379.31</td>
<td>SN: 103</td>
<td>time: 00:09:16</td>
</tr>
</tbody>
</table>

Here is the output data set:

<table>
<thead>
<tr>
<th>exported: 153, 657.65</th>
<th>time: 00:08:36</th>
</tr>
</thead>
<tbody>
<tr>
<td>exported: 13, 652.16</td>
<td>time: 00:08:20</td>
</tr>
<tr>
<td>exported: 391, 306.65</td>
<td>time: 00:08:52</td>
</tr>
<tr>
<td>exported: 299, 292.65</td>
<td>time: 00:09:24</td>
</tr>
<tr>
<td>exported: 493, 365.65</td>
<td>time: 00:08:28</td>
</tr>
<tr>
<td>exported: 580, 449.46</td>
<td>time: 00:09:00</td>
</tr>
<tr>
<td>exported: 303, 271.79</td>
<td>time: 00:09:32</td>
</tr>
<tr>
<td>exported: 216, 098.15</td>
<td>time: 00:09:08</td>
</tr>
</tbody>
</table>
Note that the operator has reversed the order of the value and SN fields and combined their contents. The time field is transferred to the output with no change, as in the following diagram:

The field_import Operator

The field_import operator exports an input string or raw field to the output fields specified in your import schema. The string or raw input field is dropped, and the other input fields are transferred to output without change.

You can optionally save rejected records in a separate reject data set. The default behavior is to continue processing and report a count of failures.

Data Flow Diagram
Properties

Table 137 field_import Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1, and optionally a reject data set</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>r:raw OR string;inRec:*</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>field1:type1;...fieldn:typen;outRec:* with the string or raw-type field dropped.</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>inRec -&gt; outRec (the string- or raw-type field is dropped)</td>
</tr>
</tbody>
</table>

Syntax and Options

The -field option is required. You must also specify either the -schema or -schemafile.

field_import
  -field field_name
  -schema schema_name | -schemafile file_name
  [-keepField]
  [-failRejects] | [-saveRejects]

Table 138 field_import Operator Properties

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-field</td>
<td>-field field_name</td>
</tr>
<tr>
<td></td>
<td>Specifies the name of the field containing the string or raw data to import.</td>
</tr>
<tr>
<td>-keepField</td>
<td>-keepField</td>
</tr>
<tr>
<td></td>
<td>Specifies that the operator continues when it encounters a reject record and writes the record to the output data set.</td>
</tr>
<tr>
<td>-failRejects</td>
<td>-failRejects</td>
</tr>
<tr>
<td></td>
<td>Specifies that the operator fails when it encounters a record whose import is rejected. The default action of the operator is to continue and report a count of the failures to the message stream. This option is mutually exclusive with -saveRejects.</td>
</tr>
</tbody>
</table>
Example

In this example, the **field_import** operator imports a 16-byte raw field to four 32-bit integer fields.

Here is the schema of the input data set:

```plaintext
record (rawfield[16]:raw; x:decimal[6,1];)
```

Here is the import schema that guides the import of the 16-byte raw field into four integer output fields. The schema also assures that the contents of the raw[16] field are interpreted as binary integers:

```plaintext
record {binary, delim=none}
 (a:int32;
  b:int32;
  c:int32;
  d:int32;
 )
```

Here is the schema of the output data set:

```plaintext
record (a:int32;
 b:int32;
 c:int32;
 d:int32;
 x:decimal[6,1];
)
```

Here are the **osh** commands for this example:

```bash
$ intSch=`record { binary, delim=none } 
 (a:int32;
```
Chapter 30  The Restructure Library

The field_import Operator

```c
b: int32;
c: int32;
d: int32;
}
```

```bash
$ osh "field_import -field raw -schema $intSch ...
```

Here is the input data set:

<table>
<thead>
<tr>
<th>rawfield</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>00087.2</td>
</tr>
<tr>
<td>04 04 04 04 04 04 04 04 04 04 04 04 04 04 04 04</td>
<td>00004.8</td>
</tr>
<tr>
<td>08 08 08 08 08 08 08 08 08 08 08 08 08 08 08 08</td>
<td>00042.7</td>
</tr>
<tr>
<td>01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01</td>
<td>00091.3</td>
</tr>
<tr>
<td>05 05 05 05 05 05 05 05 05 05 05 05 05 05 05 05</td>
<td>00075.9</td>
</tr>
<tr>
<td>09 09 09 09 09 09 09 09 09 09 09 09 09 09 09 09</td>
<td>00081.3</td>
</tr>
<tr>
<td>02 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02</td>
<td>00040.6</td>
</tr>
<tr>
<td>06 06 06 06 06 06 06 06 06 06 06 06 06 06 06 06</td>
<td>00051.5</td>
</tr>
<tr>
<td>03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03</td>
<td>00061.7</td>
</tr>
<tr>
<td>07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07</td>
<td>00015.2</td>
</tr>
</tbody>
</table>

Here is the output data set:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00087.2</td>
</tr>
<tr>
<td>67372036</td>
<td>67372036</td>
<td>67372036</td>
<td>67372036</td>
<td>00004.8</td>
</tr>
<tr>
<td>134744072</td>
<td>134744072</td>
<td>134744072</td>
<td>134744072</td>
<td>00042.7</td>
</tr>
<tr>
<td>16843009</td>
<td>16843009</td>
<td>16843009</td>
<td>16843009</td>
<td>00091.3</td>
</tr>
<tr>
<td>84215045</td>
<td>84215045</td>
<td>84215045</td>
<td>84215045</td>
<td>00075.9</td>
</tr>
<tr>
<td>151587081</td>
<td>151587081</td>
<td>151587081</td>
<td>151587081</td>
<td>00081.3</td>
</tr>
<tr>
<td>33686018</td>
<td>33686018</td>
<td>33686018</td>
<td>33686018</td>
<td>00040.6</td>
</tr>
<tr>
<td>101058054</td>
<td>101058054</td>
<td>101058054</td>
<td>101058054</td>
<td>00051.5</td>
</tr>
<tr>
<td>50529027</td>
<td>50529027</td>
<td>50529027</td>
<td>50529027</td>
<td>00061.7</td>
</tr>
<tr>
<td>117901063</td>
<td>117901063</td>
<td>117901063</td>
<td>117901063</td>
<td>00015.2</td>
</tr>
</tbody>
</table>
Note that the operator has imported four bytes of binary data in the input raw field as one decimal value in the output. The input field x (decimal[6,1]) is transferred to the output with no change, as in the following diagram:

```
raw: 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 x:0001
```

In the example above each byte contributes the value shown for a total of 117901963:

```
<table>
<thead>
<tr>
<th>byte</th>
<th>07 * 224 +</th>
<th>117440512</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>07 * 216 +</td>
<td>458752</td>
</tr>
<tr>
<td>byte</td>
<td>07 * 28 +</td>
<td>1792</td>
</tr>
<tr>
<td>byte</td>
<td>07 * 20 +</td>
<td>7</td>
</tr>
</tbody>
</table>

Total: 117901963
```

The makesubrec Operator

The `makesubrec` operator combines specified vectors in an input data set into a vector of subrecords whose fields have the names and data types of the original vectors. You specify the vector fields to be made into a vector of subrecords and the name of the new subrecord.

The `splitsubrec` operator performs the inverse operation. See “The splitsubrec Operator” on page 30-27.

Data Flow Diagram

The following figure is a data flow diagram of the `makesubrec` operator, where vtype denotes any valid Orchestrate data type that can figure in a vector, that is,
any data type except tagged and * (schema variable). Different subrecord elements can be of different vtypes.

```
input data set

inRec:*; a[]:vtype; b[]:vtype; ...n[]:vtype;
```

`makesubrec`

```
outRec:*; subrecname[]:subrec( a:vtype; b:vtype; ...n:vtype; )
```

output data set

**Properties**

Table 139  **makesubrec Operator Options**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*; a[]:vtype; b[]:vtype; ...n[]:vtype;</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*; subrecname[]:subrec( a:vtype; b:vtype; ...n:vtype; )</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>See “Transfer Behavior” on page 30-18.</td>
</tr>
</tbody>
</table>

**Transfer Behavior**

The input interface schema is as follows:

```
record ( inRec:*; a[]:vtype; b[]:vtype; b[]:vtype; ...n[]:vtype; )
```

The `inRec` schema variable captures all the input record’s fields except those that are combined into subrecords. In the interface, each field that is combined in a subrecord is a vector of indeterminate type.

The output interface schema is as follows:

```
record ( outRec:*;
  subrecname[]:subrec( a:vtype; b:vtype; ...n:vtype; )
)
```
**Subrecord Length**

The length of the subrecord vector created by this operator equals the length of the longest vector field from which it is created. If a variable-length vector field was used in subrecord creation, the subrecord vector is also of variable length.

Vectors that are smaller than the largest combined vector are padded with default values: NULL for nullable fields and the corresponding type-dependent value for non-nullable fields. For example, in the following record, the vector field \( a \) has five elements and the vector field \( b \) has four elements.

\[
a: [0:0 1:2 2:4 3:6 4:8] \\
\]

The \texttt{makesubrec} operator combines the fields into a subrecord named \texttt{sub} as follows:

\[
\begin{align*}
\text{sub}: & \quad 0: (\text{sub.a}: 0 \text{ sub.b}: 1960-08-01) \\
& \quad 1: (\text{sub.a}: 2 \text{ sub.b}: 1960-11-23) \\
& \quad 2: (\text{sub.a}: 4 \text{ sub.b}: 1962-06-25) \\
& \quad 3: (\text{sub.a}: 6 \text{ sub.b}: 1961-08-18) \\
& \quad 4: (\text{sub.a}: 8 \text{ sub.b}: 1-1-0001)
\end{align*}
\]

Subfield \( b \) of the subrecord's fifth element, shown in boldface type, contains the data type's default value.

When the \texttt{makesubrec} operator encounters mismatched vector lengths, it displays the following message:

\[
\text{When checking operator: Not all vectors are of the same length.} \\
\text{field\_nameA and field\_nameB differ in length. Some fields will contain} \\
\text{their default values or Null, if nullable}
\]

where \texttt{field\_nameA} and \texttt{field\_nameB} are the names of vector fields to be combined into a subrecord. You can suppress this message by means of the \texttt{-variable} option.

This operator treats scalar components as if they were a vector of length 1. This allows scalar and vector components to be combined.

**Syntax and Options**

The \texttt{-name} and \texttt{-subrecname} options are required.

The syntax for the \texttt{makesubrec} operator is as follows:

\[
\text{makesubrec} \quad \texttt{-name field\_name [-name field\_name...]} \\
\quad \texttt{-subrecname subrecname} \\
\quad [\texttt{-variable}]
\]
Example

The `makesubrec` operator combines fields `a` and `c` of the following input data set into a subrecord called `sub`. In the example, the operation runs sequentially.

Here are the input and output data set record schemas:

```
a[5]:uint8; b[4]:decimal[2,1]; c[4]:string[1]
b[4]:decimal[2,1]; sub [5](a:uint8; c:string[1];)
```

Here is the `osh` command:

```
$ osh "... makesubrec -name a -name c -subrecname sub ..."
```

Here is the input data set:

```
a:[0:10 1:0 2:2 3:4 4:6] b:[0:-2.4 1:-0.7 2:0.0 3:0.0]  
c:[0:A 1:B 2:C 3:D]
a:[0:6 1:8 2:10 3:0 4:2] b:[0:0.0 1:0.0 2:0.0 3:-2.0]  
c:[0:A 1:B 2:C 3:D]
a:[0:0 1:2 2:4 3:6 4:8] b:[0:0.0 1:0.0 2:0.1 3:-3.0]  
c:[0:A 1:B 2:C 3:D]
a:[0:4 1:6 2:8 3:10 4:0] b:[0:-3.5 1:0.0 2:4.9 3:0.0]  
c:[0:A 1:B 2:C 3:D]
a:[0:8 1:10 2:0 3:2 4:4] b:[0:0.0 1:6.1 2:0.0 3:0.0]  
c:[0:A 1:B 2:C 3:D]
```

Here is the output data set:

```
b:[0:0.0 1:0.0 2:0.0 3:-2.0] sub:[0:(sub.a:6 sub.c:A)  
1:(sub.a:8 sub.c:B)  
2:(sub.a:10 sub.c:C)  
3:(sub.a:0 sub.c:D)  
4:(sub.a:2 sub.c:\0)]
```
Note that vector field \( c \) was shorter than vector field \( a \) and so was padded with the data type's default value when the operator combined it in a subrecord. The default value is shown in boldface type.

### The makevect Operator

The **makevect** operator combines specified fields of an input data record into a vector of fields of the same type. The input fields must be consecutive and numbered in ascending order. The numbers must increase by one. The fields must be named `field_name0` to `field_namen`, where `field_name` starts the name of a field and `0` and `n` are the first and last of its consecutive numbers. All these fields are combined into a vector of the same length as the number of fields \((n+1)\). The vector is called `field_name`.

The **splitvect** operator performs the inverse operation. See “The splitvect Operator” on page 30-30.
Data Flow Diagram

The following figure is a conceptual diagram of the `makevect` operator, where `vtype` denotes any valid Orchestrate data type that can figure in a vector, that is, any data type except tagged and * (schema variable).

Properties

Table 141 makevect Operator Options

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td><code>inRec:*; name0:vtype; name1:vtype; ... namen:vtype;</code></td>
</tr>
<tr>
<td>Output interface schema</td>
<td><code>outRec:*; name[n+1]:vtype;</code></td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>See “Transfer Behavior” on page 30-22.</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
</tbody>
</table>

Transfer Behavior

The input interface schema is:

```plaintext
record ( inRec:*; name0:vtype; name1:vtype; ... namen:vtype; )
```

The `inRec` schema variable captures all the input record’s fields except those that are combined into the vector. The data type of the input fields determines that of the output vector. All fields to be combined in a vector must have compatible data types so that Orchestrate can cast the data types of all fields into one data type.
Data type casting is based on the following rules:

- Any integer, signed or unsigned, when compared to a floating-point type, is converted to floating-point.
- Comparisons within a general type convert the smaller to the larger size (sfloat to dfloat, uint8 to uint16, etc.)
- When signed and unsigned integers are compared, unsigned are converted to signed.
- Decimal, raw, string, time, date, and timestamp do not figure in type conversions. When any of these is compared to another type, makevect returns an error and terminates.

The output data set has the schema:

```plaintext
record ( outRec:*; name[n]:vtype; )
```

The `outRec` schema variable does not capture the entire input record. It does not include the fields that have been combined into the vector.

### Non-Consecutive Fields

If a field between `field_name0` and `field_namen` is missing from the input, the operator stops writing fields to a vector at the missing number and writes the remaining fields as top-level fields. For example, data with this input schema:

```plaintext
record ( a0:uint8; a1:uint8; a2:uint8; a4:uint8; a5:uint8; )
```

is combined as follows:

```plaintext
record ( a4:uint8; a5:uint8; a[3]:uint8; )
```

The operator combines fields `a0`, `a1`, and `a2` in a vector and writes fields `a4` and `a5` as top-level fields.

### Syntax and Option

There is one option, it is required, and you specify it only once. The syntax is as follows:

```plaintext
makevect
   -name field_name
```
Example 1: The makevect Operator

Here is the schema of a data set given as input to the makevect operator on fields whose names begin with a. In the example, the operator runs sequentially.

```plaintext
record ( a0:uint8; a1:uint8; a2:uint8; a3:uint8; a4:uint8; )
```

Here is the schema of the corresponding output data set:

```plaintext
record ( a[5]:uint8; )
```

The os command is:

```plaintext
$ osh "... makevect -name a ..."
```

Here are the input and output of the operation:

<table>
<thead>
<tr>
<th>Input Data Set</th>
<th>Output Data Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0:0 a1:6 a2:0 a3:2 a4:8</td>
<td>a: [0: 0 1:6 2:0 3:2 4:8]</td>
</tr>
<tr>
<td>a0:20 a1:42 a2:16 a3:10 a4:28</td>
<td>a: [0:20 1:42 2:16 3:10 4:28]</td>
</tr>
<tr>
<td>a0:40 a1:78 a2:32 a3:18 a4:48</td>
<td>a: [0:40 1:78 2:32 3:18 4:48]</td>
</tr>
<tr>
<td>a0:5 a1:15 a2:4 a3:4 a4:13</td>
<td>a: [0:5 1:15 2:4 3:4 4:13]</td>
</tr>
<tr>
<td>a0:25 a1:51 a2:20 a3:12 a4:33</td>
<td>a: [0:25 1:51 2:20 3:12 4:33]</td>
</tr>
<tr>
<td>a0:45 a1:87 a2:36 a3:20 a4:53</td>
<td>a: [0:45 1:87 2:36 3:20 4:53]</td>
</tr>
<tr>
<td>a0:10 a1:24 a2:8 a3:6 a4:18</td>
<td>a: [0:10 1:24 2:8 3:6 4:18]</td>
</tr>
<tr>
<td>a0:30 a1:60 a2:24 a3:14 a4:38</td>
<td>a: [0:30 1:60 2:24 3:14 4:38]</td>
</tr>
<tr>
<td>a0:15 a1:33 a2:12 a3:8 a4:23</td>
<td>a: [0:15 1:33 2:12 3:8 4:23]</td>
</tr>
<tr>
<td>a0:35 a1:69 a2:28 a3:16 a4:43</td>
<td>a: [0:35 1:69 2:28 3:16 4:43]</td>
</tr>
</tbody>
</table>

Note that the operator did not write the output records in the same order as the input records, although all fields appear in the correct order.

Example 2: The makevect Operator with Missing Input Fields

Here is the schema of a data set given as input to the makevect operator on fields whose names begin with a. Field a3 is missing from the series of fields that are combined into a vector. One field (c) does not have the same name as those in the series.

```plaintext
record ( a0:uint8; a1:uint8; a2:uint8; a3:uint8; a4:uint8; c:decimal[2,1]; )
```
Here is the output schema of the same data after the `makevect` operation.

```plaintext
record ( a4:uint8; c:decimal[2,1]; a[3]:uint8 ;)
```

Here is the `osh` command for this example:

```plaintext
$ osh "... makevect -name a ..."
```

Note that the operator:

- Stopped combining "a" fields into a vector when it encountered a field that
  was not consecutively numbered.
- Wrote the remaining, non-consecutively numbered field (a4) as a top-level
  one.
- Transferred the unmatched field (c) without alteration.
- Did not write the output records in the same order as the input records,
  although all fields appear in the correct order.

Here are the input and output of a `makevect` operation on the field `a`:

<table>
<thead>
<tr>
<th>Input Data Set</th>
<th>Output Data Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0: 0 a1: 6 a2: 0 a4: 8 c: 0.0</td>
<td>a4: 8 c: 0.0 a: [0:0 1:6 2:0]</td>
</tr>
<tr>
<td>a0: 20 a1: 42 a2: 16 a4: 28 c: 2.8</td>
<td>a4: 28 c: 2.8 a: [0:20 1:42 2:16]</td>
</tr>
<tr>
<td>a0: 40 a1: 78 a2: 32 a4: 48 c: 5.6</td>
<td>a4: 48 c: 5.6 a: [0:40 1:78 2:32]</td>
</tr>
<tr>
<td>a0: 5 a1: 15 a2: 4 a4: 13 c: 0.7</td>
<td>a4: 13 c: 0.7 a: [0:5 1:15 2:4]</td>
</tr>
<tr>
<td>a0: 25 a1: 51 a2: 20 a4: 33 c: 3.5</td>
<td>a4: 23 c: 2.0 a: [0:25 1:51 2:20]</td>
</tr>
<tr>
<td>a0: 45 a1: 87 a2: 36 a4: 53 c: 6.3</td>
<td>a4: 33 c: 3.5 a: [0:45 1:87 2:36]</td>
</tr>
<tr>
<td>a0: 10 a1: 24 a2: 8 a4: 18 c: 1.4</td>
<td>a4: 18 c: 1.4 a: [0:10 1:24 2:8]</td>
</tr>
<tr>
<td>a0: 30 a1: 60 a2: 24 a4: 38 c: 4.2</td>
<td>a4: 38 c: 4.2 a: [0:30 1:60 2:24]</td>
</tr>
<tr>
<td>a0: 15 a1: 33 a2: 12 a4: 23 c: 2.0</td>
<td>a4: 38 c: 4.2 a: [0:30 1:60 2:24]</td>
</tr>
<tr>
<td>a0: 35 a1: 69 a2: 28 a4: 43 c: 4.9</td>
<td>a4: 43 c: 4.9 a: [0:35 1:69 2:28]</td>
</tr>
</tbody>
</table>

The `promotesubrec` Operator

The `promotesubrec` operator promotes the fields of an input subrecord to top-level fields. The number of output records equals the number of subrecord elements. The data types of the input subrecord fields determine those of the corresponding top-level fields.

The `aggtorec` operator often performs the inverse operation; see “The `aggtorec` Operator” on page 30-3.
The input interface schema is as follows:

```plaintext
record ( inRec:*; subrecname[ ]:subrec (rec:*); )
```

where `inRec` does not include the subrecord to be promoted.

The output interface schema is as follows:

```plaintext
record ( outRec:*; rec:*; )
```

where `outRec` includes the same fields as `inRec` (top-level fields), and `rec` includes the fields of the subrecord.

### Syntax and Option

There is one option. It is required. You may specify it only once.

```plaintext
promotesubrec
  • subrecname subrecname
```
The splitsubrec Operator

The `splitsubrec` operator separates an input subrecord into a set of top-level vector fields. The operator creates one new vector field for each element of the original subrecord. That is, each top-level vector field that is created has the same number of elements as the subrecord from which it was created. The operator

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-subrecname</code></td>
<td><code>-subrecname subrecname</code></td>
</tr>
</tbody>
</table>

Specifies the name of the subrecord whose elements are promoted to top-level records.
outputs fields of the same name and data type as those of the fields that comprise the subrecord.

**Note**  
Input subrecords fields cannot be vectors.

The `makesubrec` operator performs the inverse operation. See “The makesubrec Operator” on page 30-17.

**Data Flow Diagram**

```
input data set

inRec:*; name[] subrec (fielda:atype; fieldb:btype; ...fieldn:ntype;)

splitsubrec

outRec:*; fielda[]:atype; fieldb[]:btype; ...fieldn[]:ntype;

output data set
```

**Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:*; name[] subrec (a:atype; b:btype; ...n:ntype;)</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:*; a[]:atype; b[]:btype; ...n[]:ntype;</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>See below.</td>
</tr>
</tbody>
</table>

The input interface schema is as follows:

```
record { inRec:*; name[] subrec (subfielda:atype; subfieldb:btype; ...subfieldn:ntype; ) }
```
The \text{inRec} schema variable omits the input field whose elements are separated and promoted. The subfields to be promoted can be of any data type except tagged or vector.

The output interface schema is as follows:

\begin{verbatim}
record ( outRec:*; fielda[]:atype; fieldb[]:btype; ... fieldn[]:ntype; )
\end{verbatim}

Each new vector field has the same name and data type as its corresponding input subfield.

This operator also works with scalar components, treating them as if they were vectors of length 1.

**Syntax and Option**

There is one option. It is required. You may specify it only once. The syntax is as follows:

\begin{verbatim}
splitsubrec -subrecname subrecname ...
\end{verbatim}

**Example**

In this example, which runs sequentially:

- The input schema is as follows:

\begin{verbatim}
record ( s[5]:subrec( a:uint8; b:decimal[2,1]; ) )
\end{verbatim}

- The \text{splitsubrec} operator separates and promotes subfields a and b of subrecord s.

- The output schema is as follows:

\begin{verbatim}
record ( a[5]:uint8; b[5]:decimal[2,1]; )
\end{verbatim}

Here is one record of the input data set:

\begin{verbatim}
s:[0:(s.a:8 s.b:-0.6) 1:(s.a:10 s.b:4.5) 2:(s.a:0 s.b:5.1) 3:(s.a:2 s.b:-8.5) 4:(s.a:4 s.b:2.1)]
\end{verbatim}

Here is the corresponding output record:
Here is the osh command for this example:

$ osh "... splitsubrec -subrecname s ..."

Here is the entire input data set:

```
s:
  0:(s.a:8 s.b:-0.6)
  1:(s.a:10 s.b:4.5)
  2:(s.a:0 s.b:5.1)
  3:(s.a:2 s.b:-8.5)
  4:(s.a:4 s.b:2.1)

s:
  0:(s.a:10 s.b:5.8)
  1:(s.a:0 s.b:-1.7)
  2:(s.a:2 s.b:1.6)
  3:(s.a:4 s.b:-5.0)
  4:(s.a:6 s.b:4.6)

s:
  0:(s.a:6 s.b:9.1)
  1:(s.a:8 s.b:4.1)
  2:(s.a:10 s.b:-2.3)
  3:(s.a:0 s.b:9.0)
  4:(s.a:2 s.b:2.2)

s:
  0:(s.a:0 s.b:3.6)
  1:(s.a:2 s.b:-5.3)
  2:(s.a:4 s.b:0.5)
  3:(s.a:6 s.b:-3.2)
  4:(s.a:8 s.b:4.6)

s:
  0:(s.a:4 s.b:8.5)
  1:(s.a:6 s.b:-6.6)
  2:(s.a:8 s.b:6.5)
  3:(s.a:10 s.b:8.9)
  4:(s.a:0 s.b:4.4)
```

Here is the entire output data set:

```
a:
  0:6 1:8 2:10 3:0 4:2
  b:
   0:6 1:4.1 2:-2.3 3:9 0 4:2.2

a:
  0:0 1:2 2:4 3:6 4:8
  b:
   0:3.6 1:-5.3 2:-0.5 3:-3.2 4:4.6

a:
  0:4 1:6 2:8 3:10 4:0
  b:
   0:8.5 1:-6.6 2:6.5 3:8 9 4:4.4

a:
  0:8 1:10 2:0 3:2 4:4
  b:
   0:-0.6 1:4.5 2:5 1 3:-8 5 4:2.1

a:
  0:10 1:0 2:2 3:4 4:6
  b:
   0:5.8 1:-1.7 2:1.6 3:-5 0 4:4.6
```

The splitvect Operator

The **splitvect** operator promotes the elements of a fixed-length vector to a set of similarly named top-level fields. The operator creates fields of the format `name0` to `namen`, where `name` is the original vector’s name and `0` and `n` are the first and last elements of the vector.

The **makevect** operator performs the inverse operation. See “The makevect Operator” on page 30-21.
Data Flow Diagram

The following figure is a conceptual diagram of the `splitvect` operator, where `vtype` denotes any valid Orchestrate data type that can figure in a vector, that is, any data type except tagged and * (schema variable):

Properties

Table 147  splitvect Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td><code>inRec:*; name[n]:vtype;</code></td>
</tr>
<tr>
<td>Output interface schema</td>
<td><code>outRec:*; name0:vtype; name1:vtype; namen-1:vtype;</code></td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>See below.</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
</tbody>
</table>

The input interface schema is:
```orchestrate
record ( inRec:*; name[n]:vtype; )
```

The `inRec` schema variable captures all the input record’s fields except the vector field whose elements are separated from each other and promoted. The data type of this vector is `vtype`, which denotes any valid Orchestrate data type that can figure in a vector, that is, any data type except tagged and * (schema variable). This data type determines the data type of the output, top-level fields. The vector must be of fixed length.
The output data set has the schema:

```
record ( outRec:*; name0:vtype; name1:vtype; namen-1:vtype; )
```

### Syntax and Option

There is one option. It is required. You may specify it only once. The syntax is as follows:

```
splitvect
  -name name
```

**Table 148 splitvect Operator Option**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name</td>
<td>-name name</td>
</tr>
</tbody>
</table>

Specifies the name of the vector whose elements you want to promote to a set of similarly named top-level fields.

### Example

Here is the input schema of a data set to be processed by `splitvect`:

```
record ( a[5]:uint8; )
```

Here is the output data set's schema:

```
record ( a0:uint8; a1:uint8; a2:uint8; a3:uint8; a4:uint8; )
```

Here is the `osh` command:

```
splitvect -name a ...
```

Here are a sample input and output of the operator:

<table>
<thead>
<tr>
<th>Input Data Set</th>
<th>Output Data Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: [0:0 1:6 2:0 3:2 4:8]</td>
<td>a0:0 a1:6 a2:0 a3:2 a4:8</td>
</tr>
<tr>
<td>a: [0:20 1:42 2:16 3:10 4:28]</td>
<td>a0:20 a1:42 a2:16 a3:10 a4:28</td>
</tr>
<tr>
<td>a: [0:40 1:78 2:32 3:18 4:48]</td>
<td>a0:40 a1:78 a2:32 a3:18 a4:48</td>
</tr>
<tr>
<td>a: [0:15 1:33 2:12 3:8 4:23]</td>
<td>a0:10 a1:24 a2:8 a3:6 a4:18</td>
</tr>
<tr>
<td>a: [0:25 1:51 2:20 3:12 4:33]</td>
<td>a0:30 a1:60 a2:24 a3:14 a4:38</td>
</tr>
<tr>
<td>a: [0:45 1:87 2:36 3:20 4:53]</td>
<td>a0:15 a1:33 a2:12 a3:8 a4:23</td>
</tr>
<tr>
<td>a: [0:35 1:69 2:28 3:16 4:43]</td>
<td>a0:35 a1:69 a2:28 a3:16 a4:43</td>
</tr>
<tr>
<td>a: [0:10 1:24 2:8 3:6 4:18]</td>
<td>a0:25 a1:51 a2:20 a3:12 a4:33</td>
</tr>
<tr>
<td>a: [0:30 1:60 2:24 3:14 4:38]</td>
<td>a0:45 a1:87 a2:36 a3:20 a4:53</td>
</tr>
</tbody>
</table>
Note that the operator did not write the output records in the same order as the input records, although all fields appear in the correct order.

The tagbatch Operator

Tagged fields are types of aggregate fields that define a nested form of data representation. However, most Orchestrate operators do not support operations on aggregate fields. This section describes an operator that converts tagged fields to a representation usable by other Orchestrate operators.

The tagbatch operator reads input records containing a tagged aggregate and flattens the tag cases into an output record whose schema accommodates all possible fields of the various tag cases.

The tagswitch operator copies each input record to a separate output data set based on the active case of a tagged field in the input record. (See “The tagswitch Operator” on page 30-43.)

Tagged Fields and Operator Limitations

Tagged fields are called aggregate fields in Orchestrate. Most Orchestrate operators do not directly support operations on aggregate fields but act on toplevel fields only. For example, the Orchestrate statistics operator can calculate statistical information only on the top-level fields of an input data set. It cannot process subrecords or nested fields.

The tagbatch operator removes this limitation, because it extracts the cases of a tagged field and raises them to the top level of a record schema.

Tagged fields let you access the data storage of a single record field as different data types. For example, you may have one record where the tagged field contains string data and another record where the field contains a floating-point value. By using a tagged field, each record can use the data type for the field that matches the data stored within it.

You typically use tagged fields when importing data from a COBOL data file, if the COBOL data definition contains a REDEFINES statement. A COBOL REDEFINES statement specifies alternative data types for a single field.

A tagged field has multiple data types, called tag cases. Each case of the tagged is defined with both a name and a data type. The following example shows a record schema that defines a tagged field that has three cases corresponding to three different data types:

```
record ( tagField:tagged

{ aField:string;     // tag case 0
  bField:int32;     // tag case 1
  cField:sfloat;    // tag case 2
}
```

The content of a tagged field can be any one of the subfields defined for the tagged field. In this example, the content of tagField is one of the following: a variable-length string, a 32-bit integer, a single-precision floating-point value. A tagged field in a record can have only one item from the list. This case is called the active case of the tagged.

"Dot" addressing refers to the components of a tagged field: For example, tagField.aField references the aField case, tagField.bField references the bField case, and so on.

Tagged fields can contain subrecord fields. For example, the following record schema defines a tagged field made up of three subrecord fields:

```c
record { tagField:tagged
    aField:subrec( // tag case 0
        aField_s1:int8;
        aField_s2:uint32;)
    bField:subrec( // tag case 1
        bField_s1:std::string[10];
        bField_s2:uint32;)
    cField:subrec( // tag case 2
        cField_s1:uint8;
        cField_s2:dfloat;)
};
```

In this example, each subrecord is a case of the tagged. You reference cField_s1 of the cField case as tagField.cField.cField_s1.

See the chapter on Orchestrate data sets in the Orchestrate 7.0 User Guide for more information on tagged fields.

**Operator Action and Transfer Behavior**

The `tagbatch` operator reads input records containing a tagged aggregate and flattens the tag cases into an output record whose schema accommodates all possible fields of the various tag cases. The operator groups input records based on one or more key fields and copies all records in the group to a single output record. The output record contains both all non-tagged fields of the input record and all cases of the tagged field as top-level fields.
Data Flow Diagram

Properties

Table 149  tagbatch Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>inRec:<em>; tField:tagged(c0:subrec(rec:</em>); c1:subrec(rec:*); ...)</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>outRec:<em>; c0_present:int8; c0_rec:</em>; c1_present:int8; c1_rec:*; ...</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>See below.</td>
</tr>
</tbody>
</table>

The input interface schema is:

```
inRec:*; tField:tagged(c0:subrec(rec:*); c1:subrec(rec:*); ...);
```

By default, the `inRec` schema variable does not include the original tagged field.

The output interface schema is:

```
outRec:*; c0_present:int8; c0_rec:*; c1_present:int8; c1_rec:*; ...
```

Added, Missing, and Duplicate Fields

When tagged fields are promoted to top-level fields, each is preceded by an `int8` field called `case_present`, where `case` is the name of the case. If the promoted field contains a value, the `case_present` field is set to 1. If the promoted field does not contain a value, the `case_present` field is set to 0 and the promoted field is set to the default value for its data type.
If two records with the same key value have the same case for the tagged field, the operator outputs the case value of the first record processed by the operator. However, you can modify this behavior by means of the `-ifD u l i p l i c a t e C a s e` option.

If a tagged case is missing from a record, the operator outputs a record with the corresponding present field set to 0 and the field of the tag case is set to the default value for its data type. However, you can modify this behavior by means of the `-i f M i s s i n g C a s e` option.

"Example 2: The tagbatch Operator, Missing and Duplicate Cases" on page 30-40 shows the behavior described in the three preceding paragraphs.

**Input Data Set Requirements**

The input data set must meet these requirements:

- The input data set must contain a tagged field.
- Each case of the tag must consist of a subrecord field.
- Input records with identical key field values must be adjacent in the input data set. That is, you must first hash partition and sort the input data set by the key fields you then specify to the `tagbatch` operator.

**Syntax and Options**

The `-k e y` option is required. The syntax is as follows:

```bash
tagbatch
   -k e y field_name [-c i | -c s] [-p a r a m params]
   [-k e y field_name [-c i | -c s] [-p a r a m params] ...]
   [-collation_sequence locale|collation_file_pathname|OFF]
   [-ifDuplicateCase drop | fail | keepFirst | keepLast]
   [-ifMissingCase continue | drop | fail]
   [-nobatch]
   [-tag tag_field]
```

Orchestrate 7.0 Operators Reference
Table 150  tagbatch Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-key</td>
<td>-key field_name [-ci</td>
</tr>
<tr>
<td></td>
<td>Specifies the name of the key field used to combine input records. The key option may be repeated for multiple key fields. See “Example 3: The tagbatch Operator with Multiple Keys” on page 30-41.</td>
</tr>
<tr>
<td></td>
<td>-ci or -cs are optional arguments for specifying case-insensitive or case-sensitive keys. By default, the operator uses case-sensitive matching. Be sure to specify the same case sensitivity here as you did when you sorted the input data set.</td>
</tr>
<tr>
<td></td>
<td>The -param suboption allows you to specify extra parameters for a field. Specify parameters using property=value pairs separated by commas.</td>
</tr>
<tr>
<td>-collation_sequence</td>
<td>-collation_sequence locale</td>
</tr>
<tr>
<td></td>
<td>This option determines how your string data is sorted. You can:</td>
</tr>
<tr>
<td></td>
<td>• Specify a predefined IBM ICU locale</td>
</tr>
<tr>
<td></td>
<td>• Write your own collation sequence using ICU syntax, and supply its collation_file_pathname</td>
</tr>
<tr>
<td></td>
<td>• Specify OFF so that string comparisons are made using Unicode code-point value order, independent of any locale or custom sequence.</td>
</tr>
<tr>
<td></td>
<td>By default, Orchestrate sorts strings using byte-wise comparisons.</td>
</tr>
<tr>
<td></td>
<td>For more information, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:</td>
</tr>
<tr>
<td></td>
<td><a href="http://oss.software.ibm.com/icu/userguide/Collate_Intro.htm">http://oss.software.ibm.com/icu/userguide/Collate_Intro.htm</a></td>
</tr>
<tr>
<td>-ifDuplicateCase</td>
<td>-ifDuplicateCase drop</td>
</tr>
<tr>
<td></td>
<td>Controls how the operator handles two records with the same key value that have the same active case for the tagged field. Suboptions include:</td>
</tr>
<tr>
<td></td>
<td>drop: Drop the output record. This means that no record is output for that key value. See “Example 2: The tagbatch Operator, Missing and Duplicate Cases” on page 30-40.</td>
</tr>
<tr>
<td></td>
<td>fail: Fail the operator when two or more records have the same active case.</td>
</tr>
<tr>
<td></td>
<td>keepFirst: Default. The operator outputs the case value from the first record processed by the operator.</td>
</tr>
<tr>
<td></td>
<td>keepLast: Output the case from the last record. See “Example 2: The tagbatch Operator, Missing and Duplicate Cases” on page 30-40.</td>
</tr>
</tbody>
</table>
Table 150  tagbatch Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-nobatch</td>
<td>-nobatch</td>
</tr>
<tr>
<td></td>
<td>Optionally specify that a record is output for every input record. The default is to place input records into groups based on key values and output them as a single record.</td>
</tr>
<tr>
<td>-ifMissingCase</td>
<td>-ifMissingCase continue</td>
</tr>
<tr>
<td></td>
<td>Optionally specifies how the operator handles missing tag cases for a specified key field. By default, the operator outputs a record with the corresponding present field set to 0 and the field of the tag case set to their default values (0 for numeric fields, 0 length for variable-length strings, and so on). The suboptions are:</td>
</tr>
<tr>
<td></td>
<td>continue: Default. Outputs a record with the corresponding present field set to 0 and the field of the tag case set to their default values (0 for numeric fields, 0 length for variable-length strings, and so on).</td>
</tr>
<tr>
<td></td>
<td>drop: Drop the record. See “Example 2: The tagbatch Operator, Missing and Duplicate Cases” on page 30-40.</td>
</tr>
<tr>
<td></td>
<td>fail: Fail the operator and step.</td>
</tr>
<tr>
<td>-tag</td>
<td>-tag tag_field</td>
</tr>
<tr>
<td></td>
<td>Specifies the name of the tagged field in the input data set used by the operator. If you do not specify this option, the input data set should have only one tagged field. If it has more than one tagged field, the operator processes only the first one and issues a warning when encountering others.</td>
</tr>
</tbody>
</table>

Example 1: The tagbatch Operator, Simple Flattening of Tag Cases

This example shows an input data set and its processing by the tagbatch operator. The name of the key field used to combine input records is key. The data set has been hash partitioned and sorted.
The operator runs sequentially. Here are the input and output schemas:

<table>
<thead>
<tr>
<th>Input Schema</th>
<th>Output Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>key:int32; t:tagged</td>
<td>key:int32;</td>
</tr>
<tr>
<td>(A:subrec(fname:string; lname:string);)</td>
<td>A_present:int8; fname:string; lname:string;</td>
</tr>
<tr>
<td>B:subrec(income:int32;);</td>
<td>B_present:int8; income:int32;</td>
</tr>
<tr>
<td>C:subrec(birth:date; retire:date;);</td>
<td>C_present:int8; birth:date; retire:date;</td>
</tr>
</tbody>
</table>

In the following data set representations, the elements of a tagged field are enclosed in angle brackets (<>); the elements of a subrecord are enclosed in parentheses ()

Here are three tagged records whose key field is equal:

1 t:<t.A:(t.A.fname:booker t.A.lname:faubus)>
   key:11 t:<t.B:(t.B.income:27000)>

Here is the equivalent output record:


The fields A_present, B_present, and C_present now precede each output field and are set to 1, since the field that follows each contains a value.

Here is the osh command for this example:

$ osh "... tagbatch -key key ..."
Here are the input and output data sets without field names:

<table>
<thead>
<tr>
<th>Input Data Set</th>
<th>Output Data Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 &lt;(booker faubus)&gt;</td>
<td>11 1 booker faubus 1 27000 1 1962-02-06 2024-02-06</td>
</tr>
<tr>
<td>11 &lt;(27000)&gt;</td>
<td></td>
</tr>
<tr>
<td>11 &lt;(1962-02-06 2024-0206)&gt;</td>
<td></td>
</tr>
<tr>
<td>22 &lt;(marilyn hall)&gt;</td>
<td>22 1 marilyn hall 1 35000 1 1950-09-20 2010-09-20</td>
</tr>
<tr>
<td>22 &lt;(35000)&gt;</td>
<td></td>
</tr>
<tr>
<td>22 &lt;(1950-09-20 2010-0920)&gt;</td>
<td></td>
</tr>
<tr>
<td>33 &lt;(gerard devries)&gt;</td>
<td>33 1 gerard devries 1 50000 1 1944-12-23 2009-12-23</td>
</tr>
<tr>
<td>33 &lt;(50000)&gt;</td>
<td></td>
</tr>
<tr>
<td>33 &lt;(1944-12-23 2009-1223)&gt;</td>
<td></td>
</tr>
<tr>
<td>44 &lt;(ophelia oliver)&gt;</td>
<td>44 1 ophelia oliver 1 65000 1 1970-04-11 2035-04-11</td>
</tr>
<tr>
<td>44 &lt;(65000)&gt;</td>
<td></td>
</tr>
<tr>
<td>44 &lt;(1970-04-11 2035-0411)&gt;</td>
<td></td>
</tr>
<tr>
<td>55 &lt;(omar khayam)&gt;</td>
<td>55 1 omar khayam 1 42000 1 1980-06-06 2040-06-06</td>
</tr>
<tr>
<td>55 &lt;(42000)&gt;</td>
<td></td>
</tr>
<tr>
<td>55 &lt;(1980-06-06 2040-0606)&gt;</td>
<td></td>
</tr>
</tbody>
</table>

**Example 2: The tagbatch Operator, Missing and Duplicate Cases**

This example shows an input data set and its processing by the `tagbatch` operator. The name of the key field used to combine input records is `key`. The data set has been hash-partitioned and sorted. The operator runs sequentially.

The input and output schemas are identical to those shown in “Example 1: The `tagbatch` Operator, Simple Flattening of Tag Cases”. The input data set is identical to that shown in the same example, except that one field is missing from the input data set and one occurs twice. Accordingly:

- The `-ifMissingCase` option has been set to `drop`.
- The `-ifDuplicateCase` option has been set to `keepLast`.

Here is the `osh` command for this example:

```bash
$ osh "... tagbatch -key key -ifMissingCase drop -ifDuplicateCase keepLast ...
```

Here is the input data set without field names. The income field is missing from those containing a `-key` whose value is 33. The corresponding fields are shown in boldface type. The income field occurs twice in those containing a key whose value is 44. The corresponding fields are shown in italic type.

```
11 <(booker faubus)>
11 <(27000)>
```

```
22 <(marilyn hall)>
22 <(35000)>
22 <(1950-09-20 2010-0920)>
33 <(gerard devries)>
33 <(50000)>
33 <(1944-12-23 2009-1223)>
44 <(ophelia oliver)>
44 <(65000)>
44 <(1970-04-11 2035-0411)>
55 <(omar khayam)>
55 <(42000)>
55 <(1980-06-06 2040-0606)>
```
Here is the output data set without field names.

```
11 1 booker faubus 1 27000 1 1962-02-06 2024-02-06
22 1 marilyn hall 1 35000 1 1950-09-20 2010-09-20
44 1 ophelia oliver 1 65000 1 1970-04-11 2035-04-11
55 1 omar khayam 1 42000 1 1980-06-06 2040-06-06
```

The operator has dropped the field containing a key whose value is 33 and has written the last value of the duplicate field (shown in italic type).

**Example 3: The tagbatch Operator with Multiple Keys**

This example shows an input data set and its processing by the `tagbatch` operator. Two key fields are used to combine input records, `fname` and `lname`. The data set has been hash partitioned and sorted. Here are the input and output schemas:

<table>
<thead>
<tr>
<th>Input Schema</th>
<th>Output Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fname:string; lname:string; t:tagged (emp_info:subrec (start_date:date; emp_id:int32;)); compensation:subrec (title:string {delim=' OR '}; grade:int16; salary:int32; ); personal:subrec (birth:date; ssn:string; ); )</code></td>
<td><code>fname:string; lname:string; emp_info_present:int8; start_date:date; emp_id:int32; compensation_present:int8; title:string; grade:int16; salary:int32; personal_present:int8; birth:date; ssn:string;</code></td>
</tr>
</tbody>
</table>

Here is the `osh` command for this example:

```
$ osh "... tagbatch -key fname -key lname ...
```
Here are three fields of the input data set:

Rosalind Fleur <(VP sales 1 120000)>
Rosalind Fleur <(1998-01-03 456)>
Rosalind Fleur <(1950-04-16 333-33-3333)>

Here is their corresponding output:

Rosalind Fleur 1 1998-01-03 456 1 VP sales 1 120000 1 1950-04-16 333-33-3333

Here is the entire input data set:

Jane Doe <(principal engineer 3 72000)>
Jane Doe <(1991-11-01 137)>
Jane Doe <(1950-04-16 111-11-111100)>
John Doe <(senior engineer 3 42000)>
John Doe <(1992-03-12 136)>
John Doe <(1960-11-12 000-00-0000)>
Rosalind Elias <(senior marketer 2 66000)>
Rosalind Elias <(1994-07-14 208)>
Rosalind Elias <(1959-10-16 222-22-2222)>
Rosalind Fleur <(VP sales 1 120000)>
Rosalind Fleur <(1998-01-03 456)>
Rosalind Fleur <(1950-04-16 333-33-3333)>

Here is the entire output data set:

Jane Doe 1 1991-11-01 137 1 principal engineer 3 72000 1 1950-04-16 111-11-111100
John Doe 1 1992-03-12 136 1 senior engineer 3 42000 1 1960-11-12 000-00-0000
Rosalind Elias 1 1994-07-14 208 1 senior marketer 2 66000 1 1959-10-16 222-22-2222
Rosalind Fleur 1 1998-01-03 456 1 VP sales 1 120000 1 1950-04-16 333-33-3333

The tagbatch operator has processed the input data set according to both the fname and lname key fields, because choosing only one of these as the key field gives incomplete results.

If you specify only the fname field as the key, the second set of tagged fields whose fname is Rosalind is dropped from the output data set. The output data set, which is as follows, does not contain the data corresponding to Rosalind Fleur:

Jane Doe 1 1991-11-01 137 1 principal engineer 3 72000 1 1950-04-16 111-11-111100
John Doe 1 1992-03-12 136 1 senior engineer 3 42000 1 1960-11-12 000-00-0000
Rosalind Elias 1 1994-07-14 208 1 senior marketer 2 66000 1 1959-10-16 222-22-2222
Rosalind Fleur 1 1998-01-03 456 1 VP sales 1 120000 1 1950-04-16 333-33-3333

If you specify only the lname field as the key, the second set of tagged fields whose lname is Doe is dropped from output data set. The output data set, which follows, does not contain the data corresponding to John Doe.

Jane Doe 1 1991-11-01 137 1 principal engineer 3 72000 1 1950-04-16 111-11-111100
The tagswitch Operator

“Tagged Fields and Operator Limitations” on page 30-33 discusses the characteristics and limitations of tagged fields. Orchestrate operators do not support operations on this type of field. This section describes an operator that converts tagged fields to a representation usable by other Orchestrate operators.

The `tagswitch` operator transfers the contents of an input data set containing tagged aggregates to one or more output data sets. The output set or sets to which input data is transferred is determined by the tag value of the tagged aggregate. The output data sets contain the cases of the tagged field promoted to the top level of the data set’s record schema. Top-level fields of the input records are transferred to all output data sets.

Data Flow Diagram

![Data Flow Diagram](image)

Properties

Table 151  **tagswitch Operator Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>(1 \leq n), where (n) is the number of cases; (1 \leq n+1), if the operator transfers unlisted cases.</td>
</tr>
</tbody>
</table>
**Input and Output Interface Schemas**

The input interface schema is:

\[
\text{inRec:}; \text{tField: tagged(c0: subrec(rec:*); c1: subrec(rec:*); ..); otherRec:};
\]

By default, the `inRec` schema variable does not include the original tagged field or fields. However, you can override this default by means of the `modify` operator to force the operator to retain the tagged field, so that it is transferred from the input data set to the output data sets.

The output interface schema is:

\[
\text{Output data set N: outRec:}; \text{rec:}; \text{Optional other output data set: otherRec:};
\]

Top-level fields are always copied from the input data set to all output data sets.

**The case Option**

By default, all cases of the tagged field are copied to the output data sets. However, you can override this default by means of the `-case` option. When you specify this option, only the cases that you identify are copied to the output. The order in which you specify the `-case` option is important: the first instance of `-case` defines the tag case copied to output data set 0, the next defines the tag case copied to data set 1, and so on.

When you specify the `-case` option, by default the operator drops any record containing an unspecified case. You can override the default behavior by means of the `-ifUnlistedCase` option. This option controls operator behavior when an unspecified case is encountered: the option lets you either terminate the operator and the step that contains it or copy the record to the last data set attached to the operator.

**Note**

If you choose `-ifUnlistedCase` without first specifying `-case`, the operator ignores the option and defaults to all cases.
Using the Operator

The input data set must contain a tagged field. Each case of the tag must consist of a subrecord field.

There are no required options. By default, the operator processes all cases of the input tagged field. In this event, connect one output data set for each case of the tagged field in the input data set.

You can override the default and explicitly list tag cases to be processed by means of the tag option. If you choose the tag option, you can control how the operator behaves when it encounters an unlisted case.

Syntax and Options

The syntax is as follows. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose the value in single quotes.

```
tagswitch
    [-tag tag _field]
    [-allCases] | [-case case ...]
    [-ifUnlistedCase drop | fail | other]
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-allCases</td>
<td>-allCases</td>
</tr>
<tr>
<td></td>
<td>Default: The operator processes all cases of the input tagged field. Specify this option only to explicitly set the action of the operator. This option is mutually exclusive with the -case option.</td>
</tr>
<tr>
<td>-case</td>
<td>-case case</td>
</tr>
<tr>
<td></td>
<td>Specifies the case of the tagged field that you want to extract from the input tagged field. You can specify multiple -case options. For each, specify both the option and its argument. The operator produces one output data set for each -case option.</td>
</tr>
<tr>
<td></td>
<td>The order in which you specify the -case option is important: the first instance of -case defines the tag case copied to output data set 0, the next defines those records copied to data set 1, and so on.</td>
</tr>
<tr>
<td></td>
<td>Connect one output data set for each specified case. However, if you specify the -other suboption of the -ifUnlistedCase option (see below), connect n+1 data sets to the operator, where n is the number of cases that you have specified.</td>
</tr>
<tr>
<td></td>
<td>This option is mutually exclusive with the -allCases option.</td>
</tr>
</tbody>
</table>
Example 1: The tagswitch Operator, Default Behavior

This example shows the default behavior of the tagswitch operator. No option has been chosen.

The input data set contains three tag cases. Accordingly, three output data sets have been attached to the operator. The operator automatically writes the same number of data sets as there are tagged cases. The input data set has been sorted on the -key field for the purpose of demonstration, although the field does not have to be. The operator runs sequentially.
Here are the input and output schemas:

<table>
<thead>
<tr>
<th>Input Schema</th>
<th>Output Schemas</th>
</tr>
</thead>
<tbody>
<tr>
<td>key:int32; t:tagged (A:subrec(fname:string; lname:string; ); B:subrec(income:int32; ); C:subrec(birth:date; retire:date; ); )</td>
<td>data set # 1 (Case A) key:int32; fname:string; lname:string;</td>
</tr>
<tr>
<td></td>
<td>data set # 2 (Case B) key:int32; income:int32;</td>
</tr>
<tr>
<td></td>
<td>data set # 3 (Case C) key:int32; birth:date; retire:date;</td>
</tr>
</tbody>
</table>

Here is the osh command for this example:

```
tagswitch -tag tag < in.ds > out0.ds > out1.ds > out2.ds``
Here are the input data set and output data sets without field names:

<table>
<thead>
<tr>
<th>Input Data Set</th>
<th>Output Data Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Set # 1 (Case A)</td>
<td>11 booker faubus</td>
</tr>
<tr>
<td>Data Set # 2 (Case B)</td>
<td>11 27000</td>
</tr>
<tr>
<td>Data Set # 3 (Case C)</td>
<td>11 1962-02-06 2024-02-06</td>
</tr>
</tbody>
</table>

Each case (A, B, and C) has been promoted to a record. Each subrecord of the case has been promoted to a top-level field of the record.

Example 2: The tagswitch Operator, One Case Chosen

This example shows the action of the tagswitch operator on the same input data set as in “Example 1: The tagswitch Operator, Default Behavior” on page 30-46.

In this program, one case of three possible cases (case B) is chosen as the case of the tagged aggregate from which output is extracted. All other cases, which are unspecified, are copied to an other output data set. Accordingly, two output data
sets are attached to the operator, one for the specified case and one for any unspecified cases.

The input data set has been sorted on the key field for the purpose of demonstration, although the field does not have to be. The operator runs sequentially.

input schema:

key:int32;
t:tagged
(A:subrec(fname:string;
  lname:string;
);
B:subrec(income:int32;
);
C:subrec(birth:date;
  retire:date;
);
)

output schema data set # 1
(Case B)
key:int32;
income:int32;

output schema data set # 2
(other)
key:int32;
t:tagged
(A:subrec(fname:string;
  lname:string;
);
B:subrec(income:int32;
);
C:subrec(birth:date;
  retire:date;
);
)
Here are the input and output schemas:

<table>
<thead>
<tr>
<th>Input Schema</th>
<th>Output Schemas</th>
</tr>
</thead>
<tbody>
<tr>
<td>key:int32;</td>
<td>Data set # 1 (Case B) key:int32;</td>
</tr>
<tr>
<td>t: tagged</td>
<td>income:int32;</td>
</tr>
<tr>
<td>A: subrec(fname:string; lname:string; )</td>
<td></td>
</tr>
<tr>
<td>B: subrec(income:int32; )</td>
<td>Data set # 2 (other)</td>
</tr>
<tr>
<td>C: subrec(birth:date; retire:date; )</td>
<td>key:int32;</td>
</tr>
<tr>
<td></td>
<td>t: tagged</td>
</tr>
<tr>
<td></td>
<td>A: subrec(fname:string; lname:string; )</td>
</tr>
<tr>
<td></td>
<td>B: subrec(income:int32; )</td>
</tr>
<tr>
<td></td>
<td>C: subrec(birth:date; retire:date; )</td>
</tr>
</tbody>
</table>

Note that the input schema is copied intact to the output schema. Case B has not been removed from the output schema. If it were, the remaining tags would have to be rematched.

Here is the osh command for this example:

```
osh "tagswitch -case B -ifUnlistedCase other < in.ds > out0.ds 1> out1.ds >"
```
Here are the input and output data sets.

<table>
<thead>
<tr>
<th>Input Data Set</th>
<th>Output Data Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 &lt;(booker faubus)&gt;</td>
<td>Data set # 1</td>
</tr>
<tr>
<td>11 &lt;(27000)&gt;</td>
<td>11 27000</td>
</tr>
<tr>
<td>11 &lt;(1962-02-06 2024-02-06)&gt;</td>
<td>22 35000</td>
</tr>
<tr>
<td>22 &lt;(marilyn hall)&gt;</td>
<td>33 50000</td>
</tr>
<tr>
<td>22 &lt;(35000)&gt;</td>
<td>44 65000</td>
</tr>
<tr>
<td>22 &lt;(1950-09-20 2010-09-20)&gt;</td>
<td>55 42000</td>
</tr>
<tr>
<td>33 &lt;(gerard devries)&gt;</td>
<td></td>
</tr>
<tr>
<td>33 &lt;(50000)&gt;</td>
<td>Data set # 2</td>
</tr>
<tr>
<td>33 &lt;(1944-12-23 2009-12-23)&gt;</td>
<td>11 &lt;(booker faubus)&gt;</td>
</tr>
<tr>
<td>44 &lt;(ophelia oliver)&gt;</td>
<td>11 &lt;(1962-02-06 2024-02-06)&gt;</td>
</tr>
<tr>
<td>44 &lt;(65000)&gt;</td>
<td>22 &lt;(marilyn hall)&gt;</td>
</tr>
<tr>
<td>44 &lt;(1970-04-11 2035-04-11)&gt;</td>
<td>22 &lt;(1950-09-20 2010-09-20)&gt;</td>
</tr>
<tr>
<td>55 &lt;(omar khayam)&gt;</td>
<td>33 &lt;(gerard devries)&gt;</td>
</tr>
<tr>
<td>55 &lt;(42000)&gt;</td>
<td>33 &lt;(1944-12-23 2009-12-23)&gt;</td>
</tr>
<tr>
<td>55 &lt;(1980-06-06 2040-06-06)&gt;</td>
<td>44 &lt;(ophelia oliver)&gt;</td>
</tr>
<tr>
<td>55 &lt;(omar khayam)&gt;</td>
<td>55 &lt;(omar khayam)&gt;</td>
</tr>
</tbody>
</table>
The Sorting Library

Describes the operators that sort the records in your data sets.

Overview of the Sort Operators 31 2

The tsort Operator 31 2
  Configuring the tsort Operator 31 4
  Using a Sorted Data Set 31 4
    Using a Sorted Data Set with a Sequential Operator 31 5
    Passing a Sorted Data Set to an RDBMS: General Information 31 5
  Specifying Sorting Keys 31 5
Data Flow Diagram 31 7
Properties 31 7
Syntax and Options 31 8
Example: Using a Sequential tsort Operator 31 11
Example: Using a Parallel tsort Operator 31 12
Performing a Total Sort 31 13
Example: Performing a Total Sort 31 16

The psort Operator 31 17
  Introduction 31 17
  Performing a Partition Sort 31 18
  Configuring the Partition Sort Operator 31 19
  Using a Sorted Data Set 31 20
    Using a Sorted Data Set with a Sequential Operator 31 21
    Passing a Sorted Data Set to an RDBMS: General Information 31 21
    Specifying Sorting Keys 31 21
Data Flow Diagram 31 22
Properties 31 23
Syntax and Options 31 24
Example: Using a Sequential Partition Sort Operator 31 27
Example: Using a Parallel Partition Sort Operator 31 27
Performing a Total Sort 31 29
Overview of the Sort Operators

This chapter of the Orchestrate 7.0 Operators Reference describes how to use the sorting operators. To understand the information in this chapter you should be familiar with the operator concepts defined in the chapter on operators in the Orchestrate 7.0 User Guide.

By default, Orchestrate inserts partition and sort components in your data flow to meet the partitioning and sorting needs of your use of Orchestrate's predefined operators. See the section “Orchestrate-Inserted Partition and Sort Components” on page -xxxvi of this Reference for information on this facility. Use the sort operators described in this chapter when you want to explicitly control the sorting behavior of an operator.

The sorting library contains these operators:

• The **tsort** operator, which requires no additional sorting software to run. This operator is described on page 2. Because of the additional capabilities in **tsort**, it is recommended that you use this operator rather than **psort**.

• The **psort** operator, which requires SyncSort or UNIX sort. This operator is described on page 17.

The sorting operators sort the records of a data set by the values of one or more key fields in each record. The operators can be used in either parallel or sequential mode.

In sequential mode, the entire data set is sorted, but the sort is executed on a single node. In parallel mode, the data is partitioned and sorted on multiple nodes, but the resulting data set is sorted only within each partition. This is adequate for many tasks, such as removing duplicate records, if an appropriate partitioning method is used.

The **tsort** Operator

Orchestrate provides the sort operator, **tsort**, that you can use to sort the records of a data set. The **tsort** operator can run as either a sequential or a parallel operator. The execution mode of the **tsort** operator determines its action:
• Sequential mode: The `tsort` operator executes on a single processing node to sort an entire data set. On completion, the records of the data set are sorted completely.

• Parallel mode: The `tsort` operator executes on multiple processing nodes in your system. On completion, the records within each partition of a data set are sorted. This type of sort is called a partition sort.

The following figure shows the difference between these two operating modes:

The `tsort` operator on the left side of this figure runs sequentially to completely sort the records of a data set. Typically, you use the `tsort` operator sequentially when you need a total sort of a data set, or when you create a sequential application.

The `tsort` operator on the right side of this figure executes in parallel and sorts records within each partition. Remember that a parallel operator executes on multiple nodes in a system, where each node receives a partition of a data set. A parallel `tsort` operator outputs multiple partitions, where the records are sorted within the partition.

Typically, you use a parallel `tsort` operator as part of a series of operators that requires sorted partitions. For example, you can combine a sort operator with an operator that removes duplicate records from a data set. After the partitions of a data set are sorted, duplicate records in a partition are adjacent.
To perform a parallel sort, you insert a partitioner in front of the `tsort` operator. This lets you control how the records of a data set are partitioned before the sort. For example, you could hash partition records by a name field, so that all records whose name field begins with the same one-, two-, or three-letter sequence are assigned to the same partition. See “Example: Using a Parallel tsort Operator” on page 31-12 for more information.

If you combine a parallel sort with a sort merge collector, you can perform a total sort. A totally sorted data set output is completely ordered, meaning the records in each partition of the output data set are ordered, and the partitions themselves are ordered. See “Performing a Total Sort” on page 31-29 for more information.

### Configuring the tsort Operator

The `tsort` operator uses temporary disk space when performing a sort. The operator looks in the following locations, in the following order, for this temporary space:

1. **Scratch disks in the disk pool `sort`**.
   - You can create these pools by editing the Orchestrate configuration file, `config.apt`.

2. **Scratch disks in the default disk pool**.
   - All scratch disks are normally included in the default disk pool.

3. **The directory specified by the `TMPDIR` environment variable**.

4. **The directory `/tmp`**.

The Orchestrate 7.0 Installation and Administration Manual chapter on configuration gives details on how to allocate disk space for sorting.

### Using a Sorted Data Set

You may perform a sort for several reasons. For example, you may want to sort a data set by a zip code field, then by last name within the zip code. Once you have sorted the data set, you can filter the data set by comparing adjacent records and removing any duplicates.

However, you must be careful when processing a sorted data set: many types of processing, such as repartitioning, can destroy the sort order of the data. For example, assume you sort a data set on a system with four processing nodes and store the results to a persistent data set. The data set will therefore have four partitions. You then use that data set as input to an operator executing on a different number of nodes, possibly due to node constraints.

Orchestrate automatically repartitions a data set to spread out the data set to all nodes in the system, possibly destroying the sort order of the data. To prevent an
accidental repartitioning of the sorted output data set, the tsort operator sets the preserve-partitioning flag in its output data set. If set, the preserve-partitioning flag prevents an Orchestrate operator using a partitioning method of any from repartitioning the data set. See the chapter on partitioning in Orchestrate in the Orchestrate 7.0 User Guide for more information on the preserve-partitioning flag.

An operator that takes a sorted data set as input could also use the partitioning method same. An operator using this partitioning method does not perform any repartitioning as it reads the input data set; that is, the partitions of an input data set are unchanged by the processing nodes executing the operator.

Using a Sorted Data Set with a Sequential Operator

You must also be careful when using a sequential operator to process a sorted data set. A sequential operator executes on a single processing node to perform its action. Sequential operators always repartition a data set when the data set has more than one partition; therefore, a sequential operator may also destroy the sorting order of its input data set.

If the input to tsort was partitioned using a range partitioner, you can use a partition-sorted data set as input to a sequential operator using an ordered collector to preserve the sort order of the data set. Using this collection method causes all the records from the first partition of a data set to be read first, then all records from the second partition, and so on. See the chapter on using Orchestrate collectors in the Orchestrate 7.0 User Guide for more information.

If the input to tsort was partitioned using a hash partitioner, you can use a partition-sorted data set as input to a sequential operator using a sortmerge collector with the keys. Using this collection method causes the records to be read in sort order.

Passing a Sorted Data Set to an RDBMS: General Information

Orchestrate allows you to read RDBMS data into a data set and to write a data set to an RDBMS table. Note that an RDBMS does not guarantee deterministic ordering behavior unless an SQL operation constrains it to do so. For example, if you write a sorted data set (using tsort or psort) to DB2, then read the data set back, the records of the data set are not guaranteed to be in sorted order. Also, if you read the same DB2 table multiple times, DB2 does not guarantee to deliver the records in the same order every time.

See the chapter on Orchestrate’s interface to your particular RDBMS for more information.

Specifying Sorting Keys

Sorting keys specify the criteria used to perform the sort. The tsort operator allows you to set a primary sorting key and multiple secondary sorting keys.
The `tsort` operator uses the sorting keys to determine the sorting order of a data set. The sort operator first sorts the records by the primary sorting key. If multiple records have the same primary key value, the `tsort` operator then sorts these records by any secondary keys. By default, if all the keys are identical in two records their sort order is undefined. Specifying the `-stable` option to `tsort` guarantees the records are output in the same order in which they were received, at a slight loss in speed.

You must define a single primary sorting key for the `tsort` operator. You may optionally define as many secondary keys as required by your application. Note, however, that each record field can be used only once as a sorting key. Therefore, the total number of primary and secondary sorting keys must be less than or equal to the total number of fields in the record.

The following figure shows four records whose schema contains three fields:

```
<table>
<thead>
<tr>
<th>fName</th>
<th>lName</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td>Smith</td>
<td>42</td>
</tr>
<tr>
<td>Paul</td>
<td>Smith</td>
<td>34</td>
</tr>
<tr>
<td>Mary</td>
<td>Davis</td>
<td>41</td>
</tr>
<tr>
<td>Bob</td>
<td>Jones</td>
<td>42</td>
</tr>
</tbody>
</table>
```

This figure also shows the results of three sorts using different combinations of sorting keys. In this figure, the `fName` and `lName` fields are string fields and the `age` field represents an integer.

Here are `tsort` defaults and optional settings:

- By default, the `tsort` operator uses a case-sensitive algorithm for sorting. This means that uppercase strings appear before lowercase strings in a sorted data set. You can use an option to the `tsort` operator to select case-insensitive sorting.
- By default, the `tsort` operator uses ascending sort order, so that smaller values appear before larger values in the sorted data set. You can use an option to the `tsort` operator to perform descending sorting.
- By default, the `tsort` operator uses ASCII. You can also set an option to `tsort` to specify EBCDIC.
By default, nulls sort low. You can set `tsort` to specify sorting nulls high.

**Data Flow Diagram**

The `tsort` operator:
- Takes a single data set as input.
- Writes its results to a single output data set and sets the `preserve-partitioning` flag in the output data set.
- Has a dynamic input interface schema that allows you to specify as many input key fields as are required by your sort operation.
- Has an input interface schema containing the schema variable `sortRec` and an output interface schema consisting of a single schema variable `sortRec`.
- Does not add any new fields to the sorted records.
- Transfers all key and non-key fields to the output.

**Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td><code>sortRec:*</code>;</td>
</tr>
<tr>
<td></td>
<td>The dynamic input interface schema lets you specify as many input key fields as are required by your sort operation.</td>
</tr>
<tr>
<td>Output interface schema</td>
<td><code>sortRec:*</code>;</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td><code>sortRec -&gt; sortRec without record modification</code></td>
</tr>
</tbody>
</table>
Syntax and Options

The syntax for the `tsort` operator in an `osh` command is shown below:

```
  tsort
  -key field[c | cs] [-ebcdic] [-nulls first | last] [-asc | -desc]
  [-sorted | -clustered] [-param params]
  [-key field[c | cs] [-ebcdic] [-nulls first | last] [-asc | -desc]
  [-sorted | -clustered] [-param params] ...
  [-collation_sequence locale|collation_file_pathname|OFF]
  [-flagKey]
  [-flagCluster]
  [-memory num_megabytes]
  [-stable | -nonstable]
  [-stats]
  [-unique]
```

You must use `-key` to specify at least one sorting key to the operator.

### Table 153  `tsort` Operator Properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>parallel mode: any</td>
</tr>
<tr>
<td>Collection method</td>
<td>sequential mode: any</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>set</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
<tr>
<td>Combinable operator</td>
<td>yes</td>
</tr>
</tbody>
</table>
Table 154  tsort Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
</table>

Specifies a key field for the sort. The first -key defines the primary key field for the sort; lower-priority key fields are supplied on subsequent key specifications.

-**key** requires that field be a field of the input data set.

-**ci** | -**cs** are optional arguments for specifying case-sensitive or case-insensitive sorting. By default, the operator does case-sensitive sorting. This means that uppercase strings appear before lowercase strings in a sorted data set. You can override this default to perform case-insensitive sorting on string fields only.

-**asc** | -**desc** are optional arguments for specifying ascending or descending sorting. By default, the operator uses ascending sort order, so that smaller values appear before larger values in the sorted data set. You can specify descending sorting order instead, so that larger values appear before smaller values in the sorted data set.

-**ebcdic** (string fields only) specifies to use EBCDIC collating sequence for string fields. Note that Orchestrate stores strings as ASCII text; this property only controls the collating sequence of the string.

For example, using the EBCDIC collating sequence, lowercase letters sort before uppercase letters (unless you specify the -ci option to select case-insensitive sorting). Also, the digits 0-9 sort after alphabetic characters. In the default ASCII collating sequence used by the operator, numbers come first, followed by uppercase, then lowercase letters.

-**sorted** specifies that input records are already sorted by this field. The operator then sorts on secondary key fields, if any. This option can increase the speed of the sort and reduce the amount of temporary disk space when your records are already sorted by the primary key field(s) because you only need to sort your data on the secondary key field(s).

-**sorted** is mutually exclusive with -**clustered**; if any sorting key specifies -**sorted**, no key can specify -**clustered**.

If you specify -**sorted** for all sorting key fields, the operator verifies that the input data set is correctly sorted, but does not perform any sorting. If the input data set is not correctly sorted by the specified keys, the operator fails.

-**clustered** specifies that input records are already grouped by this field, but not sorted. The operator then sorts on any secondary key fields. This option is useful when your records are already grouped by the primary key field(s), but not necessarily sorted, and you want to sort your data only on the secondary key field(s) within each group.

continued
- **-key**

  `-clustered` is mutually exclusive with `-sorted`; if any sorting key specifies `-clustered`, no key can specify `-sorted`.

  `-nulls` specifies whether null values should be sorted first or last. The default is first.

  The `-param` suboption allows you to specify extra parameters for a field. Specify parameters using `property=value` pairs separated by commas.

- **-collation_sequence**

  `-collation_sequence locale|collation_file_pathname|OFF`

  This option determines how your string data is sorted. You can:
  
  - Specify a predefined IBM ICU locale
  - Write your own collation sequence using ICU syntax, and supply its `collation_file_pathname`
  - Specify `OFF` so that string comparisons are made using Unicode code-point value order, independent of any locale or custom sequence.

  By default, Orchestrate sorts strings using byte-wise comparisons.

  For more information, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site: [http://oss.software.ibm.com/icu/userguide/Collate_Intro.htm](http://oss.software.ibm.com/icu/userguide/Collate_Intro.htm)

- **-flagCluster**

  `-flagCluster`

  Tells the operator to create the `int8` field `clusterKeyChange` in each output record. The `clusterKeyChange` field is set to 1 for the first record in each group where groups are defined by the `-sorted` or `-clustered` argument to `-key`. Subsequent records in the group have the `clusterKeyChange` field set to 0.

  You must specify at least one sorting key field that uses either `-sorted` or `-clustered` to use `-flagCluster`, otherwise the operator ignores this option.

- **-flagKey**

  Optionally specify whether to generate a flag field that identifies the key-value changes in output.

- **-memory**

  `-memory num_megabytes`

  Causes the operator to restrict itself to `num_megabytes` megabytes of virtual memory on a processing node.

  `-memory` requires that `1 < num_megabytes <` the amount of virtual memory available on any processing node. Acential recommends that `num_megabytes` be smaller than the amount of physical memory on a processing node.
Example: Using a Sequential tsort Operator

This section contains an example using a sequential tsort operator, as shown in the following figure:

```
input data set
schema:
a:int32;
b:int32;
c:int16;
d:sfloat;
e:string[10];
```

This step uses a sequential tsort operator to completely sort the records of an input data set. The primary key field is `a`, the secondary sorting key field is `e`.

Since the tsort operator runs by default in parallel, you use the `{seq}` framework argument to configure the operator to run sequentially.
Shown below is the `osh` command line for this step:

```bash
$ osh 'tsort -key a -key e -stable [seq] < unSortedDS.ds > sortedDS.ds '
```

**Example: Using a Parallel tsort Operator**

A parallel `tsort` operator runs on multiple processing nodes in your system to sort the records within each partition of a data set. The default execution mode of the operator is parallel, using the *any* partitioning method. However, you typically use a partitioning operator with the `tsort` operator to set an explicit partitioning method.

Choose a partitioning method that is correct for the sorting operation. For example, assume that you are sorting records in a data set based on the last name field of each record. If you randomly allocate records into any partition, records with similar last names are not guaranteed to be in the same partition and are not, therefore, processed by the same node. Similar records can be sorted by an operator only if they are in the same partition of the data set.

A better method of partitioning data in this case is to hash the records by the first five or six characters of the last name. All records containing similar names would be in the same partition and, therefore, would be processed by the same node. The `tsort` operator could then compare the entire last names, first names, addresses, or other information in the records, to determine the sorting order of the records.

For example:

```plaintext
record ( fname:string[30]; lname:string[30]; )
  ... | modify -spec "lname_hash:string[6] = substring[0,6](lname)"
  | hash -key lname_hash
  | tsort -key lname | ...
```

Orchestrate supplies a `hash` partitioner operator that allows you to hash records by one or more fields. See “The hash Partitioner” on page 27-5 for more information on the `hash` operator. You can also use any one of the supplied Orchestrate partitioning methods.

The following example is a modification of the previous example, “Example: Using a Sequential tsort Operator” on page 31-11, to execute the `tsort` operator in parallel using a `hash` partitioner operator. In this example, the `hash` operator partitions records using the integer field `a`, the primary sorting key. Thus all
records containing the same value for field a will be assigned to the same partition. The figure below shows this example:

```
$ osh "hash -key a -key e < unSortedDS.ds | tsort -key a -key e > sortedDS.ds"
```

**Performing a Total Sort**

The previous section showed an example of a parallel sorter using the `hash` partitioner to sort the partitions of a data set. When using a hash partitioner, all records containing the same key values would be in the same partition and therefore are processed and sorted by the same node. The partition-sorted data set can be collected into a totally sorted sequential data set using a sort merge collector with the same keys.

The `osh` command for this is:

```
$ osh "hash -key a -key e < unsortedDS.ds | tsort -key a -key e | sortmerge -key a -key e > sorted.ds"
```

In contrast to a partition sort, you can also use the `tsort` operator to perform a total sort on a parallel data set. A totally sorted parallel data set means the records in each partition of the data set are ordered, and the partitions themselves are ordered.
The following figure shows a total sort performed on a data set with two partitions:

In this example, the partitions of the output data set contain sorted records, and the partitions themselves are sorted. A total sort requires that all similar and duplicate records be located in the same partition of the data set. Similarity is based on the key fields in a record.

Because each partition of a data set is sorted by a single processing node, another requirement must also be met before a total sort can occur. Not only must similar records be in the same partition, but the partitions themselves should be approximately equal in size so that no one node becomes a processing bottleneck.

To meet these two requirements, use the range partitioner on the input data set before performing the actual sort. The range partitioner guarantees that all records with the same key fields are in the same partition, and calculates partition boundaries based on the key fields, in order to evenly distribute records across all partitions. See the next section for more information on using the range partitioner.

You need to perform a total sort only when your application requires a completely ordered data set, either as output or as a preprocessing step. For example, you may want to perform a total sort on a mailing list so that the list is sorted by zip code, then by last name within zip code.
For most applications, a partition sort is the correct sorting approach. For example, if you are sorting a data set as a preprocessing step to remove duplicate records, you use a hash partitioner with a partition sort to partition the data set by the key fields; a total sort is unnecessary. A hash partitioner assigns records with the same key fields to the same partition, then sorts the partition. A second operator can compare adjacent records in a partition to determine if they have the same key fields to remove any duplicates.
Example: Performing a Total Sort

This section contains an example that uses the range partitioner and the tsort operator to perform a total sort on a parallel data set. The figure shows the data flow for this example:

- SampleStep
  - Sample
    - Tsort
      - key a ...
        (sequential mode)
    - Writerangemap
  - Range
    - Tsort
      - key a ...
        (parallel mode)
  - Sorted, sampled file for creating the range partitioner
  - Input data set
    schema:
    - a: int32
    - b: int8
    - c: string[5]
    - d: int16
    - e: string
As the figure shows, this example uses two steps. The first step creates the sorted sample of the input data set used to create the range map required by the range partitioner. The second step performs the parallel sort using the range partitioner.

Orchestrate supplies the UNIX command line utility, makerangemap, that performs the first of these two steps for you to create the sorted, sampled file, or range map. See “The range Partitioner” on page 27-15 for more information on makerangemap.

Shown below are the commands for the two steps shown above using makerangemap:

```
$ makerangemap -rangemap sampledData -key a -key e inDS.ds
$ osh 'range -sample sampledData -key a -key c < inDS.ds | tsort -key a -key e > sortDS.ds'
```

### The psort Operator

**Introduction**

Many data mining applications require the records of a data set to be sorted. The sorting operation is usually based on one or more fields of the records in the data set. For example, you may want to sort a data set by a zip code field, then by last name within the zip code.

The partition sort operator, psort, sorts the records of a data set. The execution mode of the operator, parallel or sequential, determines how the operator sorts the data set. The first section of this chapter describes these different execution modes.

Fields used to sort records are referred to as key fields of the sort. In the example mentioned above, the zip code field of each record would be the primary sorting key, and the last name field would be the secondary sorting key. See “Specifying Sorting Keys” on page 31-21 for more information on sorting keys.

The partition sort operator can use either SyncSort UNIX Version, Release 1, or the UNIX sort command to perform the actual sorting operation. See “Configuring the Partition Sort Operator” on page 31-19 for information on configuring the operator.

Orchestrate also provides the sorting operator tsort. That operator does not require you to have a copy of SyncSort. It is described in “The tsort Operator” on page 31-2.
Performing a Partition Sort

Orchestrate includes the partition sort operator, \texttt{psort}, which you can use to sort the records of a data set. The \texttt{psort} operator can execute as either a sequential or a parallel operator. The execution mode of the \texttt{psort} operator determines its action:

- **Sequential mode**: The \texttt{psort} operator executes on a single processing node to sort an entire data set. On completion, the records of the data set are sorted completely.

- **Parallel mode**: The \texttt{psort} operator executes on multiple processing nodes in your system. On completion, the records within each partition of the data set are sorted.

The following figure shows the difference between these two operating modes:

![Sequential vs Parallel Sorting Diagram]

The \texttt{psort} operator on the left side of this figure runs sequentially to completely sort the records of a data set. Typically, you use the \texttt{psort} operator sequentially when you require a total sort of a data set, or when you create a sequential application.

The \texttt{psort} operator on the right side of this figure runs in parallel and sorts records within a partition. Remember that a parallel operator executes on multiple nodes in a system, where each node receives a partition of a data set. A parallel \texttt{psort} operator outputs partitions, where the records are sorted within the partition.
Typically you use a parallel \texttt{psort} operator as part of a series of operators that requires sorted partitions. For example, you can combine a sort operator with an operator that removes duplicate records from a data set. After the partitions of a data set are sorted, duplicate records are adjacent in a partition.

In order to perform a parallel sort, you specify a partitioning method to the operator, enabling you to control how the records of a data set are partitioned before the sort. For example, you could partition records by a name field, so that all records whose name field begins with the same one-, two-, or three-letter sequence are assigned to the same partition. See “Example: Using a Parallel Partition Sort Operator” on page 31-27 for more information.

If you combine a parallel sort with a range partitioner, you can perform a total sort. A totally sorted data set output is completely ordered. See “Performing a Total Sort” on page 31-29 for more information.

The example shown above for a parallel \texttt{psort} operator illustrates an input data set and an output data set containing the same number of partitions. However, the number of partitions in the output data set is determined by the number of processing nodes in your system configured to run the \texttt{psort} operator. The \texttt{psort} operator repartitions the input data set such that the output data set has the same number of partitions as the number of processing nodes. See the next section for information on configuring the \texttt{psort} operator.

\section*{Configuring the Partition Sort Operator}

Several UNIX sorting commands, including the UNIX \texttt{sort} command and SyncSort UNIX Version, Release 1 or Release 2, exist for single-processor workstations. The \texttt{psort} operator can use either the UNIX \texttt{sort} command or SyncSort (if it is installed) to sort the records of a data set. The \texttt{psort} operator can sort either in a parallel or a sequential mode for UNIX sort and SyncSort.

To use the sequential mode of the \texttt{psort} operator, you require only a single SyncSort license. To execute the \texttt{psort} operator in parallel, you must have a separate SyncSort license for every processing node that executes the \texttt{psort} operator.

Records processed by the \texttt{psort} operator must be less than 32 KB in length. If the \texttt{psort} operator encounters a record longer than 32 KB, the operator aborts execution. The step containing the operator fails, and an error message is written to the error log. Any other errors detected by SyncSort during the execution of the \texttt{psort} operator are written to the error log.

Your selection for the sorting utility, UNIX sort or SyncSort, is based on the setting of the \texttt{APT_SYNCSORT_DIR} environment variable, or on the \texttt{-sorter} option (the preferred method of selection). If \texttt{APT_SYNCSORT_DIR} is not defined, the operator uses UNIX sort.
If you want to use the `-sorter syncsort` option, you must set the `APT_SYNCSORT_DIR` environment variable to the location of SyncSort on your processing nodes. Typically, SyncSort is installed in the same location on every node. Therefore, you can set `APT_SYNCSORT_DIR` on the system from which you invoke your Orchestrate application. Orchestrate then copies `APT_SYNCSORT_DIR` to all nodes that execute the `psort` operator.

For example, if you installed SyncSort in `/usr/local/syncsort`, you must set the environment variable as shown below.

For Korn and Bourne shells:

```bash
APT_SYNCSORT_DIR=/usr/local/syncsort # in your .profile or .kshrc
export APT_SYNCSORT_DIR
```

If SyncSort is installed in a different location on each node, you can use an Orchestrate startup script to set `APT_SYNCSORT_DIR` to a different value on each node. See the Orchestrate 7.0 Installation and Administration Manual for more information on how to do this.

Though it is not required for normal execution of the `psort` operator, you may want to create node pools and resources for use by the `psort` operator. See the Orchestrate 7.0 Installation and Administration Manual for more information on configuring SyncSort.

### Using a Sorted Data Set

You perform a partition sort for several reasons. For example, you may want to sort a data set by a zip code field, then by last name within the zip code. Once you have sorted the data set, you can filter the data set by comparing adjacent records and removing any duplicates.

However, you must be careful when processing a sorted data set: many types of processing, such as repartitioning, can destroy the sorting order of the data. For example, assume you sort a data set on a system with four processing nodes and store the results to a persistent data set. The data set will therefore have four partitions. You then use that data set as input to an operator executing on a different number of nodes. This may be because your system does not have a SyncSort license for every node, or because of a node failure.

Orchestrate automatically repartitions a data set to spread out the data set to all nodes in the system, possibly destroying the sort order of the data. To prevent an accidental repartitioning of the sorted output data set, the `psort` operator sets the `preserve-partitioning` flag in its output data set. If set, the `preserve-partitioning` flag prohibits an Orchestrate operator using a partitioning method of any from repartitioning the data set. See the chapter on partitioning in Orchestrate in the Orchestrate 7.0 User Guide for more information on the `preserve-partitioning` flag.
An operator that takes a sorted data set as input could also use the partitioning method same. An operator using this partitioning method does not perform any repartitioning as it reads the input data set; that is, the partitions of an input data set are unchanged by the processing nodes executing the operator.

Using a Sorted Data Set with a Sequential Operator

You must also be careful when using a sequential operator to process a sorted data set. A sequential operator executes on a single processing node to perform its action. Sequential operators always repartition a data set when the data set has more than one partition; therefore, a sequential operator may also destroy the sorting order of its input data set.

You can use a partition-sorted data set as input to a sequential operator using the collection method sortmerge with the appropriate keys to preserve the sorting order of the data set. A sequential operator using this collection method reads all records from the first partition of a data set, then all records from the second partition, and so on. See the chapter on using Orchestrate collectors in the Orchestrate 7.0 User Guide for more information.

Passing a Sorted Data Set to an RDBMS: General Information

Orchestrate allows you to read RDBMS data into a data set and to write a data set to an RDBMS table. Note that an RDBMS does not guarantee deterministic ordering behavior unless the SQL operation used to retrieve the records constrains it to do so. For example, if you write a data set (using psort or tsort) to DB2, then read the data set back, the records of the data set are not guaranteed to be in the original order. Also, if you read the same DB2 table multiple times, DB2 does not guarantee to deliver the records in the same order every time.

See the chapter on your particular RDBMS for more information.

Specifying Sorting Keys

Sorting keys specify the criteria used to perform the sort. The psort operator allows you to set a primary sorting key and multiple secondary sorting keys.

The psort operator uses the sorting keys to determine the sorting order of a data set. The sort operator first sorts the records by the primary sorting key. If multiple records have the same primary key value, the psort operator then sorts these records by any secondary keys.

You must define a single primary sorting key for the psort operator. You may optionally define as many secondary keys as required by your application. Note, however, that each record field can be used only once as a sorting key. Therefore, the total number of primary and secondary sorting keys must be less than or equal to the total number of fields in the record.
The following figure shows four records whose schema contains three fields:

<table>
<thead>
<tr>
<th>fName</th>
<th>lName</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td>Smith</td>
<td>42</td>
</tr>
<tr>
<td>Paul</td>
<td>Smith</td>
<td>34</td>
</tr>
<tr>
<td>Mary</td>
<td>Davis</td>
<td>41</td>
</tr>
<tr>
<td>Bob</td>
<td>Jones</td>
<td>42</td>
</tr>
</tbody>
</table>

This figure also shows the results of three sorts using different combinations of sorting keys. In this figure, the lName field represents a string field and the age field represents an integer. By default, the psort operator uses a case-sensitive algorithm for sorting. This means that uppercase strings appear before lowercase strings in a sorted data set. You can use an option to the psort operator to select case-insensitive sorting.

By default, the psort operator uses ascending sort order, so that smaller values appear before larger values in the sorted data set. You can use an option to the psort operator to select descending sorting.

### Data Flow Diagram

The psort operator:
- Takes a single data set as input.
- Writes its results to a single output data set and sets the `preserve-partitioning` flag in the output data set.
- Has a dynamic input interface schema that allows you to specify as many input key fields as are required by your sort operation.
- Has an input interface schema containing the schema variable `inRec` and an output interface schema consisting of a single schema variable `outRec`.
- Does not add any new fields to the sorted records.

**Properties**

Table 155  **psort Operator Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td><code>inRec:*</code>&lt;br&gt;The dynamic input interface schema lets you specify as many input key fields as are required by your sort operation.</td>
</tr>
<tr>
<td>Output interface schema</td>
<td><code>outRec:*</code></td>
</tr>
<tr>
<td>Transfer behavior</td>
<td><code>inRec -&gt; outRec</code> without record modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>sequential or parallel</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>parallel mode: any</td>
</tr>
<tr>
<td>Collection method</td>
<td>sequential mode: any</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>set</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>
Syntax and Options

The syntax for the psort operator in an osh command is shown below. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose the value in single or double quotes.

```
psort
  -key field_name [-ci | -cs] [-asc | -desc] [-ebcdic]
  [-key field_name [-ci | -cs] [-asc | -desc] [-ebcdic]] ...
  [-extraOpts syncsort_options]
  [-memory num_megabytes]
  [-sorter unix | syncsort]
  [-stable]
  [-stats]
  [-unique]
  [-workspace workspace]
```

You must use `-key` to specify at least one sorting key to the operator. You use the `-part` option to configure the operator to run in parallel, and the `-seq` option to specify sequential operation.

If you include the `-ebcdic` option, you must also include the `-sorter` option with a value of `syncsort`. When you do not include the `-sorter` option, the default sort is `unix` which is incompatible with the EBCDIC collation sequence.

Example usage:
```
psort -sorter syncsort -key a -ebcdic
```
Table 156  **psort Operator Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-key</td>
<td>-key  field_name[-ci</td>
</tr>
<tr>
<td></td>
<td>If the -ebcdic suboption is specified, you must also include the -sorter option with a value of syncsort.</td>
</tr>
<tr>
<td></td>
<td>The -key option specifies a key field for the sort. The first -key option defines the primary key field for the sort; lower-priority key fields are supplied on subsequent -key specifications.</td>
</tr>
<tr>
<td></td>
<td>You must specify this option to psort at least once.</td>
</tr>
<tr>
<td>-key</td>
<td>requires that field_name be a field of the input data set. The data type of the field must be one of the following data types:</td>
</tr>
<tr>
<td></td>
<td>• int8 , int16 , int32 , int64 , uint8, uint16, uint32 , uint64</td>
</tr>
<tr>
<td></td>
<td>• sfloat, dfloat</td>
</tr>
<tr>
<td></td>
<td>• string[n] , where n is an integer literal specifying the string length</td>
</tr>
<tr>
<td>-ci or -cs</td>
<td>are optional arguments for specifying case-sensitive or case-insensitive sorting. By default, the operator uses a case-sensitive algorithm for sorting. This means that uppercase strings appear before lowercase strings in a sorted data set. You can override this default to perform case-insensitive sorting on string fields.</td>
</tr>
<tr>
<td>-asc or -desc</td>
<td>specify optional arguments for specifying ascending or descending sorting. By default, the operator uses ascending sort order, so that smaller values appear before larger values in the sorted data set. You can use descending sorting order as well, so that larger values appear before smaller values in the sorted data set.</td>
</tr>
<tr>
<td>-ebcdic</td>
<td>(string fields only) specifies to use EBCDIC collating sequence for string fields. Note that Orchestrate stores strings as ASCII text; this property only controls the collating sequence of the string.</td>
</tr>
<tr>
<td></td>
<td>If you include the -ebcdic option, you must also include the -sorter option with a value of syncsort. When you do not include the -sorter, the default sort is unix which is incompatible with the EBCDIC collation sequence.</td>
</tr>
<tr>
<td></td>
<td>When you use the EBCDIC collating sequence, lower-case letters sort before upper-case letters (unless you specify the -ci option to select case-insensitive sorting). Also, the digits 0-9 sort after alphabetic characters. In the default ASCII collating sequence used by the operator, numbers come first, followed by uppercase, then lower case letters.</td>
</tr>
<tr>
<td>-extraOpts</td>
<td>-extraOpts  syncsort_options</td>
</tr>
<tr>
<td></td>
<td>Specifies command-line options passed directly to SyncSort. syncsort_options contains a list of SyncSort options just as you would normally type them on the SyncSort command line.</td>
</tr>
</tbody>
</table>
Table 156  psort Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-memory</td>
<td>-memory num_megabytes</td>
</tr>
<tr>
<td></td>
<td>Causes the operator to restrict itself to num_megabytes megabytes of virtual memory on a processing node.</td>
</tr>
<tr>
<td>-memory</td>
<td>requires that 1 &lt; num_megabytes &lt; the amount of virtual memory available on any processing node. Acential recommends that num_megabytes be smaller than the amount of physical memory on a processing node.</td>
</tr>
<tr>
<td>-part</td>
<td>-part partitioner</td>
</tr>
<tr>
<td></td>
<td>This is a deprecated option. It is included for backward compatibility.</td>
</tr>
<tr>
<td>-seq</td>
<td>-seq</td>
</tr>
<tr>
<td></td>
<td>This is a deprecated option. It is included for backward compatibility.</td>
</tr>
<tr>
<td>-sorter</td>
<td>-sorter unix</td>
</tr>
<tr>
<td></td>
<td>Specifies the sorting utility used by the operator. The default is unix, corresponding to the UNIX sort utility.</td>
</tr>
<tr>
<td>-stable</td>
<td>-stable</td>
</tr>
<tr>
<td></td>
<td>Specifies that this sort is stable. A stable sort guarantees not to rearrange records that are already sorted properly in a data set.</td>
</tr>
<tr>
<td></td>
<td>The default sorting method is unstable. In an unstable sort, no prior ordering of records is guaranteed to be preserved by the sorting operation, but may processing may be slightly faster.</td>
</tr>
<tr>
<td>-stats</td>
<td>-stats</td>
</tr>
<tr>
<td></td>
<td>Configures psort to generate output statistics about the sorting operation and to print them to the screen.</td>
</tr>
<tr>
<td>-unique</td>
<td>-unique</td>
</tr>
<tr>
<td></td>
<td>Specifies that if multiple records have identical sorting key values, only one record is retained. If stable is set, then the first record is retained.</td>
</tr>
<tr>
<td>-workspace</td>
<td>-workspace workspace</td>
</tr>
<tr>
<td></td>
<td>Optionally supply a string indicating the workspace directory to be used by the sorter</td>
</tr>
</tbody>
</table>
Example: Using a Sequential Partition Sort Operator

This section contains an example using a sequential `psort` operator, as shown in the following figure:

This step uses a sequential `psort` operator to completely sort the records of an input data set. The primary key field is `a`; the secondary sorting key field is `e`.

By default, the `psort` operator executes sequentially; you do not have to perform any configuration actions.

Note that case-sensitive and ascending sort are the defaults.

Shown below is the `osh` command line for this step:

```
$ osh "psort -key a -key e -stable [seq] < unSortedDS.ds > sortedDS.ds"
```

Example: Using a Parallel Partition Sort Operator

A parallel `psort` operator executes on multiple processing nodes in your system to sort the records within each partition of a data set. To use the `psort` operator to execute the sort in parallel, you must specify a partitioning method for the operator. It is by specifying the partitioning method that you configure the operator to run in parallel.

Choose a partitioning method that is correct for the sorting operation. For example, assume that you are sorting records in a data set based on the last-name field of each record. If you randomly allocate records into any partition, records with similar last names are not guaranteed to be in the same partition and are not,
therefore, processed by the same node. Similar records can be sorted by an
operator only if they are in the same partition of the data set.

A better method of partitioning data in this case would be to hash the records by
the first five or six characters of the last name. All records containing similar
names would be in the same partition and, therefore, would be processed by the
same node. The \texttt{psort} operator could then compare the entire last names, first
names, addresses, or other information in the records, to determine the sorting
order of the records.

For example:

\begin{verbatim}
record { fname:string[30]; lname:string[30]; }
... | modify -spec "lname_hash:string[6] = substring[0,6](lname)"
| hash -key lname_hash
| tsort -key lname | ...
\end{verbatim}

Orchestrate supplies a \texttt{hash} partitioner operator that allows you to hash records
by one or more fields. See “The hash Partitioner” on page 27-5 for more
information on the \texttt{hash} operator. You can also use any one of the supplied
Orchestrate partitioning methods.

The following example is a modification of the previous example, “Example:
Using a Sequential Partition Sort Operator” on page 31-27, to execute the \texttt{psort}
operator in parallel using a \texttt{hash} partitioner operator. In this example, the \texttt{hash}
operator partitions records using the integer field \texttt{a}, the primary sorting key.
Therefore, all records containing the same value for field \texttt{a} are assigned to the
same partition. The figure below shows this example:
To configure the `psort` operator in `osh` to execute in parallel, use the `[par]` annotation. Shown below is the `osh` command line for this step:

```
$ osh " hash -key a -key e < unSortedDS.ds |
    psort -key a -key e [par] > sortedDS.ds"
```

### Performing a Total Sort

The previous section showed an example of a parallel sorter using the hash partitioner to sort the partitions of a data set. When you use a hash partitioner, all records containing the same hash key values are in the same partition and are therefore processed and sorted by the same node.

In contrast to a partition sort, you can also use the `psort` operator to perform a total sort.

The following figure shows a total sort performed on a data set with two partitions:

```
input data set
Dick
Tom
Mary
Bob
Stacy
Nick
Bob
Sue
Ted
Jack
Harry
Patsy

input partitions
Dick
Tom
Mary
Bob
Stacy
Nick

Bob
Sue
Ted
Jack
Harry
Patsy

sorted output data set
Bob
Bob
Dick
Harry
Jack
Mary
Nick
Patsy
Stacy
Sue
Ted
Tom

partitions after sorting
Bob
Bob
Dick
Harry
Jack
Mary
Nick
Patsy
Stacy
Sue
Ted
Tom

arrow indicates sorting order of partitions
```

In this example, the partitions of the output data set contain sorted records, and the partitions themselves are sorted. As you can see in this example, a total sort requires that all similar and duplicate records are located in the same partition of the data set. Similarity is based on the key fields in a record.

Because each partition of a data set is sorted by a single processing node, another requirement must also be met before a total sort can occur. Not only must similar
records be in the same partition, but the partitions themselves should be approximately equal in size so that no one node becomes a processing bottleneck. To meet these two requirements, you use the range partitioner on the input data set before performing the actual sort.

The range partitioner guarantees that all records with the same key fields are in the same partition, but it does more. The range partitioner also calculates partition boundaries, based on the sorting keys, in order to evenly distribute records to the partitions. All records with sorting keys values between the partition boundaries are assigned to the same partition so that the partitions are ordered and that the partitions are approximately the same size. See the next section for more information on using the range partitioning operator.

You need to perform a total sort only when your application requires a completely ordered data set, either as output or as a preprocessing step. For example, you may want to perform a total sort on a mailing list so that the list is sorted by zip code, then by last name within zip code.

For most applications, a partition sort is the correct sorting component. For example, if you are sorting a data set as a preprocessing step to remove duplicate records, you use a hash partitioner with a partition sort to partition the data set by the sorting key fields; a total sort is unnecessary. A hash partitioner assigns records with the same sorting key fields to the same partition, then sorts the partition. A second operator can compare adjacent records in a partition to determine if they have the same key fields to remove any duplicates.

**Range Partitioning**

The range partitioner guarantees that all records with the same key field values are in the same partition and it creates partitions that are approximately equal in size so that all processing nodes perform an equal amount of work when performing the sort. In order to do so, the range partitioner must determine distinct partition boundaries and assign records to the correct partition.

To use a range partitioner, you first sample the input data set to determine the distribution of records based on the sorting keys. From this sample, the range partitioner determines the partition boundaries of the data set. The range partitioner then repartitions the entire input data set into approximately equal-sized partitions in which similar records are in the same partition.
The following example shows a data set with two partitions:

<table>
<thead>
<tr>
<th>input partitions</th>
<th>Dick</th>
<th>Tom</th>
<th>Mary</th>
<th>Bob</th>
<th>Stacy</th>
<th>Nick</th>
<th>Bob</th>
<th>Sue</th>
<th>Ted</th>
<th>Jack</th>
<th>Harry</th>
<th>Patsy</th>
</tr>
</thead>
<tbody>
<tr>
<td>partitions after repartitioning</td>
<td>Dick</td>
<td>Tom</td>
<td>Mary</td>
<td>Bob</td>
<td>Stacy</td>
<td>Nick</td>
<td>Bob</td>
<td>Sue</td>
<td>Ted</td>
<td>Stacey</td>
<td>Nick</td>
<td>Patsy</td>
</tr>
<tr>
<td>partitions after sorting</td>
<td>Bob</td>
<td>Sue</td>
<td>Ted</td>
<td>Stacey</td>
<td>Nick</td>
<td>Patsy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This figure shows range partitioning for an input data set with four partitions and an output data set with four:

<table>
<thead>
<tr>
<th>input partitions</th>
<th>Dick</th>
<th>Tom</th>
<th>Mary</th>
<th>Bob</th>
<th>Stacy</th>
<th>Nick</th>
<th>Bob</th>
<th>Sue</th>
<th>Ted</th>
<th>Jack</th>
<th>Harry</th>
<th>Patsy</th>
</tr>
</thead>
<tbody>
<tr>
<td>partitions after repartitioning</td>
<td>Dick</td>
<td>Bob</td>
<td>Mary</td>
<td>Jack</td>
<td>Harry</td>
<td>Nick</td>
<td>Patsy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>partitions after sorting</td>
<td>Bob</td>
<td>Sue</td>
<td>Ted</td>
<td>Stacey</td>
<td>Nick</td>
<td>Patsy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In this case, the range partitioner calculated the correct partition size and boundaries, then assigned each record to the correct partition based on the sorting key fields of each record. See “The range Partitioner” on page 27-15 for more information on range partitioning.

**Example: Performing a Total Sort**

This section contains an example that uses the range partitioner and the psort operator to perform a total sort on a data set. The following figure shows the data-flow model for this example:
As you can see in this figure, this example uses two steps. The first step creates the sorted sample of the input data set used to create the range map required by the `range` partitioner. The second step performs the parallel sort using the `range` partitioner and the `psort` operator.

Orchestrate supplies the UNIX command line utility, `makerangemap`, that performs the first of these two steps for you to create the sorted, sampled file, or...
range map. See “The range Partitioner” on page 27-15 for more information on `makerangemap`.

Shown below are the commands for the two steps shown above using `makerangemap`:

```bash
$ makerangemap -rangemap sampledData -key a -key e inDS.ds
$ osh 'range -sample sampledData -key a -key c < inDS.ds |
  psort -key a -key e > sortDS.ds''
```
The SAS Interface Library

Tells you how to run SAS applications in parallel or sequentially using Orchestrate’s scalable application environment.

Overview of the SAS Interface Library 32 2
Using Orchestrate to Run SAS Code 32 4
  Writing SAS Programs 32 4
  Using SAS on Sequential and Parallel Systems 32 4
  Pipeline Parallelism and SAS 32 6
  Configuring Your System to Use the SAS Interface Operators 32 6
  An Example Data Flow 32 7
Representing SAS and Non-SAS Data in Orchestrate 32 8
  Orchestrate Data Set Format 32 8
  Sequential SAS Data Set Format 32 8
  Parallel SAS Data Set Format 32 8
  Orchestrated SAS Data Set Format 32 9
Getting Input from a SAS Data Set 32 9
Getting Input from an Orchestrate Data Set or an Orchestrated SAS Data Set 32 10
Converting between Data Set Types 32 11
  Converting Orchestrate Data to SAS Data 32 11
  Converting SAS Data to Orchestrate Data 32 13
An Orchestrate Example 32 13
Parallelizing SAS Steps 32 15
  Executing DATA Steps in Parallel 32 15
  Example Applications 32 16
Executing PROC Steps in Parallel 32 20
  Example Applications 32 21
Some Points to Consider in Parallelizing SAS code 32 24
  Which SAS Programs Benefit from Parallelization? 32 24
  Rules of Thumb 32 25
Using Orchestrate to Do ETL 32 28
The SAS Interface Operators 32 30
Overview of the SAS Interface Library

Orchestrate makes it possible for you to execute SAS applications using the computing power and data handling capabilities of Orchestrate's parallel processing system. You need to make only minor modifications to your existing SAS applications for execution within Orchestrate. The section “Parallelizing SAS Steps” on page 32-15 describes these modifications.
The Orchestrate SAS Interface Library contains four operators:

- **“The sasin Operator” on page 32-38**
  This operator converts a standard Orchestrate data set into an Orchestrated SAS data set capable of being processed in parallel by the `sas` and `sasout` operators. The `sasin` operator options include an option for defining the input data-set schema.

- **“The sas Operator” on page 32-44**
  This operator allows you to execute part or all of a SAS application in parallel. It takes SAS code in the form of DATA and PROC steps as its argument.

- **“The sasout Operator” on page 32-51**
  The `sasout` operator converts the Orchestrated SAS data set output by the `sas` operator to the standard Orchestrate data set format. The operator has a number of options including the `-schema` and `-schemaFile` options. You use either of these options to specify the schema of the output Orchestrate data set.

- **“The sascontents Operator” on page 32-55**
  This operator generates a report about a self-describing input data set. The report is similar to the report generated by the SAS procedure PROC CONTENTS.

The remainder of this chapter is divided into three major sections:

- **“Using Orchestrate to Run SAS Code” on page 32-4** describes how to structure your SAS programs to take advantage of Orchestrate’s parallel processing capabilities.

- **“Using Orchestrate to Do ETL” on page 32-28** describes the process of extracting data from a SAS file.

- **“The SAS Interface Operators” on page 32-30** tells you how to use the options of the Orchestrate SAS-interface operators.
Using Orchestrate to Run SAS Code

Orchestrate lets SAS users optimally exploit the performance potential of parallel relational database management systems (RDBMS) running on scalable hardware platforms. Orchestrate extends SAS by coupling the parallel data transport facilities of Orchestrate with the rich data access, manipulation, and analysis functions of SAS.

While Orchestrate allows you to execute your SAS code in parallel, sequential SAS code can also take advantage of Orchestrate to increase system performance by making multiple connections to a parallel database for data reads and writes, as well as through pipeline parallelism.

The SAS system consists of a powerful programming language and a collection of ready-to-use programs called procedures (PROCS). This section introduces SAS application development and explains the differences in execution and performance of sequential and parallel SAS programs.

Writing SAS Programs

You develop SAS applications by writing SAS programs. In the SAS programming language, a group of statements is referred to as a SAS step. SAS steps fall into one of two categories: DATA steps and PROC steps.

SAS DATA steps usually process data one row at a time and do not look at the preceding or following records in the data stream. This makes it possible for the sas operator to process data steps in parallel. SAS PROC steps, however, are precompiled routines which cannot always be parallelized.

Using SAS on Sequential and Parallel Systems

The SAS system is available for use on a variety of computing platforms, including single-processor UNIX workstations. Because SAS is written as a sequential application, Orchestrate can provide a way to exploit the full power of scalable computing technology within the traditional SAS programming environment.

For example, the left side of the next figure shows a sequential SAS application that reads its input from an RDBMS and writes its output to the same database. This application contains a number of sequential bottlenecks that can negatively impact its performance.
In a typical client/server environment, a sequential application such as SAS establishes a single connection to a parallel RDBMS in order to access the database. Therefore, while your database may be explicitly designed to support parallel access, SAS requires that all input be combined into a single input stream. One effect of single input is that often the SAS program can process data much faster than it can access the data over the single connection, therefore the performance of a SAS program may be bound by the rate at which it can access data. In addition, while a parallel system, either MPP or SMP, contains multiple CPUs, a single SAS job in SAS itself executes on only a single CPU; therefore, much of the processing power of your system is unused by the SAS application.

In contrast, Orchestrate's parallel processing model, shown above on the right, allows the database reads and writes, as well as the SAS program itself, to run simultaneously, reducing or eliminating the performance bottlenecks that might otherwise occur when SAS is run on a parallel computer.

Orchestrate enables SAS users to:

- Access, for reading or writing, large volumes of data in parallel from parallel relational databases, with much higher throughput than is possible using PROC SQL.
- Process parallel streams of data with parallel instances of SAS DATA and PROC steps, enabling scoring or other data transformations to be done in parallel with minimal changes to existing SAS code.
- Store large data sets in parallel, circumventing restrictions on data-set size imposed by your file system or physical disk-size limitations. Parallel data
sets are accessed from SAS programs in the same way as conventional SAS data sets, but at much higher data I/O rates.

• Realize the benefits of pipeline parallelism, in which some number of Orchestrate sas operators run at the same time, each receiving data from the previous process as it becomes available.

Pipeline Parallelism and SAS

Orchestrate applications containing multiple sas operators take advantage of pipeline parallelism. With pipeline parallelism, all Orchestrate sas operators in your application run at the same time. A sas operator may be active, meaning it has input data available to be processed, or it may be blocked as it waits to receive input data from a previous sas operator or other Orchestrate operator.

Steps that are processing on a per-record or per-group basis, such as a DATA step or a PROC MEANS with a BY clause, can feed individual records or small groups of records into downstream sas operators for immediate consumption, bypassing the usual write to disk.

Steps that process an entire data set at once, such as a PROC SORT, can provide no output to downstream steps until the sort is complete. The pipeline parallelism enabled by Orchestrate allows individual records to feed the sort as they become available. The sort is therefore occurring at the same time as the upstream steps. However, pipeline parallelism is broken between the sort and the downstream steps, since no records are output by the PROC SORT until all records have been processed. A new pipeline is initiated following the sort.

Configuring Your System to Use the SAS Interface Operators

To enable use of the SAS interface operators, you or your Orchestrate administrator should specify several SAS-related resources in your configuration file. You need to specify the following information:

• The location of the SAS executable;
• Optionally, a SAS work disk, which is defined as the path of the SAS work directory;
• Optionally, a disk pool specifically for parallel SAS data sets, called sasdataset.

Here is an example of each of these declarations:

```
resource sas ’/usr/sas612/’ { }
resource sasworkdisk ’/usr/sas/work/’ { }
resource disk ’/data/sas/’ {pool ’sasdataset’}
```
The resource names `sas` and `sasworkdisk` and the disk pool name `sasdataset` are reserved words. For more detailed information on configuration files and their contents and syntax, and in particular on configuring Orchestrate for use with SAS, see the Orchestrate 7.0 Installation and Administration Manual.

**An Example Data Flow**

This example shows a SAS application reading data in parallel from DB2. The Orchestrate `db2read` operator streams data from DB2 in parallel and converts it automatically into the standard Orchestrate data set format. The `sasin` operator then converts it, by default, into an internal Orchestrated SAS data set usable by the `sas` operator. Finally, the `sasout` operator inputs the Orchestrated SAS data set and outputs it as a standard Orchestrate data set.

The SAS interface operators can also input and output data sets in Parallel SAS data set format. Descriptions of the data set formats are in the next section.
Representing SAS and Non-SAS Data in Orchestrate

Orchestrate Data Set Format
This is data in the normal Orchestrate data set format. Orchestrate operators, such as the database read and write operators, process data sets in this format.

Sequential SAS Data Set Format
This is SAS data in its original SAS sequential format. You include statements in the SAS code of the sas operator to read in a SAS data set for processing. A SAS data set cannot be an input to an Orchestrate operator except through a SAS read statement.

Parallel SAS Data Set Format
This is an Orchestrate-provided data-set format consisting of one or more sequential SAS data sets. Each data set pointed to by the file corresponds to a single partition of data flowing through Orchestrate. The file name for this data set is *.psds. It is analogous to a persistent *.ds data set.

The header is an ASCII text file composed of a list of sequential SAS data set files, with the following format:

```
Orchestrate SAS Dataset
Lfile:
  HOSTNAME:DIRECTORY/FILENAME
Lfile:
  HOSTNAME:DIRECTORY/FILENAME
Lfile:
  HOSTNAME:DIRECTORY/FILENAME
```

To save your data as a parallel SAS data set, either add the framework argument [psds] to the output redirection or add the .psds extension to the outfile name on your osh command line. These two commands are equivalent, except that the resulting data sets have different file names. The first example outputs a file without the .psds extension and second example outputs a file with the .psds extension.

```
$ osh "... sas options > [psds] outfile"
$ osh "... sas options > outfile.psds"
```

You can likewise use a parallel SAS data set as input to the sas operator, either by specifying [psds] or by having the data set in a file with the .psds extension:

```
$ osh "... sas options < [psds] infile"
$ osh "... sas options < infile.psds"
```
A persistent parallel SAS data set is automatically converted to an Orchestrate data set if the next operator in the flow is not a SAS interface operator.

**Orchestrated SAS Data Set Format**

When data is being moved into or out of a parallel SAS interface operator, the data must be in a format that allows Orchestrate to partition it to multiple processing nodes. For this reason, Orchestrate provides an internal data set representation called an Orchestrated SAS data set, which is specifically intended for capturing the record structure of a SAS data set for use by the Orchestrate SAS engine.

This is a non-persistent Orchestrate version of the SAS data set format that is the default output from SAS interface operators. If the input data to be processed is in a SAS data set or an Orchestrate data set, the conversion to an Orchestrated SAS data set is done automatically by the SAS interface operator.

In this format, each SAS record is represented by a single Orchestrate raw field named `SASdata`:

```plaintext
record ( SASdata: raw; )
```

**Getting Input from a SAS Data Set**

Using a SAS data set in a `sas` operator entails reading it in from the library in which it is stored using a SAS step, typically, a DATA step. The data is normally then output to the `liborch` library in Orchestrated SAS data set format.

`liborch` is the SAS engine provided by Orchestrate that you use to reference Orchestrated SAS data sets as inputs to and outputs from your SAS code. The `sas` operator converts a SAS data set or a standard Orchestrate data set into the Orchestrated SAS data set format.

Here is a schematic of this process:

```
libname temp '/tmp';
data liborch.p_out;
set temp.p_in;
...
run;
```

In this example

- `libname temp '/tmp'` specifies the library name of the SAS input
**Chapter 32 The SAS Interface Library Using Orchestrate to Run SAS Code**

- `data liborch.p_out` specifies the output Orchestrated SAS data set
- `temp.p_in` is the input SAS data set.

The output Orchestrated SAS data set is named using the following syntax:

```
liborch.osds_name
```

As part of writing the SAS code executed by the operator, you must associate each input and output data set with an input and output of the SAS code. In the figure above, the `sas` operator executes a DATA step with a single input and a single output. Other SAS code to process the data would normally be included.

In this example, the output data set is prefixed by `liborch`, a SAS library name corresponding to the Orchestrate SAS engine, while the input data set comes from another specified library; in this example the data set file would normally be named `/tmp/p_in.ssd01`.

**Getting Input from an Orchestrate Data Set or an Orchestrated SAS Data Set**

You may have data already formatted as an Orchestrate data set that you want to process with SAS code, especially if there are other Orchestrate operators earlier in the data flow. In that case, the Orchestrate SAS interface operators automatically convert a data set in the normal Orchestrate data set format into an Orchestrated SAS data set. However, you still need to provide the SAS step that connects the data to the inputs and outputs of the `sas` operator.

For Orchestrate data sets or Orchestrated SAS data sets the `liborch` library is referenced. The operator creates an Orchestrated SAS data set as output. Shown below is a `sas` operator with one input and one output data set:

```
input Orchestrate or Orchestrated SAS data set
```

![SAS Diagram](image)

As part of writing the SAS code executed by the operator, you must associate each input and output data set with an input and output of the SAS code. In the figure above, the `sas` operator executes a DATA step with a single input and a single output. Other SAS code to process the data would normally be included.
To define the input data set connected to the DATA step input, you use the **-input** option to the **sas** operator.

This option has the following **osh** format:

```
- input in_port_# ds_name
```

where:

- **ds_name** is the member name of a standard Orchestrate data set or an Orchestrated SAS data set used as an input by the SAS code executed by the operator. You need only to specify the member name here; do not include any SAS library name prefix. You always include this member name in your **sas** operator SAS code, prefixed with **liborch**.

- **in_port_#** is an input data set number. Input data sets are numbered starting from 0.

You use the corresponding **-output** option to connect an output Orchestrated SAS data set to an output of the DATA step.

The **osh** syntax of **-output** is:

```
- output out_port_# ods_name
```

For the example above, the settings for **-input** and **-output** are:

```
$ osh "... sas -input 0p_in
    -output 0p_out
    other_options"
```

## Converting between Data Set Types

You may have an existing Orchestrate data set, which is the output of an upstream Orchestrate operator or is the result of reading data from a database such as INFORMIX or Oracle using an Orchestrate database read operator. Also, the Orchestrate database write operators, like most Orchestrate operators, take as input standard Orchestrate data sets. Therefore, the SAS interface operators must convert Orchestrate data to SAS data and also convert SAS data to Orchestrate data. These two kinds of data-set conversion are covered in the following sections.

### Converting Orchestrate Data to SAS Data

Once you have provided the **liborch** statements to tell the **sas** operator where to find the input data set, as described in “Getting Input from an Orchestrate Data Set or an Orchestrated SAS Data Set” on page 32-10, the operator automatically converts the Orchestrate data set into an Orchestrated SAS data set before executing your SAS code. In order to do so, a SAS interface operator must convert both the field names and the field data types in the input Orchestrate data set.
Orchestrate supports 32-character field and data set names for SAS Version 8. However, SAS Version 6 accepts names of only up to eight characters. Therefore, you must specify whether you are using SAS Version 6 or SAS Version 8 when you install Orchestrate. When you specify SAS Version 6, Orchestrate automatically truncates names to eight characters when converting a standard Orchestrate data set to an Orchestrated SAS data set.

The SAS interface operators convert Orchestrate data types to corresponding SAS data types. All Orchestrate numeric data types, including integer, float, decimal, date, time, and timestamp, are converted to the SAS numeric data type. Orchestrate raw and string fields are converted to SAS string fields. Orchestrate raw and string fields longer than 200 bytes are truncated to 200 bytes, which is the maximum length of a SAS string.

The following table summarizes these data type conversions:

Table 157  **Conversions to SAS Data Types**

<table>
<thead>
<tr>
<th>Orchestrate Data Type</th>
<th>SAS Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>date</strong></td>
<td>SAS numeric date value</td>
</tr>
<tr>
<td><strong>decimal</strong>[p,s]</td>
<td>SAS numeric (p is precision and s is scale)</td>
</tr>
<tr>
<td><strong>int64</strong> and <strong>uint64</strong></td>
<td>not supported</td>
</tr>
<tr>
<td><strong>int8</strong>, <strong>int16</strong>, <strong>int32</strong>, <strong>uint8</strong>, <strong>uint16</strong>, <strong>uint32</strong>, <strong>sfloat</strong>, <strong>dfloat</strong></td>
<td>SAS numeric</td>
</tr>
</tbody>
</table>
| fixed-length **raw**, in the form: raw[n] | SAS string with length n
The maximum string length is 200 bytes; strings longer than 200 bytes are truncated to 200 bytes. |
| variable-length **raw**, in the form: raw[max=n] | SAS string with a length of the actual string length
The maximum string length is 200 bytes; strings longer than 200 bytes are truncated to 200 bytes. |
| variable-length **raw** in the form: raw | not supported |
| fixed-length **string** or **ustring**, in the form: string[n] or ustring[n] | SAS string with length n
The maximum string length is 200 bytes; strings longer than 200 bytes are truncated to 200 bytes. |
Converting SAS Data to Orchestrate Data

After your SAS code has run, it may be necessary for the SAS interface operator to convert an output Orchestrated SAS data set to the standard Orchestrate data-set format. This is required if you want to further process the data using the standard Orchestrate operators, including writing to a database using an Orchestrate database write operator.

In order to do this, the operator needs a record schema for the output data set. You can provide a schema for the Orchestrate data set in one of two ways:

- Using the -schema option or the -schemaFile option of the sasout operator, you can define a schema definition for the output data set.
- A persistent parallel SAS data set is automatically converted to an Orchestrate data set if the next operator in the flow is not a SAS interface operator.

When converting a SAS numeric field to an Orchestrate numeric, you can get a numeric overflow or underflow if the destination data type is too small to hold the value of the SAS field. You can avoid the error this causes by specifying all fields as nullable so that the destination field is simply set to null and processing continues.

An Orchestrate Example

A Specialty Freight Carrier charges its customers based on distance, equipment, packing, and license requirements. They need a report of distance traveled and charges for the month of July grouped by license type.

Here is the osh syntax for this example:

```
osh "sas -output 0 in_data
-source 'libname shipments "/july/distances/input";
data liborch.in_data;
```
set shipments.ship_data;
if substr(SHIPDATE,1,3)="Jul``;
run;' [seq]

hash key LICENSE
sas -input 0 in_data
-sas -source "proc sort data=liborch.in_data out=liborch.out_sort;
by LICENSE;
run;' -output 0 out_sort
sas -input 0 out_sort
-sas -source 'proc means data=liborch.out_sort N MEAN SUM;
by LICENSE;
var DISTANCE CHARGE;
run;' -options nocenter"

The table below shows some representative input data:

<table>
<thead>
<tr>
<th>Ship Date</th>
<th>District</th>
<th>Distance</th>
<th>Equipment</th>
<th>Packing</th>
<th>License</th>
<th>Charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun 2 2000</td>
<td>1</td>
<td>1540</td>
<td>D</td>
<td>M</td>
<td>BUN</td>
<td>1300</td>
</tr>
<tr>
<td>Jul 12 2000</td>
<td>1</td>
<td>1320</td>
<td>D</td>
<td>C</td>
<td>SUM</td>
<td>4800</td>
</tr>
<tr>
<td>Aug 2 2000</td>
<td>1</td>
<td>1760</td>
<td>D</td>
<td>C</td>
<td>CUM</td>
<td>1300</td>
</tr>
<tr>
<td>Jun 22 2000</td>
<td>2</td>
<td>1540</td>
<td>D</td>
<td>C</td>
<td>CUN</td>
<td>13500</td>
</tr>
<tr>
<td>Jul 30 2000</td>
<td>2</td>
<td>1320</td>
<td>D</td>
<td>M</td>
<td>SUM</td>
<td>6000</td>
</tr>
</tbody>
</table>

Here is the SAS output:

The SAS System
17:39, October 26, 2002

LICENSE=SUM

<table>
<thead>
<tr>
<th>Variable</th>
<th>Label</th>
<th>N</th>
<th>Mean</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISTANCE</td>
<td>DISTANCE</td>
<td>720</td>
<td>1563.93</td>
<td>1126030.00</td>
</tr>
<tr>
<td>CHARGE</td>
<td>CHARGE</td>
<td>720</td>
<td>28371.39</td>
<td>20427400.00</td>
</tr>
</tbody>
</table>

Step execution finished with status = OK.
Parallelizing SAS Steps

This section describes how to write SAS applications for parallel execution. Parallel execution allows your SAS application and data to be distributed to the multiple processing nodes within a scalable system.

The first section describes how to execute SAS DATA steps in parallel; the second section describes how to execute SAS PROC steps in parallel. A third section provides rules of thumb that may be helpful for parallelizing SAS code.

Executing DATA Steps in Parallel

This section describes how to convert a sequential SAS DATA step into a parallel DATA step. Characteristics of DATA steps that are candidates for parallelization using the roundrobin partitioning method include:

- Perform the same action for every record
- No RETAIN, LAGN, SUM, or + statements
- Order does not need to be maintained.

Characteristics of DATA steps that are candidates for parallelization using hash partitioning include:

- BY clause
- Summarization and/or accumulation in retained variables
- Sorting or ordering of inputs.

Often, steps with these characteristics require that you group records on a processing node or sort your data as part of a preprocessing operation.

Some DATA steps should not be parallelized. These include DATA steps that:

- Process small amounts of data (startup overhead outweighs parallel speed improvement)
- Perform sequential operations that cannot be parallelized (such as a SAS import from a single file)
- Need to see every record to produce a result

Parallelizing these kinds of steps would offer little or no advantage to your code. Steps of these kinds should be executed within a sequential Orchestrate sas operator.
Example Applications

Example 1: Parallelizing a SAS Data Step

This section contains an example that executes a SAS DATA step in parallel. Here is a figure describing this step:

```
libname prod '/prod';
data prod.sal_scl;
  set prod.sal;
  if (salary < 10000)
    then salary = 1;
  else if (salary < 25000)
    then salary = 2;
  else if (salary < 50000)
    then salary = 3;
  else if (salary < 100000)
    then salary = 4;
  else salary = 5;
run;
```

The step takes a single SAS data set as input and writes its results to a single SAS data set as output. The DATA step recodes the salary field of the input data to replace a dollar amount with a salary-scale value. Here is the original SAS code:

This DATA step requires little effort to parallelize because it processes records without regard to record order or relationship to any other record in the input. Also, the step performs the same operation on every input record and contains no BY clauses or RETAIN statements.
The following figure shows the Orchestrate data flow diagram for executing this DATA step in parallel:

In this example, you:

- Get the input from a SAS data set using a sequential `sas` operator;
- Execute the DATA step in a parallel `sas` operator;
- Output the results as a standard Orchestrate data set (you must provide a schema for this) or as a parallel SAS data set. You may also pass the output to another `sas` operator for further processing. The schema required may be generated by first outputting the data to a Parallel SAS data set, then referencing that data set. Orchestrate automatically generates the schema.

The first sequential `sas` operator executes the following SAS code as defined by the `-source` option to the operator:

```sas
libname prod "/prod";
data liborch.out;
  set prod.sal;
run;
```

This parallel `sas` operator executes the following SAS code:

```sas
libname prod "/prod";
data liborch.p_sal;
  set liborch.sal;
  . . . (salary field code from previous page)
run;
```

The `sas` operator can then do one of three things: use the `sasout` operator with its `-schema` option to output the results as a standard Orchestrate data set, output the results as a Parallel SAS data set, or pass the output directly to another `sas`
operator as an Orchestrated SAS data set. The default output format is Orchestrated SAS data set. When the output is to a Parallel SAS data set or to another sas operator, for example, as a standard Orchestrate data set, the liborch statement must be used. Conversion of the output to a standard Orchestrate data set or a Parallel SAS data set is discussed in “Orchestrated SAS Data Set Format” on page 32-9 and “Parallel SAS Data Set Format” on page 32-8.

Example 2: Using the hash Partitioner with a SAS DATA Step

This example reads two INFORMIX tables as input, hash partitions on the workloc field, then uses a SAS DATA step to merge the data and score it before writing it out to a parallel SAS data set.

The sas operator in this example runs the following DATA step to perform the merge and score:

```
data liborch.emptabd;
  merge liborch.wltab liborch.emptab;
  by workloc;
  a_13 = (f1-f3)/2;
  a_15 = (f1-f5)/2;
  .
  .
  run;
```

Records are hashed based on the workloc field. In order for the merge to work correctly, all records with the same value for workloc must be sent to the same processing node and the records must be ordered. The merge is followed by a
parallel Orchestrate sas operator that scores the data, then writes it out to a parallel SAS data set.

**Example 3: Using a SAS SUM Statement**

This section contains an example using the SUM clause with a DATA step. In this example, the DATA step outputs a SAS data set where each record contains two fields: an account number and a total transaction amount. The transaction amount in the output is calculated by summing all the deposits and withdrawals for the account in the input data where each input record contains one transaction.

Here is the SAS code for this example:

```sas
libname prod '/prod';
proc sort data=prod.trans;
   out=prod.s_trans
   by acctno;
data prod.up_trans (keep = acctno sum);
   set prod.s_trans;
   by acctno;
   if first.acctno then sum=0;
   if type = "D"
      then sum + amt;
   if type = "W"
      then sum - amt;
   if last.acctno then output;
run;
```

The SUM variable is reset at the beginning of each group of records where the record groups are determined by the account number field. Because this DATA step uses the BY clause, you use the Orchestrate hash partitioning operator with this step to make sure all records from the same group are assigned to the same node.

Note that DATA steps using SUM without the BY clause view their input as one large group. Therefore, if the step used SUM but not BY, you would execute the step sequentially so that all records would be processed on a single node.
Shown below is the data flow diagram for this example:

The SAS DATA step executed by the second sas operator is:

```sas
data liborch.nw_trans (keep = acctno sum);
set liborch.p_trans;
by acctno;
if first.acctno then sum=0;
if type = 'D'
  then sum + amt;
if type = 'W'
  then sum - amt;
if last.acctno then output;
run;
```

**Executing PROC Steps in Parallel**

This section describes how to parallelize SAS PROC steps using Orchestrate. Before you parallelize a PROC step, you should first determine whether the step is a candidate for parallelization.

When deciding whether to parallelize a SAS PROC step, you should look for those steps that take a long time to execute relative to the other PROC steps in your application. By parallelizing only those PROC steps that take the majority of your application’s execution time, you can achieve significant performance improvements without having to parallelize the entire application. Parallelizing steps with short execution times may yield only marginal performance improvements.
Many of the characteristics that mark a SAS DATA step as a good candidate for parallelization are also true for SAS PROC steps. Thus PROC steps are likely candidates if they:

- do not require sorted inputs
- perform the same operation on all records.

PROC steps that generate human-readable reports may also not be candidates for parallel execution unless they have a BY clause. You may want to test this type of PROC step to measure the performance improvement with parallel execution.

The following section contains two examples of running SAS PROC steps in parallel.

**Example Applications**

**Example 1: Parallelize PROC Steps Using the BY Keyword**

This example parallelizes a SAS application using PROC SORT and PROC MEANS. In this example, you first sort the input to PROC MEANS, then calculate the mean of all records with the same value for the `acctno` field.

The following figure illustrates this SAS application:

```
libname prod "/prod"
proc means data=prod.dhist;
   BY acctno;
run;
```

The BY clause in a SAS step signals that you want to hash partition the input to the step. Hash partitioning guarantees that all records with the same value for acctno are sent to the same processing node. The SAS PROC step executing on
each node is thus able to calculate the mean for all records with the same value for acctno.

Shown below is the Orchestrate implementation of this example:

PROC MEANS pipes its results to standard output, and the sas operator sends the results from each partition to standard output as well. Thus the list file created by the sas operator, which you specify using the -listds option, contains the results of the PROC MEANS sorted by processing node.

Shown below is the SAS PROC step for this application:

```sas
proc means data=liborch.p_dhist;
by acctno;
run;
```

Example 2: Parallelizing PROC Steps Using the CLASS Keyword

One type of SAS BY GROUP processing uses the SAS keyword CLASS. CLASS allows a PROC step to perform BY GROUP processing without your having to first sort the input data to the step. Note, however, that the grouping technique used by the SAS CLASS option requires that all your input data fit in memory on the processing node.

Note also that as your data size increases, you may want to replace CLASS and NWAY with SORT and BY.

Whether you parallelize steps using CLASS depends on the following:

- If the step also uses the NWAY keyword, parallelize it.

  When the step specifies both CLASS and NWAY, you parallelize it just like a step using the BY keyword, except the step input doesn't have to be sorted.
This means you hash partition the input data based on the fields specified to the CLASS option. See the previous section for an example using the hash partitioning method.

- If the CLASS clause does not use NWAY, execute it sequentially.
- If the PROC STEP generates a report, execute it sequentially, unless it has a BY clause.

For example, the following SAS code uses PROC SUMMARY with both the CLASS and NWAY keywords:

```sas
libname prod '/prod';
proc summary data=prod.txdlst missing NWAY;
CLASS acctno lstrxd fpdx;
var xdamt xdcnt;
output out=prod.xnlstr(drop=_type_ _freq_) sum=;
run;
```

In order to parallelize this example, you hash partition the data based on the fields specified in the CLASS option. Note that you do not have to partition the data on all of the fields, only to specify enough fields that your data is be correctly distributed to the processing nodes.

For example, you can hash partition on acctno if it ensures that your records are properly grouped. Or you can partition on two of the fields, or on all three. An important consideration with hash partitioning is that you should specify as few fields as necessary to the partitioner because every additional field requires additional overhead to perform partitioning.
The following figure shows the Orchestrate application data flow for this example:

```
libname prod '/prod';
data liborch.p_txdlst
    set prod.txdlst;
run;
```

```
proc summary data=liborch.p_txdlst
    missing NWAY;
    CLASS acctno lstrxd fpxd;
    var xdamt xdcnt;
    output out=liborch.p_xnlstr(drop=_type_ _freq_) sum=;
run;
```

**Some Points to Consider in Parallelizing SAS code**

**Which SAS Programs Benefit from Parallelization?**

There are four basic points you should consider when parallelizing your SAS code. If your SAS application satisfies any of the following criteria it is probably a good candidate for parallelization, provided the application is run against large volumes of data. SAS applications that run quickly, against small volumes of data, do not benefit.
1 Does your SAS application extract a large number of records from a parallel relational database? A few million records or more is usually sufficiently large to offset the cost of initializing parallelization.

2 Is your SAS program CPU intensive? CPU-intensive applications typically perform multiple CPU-demanding operations on each record. Operations that are CPU-demanding include arithmetic operations, conditional statements, creation of new field values for each record, etc. For SMP users, Orchestrate provides you with the biggest performance gains if your code is CPU-intensive.

3 Does your SAS program pass large records of lengths greater than 100 bytes? Orchestrate introduces a small record-size independent CPU overhead when passing records into and out of the sas operator. You may notice this overhead if you are passing small records that also perform little or no CPU-intensive operations.

4 Does your SAS program perform sorts? Sorting is a memory-intensive procedure. By performing the sort in parallel, you can reduce the overall amount of required memory. This is always a win on an MPP platform and frequently a win on an SMP.

Rules of Thumb

There are rules of thumb you can use to specify how a SAS program runs in parallel. Once you have identified a program as a potential candidate for use in Orchestrate, you need to determine how to divide the SAS code itself into Orchestrate steps.

The sas operator can be run either parallel or sequentially. Any converted SAS program that satisfies one of the four criteria outlined above will contain at least one parallel segment. How much of the program should be contained in this segment? Are there portions of the program that need to be implemented in sequential segments? When does a SAS program require multiple parallel segments? Here are some guidelines you can use to answer such questions.

1 Identify the slow portions of the sequential SAS program by inspecting the CPU and real-time values for each of the PROC and DATA steps in your application. Typically, these are steps that manipulate records (CPU-intensive) or that sort or merge (memory-intensive). You should be looking for times that are a relatively large fraction of the total run time of the application and that are measured in units of many minutes to hours, not seconds to minutes. You may need to set the SAS fullstimer option on in your config.sas612 or in your SAS program itself to generate a log of these sequential run times.

2 Start by parallelizing only those slow portions of the application that you have identified in Step 1 above. As you include more code within the parallel segment, remember that each parallel copy of your code (referred to as a partition) sees only a fraction of the data. This fraction is determined by the
partitioning method you specify on the input or inpipe lines of your sas operator source code.

3 Any two sas operators should only be connected by one pipe. This ensures that all pipes in the Orchestrate program are simultaneously flowing for the duration of the execution. If two segments are connected by multiple pipes, each pipe must drain entirely before the next one can start.

4 Keep the number of sas operators to a minimum. There is a performance cost associated with each operator that is included in the data flow. Rule 3 takes precedence over this rule. That is, when reducing the number of operators means connecting any two operators with more than one pipe, don't do it.

5 If you are reading or writing a sequential file such as a flat ASCII text file or a SAS data set, you should include the SAS code that does this in a sequential sas operator. Use one sequential operator for each such file. You will see better performance inputting one sequential file per operator than if you lump many inputs into one segment followed by multiple pipes to the next segment, in line with Rule 2 above.

6 When you choose a partition key or combination of keys for a parallel operator, you should keep in mind that the best overall performance of the parallel application occurs if each of the partitions is given approximately equal quantities of data. For instance, if you are hash partitioning by the key field year (which has five values in your data) into five parallel segments, you will end up with poor performance if there are big differences in the quantities of data for each of the five years. The application is held up by the partition that has the most data to process. If there is no data at all for one of the years, the application will fail because the SAS process that gets no data will issue an error statement. Furthermore, the failure will occur only after the slowest partition has finished processing, which may be well into your application. The solution may be to partition by multiple keys, for example, year and storeNumber, to use roundrobin partitioning where possible, to use a partitioning key that has many more values than there are partitions in your Orchestrate application, or to keep the same key field but reduce the number of partitions. All of these methods should serve to balance the data distribution over the partitions.

7 Multiple parallel segments in your Orchestrate application are required when you need to parallelize portions of code that are sorted by different key fields. For instance, if one portion of the application performs a merge of two data sets using the patientID field as the BY key, this PROC MERGE will need to be in a parallel segment that is hash partitioned on the key field patientID. If another portion of the application performs a PROC MEANS of a data set using the procedureCode field as the CLASS key, this PROC MEANS will have to be in a parallel sas operator that is hash partitioned on the procedureCode key field.

8 If you are running a query that includes an ORDER BY clause against a relational database, you should remove it and do the sorting in parallel, either
using SAS PROC SORT or an Orchestrate input line order statement. Performing the sort in parallel outside of the database removes the sequential bottleneck of sorting within the RDBMS.

9 A sort that has been performed in a parallel operator will order the data only within that operator. If the data is then streamed into a sequential operator, the sort order will be lost. You will need to re-sort within the sequential operator to guarantee order.

10 Within a parallel sas operator you may only use SAS work directories for intermediate writes to disk. SAS generates unique names for the work directories of each of the parallel operators. In an SMP environment this is necessary because it prevents the multiple CPUs from writing to the same work file. Do not use a custom-specified SAS library within a parallel operator.

11 Do not use a liborch directory within a parallel segment unless it is connected to an inpipe or an outpipe. A liborch directory may not be both written and read within the same parallel operator.

12 A liborch directory can be used only once for an input, inpipe, output or outpipe. If you need to read or write the contents of a liborch directory more than once, you should write the contents to disk via a SAS work directory and copy this as needed.

13 Remember that all Orchestrate operators in a step run simultaneously. This means that you cannot write to a custom-specified SAS library as output from one Orchestrate operator and simultaneously read from it in a subsequent operator. Connections between operators must be via Orchestrate pipes which are virtual data sets normally in Orchestrated SAS data set format.
Chapter 32

The SAS Interface Library

Using Orchestrate to Do ETL

Using Orchestrate to Do ETL
Only a simple SAS step is required to extract data from a SAS file. The Little SAS
Book from the SAS Institute provides a good introduction to the SAS step
language.
In the following example, SAS is directed to read the SAS file cl_ship, and to
deliver that data as an Orchestrate data stream to the next Orchestrate step. In this
example, the next step consists of the peek operator.
See “Representing SAS and Non-SAS Data in Orchestrate” on page 32-8 for
information on data-set formats, and “Getting Input from a SAS Data Set” on
page 32-9 for a description of liborch.
The -schemaFile option instructs Orchestrate to generate the schema that defines
the Orchestrate virtual data stream from the meta data in the SAS file cl_ship.
osh sas -source 'libname curr_dir \'.\';
DATA liborch.out_data;
SET curr_dir.cl_ship;
RUN;'
-output 0 out_data -schemaFile 'cl_ship' -workingdirectory '.'
[seq] 0> Link2.v;
peek -all [ seq] 0< Link2.v;

If you know the fields contained in the SAS file, the step can be written as:
osh sas -source 'libname curr_dir \'.\';
DATA liborch.out_data;
SET curr_dir.cl_ship;
RUN;'
-output 0 out_data
-schema record(SHIP_DATE:nullable string[50];
DISTRICT:nullable string[10];
DISTANCE:nullable sfloat;
EQUIPMENT:nullable string[10];
PACKAGING:nullable string[10];
LICENSE:nullable string[10];
CHARGE:nullable sfloat;)
[seq] 0> DSLink2.v;
peek –all [seq] 0< DSLink2.v;

It is also easy to write a SAS file from a Orchestrate virtual datastream. The
following osh commands generated the SAS file described above.
osh import
-schema record {final_delim=end, delim=',', quote=double}
(Ship_Date:string[max=50];
District:string[max=10];
Distance:int32;

32 28

Orchestrate 7.0 Operators Reference


Chapter 32

The SAS Interface Library

Using Orchestrate to Do ETL

Equipment:string[max=10];
Packaging:string[max=10];
License:string[max=10];
Charge:int32;)
-file '/home/cleonard/sas_in.txt'
-rejects continue
-reportProgress yes
0> DSLink2a.v;
osh sas -source 'libname curr_dir \'.\';
DATA curr_dir.cl_ship;
SET liborch.in_data;
RUN;'
-input 0 in_data

Orchestrate 7.0 Operators Reference

[seq] 0< DSLink2a.v;

32 29


The SAS Interface Operators

The *sasin* and *sasout* operators provide some options that the *sas* operator does not provide. However, if your application does not require that schema fields be dropped or modified, it is not necessary to explicitly include the *sasin* and/or *sasout* operators in your data flow. You can simply specify the *-input* and/or *-output* options to the *sas* operator, and Orchestrate automatically propagates the appropriate option values and commands of the *sasin* and *sasout* operators.

However, the *-sas_cs* option is not propagated when you explicitly include the *sasin* and/or *sasout* operators. If your Orchestrate data includes *ustring* values, see the next section, “Specifying a Character Set and SAS Mode,” for information on handling *ustring* values consistently.

Specifying a Character Set and SAS Mode

Using *-sas_cs* to Specify a Character Set

If your Orchestrate data includes *ustring* values, you can specify what character set Orchestrate uses to map between your *ustring* values and the *char* data stored in SAS files. You use the *-sas_cs* option of the *sas* operator to indicate your character-set choice. Setting this option also causes SAS to be invoked in international mode. See the next section, “Using *-sas_cs* to Determine SAS mode,” for information on modes.

**Important**

If your data flow also includes the *sasin* and/or the *sasout* operator, use the *-sas_cs* option of these operators to specify the same character set you specified to the *sas* operator.

The *sasin* and *sas* operators use the *-sas_cs* option to determine what character-set encoding should be used when exporting *ustring* (UTF-16) values to SAS. The *sas* and *sasout* operators use this option to determine what character set Orchestrate uses to convert SAS *char* data to Orchestrate *ustring* values.

The syntax for the *-sas_cs* option is:

```
-sas_cs icu_character_set | DBCSLANG
```

In `$APT.ORCHHOME/apt/etc/platform/` there are platform-specific `sascs.txt` files that show the default ICU character set for each DBCSLANG setting. The platform directory names are: `sun`, `aix`, `osf1` (Tru64), `hpux`, `lunix`, and `uss`. For example:

```
$APT.ORCHHOME/apt/etc/aix/sascs.txt
```

When you specify a character setting, the `sascs.txt` file must be located in the platform-specific directory for your operating system. Orchestrate accesses the setting that is equivalent to your *-sas_cs* specification for your operating system. ICU settings can differ between platforms.
You may change the ICU equivalents. The default values are:

<table>
<thead>
<tr>
<th>DBCSLANG Setting</th>
<th>ICU Character Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAPANESE</td>
<td>icu_character_set</td>
</tr>
<tr>
<td>KATAKANA</td>
<td>icu_character_set</td>
</tr>
<tr>
<td>KOREAN</td>
<td>icu_character_set</td>
</tr>
<tr>
<td>HANGUL</td>
<td>icu_character_set</td>
</tr>
<tr>
<td>CHINESE</td>
<td>icu_character_set</td>
</tr>
<tr>
<td>TAIWANESE</td>
<td>icu_character_set</td>
</tr>
</tbody>
</table>

**Note** If Orchestrate encounters a ustring value but you have not specified the `-sas_cs` option, SAS remains in standard mode and Orchestrate then determines what character setting to use when it converts your ustring values to Latin1 characters.

Orchestrate first references your `APT_SAS_CHARSET` environment variable. If it is not set and your `APT_SAS_CHARSET_ABORT` environment variable is also not set, it uses the value of your `-impexp_charset` option or the value of your `APT_IMPEXP_CHARSET` environment variable.

**Using -sas_cs to Determine SAS mode**

Specifying the `-sas_cs` option also tells Orchestrate to invoke SAS in international mode. In this mode, the SAS DBCS, DBCSLANG, and DBCSTYPE environment variables are set, and SAS processes your ustring data and step source code using multi-byte characters in your chosen character set. Multi-byte Unicode character sets are supported in SAS char fields, but not in SAS field names or data set file names.

Your SAS system is capable of running in DBCS mode if your SAS log output has this type of header:

```
NOTE: SAS (r) Proprietary Software Release 8.2 (TS2MO DBCS2944)
```

If you do not specify the `-sas_cs` option for the `sas` operator, SAS is invoked in standard mode. In standard mode, SAS processes your string fields and step source code using single-byte Latin1 characters. SAS standard mode, is also called "the basic US SAS product". When you have invoked SAS in standard mode, your SAS log output has this type of header:

```
NOTE: SAS (r) Proprietary Software Release 8.2 (TS2MO)
```
Parallel SAS Data Sets and SAS International

Automatic Orchestrate Step Insertion

When you save a .ds data set as a parallel SAS data set by using the [psds] directive or adding a .psds suffix to the output file, Orchestrate automatically inserts a step in your data flow which contains the sasin, sas, and sasout operators. Using the -source option, the sas operator specifies specialized SAS code to be executed by SAS.

The SAS executable used is the one on your $PATH unless its path is overwritten by the the APT_SAS_COMMAND or APT_SASINT_COMMAND environment variable. Orchestrate SAS-specific variables are described in “Environment Variables” on page 32-35. The DBCS, DBCSLANG, and DBCSTYLE environment variables are not set for the step.

If the Orchestrate-inserted step fails, you can see the reason for its failure by rerunning the job with the APT_SAS_SHOW_INFO environment variable set.

For more information on Parallel SAS Data Sets, see “Parallel SAS Data Set Format” on page 32-8.

Handling SAS char Fields Containing Multi-Byte Unicode Data

When Orchestrate writes a .psds data set, it must reference your ICU character setting when it encounters a ustring field. Orchestrate determines your character setting by first referencing your APT_SAS_CHARSET environment variable. If this variable is not set and your APT_SAS_CHARSET_ABORT environment variable is also not set, it uses the value of your -impexp_charset option or the value of the APT_IMPEXP_CHARSET environment variable. See “Note” on page 32-31.

When reading a .psds data set, Orchestrate imports all SAS char fields as Orchestrate string fields. Use the modify operator to import those char fields that contain multi-byte Unicode characters as ustrings, and specify a character set to be used for the ustring value.

The syntax for this modify operator conversion specification is:

destField[:datatype]=ustring_from_string[charset](sourceField)

Here is the conversion specification used in an example osh command:


Specifying an Output Schema

You must specify an output schema to the sas and sasout operators when the downstream operator is a standard Orchestrate operator such as peek or copy. Orchestrate uses the schema to convert the Orchestrated SAS data used by sas or
sasout to an Orchestrate data set suitable for processing by a standard Orchestrate operator.

There are two ways to specify the schema:

- Use the -schema option to supply an Orchestrate schema. By supplying an explicit schema, you obtain better job performance and gain control over which SAS char data is to be output as ustring values and which SAS char data is to be output as string values.

- Use the -schemaFile option to specify a SAS file that has the same meta data description as the SAS output stream. Orchestrate generates the schema from that file.

Use this syntax:

```
-schemaFile schema_file_name -debug yes
```

Specifying -debug yes displays the generated schema. For example:

```
..<APT_SAS_OutputOperator> output schema from SAS file's MetaData is;
record{
    S: nullable string[8];
    B: nullable string[8];
}
```

You can use the APT_SAS_SCHEMASOURCE_DUMP environment variable to see the SAS CONTENTS report used by the -schemaFile option. The output also displays the command line given to SAS to produce the report and the pathname of the report. The input file and output file created by SAS is not deleted when this variable is set.

You can also obtain an Orchestrate schema by using the -schema option of the sascontents operator. Use the peek operator in sequential mode to print the schema to the screen. Example output is:

```
...<peek,0>Suggested schema for Orchestrated SAS dataset 'cow'
...<peek,0>record (a : nullable dfloat;
...<peek,0>     b : nullable string[10])
```

You can then fine tune the schema and specify it to the -schema option to take advantage of performance improvements.

**Note** If both the -schemaFile option and the -sas_cs option are set, all of your SAS char fields are converted to Orchestrate ustring values. If the -sas_cs option is not set, all of your SAS char values are converted to Orchestrate string values.

To obtain a mixture of string and ustring values use the -schema option.
Controlling ustring Truncation

Your ustring values may be truncated before the space pad characters and \0 because a ustring value of n characters does not fit into n bytes of a SAS char value. You can use the APT_SAS_TRUNCATION environment variable to specify how the truncation is done. This variable is described in “Environment Variables” on page 32-35.

If the last character in a truncated value is a multi-byte character, the SAS char field is padded with C_null (zero) characters. For all other values, including non-truncated values, spaces are used for padding.

You can avoid truncation of your ustring values by specifying them in your schema as variable-length strings with an upper bound. The syntax is:

```
ustring[ max=n_codepoint_units ]
```

Specify a value for n_codepoint_units that is 2.5 or 3 times larger than the number of characters in the ustring. This forces the SAS char fixed-length field size to be the value of n_codepoint_units. The maximum value for n_codepoint_units is 200.

Generating a Proc Contents Report

To obtain a report similar to the report that is generated by the SAS procedure PROC CONTENTS for a self-describing data set, use one of these methods:

- Specify the -report option of the sasin or sasout operator.
- Specify -debug yes to the sasin, sas, or sasout operator.
- Insert the sascontents operator in your data flow.

Use the peek operator in sequential mode to print the report to the screen.

Here is an example report:

```
...2:11:50(000) <APT_SAS_Driver in sas,0> input dataset 0, SAS library member moo
APT_SAS_Driver in sas (APT_SAS_Driver):
Version: 4
Number Name Type Len Offset Format Label
1   AGE   Num  8   0   4.  age
2   CAR   Char 8   8   car
3   CHILDREN Num 8   16  4.  children
4   CUR_ACT Char 8   24 cur_act
5   ID    Char 8   32  id
6   INCOME Num 8   40  4.   income
7   IRA   Char 8   48  ira
```
Orchestrate-Inserted Partition and Sort Components

By default, Orchestrate inserts partition and sort components to meet the partitioning and sorting needs of the SAS-interface operators and other operators. For a SAS-interface operator, Orchestrate selects the appropriate component from Orchestrate or from SAS. See the section “Orchestrate-Inserted Partition and Sort Components” on page xxxvi of this Reference for information on this facility.

Long Name Support

For SAS 8.2, Orchestrate handles SAS file and column names up to 32 characters. However, this functionality is dependent on your installing a SAS patch. Obtain this patch from SAS:

SAS Hotfix 82BB40 for the SAS Toolkit

For SAS 6.12, Orchestrate handles SAS file and column names up to 8 characters.

Environment Variables

These are the SAS-specific environment variables:

- **APT_SAS_COMMAND absolute_path**
  Overrides the $PATH directory for SAS with an absolute path to the US SAS executable. An example path is: /usr/local/sas/sas8.2/sas.

- **APT_SASINT_COMMAND absolute_path**
  Overrides the $PATH directory for SAS with an absolute path to the International SAS executable. An example path is:
  /usr/local/sas/sas8.2int/dbs/sas.

- **APT_SAS_CHARSET icu_character_set**
  When the -sas_cs option is not set and a SAS-interface operator encounters a ustring, Orchestrate interrogates this variable to determine what character set to use. If this variable is not set, but APT_SAS_CHARSET_ABORT is set, the operator aborts; otherwise Orchestrate accesses the -impexp_charset option or the APT_IMPEXP_CHARSET environment variable.

• **APT_SAS_CHARSET_ABORT**

Causes a SAS-interface operator to abort if Orchestrate encounters a **ustring** in the schema and neither the **-sas_cs** option or the **APT_SAS_CHARSET** environment variable is set.

• **APT_SAS_TRUNCATION_ABORT | NULL | TRUNCATE**

Because a **ustring** of \( n \) characters does not fit into \( n \) bytes of a SAS **char** value, the **ustring** value may be truncated before the space pad characters and \( \0 \). The **sasin** and **sas** operators use this variable to determine how to truncate a **ustring** value to fit into a SAS **char** field. **TRUNCATE**, which is the default, causes the **ustring** to be truncated; **ABORT** causes the operator to abort; and **NULL** exports a null field. For **NULL** and **TRUNCATE**, the first five occurrences for each column cause an information message to be issued to the log.

• **APT_SAS_ACCEPT_ERROR**

When a SAS procedure causes SAS to exit with an error, this variable prevents the SAS-interface operator from terminating. The default behavior is for Orchestrate to terminate the operator with an error.

• **APT_SAS_DEBUG_LEVEL=[0-2]**

Specifies the level of debugging messages to output from the SAS driver. The values of 0, 1, and 2 duplicate the output for the **-debug** option of the **sas** operator: **no**, **yes**, and **verbose**.

• **APT_SAS_DEBUG=1, APT_SAS_DEBUG_IO=1, APT_SAS_DEBUG_VERBOSE=1**

Specifies various debug messages output from the SA Toolkit API.

• **APT_HASH_TO_SASHASH**

The Orchestrate **hash** partitioner contains support for hashing SAS data. In addition, Orchestrate provides the **sashash** partitioner which uses an alternative non-standard hashing algorithm. Setting the **APT_HASH_TO_SASHASH** environment variable causes all appropriate instances of **hash** to be replaced by **sashash**. If the **APT_NO_SAS_TRANSFORMS** environment variable is set, **APT_HASH_TO_SASHASH** has no affect.

• **APT_NO_SAS_TRANSFORMS**

Orchestrate automatically performs certain types of SAS-specific component transformations, such as inserting a **sasout** operator and substituting **sasRoundRobin** for **eRoundRobin**. Setting the **APT_NO_SAS_TRANSFORMS** variable prevents Orchestrate from making these transformations.

• **APT_NO_SASOUT_INSERT**

This variable selectively disables the **sasout** operator insertions. It maintains the other SAS-specific transformations.

• **APT_SAS_SHOW_INFO**

Displays the standard SAS output from an import or export transaction. The SAS output is normally deleted since a transaction is usually successful.
• **APT_SAS_SCHEMA_SRC_DUMP**

  Displays the SAS CONTENTS report used by the `-schemaFile` option. The output also displays the command line given to SAS to produce the report and the pathname of the report. The input file and output file created by SAS is not deleted when this variable is set.

In addition to the SAS-specific debugging variables, you can set the **APT_DEBUG_SUBPROC** environment variable to display debug information about each subprocess operator.
## The sasin Operator

You can use the `sasin` operator to convert a standard Orchestrate data set to an Orchestrated SAS data set suitable for input to the `sas` operator.

The `sasin` operator takes a number of optional arguments that control how the Orchestrate data is converted to an Orchestrated SAS data set. For example, you can specify a schema for the input data set using the `-schema` option.

Because a `ustring` value of `n` characters does not fit into `n` bytes of a SAS `char` value, the `ustring` value may be truncated before the space pad characters and `\0`. You can use the `APT_SAS_TRUNCATION` environment variable to specify how the truncation is done. It is described in “Environment Variables” on page 32-35.

If you do not need to use any of the `sasin` options, you can bypass the `sasin` operator and input your Orchestrate data set directly to the `sas` operator.

### Note
When the `sasin` operator executes sequentially and the input Orchestrate data set has multiple partitions, the `sasin` operator also performs the conversion sequentially.

### Data Flow Diagram

```
input Orchestrate data set

sasin

output Orchestrated SAS data set
```

### Properties

Table 158  sasin Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1 standard Orchestrate data set</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1 Orchestrated SAS data set</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>from upstream Orchestrate operator or specified by the sasin operator -schema option</td>
</tr>
</tbody>
</table>
Table 158  **sasin Operator Properties (continued)**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output interface schema</td>
<td>If no key option is specified:</td>
</tr>
<tr>
<td></td>
<td>record ( sasData:raw; )</td>
</tr>
<tr>
<td></td>
<td>If a key option is specified:</td>
</tr>
<tr>
<td></td>
<td>record ( sastsort:raw; sasdata:raw; )</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>none</td>
</tr>
<tr>
<td>Execution mode</td>
<td>Parallel by default or sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>any (parallel mode)</td>
</tr>
<tr>
<td>Collection method</td>
<td>any (sequential mode)</td>
</tr>
<tr>
<td>Preserve-partitioning flag in</td>
<td>set</td>
</tr>
<tr>
<td>output data set</td>
<td></td>
</tr>
</tbody>
</table>

**Syntax and Options**

The syntax for the `sasin` operator is shown below. All options are optional.

Terms in italic typeface are option strings you supply. When your option string contains a space or a tab character, you must enclose it in single quotes.

```plaintext
sasin
[-context prefix_string]
[-debug yes | no | verbose]
[-defaultlength length]
[-drop field0 field1 ... fieldN | -keep field0 field1 ...fieldN]
[-key field_name [-a | -d] [-cs | -ci]]
[-length integer field0 field1 ... fieldN]
[-rename schema_field_name SAS_field_name...]
[-report]
[-sas_cs icu_character_set | DBCSLANG]
[-schema schema]
```
Table 159  **sasin Operator Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>-context</strong></td>
<td>context prefix_string</td>
</tr>
<tr>
<td></td>
<td>Optionally specifies a string that prefixes any informational, warning, or error messages generated by the operator.</td>
</tr>
<tr>
<td><strong>-debug</strong></td>
<td>-debug yes</td>
</tr>
<tr>
<td></td>
<td>A setting of -debug yes causes the operator to ignore errors in the SAS program and continue execution of the application. This allows your application to generate output even if a SAS step has an error.</td>
</tr>
<tr>
<td></td>
<td>By default, the setting is -debug no.</td>
</tr>
<tr>
<td></td>
<td>Setting -debug verbose is the same as -debug yes, but in addition it causes the operator to echo the SAS source code executed by the operator.</td>
</tr>
<tr>
<td><strong>-defaultlength</strong></td>
<td>-defaultlength length</td>
</tr>
<tr>
<td></td>
<td>Specifies the default length, in bytes, for all SAS numeric fields generated by the operator. The value must be between 3 and 8.</td>
</tr>
<tr>
<td></td>
<td>This option allows you to control the length of SAS numeric fields when you know the range limits of your data. Using smaller lengths reduces the size of your data.</td>
</tr>
<tr>
<td></td>
<td>You can override the default length for specific fields using the -length option.</td>
</tr>
<tr>
<td><strong>-drop</strong></td>
<td>-drop field0 field1 ... fieldN</td>
</tr>
<tr>
<td></td>
<td>Optionally specifies the fields of the input data set to be dropped. All fields not specified by -drop are written to the output data set.</td>
</tr>
<tr>
<td></td>
<td>field designates an Orchestrate data set field name. It is the original name of an input field before any renaming, as performed by the -rename option, or name truncation for input field names longer than eight characters in SAS Version 6.</td>
</tr>
<tr>
<td></td>
<td>You can also specify a range of fields to drop by specifying two field names separated by a hyphen in the form fieldN - fieldM. In this case, all fields between, and including fieldN and fieldM are dropped.</td>
</tr>
<tr>
<td></td>
<td>This option is mutually exclusive with -keep.</td>
</tr>
</tbody>
</table>
### Table 159  *sasin* Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>-keep</strong></td>
<td></td>
</tr>
<tr>
<td><code>-keep field0 field1 ... fieldN</code></td>
<td>Optionally specifies the fields of the Orchestrate data set to be retained on input. All fields not specified by <code>-keep</code> are dropped from input. Field designates an Orchestrate data set field name. It is the original name of an input field before any renaming, as performed by the <code>-rename</code> option, or name truncation for input field names longer than eight characters in SAS Version 6. You can also specify a range of fields to drop by specifying two field names separated by a hyphen in the form <code>fieldN - fieldM</code>. In this case, all fields between, and including <code>fieldN</code> and <code>fieldM</code> are dropped. This option is mutually exclusive with <code>-drop</code>.</td>
</tr>
<tr>
<td><strong>-key</strong></td>
<td></td>
</tr>
<tr>
<td>`-key field_name [-a</td>
<td>-d] [-cs</td>
</tr>
</tbody>
</table>

---
Table 159  

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>-length</strong></td>
<td>Specifies the length in bytes, of SAS numeric fields generated by the operator. The value must be between 3 and 8. This option allows you to control the length of SAS numeric fields when you know the range limits of your data. Using smaller lengths reduces the size of your data. By default, all SAS numeric fields are 8 bytes long, or the length specified by the <strong>defaultlength</strong> option to the operator. The field name is the original name of an input field before any renaming, as performed by the <strong>rename</strong> option, or name truncation for input field names longer than eight characters for SAS 6. You can also specify a range of fields by specifying two field names separated by a hyphen in the form fieldN-fieldM. All fields between, and including fieldN and fieldM use the specified length.</td>
</tr>
<tr>
<td><strong>-rename</strong></td>
<td>Specifies a mapping of an input field name from the Orchestrate data set to a SAS variable name. For multiple mappings, use multiple <strong>rename</strong> options. By default, Orchestrate truncates to eight characters the input fields with a name longer than eight characters in SAS Version 6. If truncation causes two fields to have the same name, the operator issues a syntax error. Aliases must be unique with all other alias specifications and with respect to the Orchestrated SAS data set field names.</td>
</tr>
<tr>
<td><strong>-report</strong></td>
<td>Causes the operator to output a report describing how it converts the input Orchestrate data set to the Orchestrated SAS data set.</td>
</tr>
<tr>
<td><strong>-sas_cs</strong></td>
<td>When your Orchestrate data includes ustring values, you can use the <strong>sas_cs</strong> option to specify a character set that maps between Orchestrate ustrings and the char data stored in SAS files. Use the same <strong>sas_cs</strong> character setting for all the SAS-interface operators in your data flow. See “Using -sas_cs to Specify a Character Set” on page 32-30 for more details. For information on national language support, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide, and reference this IBM ICU site: <a href="http://oss.software.ibm.com/icu/charset">http://oss.software.ibm.com/icu/charset</a></td>
</tr>
</tbody>
</table>
Table 159  **sasin Operator Options (continued)**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
</table>
| -schema | `schema schema_definition`  
  Specifies the record schema of the input Orchestrate data set. Only those fields specified in the record schema are written to the output data set. |
The sas Operator

You use the `sas` operator to execute SAS code in parallel or sequentially as part of an Orchestrate application.

**Data Flow Diagram**

![Data Flow Diagram]

**Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>N (set by user)</td>
</tr>
<tr>
<td></td>
<td>Can be either Orchestrate data sets or Orchestrated SAS data sets.</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>M (set by user)</td>
</tr>
<tr>
<td></td>
<td>All output data sets can be either Orchestrated SAS data sets or Parallel SAS data sets. If you are passing output to another <code>sas</code> operator, the data should remain in Orchestrated SAS data set format.</td>
</tr>
<tr>
<td></td>
<td>If requested, the SAS log file is written to the last output data set. For SAS 8.2, the log file contains a header and additional information at the beginning of the file.</td>
</tr>
<tr>
<td></td>
<td>If requested, the SAS list file is written to the second to last output data set if you also request a log file output, and to the last data set if no log file is requested.</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>Derived from the input data set</td>
</tr>
</tbody>
</table>
Table 160 sas Operator Properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output interface schema</td>
<td>For output data sets:</td>
</tr>
<tr>
<td></td>
<td>As specified by the -schema option or the -schemaFile option when the sasout operator is not used.</td>
</tr>
<tr>
<td></td>
<td>When the sasout operator is used and the downstream operator expects an Orchestrated SAS data set, the schema is:</td>
</tr>
<tr>
<td></td>
<td>record (sasData: raw; )</td>
</tr>
<tr>
<td></td>
<td>For list and log data sets:</td>
</tr>
<tr>
<td></td>
<td>record (partitionNumber: uint16;</td>
</tr>
<tr>
<td></td>
<td>recordNumber: uint32;</td>
</tr>
<tr>
<td></td>
<td>rec: string;</td>
</tr>
<tr>
<td>Execution mode</td>
<td>Parallel (default) or sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>Any parallel mode except modulus</td>
</tr>
<tr>
<td>Collection method</td>
<td>Any sequential mode</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>Set on all output data sets; clear on log and list data sets</td>
</tr>
</tbody>
</table>

Syntax and Options

The syntax for the sas operator is shown below. You must specify either the -source or the -sourcefile option. And, if you are not using the sasout operator, you must also specify the -output option to supply an output schema. All the other options are optional.

Terms in italic typeface are option strings you supply. When your option string contains a space or a tab character, you must enclose it in single quotes.

```
  sas
  -source SAS_code | -sourcefile SAScode_filepath
  [-sas_cs icu_character_set | DBCSLANG]
  [-options sas_options]
  [-workingdirectory directory_path | -noworkingdirectory]
  [-listds file | display | dataset | none]
  [-logds file | display | dataset | none]
  [-input in_port_# sas_dataset_name]
```
The `sas` operator options are:

```plaintext
[-output out_port_# ods_name
  [-schemaFile schema_file_name (alias -schemaSource) |
   -schema schema_definition]]
[-debug yes | no | verbose]
```

### Table 161  sas Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-source</code></td>
<td><code>-source SAS_code</code></td>
</tr>
<tr>
<td></td>
<td>Specifies the SAS code to be executed by SAS. The SAS code may contain both PROC steps and DATA steps.</td>
</tr>
<tr>
<td></td>
<td>You must specify either the <code>-source</code> or <code>-sourcefile</code> option.</td>
</tr>
<tr>
<td><code>-sourcefile</code></td>
<td><code>-sourcefile SAS_code_filepath</code></td>
</tr>
<tr>
<td></td>
<td>Specifies the path to a file containing the SAS source code to be executed. The file path and file contents should be in the UTF-8 character set.</td>
</tr>
<tr>
<td></td>
<td>You must specify either the <code>-sourcefile</code> or the <code>-source</code> option.</td>
</tr>
<tr>
<td><code>-debug</code></td>
<td>`-debug yes</td>
</tr>
<tr>
<td></td>
<td>A setting of <code>-debug yes</code> causes the <code>sas</code> operator to ignore errors in the SAS program and continue execution of the application. This allows your application to generate output even if a SAS step has an error.</td>
</tr>
<tr>
<td></td>
<td>By default, the setting is <code>-debug no</code>, which causes the operator to abort when it detects an error in the SAS program.</td>
</tr>
<tr>
<td></td>
<td>Setting <code>-debug verbose</code> is the same as <code>-debug yes</code>, but in addition it causes the operator to echo the SAS source code executed by the operator.</td>
</tr>
</tbody>
</table>
Specifies the name of the SAS data set, `sas_ds_name`, receiving its input from the data set connected to `in_port_#`.

The operator uses `-input` to connect each input data set of the operator to an input of the SAS code executed by the operator. For example, your SAS code contains a DATA step whose input you want to read from input data set 0 of the operator. The following SAS statements might be contained within your code:

```sas
libname temp '/tmp';
data liborch.parallel_out;
set temp.parallel_in;
```

In this case, you would use `-input` and set the `in_port_#` to 0, and the `sas_ds_name` to the member name `parallel_in`.

`sas_ds_name` is the member name of a SAS data set used as an input to the SAS code executed by the operator. You only need to specify the member name here; do not include any SAS library name prefix.

When referencing `sas_ds_name` as part of the SAS code executed by the operator, always prefix it with `liborch`, the name of the Orchestrate SAS engine.

`in_port_#` is the number of an input data set of the operator. Input data sets are numbered from 0, thus the first input data set is data set 0, the next is data set 1, etc.

`-listds`: 

- `-listds file | dataset | none | display`

Optionally specify that `sas` should generate a SAS list file.

Specifying `-listds file` causes the `sas` operator to write the SAS list file generated by the executed SAS code to a plain text file in the working directory. The list is sorted before being written out. The name of the list file, which cannot be modified, is `orchident.lst`, where `ident` is the name of the operator, including an index in parentheses if there are more than one with the same name. For example, `orchsas(1).lst` is the list file from the second `sas` operator in a step.

`-listds dataset` causes the list file to be written to the last output data set. If you also request that the SAS log file be written to a data set using `-logds`, the list file is written to the second to last output data set. The data set from a parallel `sas` operator containing the list information is not sorted.

If you specify `-listds none`, the list is not be generated.

`-listds display` is the default. It causes the list to be written to standard error.
### Table 161  \textbf{sas} Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-logds</td>
<td>-logds file</td>
</tr>
<tr>
<td></td>
<td>Optionally specify that \texttt{sas} write a SAS log file.</td>
</tr>
<tr>
<td></td>
<td>-logds file</td>
</tr>
<tr>
<td></td>
<td>-logds dataset</td>
</tr>
<tr>
<td></td>
<td>If you specify -logds none, the log is not generated.</td>
</tr>
<tr>
<td></td>
<td>-logds display</td>
</tr>
<tr>
<td>-noworkingdirectory</td>
<td>-noworkingdirectory</td>
</tr>
<tr>
<td>or -nowd</td>
<td>Disables the warning message generated by Orchestrate when you omit the -workingdirectory option.</td>
</tr>
<tr>
<td></td>
<td>If you omit the -workingdirectory argument, the SAS working directory is indeterminate and Orchestrate automatically generates a warning message. See the -workingdirectory option below. The two options are mutually exclusive.</td>
</tr>
<tr>
<td>-options</td>
<td>-options sas_options</td>
</tr>
<tr>
<td></td>
<td>Optionally specify a quoted string containing any options that can be specified to a SAS OPTIONS directive. These options are executed before the operator executes your SAS code. For example, you can use this argument to enable the SAS fulltimer.</td>
</tr>
<tr>
<td></td>
<td>You can specify multiple options, separated by spaces.</td>
</tr>
<tr>
<td></td>
<td>By default, the operator executes your SAS code with the SAS options notes and source. Specifying any string for sas_options configures the operator to execute your code using only the specified options. Therefore you must include notes and source in sas_options if you still want to use them.</td>
</tr>
</tbody>
</table>
Table 161  sas Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-output</td>
<td>(-output) out_port_# ods_name</td>
</tr>
<tr>
<td></td>
<td>[(-)schemaFile schema_file_name</td>
</tr>
<tr>
<td></td>
<td>Optionally specify the name of the Orchestrated SAS data set, ods_name, writing its output to the data set connected to out_port_# of the operator.</td>
</tr>
<tr>
<td></td>
<td>The operator uses -output to connect each output data set of the operator to an output of the SAS code executed by the operator.</td>
</tr>
<tr>
<td></td>
<td>For example, your SAS code contains a DATA step whose output you want to write to output data set 0 of the operator. Here is the SAS expression contained within your code:</td>
</tr>
<tr>
<td></td>
<td>data liborch.parallel_out;</td>
</tr>
<tr>
<td></td>
<td>In this case, you would use -output and set the out_port_# to 0, and the ods_name to the member name parallel_out.</td>
</tr>
<tr>
<td></td>
<td>ods_name corresponds to the name of an Orchestrated SAS data set used as an output by the SAS code executed by the operator. You only need to specify the member name here; do not include any SAS library name prefix.</td>
</tr>
<tr>
<td></td>
<td>When referencing ods_name as part of the SAS code executed by the operator, always prefix it with liborch, the name of the Orchestrate SAS engine.</td>
</tr>
<tr>
<td></td>
<td>out_port_# is the number of an output data set of the operator. Output data sets are numbered starting from 0.</td>
</tr>
<tr>
<td></td>
<td>You use the -schemaFile suboption to specify the name of a SAS file containing the metadata column information which Orchestrate uses to generate an Orchestrate schema; or you use the -schema suboption followed by the schema definition. See “Specifying an Output Schema” on page 32-32 for more details.</td>
</tr>
<tr>
<td></td>
<td>If both the -schemaFile option and the -sas_cs option are set, all of your SAS char fields are converted to Orchestrate ustring values. If the -sas_cs option is not set, all of your SAS char values are converted to Orchestrate string values. To obtain a mixture of string and ustring values use the -schema option. See “Specifying an Output Schema” on page 32-32 for more details.</td>
</tr>
<tr>
<td></td>
<td>Note: The -schemaFile and -schema suboptions are designated as optional because you should not specify them for the sas operator if you specify them for the sasout operator. It is an error to specify them for both the sas and sasout operators.</td>
</tr>
</tbody>
</table>
When your Orchestrate data includes **ustring** values, you can use the `-sas_cs` option to specify a character set that maps between Orchestrate **ustrings** and the **char** data stored in SAS files. Use the same `-sas_cs` character setting for all the SAS-interface operators in your data flow. See “Using `-sas_cs` to Specify a Character Set” on page 32-30 for more details.

For information on national language support, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:

http://oss.software.ibm.com/icu/charset

**Table 161  sas Operator Options (continued)**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-sas_cs</code></td>
<td>`-sas_cs icu_character_set</td>
</tr>
<tr>
<td></td>
<td>When your Orchestrate data includes <strong>ustring</strong> values, you can use the <code>-sas_cs</code> option to specify a character set that maps between Orchestrate <strong>ustrings</strong> and the <strong>char</strong> data stored in SAS files. Use the same <code>-sas_cs</code> character setting for all the SAS-interface operators in your data flow. See “Using <code>-sas_cs</code> to Specify a Character Set” on page 32-30 for more details. For information on national language support, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site: <a href="http://oss.software.ibm.com/icu/charset">http://oss.software.ibm.com/icu/charset</a></td>
</tr>
<tr>
<td><code>-workingdirectory</code></td>
<td><code>-workingdirectory dir_name</code></td>
</tr>
<tr>
<td>or <code>-wd</code></td>
<td>Specifies the name of the working directory on all processing nodes executing the SAS application. All relative pathnames in your application are relative to the specified directory. If you omit this argument, the directory is indeterminate and Orchestrate generates a warning message. You can use <code>-noworkingdirectory</code> to disable the warning. This option also determines the location of the file <code>config.sasversion</code>. By default, the operator searches the directory specified by <code>-workingdirectory</code>, then your home directory, then the SAS install directory for <code>config.sasversion</code>. Legal values for dir_name are fully qualified pathnames (which must be valid on all processing nodes) or &quot;.&quot; (period), corresponding to the name of the current working directory on the workstation invoking the application. Relative pathnames for dir_name are illegal.</td>
</tr>
</tbody>
</table>
The sasout Operator

You use the sasout operator to convert an Orchestrated SAS data set to the standard Orchestrate data set format suitable for input to standard Orchestrate operators. When you data includes ustring values, You must set the -sas_cs option of the SAS interface operators to specify the character set used by Orchestrate to convert between the char data stored in SAS files and Orchestrate ustring values. Use the same value for this option for the sasin, sas, and sasout operators.

This operator is only required when a standard Orchestrate operator follows a SAS interface operator.

The sasout operator requires that you specify either the -schema option or the -schemaFile option to pass a record schema that defines the layout of its output data set. You supply an Orchestrate schema definition to the -schema option. For the -schemaFile option, you specify the file path of a SAS file that has the same meta data description as the SAS output stream, and Orchestrate derives the schema definition from that file. See “Specifying an Output Schema” on page 32-32 for more details.

See the section “Data Flow Diagram” on page 32-44 for information on the schema that is derived from a SAS file when you use the -schemaFile option.

As part of converting an Orchestrated SAS data set to a standard Orchestrate data set, the sasout operator converts input SAS data types to the corresponding Orchestrate data types using that record schema.

For example, if the Orchestrated SAS data set contains a SAS numeric field named a_field that you want to convert to an int16, you include the following line as part of the sasout record schema:

```
record (
    . .
    a_field:int16;
    . .
)
```

If you want to convert the field to a decimal, you would use the appropriate decimal definition, including precision and scale.

When converting a SAS numeric field to an Orchestrate numeric, you can get a numeric overflow or underflow if the destination data type is too small to hold the value of the SAS field. By default, Orchestrate issues an error message and aborts the program if this occurs. However, if the record schema passed to sasout defines a field as nullable, the numeric overflow or underflow does not cause an error. Instead, the destination field is set to null and processing continues.
Data Flow Diagram

input Orchestrated SAS data set

sasout

output Orchestrate data set

Properties

Table 162  sasout Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1 Orchestrated SAS data set</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1 Orchestrate data set</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>none</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>as specified by the -schema option or the -schemaFile option</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>none</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>any (parallel mode)</td>
</tr>
<tr>
<td>Collection method</td>
<td>any (sequential mode)</td>
</tr>
<tr>
<td>Preserve-partitioning flag in</td>
<td>clear</td>
</tr>
<tr>
<td>output data set</td>
<td></td>
</tr>
</tbody>
</table>

Syntax and Options

The syntax for the sasout operator is shown below. Either the -schema option or the -schemaFile option is required to supply an output schema. When using sasout, do not specify an output schema to the sas operator.

Terms in italic typeface are option strings you supply. When your option string contains a space or a tab character, you must enclose it in single quotes.
sasout
 *schema* schema_definition | *schemaFile* filepath

[sas_cs icu_character_set | SAS_DBCSLANG]
[context prefix_string]
[debug no | yes | verbose]
[drop | nodrop]
[rename schema_field_name SAS_name...]
[report]

Table 163 sasout Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-schema</td>
<td>-schema schema_definition&lt;br&gt;SasOut specifies the record schema of the output Orchestrate data set. Only those fields specified in the record schema are written to the output data set. You must specify either the -schema option or the -schemaFile option. See “Specifying an Output Schema” on page 32-32 for more details.</td>
</tr>
<tr>
<td>-schemaFile</td>
<td>schemaFile filepath&lt;br&gt;You use the -schemaFile option to specify the name of a SAS file containing the metadata column information which Orchestrate uses to generate an Orchestrate schema; or you use the -schema suboption followed by the schema definition. See “Specifying an Output Schema” on page 32-32 for more details. If both the -schemaFile option and the -sas_cs option are set, all of your SAS char fields are converted to Orchestrate ustring values. If the -sas_cs option is not set, all of your SAS char values are converted to Orchestrate string values. To obtain a mixture of string and ustring values use the -schema option.</td>
</tr>
<tr>
<td>-debug</td>
<td>debug -yes</td>
</tr>
</tbody>
</table>
Table 163  sasout Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-drop</td>
<td>-drop</td>
</tr>
<tr>
<td></td>
<td>Specifies that sasout drop any input fields not included in the record schema. This is the default action of sasout. You can use the -nodrop option to cause sasout to pass all input fields to the output data set. The -drop and -nodrop options are mutually exclusive.</td>
</tr>
<tr>
<td>-nodrop</td>
<td>-nodrop</td>
</tr>
<tr>
<td></td>
<td>Specifies the failure of the step if there are fields in the input data set that are not included in the record schema passed to sasout. You can use the -drop option to cause sasout to drop all input fields not included in the record schema.</td>
</tr>
<tr>
<td>-rename</td>
<td>-rename in_field_name Orchestrated_field_name...</td>
</tr>
<tr>
<td></td>
<td>Specifies a mapping of an input field name from the Orchestrated SAS data set to an Orchestrate data set field name. For multiple mappings, use multiple -rename options. Aliases must be unique with all other alias specifications and with respect to the Orchestrated SAS data set field names.</td>
</tr>
<tr>
<td>-report</td>
<td>-report</td>
</tr>
<tr>
<td></td>
<td>Causes the operator to output a report describing how it converts the input Orchestrated SAS data set to the Orchestrate data set.</td>
</tr>
<tr>
<td>-sas_cs</td>
<td>-sas_cs icu_character_set</td>
</tr>
<tr>
<td></td>
<td>When your Orchestrate data includes ustring values, you can use the -sas_cs option to specify a character set that maps between Orchestrate ustrings and the char data stored in SAS files. Use the same -sas_cs character setting for all the SAS-interface operators in your data flow. See “Using -sas_cs to Specify a Character Set” on page 32-30 for more details. For information on national language support, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site: <a href="http://oss.software.ibm.com/">http://oss.software.ibm.com/</a> icu/ charset</td>
</tr>
</tbody>
</table>
The `sascontents` Operator

You use the `sascontents` operator to generate a report about an input self-describing data set.

The report is similar to the report generated by the SAS procedure PROC CONTENTS. It is written to the data stream. Use the `peek` operator in sequential mode to print the report to the screen.

The operator takes a single input data set and writes the generated report to output 0. Optionally, the input data set can be written to output 1 with no record modification.

Data Flow Diagram

![Data Flow Diagram](image)

Properties

Table 164  `sascontents` Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1 self-describing data set</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1 or 2</td>
</tr>
<tr>
<td></td>
<td>output 0: generated report</td>
</tr>
<tr>
<td></td>
<td>output 1: an optional copy of the input data set with no record modification</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>none</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>none</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>input to output with no record modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>same (cannot be overridden)</td>
</tr>
</tbody>
</table>
The syntax for the `sascontents` operator is shown below. No options are required. Terms in italic typeface are option strings you supply. When your option string contains a space or a tab character, you must enclose it in single quotes.

```
sascontents
  [-name name_string_for_report]
  [-recordcount]
  [-schema]
```

### Example Reports

The names in these examples can contain a maximum of 8 characters for SAS 6.12 and a maximum of 32 characters for SAS 8.2.

- **Report with no options specified:**

  **Field Count:** 2  **Record Length:** 13  **Created:** 2003-05-17 15:22:34  **Version:** 4

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Type</th>
<th>Len</th>
<th>Offset</th>
<th>Format</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>Num</td>
<td>8</td>
<td>0</td>
<td>6.</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>Char</td>
<td>5</td>
<td>8</td>
<td></td>
<td>b</td>
</tr>
</tbody>
</table>
• **Report with the `-schema` option specified:**

  Suggested schema for Orchestrated SAS dataset `sasin[00].v`  
  record (A:int32;  
           B:string[5])

• **Report with the `-schema` and `-recordcount` options specified:**

  Suggested schema for Orchestrated SAS dataset `sasin[00].v`  
  Number of partitions = 4  
  Number of data records = 10  
  Suggested schema for parallel dataset `sasin[00].v`  
  record (A:int32;  
           B:string[5])  
  Field descriptor records = 8  
  Total records in dataset = 22

• **Report with the `-schema`, `-recordcount` and `-name` options specified. The name value is 'my data set':**

  Suggested schema for Orchestrated SAS dataset `my data set`  
  Number of partitions = 4  
  Number of data records = 10  
  Suggested schema for parallel dataset `sasin[00].v`  
  record (A:int32;  
           B:string[5])  
  Field descriptor records = 8  
  Total records in dataset = 22
The Statistics Library

Describes the four operators in the Statistics Library which enable you to produce a crosstabulation of a data set, to summarize the characteristics of a group of records, to calculate basic statistical values on fields, and to perform data transform operations.

Overview of the Statistics Operators 33 3

The crosstab Operator 33 4
  Introduction 33 4
  Grouping Overview 33 4
    Defining a Group 33 5
    Specifying Aggregations 33 6
  Data Flow Diagram 33 7
  Properties 33 7
  Syntax and Options 33 8
  Modes of Aggregation 33 11
  Specifying a Weighting Field 33 12
  Re-aggregating and Two-phase Grouping 33 13
  Limiting Memory Consumption 33 13
  Generating Statistics 33 13
  Example 1: Using the crosstab Operator without Clustered Keys 33 13
  Example 2: Using the crosstab Operator with Clustered Keys 33 14

The group Operator 33 16
  Introduction 33 16
  The Grouping Operation 33 17
  Defining a Group 33 18
  Creating a Summary 33 19
  Alternatives: The crosstab and statistics Operators 33 20
  Operator Action 33 21
  Data Flow Diagram 33 22
  Properties 33 22
  group Operator Modes 33 23
  Running the group Operator 33 23
Syntax and Options 33 24
Examples 33 29
Example 1: Using The group Operator in hash Mode 33 29
Example 2: Using The group Operator in sort Mode 33 30
Example 3: Specifying a Weight Field 33 31
Example 4: Regrouping Data and Creating Summary Statistics 33 32
Example 5: Two-phase Grouping with hash Mode 33 35

The statistics Operator 33 38
Statistics Calculation and Data Types 33 38
Using the statistics Operator 33 39
Data Flow Diagram 33 40
Properties 33 40
Syntax and Options 33 40
Using the readstats Utility to Retrieve Statistics Results 33 41
Syntax and Options 33 41
Example: Using the Statistics Tools 33 43
Calculating Statistics on String Fields 33 44

The fieldtransform Operator 33 46
Overview 33 46
General Transformation Process 33 47
Using the fieldtransform Operator 33 49
Data Flow Diagram 33 49
Properties 33 50
Field Transform Operator and Nulls 33 50
Syntax and Options 33 50
Performing Z-score Transforms 33 57
Using the Z-score Transform 33 58
Forward Z-Score Transform: First Step 33 59
Forward Z-Score Transform: Second Step 33 59
Inverse Z-score Transform 33 60
Configuring Z-score Transforms 33 61
Extremely Large or Small Field Values 33 61
Standard Deviation of Zero 33 61
Performing Z-sigmoid and Z-symmetric Transforms 33 62
Using the Transforms 33 63
Forward Z-sigmoid Transform: First Step 33 64
Forward Z-sigmoid Transform: Second Step 33 65
Inverse Z-sigmoid Transform 33 65
Standard Deviation of Zero 33 66
Standard Deviation of Zero for Z-sigmoid Transforms 33 67
Standard Deviation of Zero for Z-symmetric Transforms 33 67
Performing Identity Transforms 33 67
Transforming Nominal Fields 33 68
Forward Nominal Transforms 33 68
Inverse Nominal Transforms 33 70
Overview of the Statistics Operators

This chapter of the Orchestrate 7.0 Operators Reference describes how to use the statistics operators. To understand the information in this chapter you should be familiar with the operator concepts defined in the chapter on operators in the Orchestrate 7.0 User Guide.

The statistics library contains four operators:

- The crosstabulation operator, `crosstab`, produces a crosstabulation of an input data set. A crosstabulation, also referred to as a contingency table, is a count of the number of records in each group of the input data set, where a group is determined by one or more grouping key fields. This operator is described on page 33-4.

- The data grouping operator, `group`, summarizes the characteristics of a group by applying a reduction function, or reducer, to each record in the group. One method for determining patterns in a collection of records is to group records with similar characteristics and then compare the characteristics of records in the same group. This operator is described on page 33-16.

- The `statistics` operator calculates basic statistical values for one or more fields in a data set. Calculated values include mean, minimum, maximum, sum, variance, standard deviation and error, and the number of valid and missing values. This operator and the `readstats` utility is described on page 33-38.

- The `fieldtransform` operator performs data transform operations. A transform calculates one or more output fields based on a single input field value. Transforms are often performed as a preliminary step to data modeling or other data analysis operations. The predefined transforms include Z-score calculation, sigmoid calculation, and nominal and ordinal field transformations. This operator is described on page 33-46.
The crosstab Operator

Introduction

The crosstab operator produces a crosstabulation of an input data set. A crosstabulation (also referred to as a contingency table) is a count of the number of records in each group of an input data set, where a group is determined by one or more key fields.

This chapter first defines aggregating, or grouping, data and describes how to define aggregations within a data set. The chapter then shows how to use the crosstab operator to perform a crosstabulation on a data set.

The input data set is expected to be sorted by the clustered key and hash-partitioned on all keys. By default, Orchestrates inserts partition and sort components to meet the partitioning and sorting needs of the crosstab operator and other operators. See the section “Orchestrate-Inserted Partition and Sort Components” on page xxxvi of this Reference for information on this facility.

Grouping Overview

One of the easiest ways to expose patterns in a collection of records is to group records with similar characteristics and then compare characteristics of each group. For example, records containing cash register transactions might be grouped by day of the week to see which day had the largest number of transactions, the greatest revenue, and so forth.

Grouping is a common data analysis technique. It is supported by spreadsheets (column summation), in statistical packages (the SAS PROC FREQ and PROC MEANS functions), and in SQL (using the GROUP BY clause in a statement).

Records can be grouped by several characteristics at the same time. For example, transaction records might be grouped by both day of the week and month. These aggregations might show that the busiest day of the week varies according to season.

In addition to revealing patterns in your data, grouping is also a way of reducing the volume of data by summarizing it. For example, if you group a large volume of data on the basis of one or more characteristics, the resulting data set is typically much smaller than the original and therefore easier to analyze using standard workstation- or PC-based tools.

Crosstabulation is a form of summarization in which the characteristic computed for each group is a count of the number of records in that group. See “The group Operator” on page 33-16 for information on other types of summarizations.
Defining a Group

A group is defined as all records of a data set with the same value for one or more grouping key fields. A grouping key field can be any field in an input data set, but it is generally a field with a relatively small number of possible values, such as gender, state, or account type.

For example, consider the following schema of a data set:

```
record (
  customerId: dfloat;
  gender: string[1]; // M or F
  address: string[80];
  state: string[2];
  age: uint8;
  transactionAmount: dfloat;
)
```

Each record of the data set contains information about a single credit card transaction as well as information about the customer. For each customer, the data set can contain multiple records, corresponding to multiple transactions.

If you specify `gender` as the grouping key field for this data set, your output data set will consist of only two records: one for males and one for females. The following figure shows the operation of the `crosstab` operator for this example:

Output data set 0 contains two records: one for each group. Included in the records of the output data set is the `recordCount` field, which contains a count of the number of records in each group.
If you instead specify customerID as the grouping key field, the output data set contains a single record for every unique customer ID.

You can combine grouping key fields as well. For example:

- **gender** and **state**: Aggregate the records according to purchases made by males and females in each state. The output data set then contains a maximum of 100 (2 * 50) records. The number of output records would actually be 100 only if you had records for both a man and a woman from each of the 50 states.

- **gender**, **state**, and **age**: Aggregate the records by these three fields. The output data set would contain a maximum number of records equal to 2 * 50 * (the number of unique ages of customers).

- **customerID** and **state**: Aggregate the records by customer ID in each of the 50 states. The output data set would contain a maximum of 50 * (the number of customers), but is likely to be much smaller because not every customer will have a transaction in every state.

**Specifying Aggregations**

You use options to the **crosstab** operator to identify the aggregating key fields in an input data set. You can specify some of the same key fields in more than one aggregation; however, within a given aggregation, each field in an input data set can be used only once as a key.

The output produced by grouping on two fields of an input data set is referred to as a two-dimensional or two-way crosstabulation table. In a given aggregation, you can group on as many fields as are defined in the schema of the input data set, to create an N-way crosstabulation table.

You must specify at least one aggregation, defined by at least one key field. A field specified as a grouping key must be a top-level field, and may not be a vector. It may be nullable. See the listing of options in Table 167 below for more detailed information.
As may be seen in this figure, the crosstab operator:

- Takes as input a single data set.
- Produces as output one data set for each aggregation defined, which contains a single record for each group, with the key field values for that group plus a new field, named recordCount which contains the number of records in the group. You must provide one output data set for each aggregation, to hold the operator’s results.

### Properties

**Table 166 crosstab Operator Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>$1 \leq n$, where $n$ is the number of aggregations defined</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>aggr0key0;...aggr0keyN;...aggrMkey0;...aggrMkeyN;[weightfield;]</td>
</tr>
<tr>
<td></td>
<td>Since the same keys may be used to define more than one aggregation, the input schema is actually the union of all the keys specified.</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>aggr0key0;...aggr0keyN; recordCount:dfloat; aggr1key0;...aggr1keyN; recordCount:dfloat; etc.</td>
</tr>
<tr>
<td></td>
<td>recordCount is the default name of the output field for each aggregation; you may specify a different name for this field.</td>
</tr>
<tr>
<td>Composite operator</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Syntax and Options

The syntax for the `crosstab` operator is shown below. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose the value in single quotes.

```
crosstab
  ·aggr
    ·key fieldname [-ci | -cs] [-param params] [-clustered]
    [-key fieldname [-ci | -cs] [-param params] [-clustered] ...]
    [-collation_sequence locale| collation_file_pathname| OFF]
    [-countField count_field_name]
    [-weightField weight_field_name]
    [-memory mbytes]
    [-noflush]
    [-weightField weight_field_name]
    [-aggr suboptions...]
    [-stats]
```

Each aggregation must be followed by its keys, each of them tagged as clustered or not.

The fields `-countField`, `-weightField`, `-noflush`, and `-memory` are optional arguments specific to each aggregation. The `-stats` option applies to the entire `crosstab` operation.
Table 167  crosstab Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-aggr</td>
<td>-aggr</td>
</tr>
<tr>
<td></td>
<td>· key fieldname [-ci</td>
</tr>
<tr>
<td></td>
<td>· key fieldname [-ci</td>
</tr>
<tr>
<td></td>
<td>· countField count_field_name</td>
</tr>
<tr>
<td></td>
<td>· weightField weight_field_name</td>
</tr>
<tr>
<td></td>
<td>· memory mbytes</td>
</tr>
<tr>
<td></td>
<td>· noflush</td>
</tr>
<tr>
<td></td>
<td>· weightField weight_field_name</td>
</tr>
<tr>
<td></td>
<td>[-aggr suboptions ...]</td>
</tr>
</tbody>
</table>

This option defines an aggregation by listing the key fields to be grouped, designating them as clustered, if appropriate, and supplying other optional elements: a count field name, a weight field name, and a memory limit.

You must define at least one aggregation, and for a given aggregation, you must define at least one -key option.

Each -key suboption specifies one grouping key for a given aggregation. To specify additional keys, you repeat the -key entry for each. The crosstab operator defines a group as all records with the same values for all grouping keys. For each aggregation, you can specify as many grouping keys as you like, up to the number in the input schema.

Each key field must be a top-level field in the schema of the input data set. This means that a field may not be part of a subrecord or tagged aggregate field. It may not be a vector. Keys may be nullable fields. By default, crosstab is case sensitive. You use the -ci option to override this default in order to perform case-insensitive grouping on a string field.

The -clustered flag indicates that successive records with the same value for the clustered field are to be grouped, with aggregation results output each time the value of a clustered key field changes.

The -param option stipulates extra parameters for key comparison. Specify property=value pairs without curly brackets.

The -countField, -weightField, noflush, and -memory suboptions to -aggr are explained under their option names in this table.

Brackets indicate that an option is optional; there are no such symbols in actual option input.
Table 167  crosstab Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-collation_sequence</td>
<td>-collation_sequence locale</td>
</tr>
<tr>
<td></td>
<td>This option determines how your string data is sorted. You can:</td>
</tr>
<tr>
<td></td>
<td>• Specify a predefined IBM ICU locale</td>
</tr>
<tr>
<td></td>
<td>• Write your own collation sequence using ICU syntax, and supply its collation_file_pathname</td>
</tr>
<tr>
<td></td>
<td>• Specify OFF so that string comparisons are made using Unicode code-point value order, independent of any locale or custom sequence.</td>
</tr>
</tbody>
</table>

By default, Orchestrate sorts strings using byte-wise comparisons.

For more information, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:


-countField | count_field | count_field_name
|-------------|-------------|----------------|
| This optional suboption to the -aggr option specifies the name of the count field added to the output data set. If you do not specify a name, the new field is called recordCount. The data type of the field is always dfloat.

-countField requires that count_field_name not be the same as any of the key field names.

-memory | mbytes
| This optional suboption to the -aggr option specifies the number of megabytes of memory beyond which disk storage (the resource sort scratchdisk defined in your config file) is used to process an aggregation. The value must be less than or greater than 2000.

Since this option is meaningful only if there are one or more non-clustered keys, it issues a warning if all keys are clustered, and the value of mbytes is ignored. If there are no clustered keys defined, the default value of mbytes is 20. If there are some clustered keys, mbytes defaults to 0, which puts no limit on the amount of memory used. This is based on an assumption that the presence of clustered keys considerably reduces memory consumption.

-noflush | noflush
| This optional suboption to the -aggr option specifies that output at cluster boundaries should not be flushed.
The crosstab operator may be used to specify more than one aggregation in a single run. This is equivalent to running the operator multiple times, but more efficient.

For each aggregation, a subset of the grouping keys may be designated as clustered keys. If there are keys designated as clustered, input to the operator is expected to be pregrouped on those keys, that is, all records with a given value for a clustered key should already be together in the data set. If the data is not organized in this way, do not mark keys as clustered.

If there are clustered keys, successive input records with the same value in the clustered keys are treated as a batch, and the aggregation results are output each time the value of a clustered key changes. Non-clustered keys are aggregated in the manner of the hashed group operator. (See “group Operator Modes” on page 23.)

---

### Table 167  crosstab Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-stats</td>
<td>Causes the operator to print statistics to standard error about various aspects of its execution:</td>
</tr>
<tr>
<td></td>
<td>• Average count field value—shows the degree of data reduction achieved.</td>
</tr>
<tr>
<td></td>
<td>• If there are non-clustered keys, the maximum number of hash entries used.</td>
</tr>
<tr>
<td></td>
<td>• If there are clustered keys, the number of distinct clusters.</td>
</tr>
<tr>
<td>-weightField</td>
<td>Specifies a weight field used by the operator. Typically, each record in a group increments the group count by one. However, you can use -weightField to specify that the group count be incremented by a field value for each record. Weight_field_name is a field in the schema of the input data set used as the weight field.</td>
</tr>
<tr>
<td></td>
<td>The use of -weightField requires that:</td>
</tr>
<tr>
<td></td>
<td>• Weight_field_name be present in the schema of the input data set.</td>
</tr>
<tr>
<td></td>
<td>• The data type of weight_field_name be convertible to dfloat.</td>
</tr>
<tr>
<td></td>
<td>• Weight_field_name not be a nullable field.</td>
</tr>
<tr>
<td></td>
<td>See “Specifying a Weighting Field” on page 12 for more information.</td>
</tr>
</tbody>
</table>

---

### Modes of Aggregation

The crosstab operator may be used to specify more than one aggregation in a single run. This is equivalent to running the operator multiple times, but more efficient.

For each aggregation, a subset of the grouping keys may be designated as clustered keys. If there are keys designated as clustered, input to the operator is expected to be pregrouped on those keys, that is, all records with a given value for a clustered key should already be together in the data set. If the data is not organized in this way, do not mark keys as clustered.

If there are clustered keys, successive input records with the same value in the clustered keys are treated as a batch, and the aggregation results are output each time the value of a clustered key changes. Non-clustered keys are aggregated in the manner of the hashed group operator. (See “group Operator Modes” on page 23.)
There are therefore three modes of aggregation:

- **No clustered keys**: the operator allocates an entry for each distinct key combination seen and outputs the aggregation results (in unspecified order) when all input has been consumed. This uses storage proportional to the total number of distinct groups.

- **Some keys are clustered**: like the previous case, except that aggregation results are output (in unspecified order) each time a clustered key changes value. This uses storage proportional to the number of distinct groups in a clustered key run.

- **All keys are clustered**: aggregation results are output each time a key changes value. This uses constant storage.

The first mode is analogous to the `group` operator in `hash` mode; the third is analogous to the `group` operator in `sort` mode.

### Specifying a Weighting Field

The input interface schema of the `crosstab` operator can contain a weight field. If you specify a weight field, the count for the group is incremented by the weight field value for each record, rather than by 1.

The weight field is useful if some presummarization has already been performed on the input data set. For example, assume that each record of the input data set contains information about the credit card transactions for an individual on a specific date. The schema for the input data set is shown below:

```plaintext
record {
    customerId: dfloat;
    gender: string[1];
    address: string[80];
    state: string[2];
    age: uint8;
    transactionDate: uint32;
    transactionCount: uint16;  // number of transactions
} // number of transactions
```

You could still perform crosstabulation on age and gender, but now the `transactionCount` field would be used as a weight field. Each record in a group increases the group count not by 1, but by the value of `transactionCount`. In this example, using a weight field causes the operator to produce a count of the number of transactions for each group, rather than the number of records for each group.

To specify a weight field, you use the `-weightField` option. For the schema shown above, you would use the following `osh` statement:

```bash
$ osh '... | crosstab -aggr -key age
    -key gender
    -weightField transactionCount | ...''
```
Re-aggregating and Two-phase Grouping

It is often useful to group data once, then re-aggregate, perhaps using the recordCount field of the first aggregation as a weight field. Often you will do the first aggregation in parallel and the second sequentially. These techniques are discussed in the chapter on the group operator, in “Example 4: Regrouping Data and Creating Summary Statistics” on page 33-32 and in “Example 5: Two-phase Grouping with hash Mode” on page 33-35.

Limiting Memory Consumption

When there are one or more non-clustered keys, the -memory option can be used to place an upper limit (in megabytes) on the amount of memory allocated to store entries; if the number of distinct groups is such that the memory limit is exceeded, disk storage will be used in addition to memory to process the groups.

Since this option is meaningful only if there are one or more non-clustered keys, if you set this option when all keys are clustered, the operator issues a warning and the value you specify is ignored.

If there are no clustered keys defined, the default value is 20. If there are some clustered keys, the default value is 0, which puts no limit on the amount of memory used. This is based on an assumption that the presence of clustered keys considerably reduces memory consumption.

Generating Statistics

Setting the -stats option tells the operator to print a number of statistics about the crosstabulation it has just performed. These are:

- Average count field value shows the degree of data reduction achieved.
- If there are non-clustered keys, the maximum number of hash entries is used.
- If there are clustered keys, the number of distinct clusters is used.

This information is printed to standard error.

The following examples illustrate the use of crosstab for reducing and summarizing data.

Example 1: Using the crosstab Operator without Clustered Keys

Here is a simple example use of the crosstab operator. In this example the operator is used to count the number of records in each group defined by the grouping keys gender, zipcode, and age. By default, Orchestrates inserts partition and sort components to meet the partitioning and sorting needs of the crosstab
operator and other operators. See the section “Orchestrate-Inserted Partition and Sort Components” on page xxxvi of this Reference for information on this facility.

The figure below shows the step for this example:

![Diagram of data flow](image)

The output data set of the crosstab operator, `outData0 ds`, contains one record for each unique group, that is, each unique combination of values of the fields gender, zipcode, and age. Each record in this data set contains the three grouping key values for the group, and a new field, recordCount, containing the number of records in the group. Grouped by gender, zipcode, and age, this output data set could have as many as 2 * 100,000 * (the number of unique ages in data set) records, if every zip code were represented.

Shown below is the `osh` command for this example.

```
$ osh " crosstab -aggr -key gender
- key zipcode
- key age
inData ds > outData0 ds "
```

**Example 2: Using the crosstab Operator with Clustered Keys**

Here is an example use of the crosstab operator with one clustered key. Suppose that in the same data set as for the previous example, the input data set is already organized by zip code, having been sorted or generated that way. As in Example
1. Orchestrates inserts partition and sort components to meet the partitioning and sorting needs of the `changeperture` operator and other operators. See the section “Orchestrate-Inserted Partition and Sort Components” on page xxxvi of this Reference for information on this facility.

It is useful to designate such a field as `-clustered` for the purposes of grouping. This provides certain advantages:

- Since each output record is written as soon as the end of a cluster is reached, the amount of memory required by the operator is very small.
- In addition, incremental output means that there is less waiting for results, which can improve overall throughput.

In this example we use the operator to count the number of records in each group defined by the grouping keys `zipcode` and `age`, to correlate customer ages with their locations.

The figure below shows the step for this example:

Again, the output data set contains a record for each unique group, in this case defined by the fields `zipcode` and `age`. Each record in this data set contains the two grouping key values for the group, and an additional field, `recordCount`, containing the number of records in the group. As soon as all the records for a particular `zipcode` have been created, they are output immediately, so memory use is proportional to the number of different ages encountered within the same `zipcode`. 

The figure below shows the step for this example:

```
step
inData.ds data set
schema:
customerID:dfloat;
gender:string[1];
zipcode:string[5];
state:string[2];
age:uint8;

outData0.ds data set
schema:
zipcode:string[5];
age:uint8;
recordCount:dfloat;
```

The figure below shows the step for this example:
Here is the `osh` command for this example.

```bash
$ osh " crosstab -aggr -key zipcode -clustered
    -key age
< inData.ds > outData0.ds "  
```

The group Operator

One way of finding patterns in a collection of records is to group records with similar characteristics, then compare the characteristics of the various groups. In Orchestrate, you can determine the characteristics of a group by using the `group` operator to calculate a new field containing a summary of statistical values of all records in the group.

One kind of summary value is a count of the number of records in each group. Other summaries are calculated from specific fields of each record in a group. You can calculate statistics for a field in a group to determine the maximum, minimum, sum, mean, and standard deviation, among others, for the field over the entire group.

Introduction

One of the easiest ways to expose patterns in a collection of records is to group records with similar characteristics, then compute statistics on all records in the group. You can then use these statistics to compare properties of the different groups. For example, records containing cash register transactions might be grouped by the day of the week to see which day had the largest number of transactions, the largest amount of revenue, etc.

Records can be grouped by one or more characteristics, where record characteristics correspond to field values. In other words, a group is a set of records with the same value for one or more fields. For example, transaction records might be grouped by both day of the week and by month. These groupings might show that the busiest day of the week varies by season.

In general terms, grouping performs data summarization. Grouping is a common data analysis technique and appears in spreadsheets (column summation), statistical packages (the SAS PROC FREQ and PROC MEANS functions), and in SQL (using the GROUP BY clause).

In addition to revealing patterns in your data, grouping can also reduce the volume of data by summarizing the records in each group, making it easier to manage. If you group a large volume of data on the basis of one or more characteristics of the data, the resulting data set is generally much smaller than the original and is therefore easier to analyze using standard workstation or PC-based tools.
The Grouping Operation

The input to the `group` operator is a data set to be summarized; the output is a data set containing one record for each group in the input data set. Each output record contains the fields that define the group and the output summaries calculated by the operator. One simple summary is a count of the number of records in each group. Another kind of summary is a statistical value calculated on a particular field for all records in a group.

As an example, the simplest statistic you can calculate is the number of records in each group. Shown below is the schema for a data set containing a list of customer transactions, where the schema contains a field for the gender of each customer:

```plaintext
record {
    customerId: int32;
    gender: string[1]; // 'M' or 'F'
    state: string[2]; // 'AK', 'AL', etc.
    // any other fields
}
```

For this input data set, you can calculate the number of records for all males and the number of records for all females. The following figure shows the operation of the `group` operator for this example:

If you group the records of the input data set by gender, you create two groups: one for males and one for females. Therefore, the output data set contains two records: one for each group. Note that if you choose to group records by the state field, you would have at most 50 records in the output data set; one for each of the 50 possible values for the state field.

Included in the records of the output data set are summary fields, of type dfloat, containing information about each group. In the example shown above, the summary fields consist of a single field holding the record count of each group.
data set containing a count of the number of records in each group is also referred to as a frequency table.

You can group records by more than one field. For example, if you group by the gender and state fields, the operator would generate an output data set containing at most 100 records (2 * 50). This is shown in the following figure:

In this example, the output data set contains one record for each unique combination of values for the gender and state fields. Note that 100 is the maximum number of records produced, and is reached only if the input data set contains records for at least one male and one female for each of the 50 states.

The output produced by grouping on two or more fields of an input data set is also referred to as a crosstabulation or contingency table. In this example, records were grouped by two fields, producing a 2-dimensional or 2-way crosstabulation table:

<table>
<thead>
<tr>
<th>AL</th>
<th>AK</th>
<th>AZ</th>
<th>. . .</th>
<th>WV</th>
<th>WI</th>
<th>WY</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>count</td>
<td>count</td>
<td>count</td>
<td>. .</td>
<td>count</td>
<td>count</td>
</tr>
<tr>
<td>M</td>
<td>count</td>
<td>count</td>
<td>count</td>
<td>. .</td>
<td>count</td>
<td>count</td>
</tr>
</tbody>
</table>

**Defining a Group**

A group is defined as all records of a data set with the same value(s) for zero or more grouping key fields. A grouping key field can be any field in an input data set, but it is usually a field with a relatively small number of possible values, such as
gender, state, account type, and so forth. The more values in each field, the more
groups are produced.

For example, here is the schema of a data set that you want to summarize:

```plaintext
record (  
  customerID: int32;  
  gender: string[1]; // 'M' or 'F'  
  address: string[80];  
  state: string[2];  
  age: uint8;  
  transactionAmount: dfloat;  
)
```

Each record of the data set contains information about a single credit card
transaction as well as information about the credit card customer. For each
customer, the data set can contain multiple records corresponding to multiple
transactions.

You can specify zero or more grouping key fields; however, each field in an input
data set can only be used once as a grouping key. The maximum number of
grouping keys you can specify is the number of fields in the schema of an input
data set. If you do not specify any grouping keys, all records are members of a
single group and statistical values are computed by the operator over the entire
input data set.

For example, if you specify `gender` as the grouping key field for this data set, your
output data set will consist of only two records: one for all males and one for all
females. If you specify `customerID` as the grouping key field, the output data set
will contain one record for every unique customer ID.

You can combine grouping key fields as well. For example:

- **gender, state, and age**: You could use this grouping to determine patterns in
  credit card usage based on the gender, geographical location, and age of
  customers. The output data set would contain a maximum number of records
  equal to 2 * 50 * (the number of unique ages of customers).

- **customerID and state**: Group the records by customer in each of the 50 states.
  This grouping may enable you to identify customers who travel frequently.
  The maximum number of records in the output data set is 50 * (the number of
  customers).

### Creating a Summary

The **group** operator first groups the records of a data set according to the
grouping key fields. Once the groups have been defined, the operator calculates
summary information about each group or fields being reduced or rereduced.
Each record of the output data set can contain one or more summary fields, of type \texttt{dfloat}, including:

- count of null values of field of interest
- count of non-null values
- maximum value of field of interest
- mean value of field of interest for entire group: \( \bar{x} \)
- minimum value of field of interest
- number of records in group
- standard deviation: \( s \)
- standard error: \( s/\sqrt{n} \)
- percent coefficient of variation: \( 100s/\bar{x} \)
- range of field of interest: (maximum - minimum)
- sum of field of interest for entire group: \( \sum x_i \)
- sum of weights: \( \sum w_i \)
- sum of squares corrected for the mean: \( \sum w_i(x_i - \bar{x})^2 \)
- uncorrected sum of squares of values: \( \sum w_ix_i^2 \)
- variance: \( s^2 \)

In this table, the term \( w \) corresponds to a weight value, if one has been set. See “Example 3: Specifying a Weight Field” on page 33-31 for more information.

\textbf{Alternatives: The crosstab and statistics Operators}

Orchestrate provides two other operators, the \texttt{crosstab} operator and the \texttt{statistics} operator, that can perform actions similar to those of the \texttt{group} operator. The \texttt{crosstab} operator produces a crosstabulation of an input data set. A crosstabulation, also called a contingency table, is a count of the number of records in each group of the input data set, where a group is determined by one or more grouping key fields. You may want to use \texttt{crosstab} if you want only to count the number of records in each group. See “The crosstab Operator” on page 33-4 for more information.

The \texttt{statistics} operator calculates statistical values for the records of a data set. Note that the \texttt{statistics} operator calculates statistics on a data set as a whole; the
**group** operator allows you to calculate statistics on each group of a data set as defined by the grouping key fields. See “The statistics Operator” on page 33-38 for more information.

**Operator Action**

The input to the **group** operator is a data set to be summarized; the output is a data set containing one record for each combination of grouping key fields. Each output record contains (a) the grouping key values that define the group and (b) the summary fields, of type **dfloat**, created by the operator.

The following figure shows the operation of the **group** operator for a single grouping key field:

![Diagram of group operator operation]

Included in the records of the output data set are summary fields containing information about each group. The type of information contained in the summary fields depends on which options you specify.

The output data set is required by the **group** operator; you must supply an output data set to hold the operator’s results.
### Data Flow Diagram

As shown in this figure, the **group** operator:

- Takes as input a single data set. The names of the keys are set by means of parameter arguments to the **group** operator.

- Produces as output a data set containing a single record for each group and any new fields, of type **dfloat**, added by the operator.

The schema of the output data set contains a field for every grouping key, as defined in the input interface schema, and a field for every summary field added by the operator.

### Properties

**Table 168  group Operator Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>groupKey0; ... groupKeyn; reduceField0; ... reduceFieldn;</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>groupKey0; ... groupKeyn; reduceStat0_a; reduceStat0_b; ... reduceStatn_a; reduceStatn_b; ...</td>
</tr>
</tbody>
</table>
The group operator has two modes: hash and sort. Your choice of mode depends primarily on the number of groupings in the input data set, taking into account the amount of memory available. You typically use hash mode for a relatively small number of groups; generally, fewer than about 1000 groups per megabyte of memory to be used.

When using hash mode, you should hash partition the input data set by one or more of the grouping key fields so that all the records in the same group are in the same partition. However, you are not required to hash partition the input data set; you can use any partitioning method you choose, if keeping groups together in a single partition is not important. For example, if you’re summing records in each partition and later you’ll add the sums across all partitions, you don’t need all records in a group to be in the same partition to do this. Note, though, that there will be multiple output records for each group, one from each player.

If the number of groups is large, which can happen if you specify many grouping keys, or if some grouping keys can take on many values, you would normally use sort mode. However, sort mode requires the input data set to have been partition sorted with all of the grouping keys specified as hashing and sorting keys. Sorting requires a pregrouping operation: after sorting, all records in a given group in the same partition are consecutive.

You may want to try both modes with your particular data and application to determine which gives the better performance. You may find that when calculating statistics on large numbers of groups, sort mode performs better than hash mode, assuming the input data set can be efficiently sorted before it is passed to group.

Running the group Operator

A group operator using the sort method always uses a partitioning method of any when running in parallel. When running it sequentially, if the order of
records is important, you should specify the `sortmerge` collector as a collection method before running the `group` operator, since the default collection method in `sort` mode is `any`.

For example:

```
$ osh "... sortmerge -key gender -key state |
group -method sort
    -key gender -key state [seq] summary_options ... "
```

## Syntax and Options

The syntax for the `group` operator is shown below. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose the value in single quotes.

```
group
   [ -collation_sequence locale|collation_file_pathname| OFF]
   [ -method hash | sort ]
   [ -hash -part partitioning_method | -seq ]
   [ -sort unix | syncsort ]
   [ -coll collection_method ]
   [ -key fieldname[|fieldtype] [-param param_string... ] [ci | -cs ]
   [ -reduce source_field suboptions (shown on option table) ]
   [ -rereduce source_field suboptions (shown on option table) ]
   [ -nul_res ]
   [ -countField count_field_name ]
   [ -weightField weight_field_name ]
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>collation_sequence</code></td>
<td>`collation_sequence locale</td>
</tr>
</tbody>
</table>

This option determines how your string data is sorted. You can:

- Specify a predefined IBM ICU locale
- Write your own collation sequence using ICU syntax, and supply its `collation_file_pathname`
- Specify `OFF` so that string comparisons are made using Unicode code-point value order, independent of any locale or custom sequence.

By default, Orchestrate sorts strings using byte-wise comparisons.

For more information, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:

Chapter 33 The Statistics Library

The group Operator

Table 169  **group Operator Options (continued)**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-countField</td>
<td>-countField  count_field_name</td>
</tr>
<tr>
<td></td>
<td>Optionally specify the count of the number of records in each group.</td>
</tr>
<tr>
<td></td>
<td>You cannot specify -countField if you specify either reduce or rereduce. You must supply one of the three options.</td>
</tr>
<tr>
<td>-hash</td>
<td>-hash -partition partitioning_method</td>
</tr>
<tr>
<td></td>
<td>The hash method separates records based on the values of key fields of the input data set according to one or more grouping key fields. You typically use hash mode for a relatively small number of groups, as its memory use is proportional to the number of groups.</td>
</tr>
<tr>
<td></td>
<td>The -hash option is optional. If used, you must specify either the -partition or -seq suboption.</td>
</tr>
<tr>
<td></td>
<td>The -hash option is mutually exclusive with the -sort and -method options. You must supply one of these three options.</td>
</tr>
<tr>
<td>-key</td>
<td>-key fieldname[ fieldtype] [-param param_string] [ci</td>
</tr>
<tr>
<td></td>
<td>There can be zero or more occurrences of this option. It specifies a grouping key. Supply the field name and, optionally, the field type.</td>
</tr>
<tr>
<td></td>
<td>The -key suboptions are optional. The -param option stipulates extra parameters for key comparison. Specify property=value pairs without curly brackets. By default, group is case sensitive. You use the -ci option to override this default in order to perform case-insensitive grouping on a string field.</td>
</tr>
<tr>
<td>-method</td>
<td>-method hash</td>
</tr>
<tr>
<td></td>
<td>Optionally sets the operator’s grouping method. Specify either hash or sort mode. The default method is any.</td>
</tr>
<tr>
<td></td>
<td>If the number of groups is large (which can happen either if you specify many grouping keys or if some grouping keys can take many values), sort mode is preferable.</td>
</tr>
<tr>
<td>-nul_res</td>
<td>-nul_res</td>
</tr>
<tr>
<td></td>
<td>This optional option indicates the use of nullable output for calculating the options -min, -max, -mean, -std, -ste, -sum, -sumw, and -var. For example, mean = NULL.</td>
</tr>
<tr>
<td></td>
<td>If you do not set -nul_res, Orchestrate uses integer 0 instead of NULL. For example, mean = 0.</td>
</tr>
</tbody>
</table>

Orchestrate 7.0 Operators Reference
You can supply zero or more `-reduce` options. For each, supply the field name that is the summary field of previously calculated statistical values.

You cannot specify `-reduce` if you specify either `-countField` or `-rereduce`. You must supply one of the three options.

You can specify multiple suboptions, described below. For each option, you specify both the summary to calculate and, optionally, a name for the field to be added to the output data set containing the summary value.

- **summary**: specifies generating a summary subrec of statistics
- **count**: count of (non-missing) values of `source_field`
- **css**: corrected sum of squares of `source_field`
- **max**: maximum value of `source_field`
- **min**: minimum value of `source_field`
- **mean**: mean value of `source_field`
- **missing**: count of missing values of `source_field`
- **range**: `source_field` range (maximum - minimum)
- **std**: standard deviation of `source_field`
- **ste**: standard error of `source_field`
- **sum**: sum of `source_field` for group
- **sumw**: sum of weight field for group

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-reduce</code></td>
<td><code>-reduce source_field</code></td>
</tr>
<tr>
<td></td>
<td><code>[summary summary_field]</code></td>
</tr>
<tr>
<td></td>
<td><code>[count [count_out]]</code></td>
</tr>
<tr>
<td></td>
<td><code>[css [count_out]]</code></td>
</tr>
<tr>
<td></td>
<td><code>[max [max_out_field]]</code></td>
</tr>
<tr>
<td></td>
<td><code>[min [min_out_field]]</code></td>
</tr>
<tr>
<td></td>
<td><code>[mean [mean_out_field]]</code></td>
</tr>
<tr>
<td></td>
<td><code>[missing [missing_out]]</code></td>
</tr>
<tr>
<td></td>
<td><code>[range [range_out_field]]</code></td>
</tr>
<tr>
<td></td>
<td><code>[std [std_out_field]]</code></td>
</tr>
<tr>
<td></td>
<td><code>[ste [ste_out_field]]</code></td>
</tr>
<tr>
<td></td>
<td><code>[sum [sum_out_field]]</code></td>
</tr>
<tr>
<td></td>
<td><code>[sumw [sumw_out_field]]</code></td>
</tr>
<tr>
<td></td>
<td><code>[uss [uss_out_field]]</code></td>
</tr>
<tr>
<td></td>
<td><code>[cv [cv_out_field]]</code></td>
</tr>
<tr>
<td></td>
<td><code>[var [var_out_field]]</code></td>
</tr>
<tr>
<td></td>
<td>`[vardiv [df</td>
</tr>
<tr>
<td></td>
<td><code>[mval value...]</code></td>
</tr>
<tr>
<td></td>
<td><code>[weight_field [weight_field_name]]</code></td>
</tr>
</tbody>
</table>
Table 169  group Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-reduce</td>
<td>-uss: uncorrected sum of squares of source_field</td>
</tr>
<tr>
<td></td>
<td>-cv: percent coefficient of variation of source_field</td>
</tr>
<tr>
<td></td>
<td>-var: variance value of source_field</td>
</tr>
<tr>
<td></td>
<td>-vardiv: specifies the variance divisor. By default, the operator uses a value of (the number of records in the group minus the number of records with missing values minus 1) to calculate the variance. This corresponds to a -vardiv setting of df. If you specify a -vardiv setting of n, the operator uses (the number of records in the group minus the number of records with missing values) instead. The default is -df. If you supply the -vardiv suboption, you must also supply the -var suboption</td>
</tr>
<tr>
<td></td>
<td>-mval: missing values. There can be any number of occurrences for this suboption</td>
</tr>
<tr>
<td></td>
<td>-weightField: configures the operator to increment the count for the group by the contents of the weight field for each record in the group, instead of by 1. Setting this option affects only -cv, -mean, -sum, -sumw, and -uss. It does not affect -count</td>
</tr>
</tbody>
</table>
### Table 169  group Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
</table>
| `-rereduce` | `-rereduce` `source_field`
               |   `[-summary ` `summary_field`]
               |   `-count `[ `count_out`]
               |   `-css` `[ `count_out`]
               |   `-max` `[ max_out_field`]
               |   `-min` `[ min_out_field`
               |   `-mean` `[ `mean_out_field`
               |   `-missing` `[ `missing_out`
               |   `-range` `[ range_out_field`
               |   `-std` `[ `std_out_field`
               |   `-ste` `[ `ste_out_field`
               |   `-sum` `[ `sum_out_field`
               |   `-sumw` `[ `sumw_out_field`
               |   `-uss` `[ `uss_out_field`
               |   `-cv` `[ `cv_out_field`
               |   `-var` `[ `var_out_field`
               |   `-vardiv` `[ `df` | `n`]|

There can be any number of occurrences of `-rereduce`. Supply the name of a source field that is the summary field of previously calculated statistical values.

You cannot specify `-rereduce` if you specify either `-countField` or `-reduce`. You must supply one of the three options.

You can specify multiple options, described below. For each option, you specify both the summary to calculate and, optionally, a name for the field to be added to the output data set containing the summary value.

- `summary`: specifies generating a summary `subrec` of statistics
- `count`: count of (non-missing) values of `source_field`
- `css`: corrected sum of squares of `source_field`
- `max`: maximum value of `source_field`
- `min`: minimum value of `source_field`
- `mean`: mean value of `source_field`
- `missing`: count of missing values of `source_field`
- `range`: `source_field` range (maximum - minimum)
- `std`: standard deviation of `source_field`
- `ste`: standard error of `source_field`
- `sum`: sum of `source_field` for group
- `sumw`: sum of weight field for group

continued
Examples

By default, Orchestrates inserts partition and sort components to meet the partitioning and sorting needs of the group operator and other operators. See the section “Orchestrate-Inserted Partition and Sort Components” on page xxxvi of this Reference for information on this facility.

Example 1: Using The group Operator in hash Mode

This section gives an example using the group operator in hash mode. In this example, you use the operator to determine the number of records in each group, where the groups are defined by the grouping key fields gender and age.

Because the resulting number of groups will be at most 200 (assuming a maximum age of 100 years), this application can be handled by hash mode. (This
assumes that the data set has already been partitioned by gender and age.) If you were grouping by zip and age, to create a maximum of 100 * 99,999 groups, you would use sort mode and presort the records before grouping to avoid prohibitively high memory use. See “Example 2: Using The group Operator in sort Mode” on page 30.

In the figure below, the group operator does a simple count of groups in the output data set:

```
$ osh "group -method hash
    -key gender
    -key age -countField groupCount
    < inData.ds > outData.ds"
```

Example 2: Using The group Operator in sort Mode

This example uses the group operator in sort mode to calculate statistics about the groups of a data set. In order to use a data set as input to the group operator in sort mode, it must be hash partitioned on one or more leading keys and sorted on all the grouping keys. By default, Orchestrates inserts partition and sort components to meet the partitioning and sorting needs of the group operator and other operators.
The figure below shows the step for this example:

```
$ osh "group -method sort -key gender -key age -reduce income -min incomeMin -max incomeMax -mean incomeMean > outData.ds"
```

**Example 3: Specifying a Weight Field**

Typically, the record count for a group is incremented by 1 for each record in the group. You can also configure the operator to increment the group count by a weight field. In that case, the count for the group is incremented by the value of the contents of the weight field for each record in the group, rather than by 1.
The weight field is useful if some pre-summarization has already been performed on the input data set. For example, assume that each record of the input data set contains information about all credit card transactions for an individual on a specific date. The schema for the input data set is shown below:

```plaintext
record {
    customerID: int32;
    gender: string[1];
    address: string[80];
    state: string[2];
    age: uint8;
    transactionDate: uint32;
    transactionCount: uint16;  // number of transactions
}
```

You could still perform grouping on age and gender, but now the transactionCount field would be used as a weight field. Each record in a group increases the group count not by 1, but by transactionCount. This means that the operator produces as a result a count of the number of transactions for each group, not the number of records for each group.

For the schema shown above, you would use this `osh` command:

```bash
$ osh "group options -countField groupCount -weightField transactionCount ...
```

If a record’s weight field contains a floating point `NaN` (IEEE Not a Number) value, the count field for the group containing that record is also a `NaN`.

**Example 4: Regrouping Data and Creating Summary Statistics**

You often use the `group` operator to calculate statistics on the groups of an input data set, use those statistics to compare the groups, then continue with your application. However, you may also want to compute statistics for one set of grouping keys, then recompute statistics for a different combination of grouping keys.

For example, you first group your data by gender, state, zip code, and age to determine information about the income for that set of grouping keys. This operation produces one output record for each unique combination of the input grouping key fields.

You then want to regroup the output data set by age and zip code alone to look for different patterns in the income of your groups. As long as you regroup the output data set using a subset of the previous grouping keys, and calculate statistics on the same field of interest, you can use the `group` operator to regroup the output data set.

Since the grouping operation produces an output data set with a single record for each group, subsequent grouping operations on the output data set typically run much faster because the output data set is much smaller than the original input data set.
In order to regroup an output data set, you first generate summary statistics on the field of interest in the input data set, rather than selecting individual statistics. These summary statistics are stored as a subrecord in the output data set of the group operator. While you specify the name of the subrecord field, names of the elements of the subrecord are supplied by the operator.

For example, you want to group records using the input data set fields gender, state, zip, and age as the grouping key fields and generate summary statistics for the income field of each group. The schema of the input data set is shown below:

```plaintext
record (
    gender: string[1];
    state: string[2];
    zip: string[5];
    age: uint8;
    income: dfloat;
)
```

Here is the osh command for this example:

```bash
$ osh ' ... group -method sort
    -key gender -key state -key zip -key age
    -reduce income
    -summary incomeSummary ... '
```

The output data set of the group operator in this case would have the following schema:

```plaintext
record (
    gender: string[1];
    state: string[2];
    zip: string[5];
    age: uint8;
    incomeSummary: subrec (
        n: dfloat;
        nMissing: dfloat;
        sumOfWeights: dfloat;
        minimum: dfloat;
        maximum: dfloat;
        mean: dfloat;
        css: dfloat; );
)
```

Here are the meanings of the elements of the summary subrecord:

- `n`: Number of records in the group
- `nMissing`: Number of missing values for the group
- `sumOfWeights`: Sum of the weights (if specified) for the group
- `minimum`: Minimum value for the group
- `maximum`: Maximum value for the group
- `mean`: Mean value for the group
• css: Sum of squares of the group adjusted for the mean

You can now use this output data set as input to another group operator to calculate statistics on income for a different set of grouping keys. From this data set, you could calculate statistics based on any combination of the four fields gender, state, zip, and age.

The output of the subsequent group operator can be either another summary of income, or individual statistics calculated only for the group. For example, the following figure shows the group operator taking as input a data set that contains a summary field:

In this example, you configure the group operator to generate another summary of income for the grouping keys gender and zip.

When specifying the arguments to the group operator for the figure above, you must configure the operator to use the information in the summary field in order to calculate statistics for the groups. In this case, you use the -rereduce option.

The osh command for this example is:

```bash
$ osh "group -method sort
  -key gender -key zip
  -rereduce incomeSummary -summary incomeSummary2 ..."
```

If you do not want to generate a summary, but only the mean of income, you can use the standard options.
Here is the `osh` command:

```bash
$ osh 'group -method sort
    -key gender -key zip
    -rereduce incomeSummary -mean incomeMean'
```

In this case, the schema of the output data set is:

```plaintext
record {
    gender: string[1];
    zip: string[5];
    incomeMean: dfloat;
}
```

Note that in this case you cannot again regroup the output data set, because you did not ask for summary statistics.

**Example 5: Two-phase Grouping with hash Mode**

Typically, when you use the `group` operator in **hash** mode, you hash partition the input data set (before grouping) using one or more of the grouping keys as hashing keys. If there are too few groups to make effective use of parallelism, though, you may not want to hash partition on the grouping fields.

For example, your system may have 16 processing nodes. If you group and hash partition on the field `gender`, you will create an input data set with data actually in only two partitions. In this case, you would be able to process the data set on only two nodes.

Instead of hash partitioning on a grouping key, you could specify a different partitioning method. You could:

- Hash partition the input data set on a field other than one of the grouping keys
- Use one of the built-in Orchestrate partitioning methods such as `roundrobin`

Since partitioners add processing time for each record of an input data set, use the simplest partitioner that satisfies your application requirements. Typically, you use the `roundrobin` partitioner to create equal-sized partitions. If the partitions are already of equal size, you can use the partitioning method **any** or **same**.

If you use a partitioning method other than hashing on one or more of the grouping key fields, your partitioning method may put records from the same group into different partitions, so that the records will be processed by different nodes in your system. Since each node computes information for the groups in its partition, the output data set would have multiple records for the same group, and you will need to rereduce the data.
For example, you want to calculate the record count for each group based on the single grouping key field `gender`. Shown below is an input data set to the `group` operator that is partitioned using the partitioning method `any`:

The left side of this figure shows the input data set to the `group` operator. The right side shows the individual nodes processing the partitions of the input data set. As you can see, if you partition the input data set such that multiple nodes process records from the same group, the output data set contains multiple records for the same group. In this example, the output data set contains one record for males and one for females for each partition of the input data set. The group count in each output record therefore contains the subtotal for males and females in each partition.

To later combine those multiple records, invoke the `group` operator again, configuring it to run sequentially. You configure the sequential `group` operator to compute the same statistics using the same grouping keys as did the preceding parallel operator.
Shown below is the `group` operator configured to run sequentially:

The sequential operator uses the `subCount` as a weight field in this example. In this case, each record input to the sequential operator contains a subtotal of the group count for a partition. The sequential operator, in incrementing by the subtotals, in effect adds up the subtotals to calculate the final group count for the data set. See “Example 3: Specifying a Weight Field” on page 31 for more information on weight fields.

The second operator, which is sequential, combines all partitions of the input data set into a single input stream. The right side of the figure shows the sequential operator combining the input partitions and creating an output data set with total record counts for males and females.

Here is the `osh` command for this example:

```
$ osh " group -method hash -key gender
    -countField groupCountSubTotal
    < transactionData.ds |
  group -method hash -key gender
    -countField groupCount -weightField groupCountSubTotal
  [seq]
  > summaryData.ds "
```
The **statistics** Operator

The Orchestrate **statistics** operator can be used to calculate the following values for the record fields of a data set:

- Mean
- Minimum
- Maximum
- Sum
- Variance
- Standard deviation
- Standard error
- Number of valid values
- Number of missing values

In addition to the **statistics** operator, used to calculate statistical values and store them to a file, Orchestrate provides the **readstats** utility, which is used to retrieve statistical values from the file.

This chapter describes how to use these two tools.

**Statistics Calculation and Data Types**

By default, the **statistics** operator calculates statistics on all fields of the input data set. The **statistics** operator can calculate statistics on fields of the following types:

- 8-, 16-, and 32-bit signed and unsigned integers
- Decimal
- Single-precision and double-precision floating points
- Strings

**Note**

To calculate statistics on string fields, you must use the **modify** operator and a string map to convert strings to numeric values. See “Calculating Statistics on String Fields” on page 33-44 for more information.

To calculate statistics on date, time, and timestamp fields, you can use the **modify** operator to translate the field to a numeric data type. See the chapter on the **modify** operator in this reference manual for more information on modifying field types. Any record with an input field containing a null value is omitted from calculations on that field. The **statistics** operator cannot calculate statistics on
raw, tagged aggregate, or subrecord fields. If you specify one of these, the **statistics** operator ignores the field.

In its computations, the **statistics** operator internally converts all field values to double-precision floating-point (**dfloat**) values. The **statistics** operator returns computed statistical values as double-precision floating-point values.

**Using the statistics Operator**

To calculate statistics on a data set, you insert the **statistics** operator into a step. The **statistics** operator stores its results to a binary file. After the step has finished, you can interrogate the file to determine the results of the calculations. The following figure shows the **statistics** operator in a step:

\[ \text{step} \rightarrow \text{op1} \rightarrow \text{statistics} \rightarrow \text{stats results (binary file)} \]

As part of calculating statistics, the **statistics** operator calculates the mean of all values for a specified field. Instead of calculating \( \frac{\text{sumOfValues}}{\text{numberOfValues}} \), the operator calculates a running mean after each record of the data set, that is:

\[
\text{mean}(i) = \text{mean}(i - 1) + \frac{V(i) - \text{mean}(i - 1)}{i}
\]

where:

- \( i \) is the current record number.
- \( \text{mean}(i) \) is the running mean through record \( i \).
- \( V(i) \) is the field value for record \( i \).

This method is more accurate than \( \frac{\text{sumOfValues}}{\text{numberOfValues}} \) for large data sets because it eliminates the potential for overflow errors.
Data Flow Diagram

Properties

Table 170  statistics Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>0 (results written to results file)</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>copy:*</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>any</td>
</tr>
<tr>
<td>Collection method</td>
<td>any</td>
</tr>
<tr>
<td>Composite operator</td>
<td>yes</td>
</tr>
</tbody>
</table>

Syntax and Options

The syntax for the statistics operator is shown below. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose the value in single quotes.

```
statistics
  -results statisticsResultsFileName [-results statisticsResultsFileName ...]
  [-fields commaSeparatedFieldNames ...]
  [-description description_string ...]
  [-missing field -values commaSeparatedListOfValues ...]
```

You must specify a results file to the operator using the -results option.
Using the readstats Utility to Retrieve Statistics Results

The statistics operator writes its results to a file. You use the readstats utility to access the file to obtain the results.

Syntax and Options

The readstats utility has the following syntax:

```
readstats results_file
[ -field field_name ]
[ -noverbose ]
[ -quantity quantity_name ]
[ -table ]
```

Table 171  statistics Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-description</td>
<td>-description (or -d) description_string</td>
</tr>
<tr>
<td></td>
<td>Specifies a description string that is stored with the statistics results. The readstats utility prints the description string when you use the verbose mode of readstats.</td>
</tr>
<tr>
<td></td>
<td>There can be zero or more occurrences of this option.</td>
</tr>
<tr>
<td>-fields</td>
<td>-fields commaSeparatedFieldNames</td>
</tr>
<tr>
<td></td>
<td>Optionally specifies a comma-separated list of fields, enclosed in single quotes, on which statistics are computed. The list of fields must include at least one valid field name from the input data set to the operator. You can omit the quotes if you specify only a single field name. There can be zero or more occurrences of this option.</td>
</tr>
<tr>
<td></td>
<td>Omitting this option causes the operator to calculate statistics on all fields of the input data set.</td>
</tr>
<tr>
<td>-missing</td>
<td>-missing (or -r) field-values commaSeparatedListOfValues</td>
</tr>
<tr>
<td></td>
<td>Sets the operator missing field. There can be any number of occurrences of this option, and there are be any number of occurrences of its suboption -values.</td>
</tr>
<tr>
<td>-results</td>
<td>-results (or -r) statisticsResultsFileName</td>
</tr>
<tr>
<td></td>
<td>Specifies the name of the file used to hold the results of the statistics operator. If the file already exists, the operator does not overwrite the file. There can be one or more occurrences of this option.</td>
</tr>
<tr>
<td></td>
<td>This option is required by the operator. There can be one or more occurrences.</td>
</tr>
</tbody>
</table>
Results_file specifies an output file from a statistics operator. You must specify a file name to the utility.

Table 172  readstats utility Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-field</td>
<td>-field field_name</td>
</tr>
<tr>
<td></td>
<td>Specifies the field whose statistics you want to display. field_name must specify a field name from the results file. Omitting this option causes the utility to display statistics on all fields that were specified to the statistics operator.</td>
</tr>
<tr>
<td>-noverbose</td>
<td>-noverbose</td>
</tr>
<tr>
<td></td>
<td>Specifies to display only the field name and statistics. If you specify -quantity, only the value is displayed, not the field name. This option is often used when streaming the output of readstats to another utility.</td>
</tr>
<tr>
<td>-quantity</td>
<td>-quantity quantity_name</td>
</tr>
<tr>
<td></td>
<td>Specifies a single statistical value to display. Possible values for quantity_name are: maximum, mean, minimum, numberOfMissingValues, numberOfValues, standardDeviation, standardError, sum, variance</td>
</tr>
<tr>
<td>-table</td>
<td>-table</td>
</tr>
<tr>
<td></td>
<td>Configures the utility to print statistics results in the tabular format.</td>
</tr>
</tbody>
</table>
Example: Using the Statistics Tools

This section shows a step that uses the **statistics** operator to calculate statistics for two fields, a and b, of a data set. The following figure shows part of the data flow diagram:

![Data Flow Diagram]

Here is the `osh` command for this example:

```bash
$ osh " ... | statistics -results /stats/results -fields 'a, b'  
```

You can interrogate the file containing the statistics results using the `readstats` utility. For example, the following command causes `readstats` to display all statistics for all fields:

```
$ readstats /stats/results
```

The result of this command is:

```
Statistics for 1600 records, as of Thu Feb 13 13:35:02 1997

a:
  minumum=0
  maximum=9782
  sum=58272
  mean=3642
  variance=1.26128e+07
  standardDeviation=3551.45
  standardError=887.863

b:
  minumum=0
  maximum=9992
  sum=45250
  mean=2828.12
```
The statistics Operator

```
variance=6.4004e+06
standardDeviation=2529.9
standardError=632.475
```

You can also use the `-noverbose` option, as shown below:

```
$ readstats /stats/results -noverbose
```

This commands displays the following information:

```
a 0 9782 58272 3642 1.26128e+07 3551.45 887.863
b 0 9992 45250 2828.12 6.4004e+06 2529.9 632.475
```

You can also display statistics in a tabular form using the `-table` option:

```
$ readstats /stats/results -table
```

This command displays:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Valid N</th>
<th>Mean</th>
<th>Std Dev</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1600</td>
<td>3642</td>
<td>3551.45</td>
<td>0</td>
<td>9782</td>
</tr>
<tr>
<td>b</td>
<td>1600</td>
<td>2828.12</td>
<td>2529.9</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Alternatively, you can display the results for a specific field, using the `-field` option, and for any one statistic, using the `-quantity` option. For example, the following command causes `readstats` to display the mean of field `a`:

```
readstats /stats/results -noverbose -field a -quantity mean
```

The result of this command is the string:

3642

### Calculating Statistics on String Fields

The `statistics` operator can calculate statistics on string fields if you convert them to numbers. However, you must use a `modify` operator to convert strings to numeric values. See Chapter 13, “The modify Operator” for information on how to do this.
The following figure shows the data flow diagram for this example:

Suppose you want to calculate statistics on a string field named gender. Before invoking the `statistics` operator, you use the `modify` operator to create a string lookup table such as the one shown below to translate the string values for gender to numeric values:

<table>
<thead>
<tr>
<th>Numeric Value</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;f&quot;</td>
</tr>
<tr>
<td>0</td>
<td>&quot;female&quot;</td>
</tr>
<tr>
<td>1</td>
<td>&quot;m&quot;</td>
</tr>
<tr>
<td>1</td>
<td>&quot;male&quot;</td>
</tr>
</tbody>
</table>

If a gender field contains an invalid string, the string map outputs a value of null for the field, which the `statistics` operator ignores when computing statistics.

The `osh` command for this requires two `modify` operators:

```bash
$ osh '... | modify 'gender = lookup_int16_from_string(
  {default_value = 2}
  ['f' = 0; 'female' = 0;
  'm' = 1; 'male' = 1;])(gender);'
  | modify 'gender:nullable int16 = make_null
  (gender, 2)'
  | statistics ...'
```
The fieldtransform Operator

Overview

After data is loaded into Orchestrate and cleansed, you can use the Orchestrate data mining tools to analyze the data. You can calculate statistics, classify data into categories, or detect trends and patterns. Many of the tools that Orchestrate provides for this purpose work best with data in a standardized form. Thus, before you can analyze the information you extracted from very large database systems you may need to prepare or convert the fields. Statisticians refer to such data conversion as recoding the data. Orchestrate uses the term field transformation.

Transforming data serves two purposes. First, some tools you use to analyze your data require inputs to be within certain ranges. For neural nets, for example, the required range is typically between -1.0 and 1.0. Second, by rescaling or converting input fields, you shorten the time required to build reliable tools and improve the results they produce.

All Orchestrate field transforms are reversible. You use inverse field transforms to convert transformed data back to its original form. Some tools even require inverse conversions to bring computed values back into a form that is appropriate to the particular problem.

This chapter outlines the general process for performing transforms and describes the fieldtransform operator, the operator provided by Orchestrate for this purpose. In addition to the operator, you need to specify a field transform. Orchestrate provides the following predefined field transforms, which are described in the remaining sections of this chapter:

- Z-score calculation, which computes a value in terms of standard deviations from the mean of all values for the field (see “Performing Z-score Transforms” on page 33-57).
- Z-sigmoid calculation, which computes a value for a field that is within a fixed boundary of possible values (see “Performing Z-sigmoid and Z-symmetric Transforms” on page 33-62).
- Z-symmetric calculation, which produces a value in the range -1.0 < result < 1.0 from the input (see “Performing Z-sigmoid and Z-symmetric Transforms” on page 33-62).
- Identity calculation, where the output of the transform is the same as the input (see “Performing Identity Transforms” on page 33-67).
- Nominal field transformation, which adds multiple fields to a record to represent a field as a collection of true and false field values. Only one of the added fields may be set to true for this transform (see “Transforming Nominal Fields” on page 33-68).
• Ordinal field transformation, which, like nominal field transformations, adds multiple fields to a record to represent a field as a collection of true and false field values. More than one of the added fields may be set to true for this transform (see “Transforming Ordinal Fields” on page 33-73).

General Transformation Process

Field transforms examine one or more source fields and produce as output one or more result fields. For example, a transform could examine a single variable called gender, which describes distinct categories, and convert it to a new form. For a neural network, the converted form would be two Boolean variables: isMale and isFemale. Another common transformation is Z-scoring, which adjusts the range of a collection of variables so that they all have a mean value of 0.0 and a standard deviation of 1.0. This type of conversion permits intuitive comparisons between pairs of values that originally had unrelated scales.

You use the fieldtransform operator to perform a transform. Using the operator, you can perform one or more transforms on a data set. You may need to perform more than one transform if different fields in the data set require different types of conversion. Each transform adds one or more fields to the output data set of the operator. The following figure shows an example of using the fieldtransform operator:

![Diagram showing the general transformation process](image)

Each transform object defines the field names of the inputs to the transform, the direction of the transform, and the type of conversion (such as Z-score calculation or nominal field transformation).
The example above uses:
- The Z-score transform to perform a forward transform on field a
- The Z-sigmoid transform to perform a forward transform on field b
- The Z-symmetric transform to perform a forward transform on field d

A transform creates new fields that are added to the schema of the operator's output data set. As you can see in the figure, three new fields have been added:
- a_ZScore: Contains the Z-score value calculated for field a
- b_ZSigmoid: Contains the Z-sigmoid value calculated for field b
- d_ZSymmetric: Contains the Z-symmetric value calculated for field d

To produce the name of the output field, each transform adds a suffix to the input field name.

To perform an inverse transform on a data set, you use the same operator but a different transform. The following figure shows an inverse transform that converts calculated values back into the form of the original values. The transform object specifies an inverse transform to the fieldtransform operator:
Typically, you perform an inverse transform on a field value calculated by your application, not on a field calculated by a forward transform. In the example above, the first operator calculates a result field based on four input fields. This data set is used as input to the `fieldtransform` operator. The transform object specifies that an inverse Z-score operation be performed on the result field. This process adds a new field, `result_InvZScore`, to the output data set of the `fieldtransform` operator.

**Using the fieldtransform Operator**

You use the `fieldtransform` operator to perform both forward and inverse transforms on a data set. The transform type determines the direction of the transform. Each transform adds one or more fields to output data set 0 of the `fieldtransform` operator. The transform type defines the field names of the inputs to the transform, the direction of the transform, and the type of conversion (such as Z-score calculation or nominal field transformation).

You must attach at least one output data set to the `fieldtransform` operator to hold the results of the operator. You can attach more data sets as well; each additional output data set attached to the operator receives an unmodified copy of the input data set. This allows you to propagate the input data set to subsequent operators.

**Data Flow Diagram**
Properties

Table 173  fieldtransform Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>in:*;</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>data set 0: out:* plus new fields added by transform</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>output data set 0: in -&gt; out without record modification</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>any</td>
</tr>
<tr>
<td>Collection method</td>
<td>any</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>propagated</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

Field Transform Operator and Nulls

All Orchestrate data types support a null representation called the Orchestrate indicated null representation. The fieldtransform operator recognizes when an input field is nullable. If so, any output fields created by the operator from a nullable input field are also nullable. If an input field is not nullable, the corresponding output fields are also not nullable.

See the chapter on Orchestrate data sets in the Orchestrate 7.0 User Guide for more information on Orchestrate null support.

Syntax and Options

The syntax for the fieldtransform operator in an osh command is shown below. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose the value in single quotes.

```
fieldtransform
  [-statistics statisticsResultsFileName]
  [-zscore commaSeparatedFieldNames [-missing commaSeparatedValues ...]]
  [-missing asdata | -missing aserror] [-statsfield fieldName]
  [-error errorValue ...]
```
The next table lists the options to the fieldtransform operator. Note that when the first letter of the argument name is unique, you can abbreviate it using only the first letter. For example, -values can be abbreviated as -v.
Table 174  fieldtransform Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-identity</td>
<td>-identity commaSeparatedFieldNames</td>
</tr>
<tr>
<td></td>
<td>[-missing or -m commaSeparatedValues ...]</td>
</tr>
<tr>
<td></td>
<td>[-missingasdata</td>
</tr>
<tr>
<td></td>
<td>[-statsfield FieldN ame] [-error or -e errorValue]</td>
</tr>
</tbody>
</table>

Specifies the fields, separated by commas, on which you want to perform the identity transform. You can specify single fields as well as a field list in the form field0-fieldN. The transform is performed on all fields between field0 and fieldN, inclusive, as defined in the data set’s schema.

You can specify multiple -identity arguments.

The field added to the output data set is named field.Identity.

You must use the -statistics option to specify the file containing the mean and standard deviation of fieldN. By default, the statistics file must contain the mean and standard deviation for each field being transformed. You can optionally use the -statsfield argument to specify the name of a single field in the statistics file providing the mean and standard deviation for all fields specified by -identity.

The -missingasdata and -missingaserror options are mutually exclusive.

-inverseidentity     | -inverseidentity commaSeparatedFieldNames                            |
|                      | [-missing or -m commaSeparatedValues ...]                            |
|                      | [-missingasdata | -missingaserror] [-statsfield FieldN ame]                   |

Specifies the fields, separated by commas, on which you want to perform the inverse identity transform. You can specify single fields as well as a field list in the form field0-fieldN. The transform is performed on all fields between field0 and fieldN, inclusive, as defined in the data set’s schema. There can be any number of occurrences of this option.

The field added to the output data set is named fieldN.InvIdentity.

You must use -statistics to specify the file containing the mean and standard deviation of fieldN. By default, the statistics file must contain the mean and standard deviation for each field being transformed. You can optionally use the -statsfield argument to specify the name of a single field in the statistics file providing the mean and standard deviation for all fields identified by -inverseidentity.

If fieldN contains the suffix _Identity, the operator strips off the suffix to determine the source field for the mean and standard deviation.

The suboptions -missingasdata and -missingaserror are mutually exclusive.
Table 174  fieldtransform Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
</table>
| -inversenominal      | - in versenominal FieldName  
|                      |   [ -values or -v commaSeparatedValues ... ]  
|                      |   [ -missingfield or -m ] [ -range or -r first | last | incr ]  
|                      |   [ -truth or -t falseVal, trueVal ]  
|                      | Specifies the field on which you want to perform an inverse nominal transform. The new field is named FieldName_inverse. There can be any number of occurrences of this option.  
|                      | -values and -range specify the values for fields recognized by the transform. See “Inverse Nominal Transforms” on page 70 for more information.  
|                      | -truth specifies optional values for true and false. The default values are 0.1 for false and 0.9 for true. See “Forward Nominal Transforms” on page 68 for more information.  |
| -inverseordinal      | - in verseordinal FieldName  
|                      |   [ -values or -v commaSeparatedValues ... ]  
|                      |   [ -missingfield or -m ] [ -range or -r first | last | incr ]  
|                      |   [ -truth or -t falseVal, trueVal ]  
|                      | Specifies the field on which you want to perform an inverse ordinal transform. This is the name of the field on which you originally performed the forward transform. The new field is named FieldName_inverse. There can be any number of occurrences of this option.  
|                      | -values and -range specify the values for field recognized by the transform. See “Inverse Ordinal Transforms” on page 75 for more information.  
|                      | -truth specifies optional values for true and false. The default values are 0.1 for false and 0.9 for true. See “Forward Nominal Transforms” on page 68 for more information.  |
Specifies a comma-separated list of fields on which you want to perform an inverse Z-score transform. You can specify single fields as well as a field list in the form field0-fieldN. The transform is performed on all fields between field0 and fieldN, inclusive, as defined in the data set’s schema. There can be any number of occurrences of this option.

The field added to the output data set is named fieldN_InvZScore.

You must use -statistics to specify the file containing the mean and standard deviation of fieldN. By default, the statistics file must contain the mean and standard deviation for each field being transformed. You can optionally use the -statsfield argument to specify the name of a single field in the statistics file providing the mean and standard deviation for all fields identified by -inversezscore.

If fieldN contains the suffix ZScore, the operator strips off the suffix to determine the source field for the mean and standard deviation.

The suboptions -missingasdata and -missingaserror are mutually exclusive.

Specifies a comma-separated list of fields on which you want to perform an inverse Z-sigmoid transform. You can specify single fields as well as a field list in the form field0-fieldN. The transform is performed on all fields between field0 and fieldN, inclusive, as defined in the data set’s schema. There can be any number of occurrences of this option.

The field added to the output data set is named fieldN_InvZSigmoid.

You must use -statistics to specify the file containing the mean and standard deviation of fieldN. By default, the statistics file must contain the mean and standard deviation for each field being transformed. You can optionally use the -statsfield argument to specify the name of a single field in the statistics file providing the mean and standard deviation for all fields identified by -inversezsigmoid.

If fieldN contains the suffix ZSigmoid, the operator strips off the suffix to determine the source field for the mean and standard deviation.

The suboptions -missingasdata and -missingaserror are mutually exclusive.
Specifies a comma-separated list of fields on which you want to perform an inverse Z-symmetric transform. You can specify single fields as well as a field list in the form field0-fieldN. The transform is performed on all fields between field0 and fieldN, inclusive, as defined in the data set's schema. There can be any number of occurrences of this option.

The field added to the output data set is named fieldN_invZSymmetric.

You must use -statistics to specify the file containing the mean and standard deviation of fieldN. By default, the statistics file must contain the mean and standard deviation for each field being transformed. You can optionally use the -statsfield argument to specify the name of a single field in the statistics file providing the mean and standard deviation for all fields identified by -inversezsymmetric.

If fieldN contains the suffix _ZSymmetric, the operator strips off the suffix to determine the source field for the mean and standard deviation.

The suboptions -missingasdata and -missingaserror are mutually exclusive.

Specifies the fields on which you want to perform a nominal transform. There can be any number of occurrences of this option.

-values and -range specify the values for field recognized by the transform. See “Forward Nominal Transforms” on page 68 for more information.

-truth specifies the output values for true and false. The default values are 0.1 for false and 0.9 for true. See “Forward Nominal Transforms” on page 68 for more information.
Chapter 33  The Statistics Library  The fieldtransform Operator

Table 174  fieldtransform Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ordinal</td>
<td>-ordinal commaSeparatedFieldNames</td>
</tr>
<tr>
<td></td>
<td>[-values or -v commaSeparatedValues ...]</td>
</tr>
<tr>
<td></td>
<td>[-missingfield or -m] [-range or -r first</td>
</tr>
<tr>
<td></td>
<td>[-truth or -t falseVal, trueVal]</td>
</tr>
<tr>
<td></td>
<td>Specifies the fields on which you want to perform an ordinal transform.</td>
</tr>
<tr>
<td></td>
<td>There can be any number of occurrences of this option.</td>
</tr>
<tr>
<td></td>
<td>-values and -range specify the values for fields recognized by the</td>
</tr>
<tr>
<td></td>
<td>transform. See “Forward Ordinal Transforms” on page 74 for more</td>
</tr>
<tr>
<td></td>
<td>information.</td>
</tr>
<tr>
<td></td>
<td>-truth specifies the output values for true and false. The default</td>
</tr>
<tr>
<td></td>
<td>values are 0.1 for false and 0.9 for true. See “Forward Nominal</td>
</tr>
<tr>
<td></td>
<td>Transforms” on page 68 for more information.</td>
</tr>
<tr>
<td>-statistics</td>
<td>-statistics staticsResultsFileName</td>
</tr>
<tr>
<td></td>
<td>Specifies the file containing the results of the fieldtransform operator.</td>
</tr>
<tr>
<td></td>
<td>This option is required for the -zscore, -inversezscore, -zsigmoid,</td>
</tr>
<tr>
<td></td>
<td>-inversezsigmoid, -zsymmetric, -inversezsymmetric, -identity, and</td>
</tr>
<tr>
<td></td>
<td>-inverseidentity transforms; otherwise, it is optional.</td>
</tr>
<tr>
<td>-zscore</td>
<td>-zscore commaSeparatedFieldNames</td>
</tr>
<tr>
<td></td>
<td>[-missing or -m commaSeparatedValues ...]</td>
</tr>
<tr>
<td></td>
<td>[-missingasdata</td>
</tr>
<tr>
<td></td>
<td>[-statsfield fieldName] [-error or -e errorValue]</td>
</tr>
<tr>
<td></td>
<td>Specifies the fields on which you want to perform a Z-score transform.</td>
</tr>
<tr>
<td></td>
<td>You can specify single fields as well as a field list in the form</td>
</tr>
<tr>
<td></td>
<td>field0-fieldN. The transform is performed on all fields between field0</td>
</tr>
<tr>
<td></td>
<td>and fieldN, inclusive, as defined in the data set’s schema. There</td>
</tr>
<tr>
<td></td>
<td>can be any number of occurrences of this option.</td>
</tr>
<tr>
<td></td>
<td>The field added to the output data set is named field_ZScore.</td>
</tr>
<tr>
<td></td>
<td>You must use -statistics to specify the file containing the mean and</td>
</tr>
<tr>
<td></td>
<td>standard deviation of the fields. By default, the statistics file</td>
</tr>
<tr>
<td></td>
<td>must contain the mean and standard deviation for each field being</td>
</tr>
<tr>
<td></td>
<td>transformed. You can optionally use the -statsfield argument to</td>
</tr>
<tr>
<td></td>
<td>specify the name of a single field in the statistics file providing</td>
</tr>
<tr>
<td></td>
<td>the mean and standard deviation for all fields identified by -zscore.</td>
</tr>
<tr>
<td></td>
<td>The suboptions -missingasdata and -missingaserror are mutually</td>
</tr>
<tr>
<td></td>
<td>exclusive.</td>
</tr>
</tbody>
</table>
Performing Z-score Transforms

The Z-score of a field describes a value in terms of standard deviations from the mean of all values for the field. Consider, for example, a data set that contains an
income field for each record. A Z-score of 2.00 for that field means that the income in question is two standard deviations above the mean income value. By transforming dollar amounts into a Z-score, you can determine how a person’s income compares to the mean value of all incomes in the data set. As a result, you can compare values regardless of their scale.

The equation for Z-score is shown here:

\[ Z(f) = \frac{\text{Value}(f) - \text{Mean}(f)}{\text{StdDev}(f)} \]

Where:

- \( f \) = a field of a data set
- \( Z(f) \) = the Z-score of field \( f \)
- \( \text{Value}(f) \) = the value of a field \( f \) for a record of the data set
- \( \text{Mean}(f) \) = the mean value of field \( f \) for the entire data set
- \( \text{StdDev}(f) \) = the standard deviation of field \( f \) for the entire data set

To calculate the Z-score, you need to calculate the mean and standard deviation for the field across the entire data set. Orchestrate provides statistics tools that calculate these values using the statistics operator. See “The statistics Operator” on page 38 for more information on this operator.

Using the equation below, you can invert a Z-score to recapture the original value:

\[ \text{inverseZ}(f) = (Z(f) \times \text{StdDev}(f)) + \text{Mean}(f) \]

**Using the Z-score Transform**

Orchestrate provides two predefined types of Z-score transforms:

- The Z-score transform, which creates an output field containing the Z-score of an input field.
- The inverse Z-score transform, which creates an output field containing the inverse Z-score of an input field.

You need two steps to calculate a Z-score:

1. For the fields that are inputs to the transform, compute the mean and standard deviation.
   
   For this purpose, you use the statistics operator. The output of this operator is a file that contains the mean, standard deviation, and other statistics for the fields to be transformed.
   
   The results of the statistics operator are not valid until this step is completed.

2. Compute the Z-score transform using the fieldtransform operator.
The following figure shows the two steps you use to compute the Z-score:

**Forward Z-Score Transform: First Step**

In the first step, you compute statistics for the fields that are inputs to the transform. This code example computes statistics for the fields `a` and `b`.

Shown below is the `osh` command for this example:

```bash
$ osh 'statistics -results /data/results -fields 'a, b' < inData1.ds ...
```

The `-results` argument specifies to store the results of the operator to the file `/data/results`.

After you calculate the statistics, you can use the mean and standard deviation values to calculate the Z-score for fields `a` and `b`.

**Forward Z-Score Transform: Second Step**

In the second step, you use the same input data set for the `fieldtransform` operator as you used in the first step. By appending the suffix `_ZScore` to field names `a` and `b`, the `fieldtransform` operator creates new double-precision floating-point fields in the output data set.
Shown below is the `osh` command for this example:

```bash
$ osh 'fieldtransform -statistics /data/results -zscore 'a, b' < inData1.ds ...
```

In this example, the `-statistics` argument specifies the results file from the `statistics` operator.

**Inverse Z-score Transform**

Often when you use Z-score fields, you calculate a single output result that is also a Z-score field. You can then run the inverse Z-score transform to convert the field back into its original form. The following figure uses the inverse Z-score transform to convert the Z-score in field `a_ZScore` into the new field `a_ZScore_InvZScore`. Note that the data type of a field created by an inverse Z-score is always a double-precision float:

Typically, you perform an inverse transform on a field value calculated by your application, not on a field calculated by a forward transform. Therefore, unlike the forward transform, you can only convert a single field using the inverse Z-score transform.
Shown below is the osh command for this example:

```bash
$ osh "fieldtransform -statistics /data/results
   -inversezscore 'a_ZScore' < inData3.ds ...
```

In this example, the **statistics** argument specifies the results file from the **statistics** operator.

**Configuring Z-score Transforms**

You must configure a Z-score transform to handle certain field values and properties, including:

- Extremely large or small field values
- Fields with a standard deviation of zero

This section describes each of these cases.

**Extremely Large or Small Field Values**

The Z-score transform limits the value it computes to the range -99.0 to 99.0. An extremely large field value can generate a maximum Z-score of 99.0, while an extremely small field value can generate a minimum field value of -99.0. On average, however, maximum Z-score will be \( \log_2(\text{numberOfValues}) \).

**Standard Deviation of Zero**

A standard deviation of 0 for a field means that there is no variation in the values of the field for the data set. In other words, only a single value occurs in the field and that value is the mean. If this occurs, it probably means you have not calculated standard deviation on a sufficiently representative sample, or that the field contains no useful information. Either way, you should re-examine your input.

In order to calculate the Z-score for a field with a standard deviation of 0, the Z-score transform:

- Outputs a Z-score of 0 for field values equal to the mean.
- Outputs a Z-score of 99.0 for field values above the mean and -99.0 for field values below the mean. Note that fields can only have a value different from the mean if you use the mean and standard deviation from one data set to compute the Z-score of a field in another data set. This may occur if you compute the statistics for a sample of a data set and then use the statistics to transform the larger data set that the sample represents.

In order to calculate the inverse Z-score for a field with a standard deviation of 0, the inverse Z-score transform:

- Outputs the mean if the input Z-score is 0.
• Outputs the mean plus 1 if the input Z-score is positive or the mean minus 1 if the Z-score is negative.

Performing Z-sigmoid and Z-symmetric Transforms

Computing a Z-sigmoid or Z-symmetric transform for a field performs a squashing operation; that is, you compute a value for the field that is always within a fixed boundary of possible values. This is shown in the following figure:

The original function is mapped within the maximum and minimum values specified by the transform function. As the original function grows larger, it approaches the maximum value of the transform function; as the function gets smaller, it approaches the minimum value of the transform function.

The only difference between Z-sigmoid and Z-symmetric transforms is the minimum and maximum. For sigmoids, the minimum and maximum are 0.0 and 1.0, respectively. For symmetric transforms, they are -1.0 and 1.0.

To compute the Z-sigmoid or Z-symmetric transform of a field, you need the Z-score of that field, but the transform you use handles this task for you. See “Performing Z-score Transforms” on page 33-57 for details on computation of Z-scores.

The equations for forward Z-sigmoid and Z-symmetric transforms are shown below:

$$Z_{\text{Sigmoid}}(f) = \frac{1}{1 + e^{-Z(f)}}$$

$$Z_{\text{Symmetric}}(f) = \frac{1 - e^{-2Z(f)}}{1 + e^{-2Z(f)}}$$

where:

• f = a field of a data set
• Z(f) = the Z-score of field f

The sigmoid function always produces a number in the range:
minimum value = 0.0 < Sigmoid(f) < 1.0 = maximum value

The symmetric function always produces a number in the range:
minimum value = -1.0 < Symmetric(f) < 1.0 = maximum value

You use the inverse transform to compute the inverse transform value of a field. The inverse equations are shown below:

\[
\text{inverse\text{Z}Sigmoid}(f) = -\log \left( \frac{1 - \text{ZSigmoid}(f)}{\text{ZSigmoid}(f)} \right)
\]

\[
\text{inverse\text{Z}Symmetric}(f) = \frac{1}{2} \log \left( \frac{1 + \text{ZSymmetric}(f)}{1 - \text{ZSymmetric}(f)} \right)
\]

Where:
• f = a field of a data set
• Z-Sigmoid(f) = the sigmoid value of field f in the range: 0.0 < Z-Sigmoid(f) < 1.0
• Z-Symmetric(f) = the symmetric value of field f in the range: -1.0 < Z-Symmetric(f) < 1.0

Using the Transforms

This section describes how to perform either a Z-sigmoid or a Z-symmetric transform. The example in this section uses the Z-sigmoid transform; however, the procedure is exactly the same for the Z-symmetric transform.

Orchestrate provides two Z-sigmoid transforms:
• The Z-sigmoid transform, which creates a new field containing the sigmoid value for an input field
• The inverse Z-sigmoid transform, which creates a new field containing the inverse sigmoid for an input field
You need two steps to calculate a Z-sigmoid:

1. For the fields that are inputs to the transform, compute the mean and standard deviation.

For this purpose, you use the statistics operator. The output of this operator is a file containing the mean, standard deviation, and other statistics for the fields to be transformed. See “The statistics Operator” on page 38 for more information.

The results of the statistics operator are not valid until this step is completed.

2. Compute the sigmoid transform using the fieldtransform operator.

The following figure shows these two steps:

Forward Z-sigmoid Transform: First Step

In the first step, you compute the statistics for the input fields to the transform. The following sample code computes statistics for the fields a and b.

```datanode
step1
inData1.ds data set
schema:
a:int32;
b:int32;
c:string;
d:int16;
e:string;

statistics

step2
inData1.ds data set
schema:
a:int32;
b:int32;
c:string;
d:int16;
e:string;

fieldtransform
Z-sigmoid transform

statistics results

data set
schema:
a:int32;
b:int32;
c:string;
d:int16;
e:string;
a_ZSigmoid:dfloat;
b_ZSigmoid:dfloat;
```
Shown below is the `osh` command for this example:

```bash
$ osh "statistics -results /data/results -fields 'a, b' < inData1.ds ...

The `-results` argument specifies to store the results of the operator to the file `/data/results`.

Once you create the statistics, you can use the mean and standard deviation values stored in the results file to calculate the sigmoid for fields a and b.

**Forward Z-sigmoid Transform: Second Step**

In the second step, you use the same input data set for the `fieldtransform` operator as you used for the `statistics` operator. By appending the suffix `_ZSigmoid` to field names a and b, the `fieldtransform` operator creates new double-precision floating-point fields in the output data set.

Here is the `osh` command for this example:

```bash
$ osh "fieldtransform -statistics /data/results -zsigmoid 'a, b' < inData1.ds ...

In this example, the `-statistics` argument specifies the results file from the `statistics` operator.

**Inverse Z-sigmoid Transform**

Often when you use sigmoid fields, you calculate a single output result that is also a sigmoid field. You can then run the inverse sigmoid transform to convert the field into the form of the original input fields. The following figure shows how to use the inverse Z-sigmoid transform to convert the sigmoid in field a_ZSigmoid...
into the new field a_ZSigmoid_InvZSigmoid. Note that the data type of a field created by an inverse sigmoid is always a double-precision float.

Typically, you perform an inverse transform on a field value calculated by your application; you do not perform the inverse transform on the fields containing the transform of the original fields. Therefore, unlike the forward transform, you can convert only a single field using the inverse Z-sigmoid transform.

Shown below is the $osh command for this example:

```
$ osh 'fieldtransform -statistics /data/results -inversezsigmoid 'a_ZSigmoid' < inData3.ds ...'
```

In this example, the `-statistics` argument specifies the results file from the `statistics` operator.

**Standard Deviation of Zero**

To calculate the transform of a field, you must first determine the mean and standard deviation for the field. A standard deviation of 0 for a field means that there is no variation in the values of the field for the data set. In other words, only a single value occurs in the field and that value is the mean. If you expect the
input field to have more than one value, you should re-examine your data and provide additional values.

This section describes how to handle standard deviation of 0 for both Z-sigmoid and Z-symmetric transforms.

**Standard Deviation of Zero for Z-sigmoid Transforms**

To calculate the sigmoid for a field with a standard deviation of 0, the transform:

1. Outputs a sigmoid value of 0.5 for field values equaling the mean.
2. Outputs a sigmoid of 0.9 for field values above the mean and 0.1 for field values below the mean.

Note that fields can only have a value different from the mean if you use the mean and standard deviation from one data set to compute the transform of a field in another data set. For example, this may occur if you compute the statistics for a sample of a data set and then use the statistics to transform the larger data set that the sample represents.

To calculate the inverse sigmoid for a field with a standard deviation of 0, the inverse Z-sigmoid transform:

1. Outputs the mean if the input sigmoid is 0.5.
2. Outputs the mean plus 1.0 if the input sigmoid is greater than 0.5 or the mean minus 1.0 if the sigmoid is less than 0.5.

**Standard Deviation of Zero for Z-symmetric Transforms**

To calculate the symmetric transform for a field with a standard deviation of 0, the transform:

1. Outputs a value of 0.0 for field values equaling the mean.
2. Outputs a value of 0.9 for field values above the mean and -0.9 for field values below the mean.

To calculate the inverse symmetric for a field with a standard deviation of 0, the inverse Z-symmetric transform:

1. Outputs the mean if the input sigmoid is 0.0.
2. Outputs the mean plus 1.0 if the input sigmoid is greater than 0.0 or the mean minus 1.0 if the sigmoid is less than 0.0.

**Performing Identity Transforms**

An identity transform, both forward and inverse, copies the input field to the output without modifying the field. This transform is useful in an application that
defines a transform for all fields in a data set, even if a field does not require the transform.

The forward transform appends the string _Identity to the field name on the forward transform. The inverse transform appends the string _InvIdentity to the field.

Note that when performing an identity transform, you use the same procedure and supply the same information to the fieldtransform operator as for a Z-score, sigmoid, or symmetric transform. While you must supply this information, it is ignored by the transform except to handle NaN and missing values, as described below. See “Performing Z-score Transforms” on page 33-57 for the procedure you follow in order to calculate an identity transform.

Transforming Nominal Fields

A nominal field represents data numerically. For example, you may have a string field containing the names of states. Alternatively, you can represent this information with a nominal field containing an integer ranging from 1 to 50. Nominal fields represent unordered data; thus, in this example, there would be no significance to a state being represented by 1 or by 50.

The nominal transform performs a transform on a nominal field by adding multiple fields to a record. These fields represent the nominal field as a collection of true and false field values. Only one of the added fields is set to true; all other added fields are set to false.

You use the inverse nominal transform to perform an inverse nominal transform. This transform examines multiple fields where one field contains a value of true and all other fields have a value of false. The result is a single field whose value represents the input field with a value of true. Note that on an inverse transform, if a source field contains a null, a null is produced as output.

You can use nominal transforms with the Orchestrate data modeling components. In some cases, these components work better with a series of true and false fields rather than a single field containing a value range.

This section describes both forward and inverse nominal transforms. This section also describes how you can derive your own thresholding function when performing an inverse nominal transform.

Forward Nominal Transforms

In the following example, the nominal field fruitPreference contains a number indicating a person’s preferred fruit. Acceptable field values for fruitPreference, and their meanings, are as follows:

- 10 = oranges
- 20 = grapes
A nominal transform converts the source field `fruitPreference` by adding six new fields to the record, one for each of the possible values of `fruitPreference`. The new fields are named by appending an index to `fruitPreference`, resulting in new fields with the names `fruitPreference_0` through `fruitPreference_5`.

If `fruitPreference` has a value of 40, signifying a person’s preference for bananas, the fields added to the record by the `fieldtransform` operator are given these values:

- `fruitPreference_0 = false` // corresponding to oranges
- `fruitPreference_1 = false` // corresponding to grapes
- `fruitPreference_2 = false` // corresponding to apples
- `fruitPreference_3 = true` // corresponding to bananas
- `fruitPreference_4 = false` // corresponding to peaches
- `fruitPreference_5 = false` // corresponding to other

Note that only one of the added fields has a value of true. The following figure shows the record before and after the nominal transform:

You can determine the numeric value and data type representing true and false. The `fieldtransform` operator defines the default value for true as 0.9 and 0.1 for false. You use the `--truth` option to change these values. In the example shown above, true and false were set 1.00 and 0.00, respectively.

In order to use the nominal transform to transform a field, you must define the source field values recognized by the transform. You use the operator option `--values` to specify these values.
The **-values** option defines all of the expected values for a source field. The following example configures the `fieldtransform` operator to recognize the six possible values for the source field `fruitPreference`:

```
$ osh "fieldtransform -nominal fruitPreference
   -values '10, 20, 30, 40, 50, 60' other_options ..."
```

The indices for the output fields generated by the operator are assigned sequentially from 0 to \( n - 1 \) in the order in which you specify the values. The first value sets the source field value that causes the output field with the index \(_0\) to be set to true, the next value sets the source field value corresponding to the output field with the index \(_1\), and so on. In this example, a `fruitPreference` value of 40 causes `fruitPreference_3` to be set to true and all other fields to be set to false.

The **-range** option allows you to specify an entire sequence of values with a single call, as shown below.

```
$ osh "fieldtransform -nominal fruitPreference
   -range '10, 60, 10' other_options ..."
```

The first value after **-range** is the starting value, the second is the ending value, and the third is the increment value. This statement causes the operator to generate six new fields for each record, just as the six separate specifications to **-values** did.

Note that you cannot set duplicate values; that is, a source field value cannot cause more than one output field to be set to true.

**Inverse Nominal Transforms**

The inverse nominal transform relies on information contained within a forward nominal transform object.
The following figure shows a record after a forward and an inverse nominal transform:

![Diagram of record transformation](image)

This figure uses the forward transform example from the previous section. In that example, a forward transform is calculated on the source field fruitPreference. This field can have any of six values (10, 20, 30, 40, 50, 60).

In the figure shown above, fruitPreference had a value of 40 at the time of the forward transform. Therefore, the forward transform adds six new fields to the record. The field fruitPreference_3 is set to 1.00 and the remaining five fields are set to 0.00. Note that these fields added by the forward transform have a data type of dfloat.

An inverse transform adds a single field of type dfloat whose value is determined by the six fields added by the forward transform. The name of the added field in this example is fruitPreference_inverse.

In order to calculate the value of fruitPreference_inverse, the inverse nominal transform examines the fields created by the forward transform. The field with the highest value, in this case fruitPreference_3, determines the value of fruitPreference_inverse. The inverse transform uses information from the forward transform to determine that fruitPreference_3 corresponds to a fruitPreference value of 40. Therefore, fruitPreference_inverse is set to 40.00.

Typically, you perform an inverse transform not on fields created by a forward transform, but on fields whose values are calculated by your application. For
example, after processing, the fields of the forward transform may appear as shown below:

```
<table>
<thead>
<tr>
<th>fruitPreference</th>
<th>fruitPreference_0</th>
<th>fruitPreference_5</th>
</tr>
</thead>
<tbody>
<tr>
<td>other fields</td>
<td>40</td>
<td>other</td>
</tr>
<tr>
<td></td>
<td>0.68</td>
<td>0.31</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.55</td>
</tr>
</tbody>
</table>
```

In this example, fruitPreference_3 still contains the largest value; therefore, the inverse transform would return a value of 40.00. Note, however, that processing by your application may cause a different field to have the highest value. Also, the difference in value between the fields fruitPreference_3 and fruitPreference_0 is only 0.07, a difference of less than 10%. The inverse nominal transform always returns a value based on the highest field value and makes no allowances for fields that are very close in value.

On an inverse transform on a nullable field, if the field contains a null, a null is produced as output.

**Example: Using Nominal Transforms**

In this example, you transform two nominal fields, primaryVehicle and secondaryVehicle, in each record of a data set. primaryVehicle and secondaryVehicle describe the two cars, if any, owned by each person described by a record in the data set. The 11 possible values for primaryVehicle and secondaryVehicle are:

- 0 = no vehicle
- 1 = 2-door sedan
- 2 = 4-door sedan
- 3 = station wagon
- 4 = minivan
- 5 = sport utility truck
- 12 = pick-up
- 13 = cargo van
- 31 = motorcycle
- 98 = not specified
- 99 = other
The nominal transform instance for this transform adds 22 new fields to each record of the data set: 11 new fields for primaryVehicle and 11 for secondaryVehicle. Only one of the 11 fields for primaryVehicle and one for secondaryVehicle will be set to true; the remaining fields will all be set to false. Any other values for primaryVehicle and secondaryVehicle besides those listed above cause all fields to be set to false.

Shown below is the osh command for this example:

```
$ osh "fieldtransform
  -nominal primaryVehicle
    -truth '0.1, 0.9'
    -range '0, 5, 1'
    -values '12, 13, 31, 98, 99'
  -nominal secondaryVehicle
    -truth '0.1, 0.9'
    -range '0, 5, 1'
    -values '12, 13, 31, 98, 99'
  < inDS.ds > outDS1.ds"
```

The following example performs an inverse nominal transform on the fields primaryVehicle_0 through primaryVehicle_10 created in the example above. This example adds a new field named primaryVehicle_inverse, of type dfloat, containing the value of the inverse transform.

Shown below is the osh command for this example:

```
$ osh "fieldtransform
  -inversenominal 'primaryVehicle'
    -truth '0.1, 0.9'
    -range '0, 5, 1'
    -values '12, 13, 31, 98, 99'
  < outDS1.ds > outDS2.ds"
```

### Transforming Ordinal Fields

For an ordinal field, two factors can be important for interpreting the data:

- The field’s value
- The relationship of that value to other possible values for the field.

The ordinal transform adds multiple fields to a record that represent the ordinal field as a collection of true and false field values. You can also perform an inverse ordinal transform. The inverse ordinal transform creates a single field containing the inverse ordinal transform. Note that on an inverse transform, if a source field contains a null, a null is produced as output.

This section describes both forward and inverse ordinal transforms. This section also describes how you can derive your own thresholding function when performing an inverse ordinal transform.
Forward Ordinal Transforms

This section follows an example in which the ordinal field finishPosition indicates the finishing position of a horse in a horse race. The possible values for finishPosition are:

- 0 = horse finished fourth or worse
- 1 = horse finished third
- 2 = horse finished second
- 3 = horse finished first

The forward ordinal transform creates multiple fields by appending an index to the source field name, resulting in the names finishPosition_0, finishPosition_1, and so on. If finishPosition in a particular record has a value of 2, meaning the horse came in second, the result fields added to the record are set as shown below:

- finishPosition_0 = true // corresponding to horse's finishing fourth or worse
- finishPosition_1 = true // corresponding to horse's finishing third
- finishPosition_2 = true // corresponding to horse's finishing second
- finishPosition_3 = false // corresponding to horse's finishing first

In an ordinal transform, all fields up to the specified field are set to true; the remaining fields are set to false. You could interpret the results of this transform to mean that any horse finishing second also finished at least third and at least fourth or worse.

The following figure shows the record before and after the ordinal transform:

You can determine the numeric value and data type representing true and false. The fieldtransform operator defines the default value for true as 0.9 and 0.1 for false. You use the -truth option to change these values. In the example shown above, true and false were set 1.00 and 0.00, respectively.
In order to transform a field, you must define the source field values recognized by the transform. You use the operator `-values` option in order to specify these values.

The `-values` option defines a list of expected values for a source field. The following example configures the `fieldtransform` operator to recognize the four possible values for the source field `finishPosition`:

```
$ osh "fieldtransform -ordinal finishPosition
-values '0, 1, 2, 3'
other_options ..."
```

The indices for the output fields generated by the operator are assigned sequentially from 0 to \( n - 1 \) in the order in which you specify the values. The first value sets the source field value that causes the output field with the index 0 to be set to true, the next value sets the source field value corresponding to the output field with the index 1, and so on.

The `-range` option allows you to specify an entire sequence of values with a single argument, as shown below:

```
$ osh "fieldtransform -ordinal finishPosition
-range '0, 3, 1'
other_options ..."
```

The first value after `-range` is the starting value, the second is the ending value, and the third is the increment value. This statement causes the transform to generate four new fields for each record, just as the four separate specifications to `-values` did.

**Inverse Ordinal Transforms**

The inverse ordinal transform defines the Orchestrate inverse ordinal transform mechanism. The inverse ordinal transform relies on information contained within a forward ordinal transform.

The following figure shows a record after a forward and an inverse ordinal transform:

```
record after forward ordinal transform
  finishPosition        finishPosition_0        finishPosition_3
  ↓                        ↓                        ↓
  other fields 2          other fields 1.00 1.00 1.00
  ↓                        ↓                        ↓
  record after inverse ordinal transform
  finishPosition_inverse
  ↓
  other fields 2          other fields 1.00 1.00 1.00 0.00 2.00
```

Orchestrate 7.0 Operators Reference
This figure uses the forward transform example from the previous section. In that example, you calculated a forward transform on the source field \textit{finishPosition}. This field can have any one of the values (0, 1, 2, 3).

In the figure above, \textit{finishPosition} had a value of 2 at the time of the forward transform. Therefore, the forward transform adds four new fields to the record where the fields \textit{finishPosition}_0, \textit{finishPosition}_1, and \textit{finishPosition}_2 are set to 1.00 and \textit{finishPosition}_3 is set to 0.00. Note that the fields added by the forward transform in this example have a data type of \textit{sfloat}.

An inverse transform adds a single field of type \textit{dfloat} whose value is determined by the fields added by the forward transform. The name of the added field in this example is \textit{finishPosition\_inverse}; however, you can choose any name.

In order to perform the inverse transform, the inverse ordinal transform examines the fields added by the forward transform, starting at the field with an index of \textit{0} and ending at the field with the highest index. The value of \textit{finishPosition\_inverse} is determined by the last field found whose value is greater than the cut-off value of:

\[
\frac{\text{true value} + \text{false value}}{2} = \frac{1.00 + 0.00}{2} = 0.50
\]

In this example, the last field with a value greater than 0.50 is \textit{finishPosition\_2}. Therefore, \textit{finishPosition\_inverse} is set to 2.00.

Typically, your application performs an inverse transform not on fields created by a forward transform, but on fields whose values are calculated by your application. For example, after processing, the input fields of the inverse transform may appear as shown below:

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
record after processing & \textit{other fields} & 2 & \textit{other fields} & 0.57 & 0.26 & 0.65 & 0.00 & 2.00 \\
\hline
\end{tabular}
\end{center}

In this example, \textit{finishPosition\_2} still contains the largest value; therefore, the inverse transform would return a value of 2.00. Note, however, that processing by your application may cause a different field to have the highest value.

**Example: Using Ordinal Transforms**

In this example, you create an ordinal transform to convert a single numeric field to multiple fields containing \textit{true} or \textit{false}. The field \textit{midTermGrade} can take one of the following values, which represent grades and are listed in increasing order of quality:

- 0 = F
The values for true and false are 0.9 and 0.1, and the data type of the new fields added to the output data set fields is dfloat.

Shown below is the **osh** command for this example:

```bash
$ osh "fieldtransform -ordinal 'midTermGrade' -truth '0.1, 0.9' -values '0, 1, 2, 3, 4' < inDS.ds > outDS1.ds"
```

The following code performs an inverse ordinal transformation on the fields `midTermGrade_0` through `midTermGrade_4` created in the example above. This inverse transform creates a new field named `midTermGrade_inverse`, of type **dfloat**, containing the value of the inverse transform.

The values used by the forward transform for true and false are 0.9 and 0.1; therefore, the inverse ordinal transform scans the fields `midTermGrade_0` through `midTermGrade_4` for the last added field with a value above the cut-off value of:

\[
\frac{0.9 + 0.1}{2} = 0.5
\]

The last field with a value above 0.5 determines the value of `midTermGrade_inverse`.

**Option:** `inverseordinal`  
**Value:** `midTermGrade`  
**Values:** `'0, 1, 2, 3, 4'`

Shown below is the **osh** command for this example:

```bash
$ osh "fieldtransform -inverseordinal 'midTermGrade' -values '0, 1, 2, 3, 4' < out1DS.ds > outDS2.ds"
```
The DB2 Interface Library

Describes the DB2 interface operators that enable you to convert a set of DB2 records to an Orchestrate data set, write Orchestrate data sets to DB2, insert and update DB2 table records with data contained in an Orchestrate data set, and partition an input data set as DB2 partitions it.

Overview of the DB2 Interface Operators 34 3
Configuring Orchestrate Access 34 3
Establishing a Remote Connection to a DB2 Server 34 4
Handling # and $ Characters in DB2 Column Names 34 5
    Using the External Representation 34 5
    Using the Internal Representation 34 6
Using the -padchar Option 34 6
Running Multiple DB2-Interface Operators in a Single Step 34 7
National Language Support 34 7
    Specifying Character Settings 34 8
    Preventing Character-Set Conversion 34 8
The db2read Operator 34 9
    Overview 34 9
    Data Flow Diagram 34 9
    Properties 34 9
    Operator Action 34 10
    Conversion of a DB2 Result Set to an Orchestrate Data Set 34 11
    Targeting the Read Operation 34 12
        Specifying the DB2 Table Name 34 13
        Explicitly Specifying the Query 34 13
    Specifying Open and Close Commands 34 14
    Syntax and Options 34 14
    Example 1: Reading a DB2 Table with the Table Option 34 17
Example 2: Reading a DB2 Table Sequentially with the -query Option  34 19
Example 3: Reading a Table in Parallel with the -query Option  34 19

The db2write and db2load Operators  34 20
Data Flow Diagram  34 20
Properties  34 20
Actions of the Write Operators  34 21
How Orchestrate Writes the Table: the Default SQL INSERT Statement  34 22
Field Conventions in Write Operations to DB2  34 22
Data Type Conversion  34 24
Write Modes  34 25
Matched and Unmatched Fields  34 26
Syntax and Options  34 27
db2load Special Characteristics  34 33
Example 1: Appending Data to an Existing DB2 Table  34 33
Example 2: Writing Data to a DB2 Table in truncate Mode  34 34
Example 3: Handling Unmatched Orchestrate Fields in a DB2 Write Operation  34 35
Example 4: Writing to a DB2 Table Containing an Unmatched Column  34 36

The db2upsert Operator  34 38
Partitioning for db2upsert  34 38
Data Flow Diagram  34 38
Properties  34 39
Operator Action  34 39
Syntax and Options  34 40

The db2part Operator  34 44
Syntax and Options  34 44
Example  34 46

The db2lookup Operator  34 47
Data Flow Diagram  34 48
Properties  34 48
Syntax and Options  34 48
Example  34 51

Considerations for Reading and Writing DB2 Tables  34 52
Data Translation Anomalies  34 52
Write Operations  34 52
Read Operations  34 52
Using a Node Map  34 53
Overview of the DB2 Interface Operators

This chapter of the Orchestrate 7.0 Operators Reference describes how to use the DB2 operators. To understand the information in this chapter you should be familiar with the operator concepts defined in the Orchestrate 7.0 User Guide.

The DB2 library contains five operators:

- **“The db2read Operator” on page 34-9.**
  Sets up a connection to a DB2 server, sends it a query, and converts the resulting two-dimensional array (the DB2 result set) to an Orchestrate data set. This operator allows you to read an existing DB2 table or to explicitly query the DB2 database.

- **“The db2write and db2load Operators” on page 34-20.**
  Both operators write data to DB2.

- **“The db2upsert Operator” on page 34-38.**
  Updates DB2 table records with data contained in an Orchestrate data set.

- **“The db2part Operator” on page 34-44.**
  Allows an operator to partition an input data set just as DB2 does.

- **“The db2lookup Operator” on page 34-47.**
  With this operator, you can perform a join between one or more DB2 tables and an Orchestrate data set.

See the Orchestrate 7.0 Installation and Administration Manual for information on configuring Orchestrate to communicate with DB2.

Configuring Orchestrate Access

This section assumes that Orchestrate users have been configured to access DB2 using the DB2 configuration process outlined in Chapter 8: RDBMS Configuration in the Orchestrate 7.0 Installation and Administration Manual.

User privileges and Orchestrate settings must be correct to ensure DB2 access. You must make sure your environment is configured correctly. Orchestrate connects to DB2 using your Orchestrate user name and password. Users who perform read and write operations must have valid accounts and appropriate...
privileges on the databases to which they connect. The following table lists the required DB2 privileges.

Table 175  **Required DB2 Privileges**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Required Privileges</th>
</tr>
</thead>
<tbody>
<tr>
<td>db2read</td>
<td>SELECT on the table to be read</td>
</tr>
<tr>
<td>db2write, db2load, db2upsert</td>
<td>For <strong>append</strong> or <strong>truncate</strong> mode, INSERT on an existing table&lt;br&gt;For <strong>create</strong> mode, TABLE CREATE&lt;br&gt;For <strong>replace</strong> mode, INSERT and TABLE CREATE on an existing table</td>
</tr>
<tr>
<td>db2load</td>
<td>DBADM on the database written by the operator&lt;br&gt;If you are a DB2 administrator, you can grant this privilege in several ways. One way is to start DB2, connect to a database, and grant DBADM privilege to a user:&lt;br&gt;<code>&lt;br&gt;db2&gt; CONNECT TO db_name&lt;br&gt;db2&gt; GRANT DBADM ON DATABASE TO USER user_name&lt;br&gt;</code>&lt;br&gt;where <strong>db_name</strong> is the name of the DB2 database and <strong>user_name</strong> is the login name of the Orchestrate user. If you specify the <strong>-msgFile</strong> option, the database instance must have read/write privilege on the file.</td>
</tr>
</tbody>
</table>

The DB2 environment variable, **DB2INSTANCE**, specifies the user name of the owner of the DB2 instance. DB2 uses **DB2INSTANCE** to determine the location of db2nodes.cfg. For example, if you set **DB2INSTANCE** to "Mary", the location of db2nodes.cfg is `~Mary/sqlib/db2nodes.cfg`.

The three methods of specifying the default DB2 database are listed here in order of precedence:

1. The **-dbname** option of the DB2 Interface read and write operators
2. The Orchestrate environment variable **APT_DBNAME**
3. The DB2 environment variable **DB2DBDFT**

**Establishing a Remote Connection to a DB2 Server**

You make a remote connection to DB2 on the conductor node. The players are run locally. In order to remotely connect from Orchestrate to a remote DB2 server, Orchestrate and DB2 need to be configured for remote-connection communication.
Follow these steps:

1. In your configuration file, include the node for the client and the node for the remote platform where DB2 is installed.

2. Set the client instance name using the `-client_instance` option.

3. Optionally set the server using the `-server` option. If the server is not set, you must set the `DB2INSTANCE` environment variable to the DB2 instance name.

4. Set the remote server database name using the `-dbname` option.

5. Set the client alias name for the remote database using the `-client_dbname` option.

6. Set the user name and password for connecting to DB2, using the `-user` and `-password` suboptions.

**Note**

If the name in the `-user` suboption of the `-client_instance` option differs from your UNIX user name, use an owner-qualified name when specifying the table for the `-table`, `-part`, `-upsert`, `-update`, and `-query` options. The syntax is:

```
owner_name.table_name
```

For example, `db2instance1.data23A`.

---

### Handling # and $ Characters in DB2 Column Names

The DB2 operators accept the `#` and `$` characters for DB2 table column names. Orchestrate converts these two reserved characters into an internal format when they are written to the DB2 database and reconverts them when they are read from DB2.

The internal representation for the `#` character is `'__035__'` and the internal representation for the `$` character is `'__036__'`. You should avoid using these internal strings in your DB2 column names.

### Using the External Representation

Use the external representation (with the `#` and `$` characters) when referring to table column names for these options:

- The `where` and `selectlist` options of `db2read` and `db2lookup`
- The `selectlist` options of `db2write` and `db2load`
- The `query` option of `db2read` and `db2lookup`
- The `upsert` and `update` options
Using the Internal Representation

Use the internal representation (with the __035__ and __036__ strings) when referring to table column names for these options:

- The query option of db2lookup when referring to an Orchestrate name. For example:
  ```sql
  -query 'select * from tablename where $A# = ORCHESTRATE.__035__A__035__
  and ##B$ = ORCHESTRATE.__035__B__036__'
  ```
- The upsert and update options when referring to an ORCHESTRATE name. For example:
  ```sql
  -insert 'INSERT INTO tablename (A#,B$#)
  VALUES (ORCHESTRATE.A__035__, ORCHESTRATE.B__036__)
  '
  -update 'UPDATE tablename set B$# = ORCHESTRATE.B__036__
  WHERE (A# = ORCHESTRATE.A__035__)'
  ```

Using the -padchar Option

Use the -padchar option of the db2upsert and db2lookup operators to pad string and ustring fields that are less than the length of the DB2 CHAR column.

Use this option for string and ustring fields that are inserted in DB2 or are used in the WHERE clause of an UPDATE, DELETE, or SELECT statement when all three of these conditions are met:

1. The UPDATE or SELECT statement contains string or ustring fields that map to CHAR columns in the WHERE clause.
2. The length of the string or ustring field is less than the length of the CHAR column.
3. The padding character for the CHAR columns is not the null terminator.

For example, if you add rows to a table using an INSERT statement in SQL, DB2 automatically pads CHAR fields with spaces. When you use the db2upsert operator to update the table or the db2lookup operator to query the table, you must use the -padchar option with the value of a space in order to produce the correct results. Use this syntax:
```sql
-padchar ' ' | OxS20
```

The db2write operator automatically pads with null terminators, and the default pad character for the db2upsert and db2lookup operators is the null terminator. Therefore, you do not need to include the -padchar option when the db2write operator is used to insert rows into the table and the db2upsert or db2lookup operators are used to update or query that table.
The -padchar option should not be used if the CHAR column in the table is a CHAR
FOR BIT type, and any of the input records contain null terminators embedded
within the string or ustring fields.

Running Multiple DB2-Interface Operators in a Single Step

Using Orchestrate’s remote-connection capabilities, you can run multiple DB2-interface
operators in the same step, where each operator can be connected to its own DB2 server.
Here is an osh example of this functionality:

```
osh 'db2read -table table_1 -server DB2instance_name_1
    -dbname database_1
    | db2lookup -table table_2 -key account
    -client_instance DB2instance_1
    -server remote_server_DB2instance_2
    -dbname remote_database_1
    -client_dbname alias_database_2
    -user username -password passwd
    -ifNotFound reject > lookup.out
    | db2load -table table_3 -mode replace
    -server remote_server_DB2instance_3
    -dbname remote_database_2 -client_instance DB2instance_1
    -client_dbname alias_database_3 -user username
    -password passwd''
```

In this example, `db2read` reads from `table_1` of `database_1` in `DB2instance_1`.
The results read from `table_1` are then piped to `db2lookup` to `table_2` of remote
database_1 in remote_server_DB2instance_2.
The records rejected by `db2lookup` are then piped to `db2load` where they are put
in `table_3` of `remote_database_3` in remote_server_DB2instance_3.

National Language Support

Orchestrate National Language Support (NLS) makes it possible for you to
process data in international languages using Unicode character sets.

Orchestrate uses International Components for Unicode (ICU) libraries to support
NLS functionality. For information on national language support, see Chapter 7:
National Language Support in the Orchestrate 7.0 User Guide.

The ICU home page is:

http://oss.software.ibm.com/developerworks/opensource/icu/project
Specifying Character Settings

In DB2 these components can contain multi-byte Unicode characters: database names; database objects such as tables, views, and indices; host variables; cursors; program labels; CHAR and Varchar columns; and Graphic and Vargraphic columns.

The -db_cs option of the DB2 read and write operators let you determine which ICU character set Orchestrate uses to map between DB2 char and Varchar values and Orchestrate ustring values, and to map DLL and query statements for output to DB2.

In $APT_ORCHHOME/etc/db2_cs.txt, these mappings between ICU character sets and DB2 code pages are listed:

<table>
<thead>
<tr>
<th>ICU character setting</th>
<th>DB2 code page</th>
</tr>
</thead>
<tbody>
<tr>
<td>eucJP</td>
<td>954</td>
</tr>
<tr>
<td>ISO-8859-1</td>
<td>819</td>
</tr>
<tr>
<td>UTF-8</td>
<td>1208</td>
</tr>
</tbody>
</table>

If your -db_cs ICU character setting is listed in db2_cs.txt, Orchestrate sets the DB2CODEPAGE environment variable to the corresponding code page. If your ICU character setting is not in db2_cs.txt, you can add it to that file or set the DB2CODEPAGE environment variable yourself.

If there is no code-page mapping in db2_cs.txt and DB2CODEPAGE is not set, Orchestrate uses the DB2 defaults. DB2 converts between your character setting and the DB2 code page, provided that the appropriate language support is set in your operating system; however, no conversion is performed for the data of the db2load operator. Refer to your DB2 documentation for DB2 national language support.

Preventing Character-Set Conversion

The -use_strings option of the db2read and db2lookup operators direct Orchestrate to import DB2 char and Varchars to Orchestrate as Orchestrate strings without converting them from their ASCII or binary form. This option overrides the db_cs option which converts DB2 char and Varchars as ustrings using a specified character set.
The db2read Operator

Overview

The db2read operator sets up a connection to a DB2 server, sends it a query, and converts the resulting two-dimensional array (the DB2 result set) to an Orchestrate data set. The operator allows you to either read an existing DB2 table or to explicitly query the DB2 database.

Data Flow Diagram

Properties

Table 176  db2read Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>0</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>none</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>determined by the SQL query.</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>none</td>
</tr>
<tr>
<td>Execution mode</td>
<td>determined by options passed to it (see “Targeting the Read Operation” on page 34-12)</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>not applicable</td>
</tr>
<tr>
<td>Collection method</td>
<td>not applicable</td>
</tr>
</tbody>
</table>
Here are the chief characteristics of the `db2read` operator:

- The read operation is carried out on either the default database or a database specified by another means. See “Configuring Orchestrate Access” on page 34-3.
- Its output is an Orchestrate data set that you can use as input to a subsequent Orchestrate operator. The output data set is partitioned in the same way as the input DB2 table.
- It translates the query’s result set row by row to an Orchestrate data set, as described in “Conversion of a DB2 Result Set to an Orchestrate Data Set” on page 34-11.
- The translation includes the conversion of DB2 data types to Orchestrate data types, as listed in Table 177 on Page 11.
- It specifies either a DB2 table to read or an SQL query to carry out.
  - If it specifies a read operation of a table, the operation is executed in parallel, one instance of `db2read` on every partition of the table. See “Specifying the DB2 Table Name” on page 34-13.
  - If it specifies a query, the operation is executed sequentially, unless the query directs it to operate in parallel. See “Explicitly Specifying the Query” on page 34-13.
- It optionally specifies commands to be executed on all processing nodes before the read operation is performed and after it has completed.

**Note**

An RDBMS such as DB2 does not guarantee deterministic ordering behavior unless an SQL operation constrains it to do so. Thus, if you read the same DB2 table multiple times, DB2 does not guarantee delivery of the records in the same order every time.

In addition, while DB2 allows you to run queries against tables in parallel, not all queries should be run in parallel. For example, some queries perform an operation that must be carried out sequentially, such as a group-by operation or a non-collocated join operation. These types of queries should be executed sequentially in order to guarantee correct results and finish in a timely fashion.
Conversion of a DB2 Result Set to an Orchestrate Data Set

A DB2 result set is defined by a collection of rows and columns. Here is how the db2read operator translates the query’s result set to an Orchestrate data set:

- The rows of a DB2 result set correspond to the records of an Orchestrate data set.
- The columns of a DB2 row correspond to the fields of an Orchestrate record, and the name and data type of a DB2 column correspond to the name and data type of an Orchestrate field.
- Names are translated exactly except when a component of a DB2 column name is not compatible with Orchestrate naming conventions which place no limit on field-name length, but have the following restrictions:
  - The name must start with a letter or underscore (_) character.
  - The name can contain only alphanumeric and underscore characters.
  - The name is case insensitive.
  - When there is an incompatibility, Orchestrate converts the DB2 column name as follows:
    - If the DB2 column name does not begin with a letter or underscore, the string “APT__column#” (two underscores) is prepended to the column name, where column# is the number of the column. For example, if the third DB2 column is named 5foo, the Orchestrate field is named APT__35foo.
    - If the DB2 column name contains a character that is not alphanumeric or an underscore, the character is replaced by two underscore characters.
- Both DB2 columns and Orchestrate fields support nulls, and a null contained in a DB2 column is stored as a null in the corresponding Orchestrate field.
- The DB2 read operators convert DB2 data types to Orchestrate data types, as in the next table:

Table 177  DB2 Interface Operator Data Type Conversions

<table>
<thead>
<tr>
<th>DB2 Data Type</th>
<th>Orchestrate Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)</td>
<td>string[n] or ustring[n]</td>
</tr>
<tr>
<td>CHARACTER VARYING(n,r)</td>
<td>string[max =n] or ustring[max =n]</td>
</tr>
<tr>
<td>DATE</td>
<td>date</td>
</tr>
</tbody>
</table>
Targeting the Read Operation

You can read a DB2 table using one of these methods:

- Specify the table name and allow Orchestrate to generate a default query that reads the entire table.
- Explicitly specify the query.

Table 177  DB2 Interface Operator Data Type Conversions (continued)

<table>
<thead>
<tr>
<th>DB2 Data Type</th>
<th>Orchestrate Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME</td>
<td>time or timestamp with corresponding fractional precision for time</td>
</tr>
<tr>
<td></td>
<td>If the DATETIME starts with a year component, the result is a timestamp field. If the DATETIME starts with an hour, the result is a time field.</td>
</tr>
<tr>
<td>DECIMAL[p,s]</td>
<td>decimal[p,s] where p is the precision and s is the scale</td>
</tr>
<tr>
<td></td>
<td>The maximum precision is 32, and a decimal with floating scale is converted to a dfloat.</td>
</tr>
<tr>
<td>DOUBLE-PRECISION</td>
<td>dfloat</td>
</tr>
<tr>
<td>FLOAT</td>
<td>dfloat</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int32</td>
</tr>
<tr>
<td>MONEY</td>
<td>decimal</td>
</tr>
<tr>
<td>NCHAR(n,r)</td>
<td>string[n] or ustring[n]</td>
</tr>
<tr>
<td>NVARCHAR(n,r)</td>
<td>string[max=n] or ustring[max=n]</td>
</tr>
<tr>
<td>REAL</td>
<td>sfloat</td>
</tr>
<tr>
<td>SERIAL</td>
<td>int32</td>
</tr>
<tr>
<td>SMALLFLOAT</td>
<td>sfloat</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>int16</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>string[max=n] or ustring[max=n]</td>
</tr>
</tbody>
</table>

Note Data types that are not listed in the table shown above generate an error.
Specifying the DB2 Table Name

If you use the -table option to pass the table name to db2read, the operator functions in parallel, one instance for each partition of the table.

Orchestrate issues the following SQL SELECT statement to read the table:

```
select [selectlist]
from tableName
where nodenumber(colName)=current node
[and (filter)];
```

You can specify optional parameters to narrow the read operation. They are as follows:

- **-selectlist** specifies the columns of the table to be read. By default, Orchestrate reads all columns.
- **-filter** specifies the rows of the table to exclude from the read operation. By default, Orchestrate reads all rows.

Note that the default query’s WHERE clause contains the predicate:

```
nodenumber(colName)=current node
```

where colName corresponds to the first column of the table or select list. This predicate is automatically generated by Orchestrate and is used to create the correct number of DB2 read operators for the number of partitions in the DB2 table.

You can optionally specify the -open and -close options. These commands are executed by DB2 on every processing node containing a partition of the table before the table is opened and after it is closed.

Explicitly Specifying the Query

If you choose the -query option, you pass an SQL query to the operator. The query specifies the table and the processing that you want to perform on the table as it is read into Orchestrate. The SQL statement can contain joins, views, database links, synonyms, and so on. However, the following restrictions apply to -query:

- It cannot contain bind variables.
- By default, the query is executed sequentially (on a single processing node). It can also be executed in parallel when you specify the -part option.
- If you want to include a filter or select list, you must specify them as part of the query.

You can optionally specify the -open and -close options. These commands are executed by DB2 on every processing node containing a partition of the table before the table is opened and after it is closed.
If you choose the **-query** option, the operator functions sequentially by default. It functions in parallel if two conditions are satisfied:

- The SQL query contains a WHERE clause providing information that enables Orchestrate to run the query in parallel
- The **-part table_name** option and argument are specified, so that one instance of the operator is created on every processing node containing a partition of **table_name**.

**Note** Complex queries, such as joins, executed in parallel may cause the database to consume large amounts of system resources.

### Specifying Open and Close Commands

You can optionally specify the **-open** and **-close** command options. These commands are executed by DB2 on every processing node containing a partition of the table before the table is opened and after it is closed. Their syntax is:

- **open** open_command
- **close** close_command

If you do not specify an open command and the read operation is carried out in parallel, Orchestrate runs the following default command:

```
lock table table_name in share mode;
```

where **table_name** is the table specified by the **-table** option.

This command locks the table until Orchestrate finishes reading the table. The table cannot be written to when it is locked; however, if you specify an explicit open command, the lock statement is not run. When DB2 is accessed sequentially, Orchestrate provides no default open command.

The close command is executed by the operator after Orchestrate finishes reading the DB2 table and before it disconnects from DB2. If you do not specify a close command, the connection is immediately terminated after Orchestrate finishes reading the DB2 table.

If DB2 has been accessed in parallel and **-close** is not specified, Orchestrate releases the table lock obtained by the default open_command.

### Syntax and Options

You must specify either the **-table** or the **-query** option. All others are optional. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose the value in single quotes.
db2read
 -table table_name [-filter filter] [-selectlist list]
 | -query sql_query [-part table_name]
 [-client_instance client_instance_name
  -client_dbname database
  -user user_name
  -password password]
 [-close close_command]
 [-db_cs character_set]
 [-dbname database_name]
 [-open open_command]
 [-server server_name]
 [-use_strings]

Table 178  db2read Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
</table>
| -client_instance | -client_instance client_instance_name
                 [-client_dbname database]
                 -user user_name
                 -password password |

Specifies the client DB2 instance name. This option is required for a remote connection.

The -client_dbname suboption specifies the client database alias name for the remote server database. If you do not specify this option, Orchestrate uses the value of the -dbname option, or the value of the APT_DBNAME environment variable, or DB2DBDFT; in that order.

The required -user and -password suboptions specify a user name and password for connecting to DB2.

-close

- close close_command

Specify a command to be parsed and executed by DB2 on all processing nodes accessed by the operator after Orchestrate finishes reading the DB2 table and before it disconnects from DB2.

See “Specifying Open and Close Commands” on page 34-14 for more details.
### Table 178  db2read Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-db_cs</td>
<td>-db_cs character_set</td>
</tr>
<tr>
<td></td>
<td>Specify the character set to map between DB2 char and Varchar values and Orchestrate ustring schema types and to map SQL statements for output to DB2. The default character set is UTF-8 which is compatible with your osh jobs that contain 7-bit US-ASCII data. For information on national language support, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site: <a href="http://oss.software.ibm.com/icu/charset">http://oss.software.ibm.com/icu/charset</a></td>
</tr>
<tr>
<td>-dbname</td>
<td>-dbname database_name</td>
</tr>
<tr>
<td></td>
<td>Specifies the name of the DB2 database to access.</td>
</tr>
<tr>
<td></td>
<td>By default, the operator uses the setting of the environment variable APT_DBNAME, if defined, and DB2DBDFT otherwise. Specifying -dbname overrides APT_DBNAME and DB2DBDFT. See the Orchestrate 7.0 Installation and Administration Manual for information on creating the DB2 configuration.</td>
</tr>
<tr>
<td>-open</td>
<td>-open open_command</td>
</tr>
<tr>
<td></td>
<td>Specifies any command to be parsed and executed by DB2. Orchestrate causes DB2 to execute this command on all processing nodes to be accessed before opening the table. See “Specifying Open and Close Commands” on page 34-14 for more details.</td>
</tr>
<tr>
<td>-query</td>
<td>-query sql_query [-part table_name]</td>
</tr>
<tr>
<td></td>
<td>Specifies an SQL query to read one or more tables. The query specifies the tables and the processing that you want to perform on the tables as they are read into Orchestrate. This statement can contain joins, views, database links, synonyms, and so on. The -part suboption specifies execution of the query in parallel on the processing nodes containing a partition of table_name. If you do not specify -part, the operator executes the query sequentially on a single node. If the name in the -user suboption of the -client_instance option differs from your UNIX user name, use an owner-qualified name when specifying the table. The syntax is: owner_name.table_name The default open command for a parallel query uses the table specified by the -part option. Either the -table or -query option must be specified.</td>
</tr>
</tbody>
</table>
Example 1: Reading a DB2 Table with the Table Option

The following figure shows **table_1**, a DB2 table used as an input to the **db2read** Orchestrate operator.
In this example:

- The read operation is performed in parallel, with one instance of the operator for each partition of the table, because the `-table` option is specified.
- The entire table is read because neither a `-filter` nor a `-selectlist` suboption is specified.

The `db2read` operator writes the table to an Orchestrate data set. The schema of the data set is also shown in this figure. The data set is the input of the next operator.

The DB2 table contains three columns. Each DB2 column name becomes the name of an Orchestrate field. The operator converts the data type of each column to its corresponding Orchestrate data type, as listed in the this table:

<table>
<thead>
<tr>
<th>Column/Field Name</th>
<th>DB2 Data Type</th>
<th>Converted to Orchestrate Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemNum</td>
<td>INTEGER</td>
<td>int32</td>
</tr>
<tr>
<td>price</td>
<td>DECIMAL</td>
<td>decimal</td>
</tr>
<tr>
<td>storeID</td>
<td>SMALLINT</td>
<td>int16</td>
</tr>
</tbody>
</table>

Here is the `osh` syntax for this example:

```
$ osh "db2read -table table_1 ... "
```
Example 2: Reading a DB2 Table Sequentially with the -query Option

As in Example 1 above, this query reads a DB2 table named `table_1`. In this example, the query explicitly specifies three columns to read from the table, `itemNum`, `price`, and `storeID`. The read operation is executed sequentially. This is the default execution mode of `db2read` when the `-query` option is chosen.

Here is the `osh` syntax for this example:

```bash
$ osh "db2read -query 'select itemNum, price, storeID from table_1' ...
```

Example 3: Reading a Table in Parallel with the -query Option

The following query reads every row and column of a DB2 table named `table_1`. The `-part` option and `table_1` argument direct Orchestrate to create an instance of the operator on every processing node containing a partition of this table.

Here is the `osh` syntax for this example:

```bash
$ osh 'db2read -query 'select * from table_1 where nodenumber(itemNum)=current node' -part table_1 ...
```

where `itemNum` is the name of the first column of the table or select list. This predicate is automatically generated by Orchestrate and is used to create the correct number of DB2 read operators for the number of partitions in the DB2 table.
The db2write and db2load Operators

The Orchestrate operators that write to DB2 are `db2write` and `db2load`. They function identically with these exceptions:

- The `db2load` operator takes advantage of the fast DB2 loader technology for writing data to the database.
- The `db2load` operator accepts four options that `db2write` does not: `-ascii`, `-cleanup`, `-msgfile`, and `-nonrecoverable`.
- The `db2load` operator requires that Orchestrate users have DBADM privilege on the DB2 database written by the operator.

The operators set up a connection to DB2 and insert records into a table. These operators take a single input data set. The write mode of the operators determines how the records of a data set are inserted into the table.

Both operators are documented as the write operators. The particular aspects of `db2load` are indicated in “Syntax and Options” on page 34-27 and “db2load Special Characteristics” on page 34-33.

Also see the section on the `db2upsert` operator which inserts and updates DB2 table records with data contained in an Orchestrate data set. It is described in “The db2upsert Operator” on page 34-38.

Data Flow Diagram

```
input data set
```

```
db2write | db2load
```

```
DB2 Table
```

Properties

Table 179  `db2write` and `db2load` Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>0</td>
</tr>
</tbody>
</table>
Actions of the Write Operators

Here are the chief characteristics of the Orchestrate operators that write Orchestrate data sets to DB2 tables:

- They translate the Orchestrate data set record by record to DB2 table rows by means of an SQL INSERT statement, as described in “How Orchestrate Writes the Table: the Default SQL INSERT Statement” on page 34-22.

- The fields of the Orchestrate record correspond to the columns of the DB2 table. Fields may be only of certain data types, as described in “Field Conventions in Write Operations to DB2” on page 34-22.

- Translation includes the conversion of Orchestrate data types to DB2 data types, as listed in “Data Type Conversion” on page 34-24.

- The write operators append records to an existing table, unless you set another mode of writing. “Write Modes” on page 34-25 discusses these modes.

- When you write to an existing table, the schema of the table determines the operator’s input interface and the data set schema must be compatible with the table’s schema. See “Matched and Unmatched Fields” on page 34-26 and “Syntax and Options” on page 34-27.
• Each instance of a parallel write operator running on a processing node writes its partition of the data set to the DB2 table. A failure by one instance aborts the write operation.

Operators optionally specify DB2 commands to be parsed and executed on all processing nodes before the write operation runs using the `-open` option or after it finishes using the `-close` option.

**How Orchestrate Writes the Table: the Default SQL INSERT Statement**

You cannot explicitly define an SQL statement for the write operation. Instead, Orchestrate generates an SQL INSERT statement that writes to the table. However, optional parameters of the write operators allow you to narrow the operation.

When you specify the DB2 table name to a write operator, Orchestrate creates this INSERT statement:

```sql
insert into table_name[(selectlist)] values (?, ?, ?, ...);
```

where:

• `table_name` specifies the name of a DB2 table. By default, if `table_name` does not exist, the step containing the write operator terminates, unless you use the `-mode create` option.

To set the table name, you specify the `-table` option.

• `selectlist` optionally specifies a clause of the INSERT statement to determine the fields of the data set to be written.

If you do not specify `selectlist`, Orchestrate writes all fields in the data set. You specify the write operator’s `-selectlist` suboption of `-table` to set the `selectlist`.

• `?,?,?, ...` contains one input parameter for each column written by Orchestrate. By default, this clause specifies that all fields in the input data set are written to DB2.

However, when you specify a `selectlist`, the default is modified to contain only the columns defined by the `selectlist`.

**Field Conventions in Write Operations to DB2**

The record schema of the input Orchestrate data set defines the field name and data type of each field. When the write operators write this data set to DB2, they do not modify the field names but translate the Orchestrate data types to DB2 data types, with these limitations:
DB2 column names may have a length limit. If an Orchestrate field name is longer than that limit, which varies by DB2 release, do one of the following:

- Specify **-truncate** to the write operator to truncate Orchestrate field names to 18 characters, or specify **-truncate** and **-truncationLength n**, to set the field name to some other length.

- Invoke the **modify** operator to modify the Orchestrate field name. See Chapter 13, “The modify Operator”.

The order of the fields of the Orchestrate record and that of the columns of the DB2 rows can differ but are nonetheless written successfully.

The size of Orchestrate records is limited to 32 KB, but a DB2 table row can be 4005 bytes. If you try to write an Orchestrate record of greater length, DB2 returns an error and terminates the step.

The maximum number of columns allowed in a UDB table is 500, and the maximum allowed in a view is 5000.

An Orchestrate data set written to DB2 may not contain fields of certain types. If it does, an error occurs and the corresponding step terminates. However, Orchestrate offers operators that modify certain data types to types DB2 accepts, as in the next table:

Table 180  **Operators that modify Orchestrate Types for DB2**

<table>
<thead>
<tr>
<th>Incompatible Type</th>
<th>Operator That Changes It</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw</td>
<td><strong>field_import</strong>; see “The field_import Operator” on page 30-13</td>
</tr>
<tr>
<td>Strings, fixed or variable length, longer than 4000 bytes</td>
<td>none</td>
</tr>
<tr>
<td>Subrecord</td>
<td><strong>promotesubrec</strong>; see “The promotesubrec Operator” on page 30-25</td>
</tr>
<tr>
<td>Tagged aggregate</td>
<td><strong>tagbatch</strong>; see “The tagbatch Operator” on page 30-33</td>
</tr>
<tr>
<td>Unsigned integer of any size</td>
<td><strong>modify</strong>; see Chapter 13, “The modify Operator”</td>
</tr>
</tbody>
</table>
Data Type Conversion

The `db2write` and `db2load` operators convert Orchestrate data types to DB2 data types, as listed in the next table:

Table 181  `db2write` and `db2load` Type Conversions

<table>
<thead>
<tr>
<th>Orchestrate Data Type</th>
<th>DB2 Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>DATE</td>
</tr>
<tr>
<td>decimal[p,s]</td>
<td>DECIMAL[p,s]</td>
</tr>
<tr>
<td><code>p</code> is decimal’s precision and <code>s</code> is its scale</td>
<td></td>
</tr>
<tr>
<td>int8</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>int16</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>int32</td>
<td>INTEGER</td>
</tr>
<tr>
<td>sfloat</td>
<td>FLOAT</td>
</tr>
<tr>
<td>dfloat</td>
<td>FLOAT</td>
</tr>
<tr>
<td>fixed-length string in the form <code>string[n]</code> and</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td><code>ustring[n]</code>; length &lt;= 254 bytes</td>
<td>where <code>n</code> is the string length</td>
</tr>
<tr>
<td>fixed-length string in the form <code>string[n]</code> and</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td><code>ustring[n]</code>; 255 &lt;= length &lt;= 4000 bytes</td>
<td>where <code>n</code> is the string length</td>
</tr>
<tr>
<td>variable-length string, in the form <code>string[n]</code></td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td><code>ustring[n]</code>; maximum length &lt;= 4000 bytes</td>
<td>where <code>n</code> is the maximum string length</td>
</tr>
<tr>
<td>variable-length string in the form <code>string</code> and</td>
<td>VARCHAR(32)*</td>
</tr>
<tr>
<td><code>ustring</code></td>
<td></td>
</tr>
<tr>
<td>string and <code>ustring</code>; 4000 bytes &lt; length</td>
<td>Not supported</td>
</tr>
<tr>
<td>time</td>
<td>TIME</td>
</tr>
<tr>
<td>timestamp</td>
<td>TIMESTAMP</td>
</tr>
</tbody>
</table>

*The default length of VARCHAR is 32 bytes. That is, 32 bytes are allocated for each variable-length string field in the input data set. If an input variable-length string field is longer than 32 bytes, the operator issues a warning. You can use the `-stringlength` option to modify the default length.*

Orchestrate data types not listed in this table generate an error and terminate the step. Use the `modify` operator to perform type conversions. See Chapter 13, “The modify Operator.”
Any column in the DB2 table corresponding to a nullable data set field will be nullable.

Orchestrate and DB2 integer and floating-point data types have the same range and precision. You need not worry about numeric overflow or underflow.

**Write Modes**

The write mode of the operator determines how the records of the data set are inserted into the destination table. The write mode can have one of the following values:

- **append**: The table must exist and the record schema of the data set must be compatible with the table. The write operator appends new rows to the table. This is the default mode. The schema of the existing table determines the input interface of the operator. See “Example 1: Appending Data to an Existing DB2 Table” on page 34-33.

- **create**: The operator creates a new table. If a table exists of the same name as the one you want to create, the step that contains the operator terminates with an error. You must specify either `create` mode or `replace` mode if the table does not exist. The schema of the new table is determined by the schema of the Orchestrate data set. By default, Orchestrate creates the table on all processing nodes in the default table space and uses the first column in the table, corresponding to the first field in the input data set, as the partitioning key. You can override these options for partitioning keys and table space by means of the `-dboptions` option.

- **replace**: The operator drops the existing table and creates a new one in its place. If a table exists of the same name as the one you want to create, it is overwritten. DB2 uses the default partitioning method for the new table. The schema of the new table is determined by the schema of the Orchestrate data set.

- **truncate**: The operator retains the table attributes but discards existing records and appends new ones. The schema of the existing table determines the input interface of the operator. See “Example 2: Writing Data to a DB2 Table in truncate Mode” on page 34-34.

**Note**

If a previous write operation fails, you can retry your application specifying a write mode of `replace` to delete any information in the output table that may have been written by the previous attempt to run your program.
Each mode requires specific user privileges, as listed in this table:

<table>
<thead>
<tr>
<th>Write Mode</th>
<th>Required Privileges</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>INSERT on existing table</td>
</tr>
<tr>
<td>create</td>
<td>TABLE CREATE</td>
</tr>
<tr>
<td>replace</td>
<td>INSERT and TABLE CREATE on existing table</td>
</tr>
<tr>
<td>truncate</td>
<td>INSERT on existing table</td>
</tr>
</tbody>
</table>

### Matched and Unmatched Fields

The schema of the DB2 table determines the operator's interface schema. Once the operator determines this, it applies the following rules to determine which data set fields are written to the table:

1. Fields of the input data set are matched by name with fields in the input interface schema.

   Orchestrate performs default data type conversions to match the input data set fields with the input interface schema. See the chapter on using Orchestrate operators in the *Orchestrate 7.0 User Guide* for more information on these conversions. You can also use the `modify` operator to perform explicit data type conversions. See Chapter 13, “The modify Operator”.

2. If the input data set contains fields that do not have matching components in the table, the operator causes an error and terminates the step.

   This rule means that Orchestrate does not add new columns to an existing table if the data set contains fields that are not defined in the table. Note that you can use either the `-drop` option or the `modify` operator with the `db2write` operator to drop extra fields from the data set. See “Example 3: Handling Unmatched Orchestrate Fields in a DB2 Write Operation” on page 34-35 and Chapter 13, “The modify Operator”.

3. Columns in the DB2 table that do not have corresponding fields in the input data set are set to their default value, if one is specified in the DB2 table. If no default value is defined for the DB2 column and it supports nulls, it is set to null. Otherwise, Orchestrate issues an error and terminates the step. See “Example 4: Writing to a DB2 Table Containing an Unmatched Column” on page 34-36.

4. Orchestrate data sets support nullable fields. If you write a data set to an existing table and a field contains a null, the DB2 column must also support nulls. If not, Orchestrate issues an error message and terminates the step.
You can use the modify operator to convert a null in an input data set field to a different value in order to avoid the error. See Chapter 13, “The modify Operator” for more information on handling nulls.

Syntax and Options

Following is the syntax of the db2write and db2load operators. Exactly one occurrence of the -table option is required. All other options are optional. Most options apply to both operators; the exceptions are noted.

db2write | db2load
  -table table_name[-selectlist selectlist]
  [-ascii] (db2load only)
  [-cleanup] (db2load only)
  [-client_instance client_instance_name]
  [-client_dbname database]
  [-user user_name]
  [-password password]
  [-close close-command]
  [-rowCommitInterval integer] (db2write only)
  [-db_cs character_set]
  [-dbname database_name]
  [-dboptions '{[ = table_space,]
  [  key = field0, ... key = fieldN]}'
  [-drop]
  [-mode create | replace | append | truncate]
  [-msgfile msgFile] (db2load only)
  [-nonrecoverable] (db2load only)
  [-open open_command]
  [-server server_name]
  [-stringlength string_length]
  [-truncate]
  [-truncationLength n]
Table 183  **db2write and db2load Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-ascii</code></td>
<td><code>db2load</code> only. Specify this option to configure DB2 to use the ASCII-delimited format for loading binary numeric data instead of the default ASCII-fixed format. This option can be useful when you have variable-length fields, because the database does not have to allocate the maximum amount of storage for each variable-length field. However, all numeric fields are converted to an ASCII format by DB2, which is a CPU-intensive operation. See the DB2 reference manuals for more information.</td>
</tr>
</tbody>
</table>
| `-cleanup`       | `db2load` only. Specify this option to deal with operator failures during execution that leave the loading tablespace that it was loading in an inaccessible state. For example, if the following `osh` command was killed in the middle of execution:     

```
    db2load -table upcdata -dbname my_db
```

The tablespace in which this table resides will probably be left in a quiesced exclusive or load pending state. In order to reset the state to normal, run the above command specifying `-cleanup`, as follows:

```
    db2load -table upcdata -dbname my_db -cleanup
```

The cleanup procedure neither inserts data into the table nor deletes data from it. You must delete rows that were inserted by the failed execution either through the DB2 command-level interpreter or by running the operator subsequently using the `replace` or `truncate` modes. |
| `-client_instance` | `-client_instance client_instance_name` `[ -client_dbname database]` `-user user_name` `-password password` Specifies the client DB2 instance name. This option is required for a remote connection. The `-client_dbname` suboption specifies the client database alias name for the remote server `database`. If you do not specify this option, Orchestrate uses the value of the `-dbname` option, or the value of the `APT_DBNAME` environment variable, or `DB2DBDFT`; in that order. The required `-user` and `-password` suboptions specify a user name and password for connecting to DB2. |
### db2write and db2load Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-close</code></td>
<td><code>close close_command</code></td>
</tr>
<tr>
<td></td>
<td>Specifies any command to be parsed and executed by the DB2 database</td>
</tr>
<tr>
<td></td>
<td>on all processing nodes after Orchestrate finishes processing the DB2</td>
</tr>
<tr>
<td></td>
<td>table.</td>
</tr>
<tr>
<td><code>-rowCommitInterval</code></td>
<td><code>rowCommitInterval integer</code></td>
</tr>
<tr>
<td></td>
<td>Specifies the size of a commit segment. Specify an integer that is 1</td>
</tr>
<tr>
<td></td>
<td>or larger. The specified number must be a multiple of the input</td>
</tr>
<tr>
<td></td>
<td>array size. The default size is 2000. You can also use the</td>
</tr>
<tr>
<td></td>
<td><code>APT_RDBMS_COMMIT_ROWS</code> environment to specify the size of a commit.</td>
</tr>
<tr>
<td><code>-db_cs</code></td>
<td><code>db_cs character_set</code></td>
</tr>
<tr>
<td></td>
<td>Specify the character set to map between DB2 <code>char</code> and <code>varchar</code></td>
</tr>
<tr>
<td></td>
<td>values and Orchestrate <code>ustring</code> schema types and to map SQL</td>
</tr>
<tr>
<td></td>
<td>statements for output to DB2. The default character set is <code>UTF-8</code></td>
</tr>
<tr>
<td></td>
<td>which is compatible with your <code>osh</code> jobs that contain 7-bit <code>US-ASCII</code></td>
</tr>
<tr>
<td></td>
<td>data. For information on national language support, see Chapter 7:</td>
</tr>
<tr>
<td></td>
<td>National Language Support in the Orchestrate 7.0 User Guide; and</td>
</tr>
<tr>
<td></td>
<td>reference this IBM ICU site:</td>
</tr>
<tr>
<td></td>
<td><code>http://oss.software.ibm.com/icu/charset</code></td>
</tr>
<tr>
<td><code>-dbname</code></td>
<td><code>dbname database_name</code></td>
</tr>
<tr>
<td></td>
<td>Specifies the name of the DB2 database to access. By default, the</td>
</tr>
<tr>
<td></td>
<td>operator uses the setting of the environment variable <code>APT_DBNAME</code>,</td>
</tr>
<tr>
<td></td>
<td>if defined, and <code>DB2DBDFT</code> otherwise. Specifying <code>-dbname</code> overrides</td>
</tr>
<tr>
<td></td>
<td><code>APT_DBNAME</code> and <code>DB2DBDFT</code>.</td>
</tr>
</tbody>
</table>
Chapter 34

The DB2 Interface Library

The db2write and db2load Operators

Table 183 db2write and db2load Options (continued)
Option

Use

-dboptions

-dboptions {
[-tablespace = table_space ,]
[-key = field ...]}

Specifies an optional table space or partitioning key to be used by DB2
to create the table. You can specify this option only when you perform
the write operation in either create or replace mode.
The partitioning key must be the external column name. If the external
name contains # or $ characters, the entire column name should be
surrounded by single backslashed quotes. For example:
-dboptions '{key=\'B##B$\'}'

By default, Orchestrate creates the table on all processing nodes in the
default table space and uses the first column in the table, corresponding
to the first field in the input data set, as the partitioning key.
You specify arguments to -dboptions as a string enclosed in braces.
This string can contain a single -tablespace argument, a single nodegroup argument, and multiple -key arguments, where:
-tablespace defines the DB2 table space used to store the table.
-nodegroup specifies the name of the node pool (as defined in the
Orchestrate configuration file) of nodes on which DB2 is installed.
-key specifies a partitioning key for the table.

-drop

-drop

Causes the operator to silently drop all input fields that do not
correspond to fields in an existing DB2 table. By default, the operator
reports an error and terminates the step if an input field does not have a
matching column in the destination table.

-mode

-mode create | replace | append | truncate

Specifies the write mode of the operator.
append (default): New records are appended to an existing table.
create: Create a new table. Orchestrate reports an error and terminates
the step if the DB2 table already exists. You must specify this mode if
the DB2 table does not exist.
truncate: The existing table attributes (including schema) and the DB2
partitioning keys are retained, but any existing records are discarded.
New records are then appended to the table.
replace: The existing table is first dropped and an entirely new table is
created in its place. DB2 uses the default partitioning method for the
new table.

34 30

Orchestrate 7.0 Operators Reference


Table 183  db2write and db2load Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-msgfile</td>
<td>`- msgfile msgfile</td>
</tr>
<tr>
<td></td>
<td>db2load only. Specifies the file where the DB2 loader writes diagnostic messages. The msgFile can be an absolute path name or a relative path name. Regardless of the type of path name, the database instance must have read/write privilege to the file. By default, each processing node writes its diagnostic information to a separate file named APT_DB2_LOADMSG_nodenum where nodenum is the DB2 node number of the processing node. Specifying -msgFile configures DB2 to write the diagnostic information to the file msgFile_nodenum. If you specify a relative path name, Orchestrate attempts to write the files to the following directories, in this order: 1 The file system specified by the default resource scratchdisk, as defined in the Orchestrate configuration file. 2 The directory specified by the tmpdir parameter to the Orchestrate server. 3 The directory /tmp.</td>
</tr>
<tr>
<td>-nonrecoverable</td>
<td>`- nonrecoverable</td>
</tr>
<tr>
<td></td>
<td>db2load only. Specifies that your load transaction is marked as nonrecoverable. It will not be possible to recover your transaction with a subsequent roll forward action. The roll forward utility skips the transaction, and marks the table into which data was being loaded as &quot;invalid&quot;. The utility also ignores any subsequent transactions against the table. After a roll forward is completed, the table can only be dropped. Table spaces are not put in a backup pending state following the load operation, and a copy of the loaded data is not made during the load operation.</td>
</tr>
<tr>
<td>-open</td>
<td>`- open open_command</td>
</tr>
<tr>
<td></td>
<td>Specifies any command to be parsed and executed by the DB2 database on all processing nodes. DB2 runs this command before opening the table.</td>
</tr>
<tr>
<td>-server</td>
<td>`-server server_name</td>
</tr>
<tr>
<td></td>
<td>Optionally specifies the DB2 instance name for the table. By default, Orchestrate uses the setting of the DB2INSTANCE environment variable. Specifying -server overrides DB2INSTANCE.</td>
</tr>
</tbody>
</table>
### db2write and db2load Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>-stringlength</strong></td>
<td>Sets the default string length of variable-length strings written to a DB2 table. If you do not specify a length, Orchestrate uses a default size of 32 bytes. Variable-length strings longer than the set length cause an error. The maximum length you can set is 4000 bytes. Note that the operator always allocates <code>string_length</code> bytes for a variable-length string. In this case, setting <code>string_length</code> to 4000 allocates 4000 bytes for every string. Therefore, you should set <code>string_length</code> to the expected maximum length of your largest string and no larger.</td>
</tr>
<tr>
<td><strong>-table</strong></td>
<td>Specifies the name of the DB2 table. If the output mode is <code>create</code>, the table must not exist and you must have TABLE CREATE privileges. By default, Orchestrate creates the table without setting partitioning keys. By default, DB2 uses the first column as the partitioning key. You can use <code>-dboptions</code> to override this default operation. If the output mode is <code>append</code> or <code>truncate</code>, the table must exist and you must have INSERT privileges. In <code>replace</code> mode, the table may already exist. In this case, the table is dropped and and the table is created in <code>create</code> mode. The table name may not contain a view name. If your DB2 user name does not correspond to the owner of the table, you can optionally prefix <code>table_name</code> with that of a table owner in the form <code>tableOwner.tableName</code>. The <code>tableOwner</code> string must be 8 characters or less. If you do not specify <code>tableOwner</code>, Orchestrate uses the following statement to determine your user name: <code>select user from sysibm.systables;</code> <strong>-selectlist</strong> optionally specifies an SQL select list used to determine which fields are written. The list does not support the use of bind variables.</td>
</tr>
<tr>
<td><strong>-truncate</strong></td>
<td>Select this option to configure the operator to truncate Orchestrate field names to 18 characters. To specify a length other than 18, use the <code>-truncationLength</code> option along with this one. See “Field Conventions in Write Operations to DB2” on page 34-22 for further information.</td>
</tr>
</tbody>
</table>
db2load Special Characteristics

During the table load operation, \texttt{db2load} keeps an exclusive lock on the entire tablespace into which it loads data and no other tables in that tablespace can be accessed. The operator issues the \texttt{QUIESCE TABLESPACES} command to get and retain the lock.

The DB2 load operator performs a non-recoverable load. That is, if the load operation is terminated before it is completed, the contents of the table are unusable and the tablespace is left in a load pending state. You must run \texttt{db2load} in \texttt{truncate} mode to clear the load pending state.

The \texttt{db2load} operator requires that Orchestrate users, under their Orchestrate login name, have DBADM privilege on the DB2 database written by the operator.

**Example 1: Appending Data to an Existing DB2 Table**

In this example, a DB2 write operator appends the contents of an Orchestrate data set to an existing DB2 table. This is the default action of DB2 write operators.

The record schema of the Orchestrate data set and the row schema of the DB2 table correspond to one another, and field and column names are identical (though they are not in the same order). Here are the input Orchestrate record schema and output DB2 row schema:

<table>
<thead>
<tr>
<th>Input Orchestrate Record</th>
<th>Output DB2 Table Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{itemNum int32}</td>
<td>\texttt{price DECIMAL[3,2]},</td>
</tr>
<tr>
<td>\texttt{price:decimal[3,2]}</td>
<td>\texttt{itemNum INTEGER},</td>
</tr>
<tr>
<td>\texttt{storeID:int16;}</td>
<td>\texttt{storeID SMALLINT},</td>
</tr>
</tbody>
</table>

The following figure shows the data flow and field and column names for this example. Columns that the operator appends to the DB2 table are shown in boldface type. Note that the order of the fields of the Orchestrate record and that of the columns of the DB2 rows are not the same but are nonetheless written...
Here is the osh command for this example:

```
$ osh "... db2_write_operator -table table_1
  -server db2Server
  -dbname my_db ..."
```

### Example 2: Writing Data to a DB2 Table in truncate Mode

In this example, the DB2 write operators write entirely new data to `table_1` whose schema and column names it retains. It does so by specifying a write mode of **truncate**. The operator deletes old data and writes new data under the old column names and data types.

As in “Example 1: Appending Data to an Existing DB2 Table” on page 34-33, the record schema of the Orchestrate data set and the row schema of the DB2 table correspond to one another, and field and column names are identical. The attributes of old and new DB2 tables are also identical.
Here is the `osh` command for this example:

```
$ osh ' ... db2_write_operator -table table_5
   -server db2server
   -dbname my_db -mode truncate ...'
```

**Example 3: Handling Unmatched Orchestrate Fields in a DB2 Write Operation**

In this example, the records of the Orchestrate data set contain a field that has no equivalent column in the rows of the DB2 table. If the operator attempts to write a data set with such an unmatched field, the corresponding step returns an error and terminate. The `-drop` option of the operator removes the unmatched field from processing and writes a warning to the message stream:

```
input field field_name dropped because it does not exist in table table_name
```

where:

- `field_name` is the name of the Orchestrate field that is dropped from the operation.
- `table_name` is the name of the DB2 table to which data is being written.
In the example, which operates in `truncate` mode with the `-drop` option chosen, the field named `code` is dropped and a message to that effect is displayed.

```
$ osh "... db2_write_operator -table table_5 -server db2server -dbname my_db -mode truncate -drop ..."
```

**Example 4: Writing to a DB2 Table Containing an Unmatched Column**

In this example, the DB2 column does not have a corresponding field in the Orchestrate data set, and the operator writes the DB2 default to the unmatched column. It does so without user intervention. The operator functions in its default write mode of `append`.

This table lists the Orchestrate records and DB2 row schemas:

<table>
<thead>
<tr>
<th>Input Orchestrate Record</th>
<th>Output DB2 Table Row</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>itemNum: int32;</code></td>
<td><code>code: SMALLINT,</code></td>
</tr>
<tr>
<td><code>price: decimal[3,2];</code></td>
<td><code>price: DECIMAL[3,2],</code></td>
</tr>
<tr>
<td><code>storeID: int16;</code></td>
<td><code>itemNum: INTEGER,</code></td>
</tr>
<tr>
<td></td>
<td><code>storeID: SMALLINT,</code></td>
</tr>
</tbody>
</table>
The next figure shows the write operator writing to the DB2 table. As in the previous example, the order of the columns is not necessarily the same as that of the fields of the data set.

Here is the `osh` code for this example:

```
$ osh "... db2_write_operator -dbname my_db
    -server db2server
    -table table_6 ..."
```
The db2upsert Operator

The db2upsert operator inserts, updates, and deletes DB2 table records with data contained in a DB2 data set. You provide the insert statements using the -insert option, and provide the delete and update SQL statements using the -update option.

An example -update delete statement is:

```
-update 'delete from tablename where A = ORCHESTRATE.A'
```

The db2upsert operator takes advantage of the DB2 CLI (Call Level Interface) system to optimize performance. It builds an array of records and then executes them with a single statement.

This operator receives a single data set as input and writes its output to a DB2 table. You can request an optional output data set that contains the records that fail to be inserted or updated.

Partitioning for db2upsert

To ensure that insert and update results are correct, the db2upsert operator automatically inserts the db2part partitioner which partitions the input data set the same way the table is partitioned using the partitioning key of the table. You can prevent this automatic insertion by explicitly inserting another partitioner, but the results may not be correct.

In order for the db2upsert to partition correctly, the partitioning key of the table must be an Orchestrate bind variable in the WHERE clause of the update statement. If this is not possible, db2upsert should be run sequentially.

Data Flow Diagram

```
input data set

db2upsert

optional reject dataset

output DB2 table
```
Properties

Table 184  db2upsert Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>by default, none; 1 when you select the -reject option</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>derived from your insert and update statements</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>Rejected update records are transferred to an output data set when you select the -reject option. A state field is added to each record. It contains a five-letter SQL code which identifies the reason the record was rejected.</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel by default, or sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>any</td>
</tr>
<tr>
<td>Collection method</td>
<td>any</td>
</tr>
<tr>
<td>Combinable operator</td>
<td>yes</td>
</tr>
</tbody>
</table>

Operator Action

Here are the main characteristics of db2upsert:

- An -update statement is required. The -insert and -reject options are mutually exclusive with using an -update option that issues a delete statement.

- Orchestrate uses the DB2 CLI (Call Level Interface) to enhance performance by executing an array of records with a single SQL statement. In most cases, the CLI system avoids repetitive statement preparation and parameter binding.

- The -insert statement is optional. If it is included, it is executed first. An insert record that does not receive a row status of SQL_PARAM_SUCCESS or SQL_PARAM_SUCCESS_WITH_INFO is then used in the execution of an update statement.

- When you specify the -reject option, any update record that receives a status of SQL_PARAM_ERROR is written to your reject data set. It’s syntax is:
  
  .reject filename

- You use the -arraySize option to specify the size of the input array. For example:
arraySize 600

The default array size is 2000.

- The `db2upsert` operator commits its input record array, and then waits until a specified number of records have been committed or a specified time interval has elapsed, whichever occurs first, before committing the next input array. You use the `-timeCommitInterval` and `-rowCommitInterval` options to control intervals between commits.

To specify between-commit intervals by time, specify a number of seconds using the `-timeCommitInterval` option. For example:

```
-timeCommitInterval 3
```

The default time-commit interval is 2 seconds.

To specify between-commit intervals by the number of records committed, use the `-rowCommitInterval` option. Specify a multiple of the input array size. The default value is 2000. For example:

- You use the `-open` and `-close` options to define optional DB2 statements to be executed before and after the processing of the insert array.
- At the end of execution, the `db2upsert` operator prints the number of inserted, updated, and rejected records for each processing node used in the step.
- Columns in the DB2 table that do not have corresponding fields in the input data set are set to their default value, if one is specified in the DB2 table. If no default value is defined for the DB2 column and it supports nulls, it is set to null. Otherwise, Orchestrate issues an error and terminates the step.
- The `db2upsert` operator converts Orchestrate values to DB2 values using the type conversions detailed on Table 181 on Page 24.
- The required DB2 privileges are listed on Table 182 on Page 26.

**Syntax and Options**

The syntax for `db2upsert` is shown below. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose it in single quotes. Exactly one occurrence of the `-update` option is required; all other options are optional.

```
db2upsert
  -update update_or_delete_statement
  [-arraySize n]
  [-close close_statement]
  [-client_instance client_instance_name]
  [-client_dbname database]
  [-user user_name]
  [-password password]
  [db_cs character_set]
```

Orchestrate 7.0 Operators Reference
**Table 185  db2upsert Operator Options**

<table>
<thead>
<tr>
<th>Options</th>
<th>Value</th>
</tr>
</thead>
</table>
| -arraySize        | -arraySize n
|                   | Optionally specify the size of the insert array. The default size is 2000 records. |
| -close            | -close close_command
|                   | Optionally specify an SQL statement to be executed after the insert array is processed. You cannot commit work using this option. The statements are executed only once on the Conductor node. |
| -client_instance  | -client_instance client_instance_name
|                   | [-client dbname database] |
|                   | -user user_name |
|                   | -password password |
|                   | Specifies the client DB2 instance name. This option is required for a remote connection. |
|                   | The -client dbname suboption specifies the client database alias name for the remote server database. If you do not specify this option, Orchestrate uses the value of the -dbname option, or the value of the APT_DBNAME environment variable, or DB2DBDFT; in that order. |
|                   | The required -user and -password suboptions specify a user name and password for connecting to DB2. |
| -db_cs            | -db_cs character_set
|                   | Specify the character set to map between DB2 char and Varchar values and Orchestrate ustring schema types and to map SQL statements for output to DB2. The default character set is UTF-8 which is compatible with your osh jobs that contain 7-bit US-ASCII data. |
|                   | For information on national language support, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:
### Table 185  db2upsert Operator Options (continued)

<table>
<thead>
<tr>
<th>Options</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-dbname</td>
<td>-dbname database_name</td>
</tr>
<tr>
<td></td>
<td>Specifies the name of the DB2 database to access. By default, the</td>
</tr>
<tr>
<td></td>
<td>operator uses the setting of the environment variable ${APT_DBNAME},</td>
</tr>
<tr>
<td></td>
<td>if defined, and DB2DBDFT otherwise. Specifying <code>-dbname</code> overrides</td>
</tr>
<tr>
<td></td>
<td>${APT_DBNAME} and DB2DBDFT. See the Orchestrate 7.0 Installation</td>
</tr>
<tr>
<td></td>
<td>and Administration Manual for information on creating the DB2</td>
</tr>
<tr>
<td></td>
<td>configuration.</td>
</tr>
<tr>
<td>-insert</td>
<td>-insert insert_statement</td>
</tr>
<tr>
<td></td>
<td>Optionally specify the insert statement to be executed.</td>
</tr>
<tr>
<td></td>
<td>This option is mutually exclusive with using an <code>-update</code> option</td>
</tr>
<tr>
<td></td>
<td>that issues a delete statement.</td>
</tr>
<tr>
<td>-open</td>
<td>-open open_command</td>
</tr>
<tr>
<td></td>
<td>Optionally specify an SQL statement to be executed before the insert</td>
</tr>
<tr>
<td></td>
<td>array is processed. The statements are executed only once on the</td>
</tr>
<tr>
<td></td>
<td>Conductor node.</td>
</tr>
<tr>
<td>-padchar</td>
<td>-padchar string</td>
</tr>
<tr>
<td></td>
<td>Specify a string to pad string and ustring fields that are less than</td>
</tr>
<tr>
<td></td>
<td>the length of the DB2 CHAR column. See “Using the -padchar Option”</td>
</tr>
<tr>
<td></td>
<td>on page 34-6 for more information on how to use this option.</td>
</tr>
<tr>
<td>-reject</td>
<td>-reject</td>
</tr>
<tr>
<td></td>
<td>If this option is set, records that fail to be updated or inserted</td>
</tr>
<tr>
<td></td>
<td>are written to a reject data set. You must designate an output data</td>
</tr>
<tr>
<td></td>
<td>set for this purpose.</td>
</tr>
<tr>
<td></td>
<td>This option is mutually exclusive with using an <code>-update</code> option</td>
</tr>
<tr>
<td></td>
<td>that issues a delete statement.</td>
</tr>
<tr>
<td>-rowCommitInterval</td>
<td>-rowCommitInterval n</td>
</tr>
<tr>
<td></td>
<td>Specifies the number of records that should be committed before</td>
</tr>
<tr>
<td></td>
<td>starting a new transaction. The specified number must be a multiple</td>
</tr>
<tr>
<td></td>
<td>of the input array size. The default is 2000.</td>
</tr>
<tr>
<td></td>
<td>You can also use the APT_RDBMS_COMMIT_ROWS environment to specify</td>
</tr>
<tr>
<td></td>
<td>the size of a commit.</td>
</tr>
<tr>
<td>-server</td>
<td>-server server_name</td>
</tr>
<tr>
<td></td>
<td>Specify the name of the DB2 instance name for the table name. By</td>
</tr>
<tr>
<td></td>
<td>default, Orchestrate uses the setting of the DB2INSTANCE environment</td>
</tr>
<tr>
<td></td>
<td>variable. Specifying <code>-server</code> overrides DB2INSTANCE.</td>
</tr>
</tbody>
</table>
## db2upsert Operator Options (continued)

<table>
<thead>
<tr>
<th>Options</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-timeCommitInterval</td>
<td>Specifies the number of seconds Orchestrate should allow between committing the input array and starting a new transaction. The default time period is 2 seconds.</td>
</tr>
</tbody>
</table>
| -update                | Use this required option to specify the update or delete statement to be executed. An example delete statement is:  
  - update 'delete from tablename where A = ORCHESTRATE.A'  
  A delete statement cannot be issued when using the -insert or -reject option. |
### The db2part Operator

You can use the `db2part` operator when you are creating a custom operator. The `db2part` operator replicates the DB2 partitioning method of a specific DB2 table.

Orchestrate uses the partitioning method specified by a parallel operator to partition a data set for processing by that operator. If the operator is supplied by Orchestrate, the operator has a predefined partitioning method in most cases. However, if you create an operator, you can choose its partitioning method. DB2 also implements partitioning. DB2 hash partitions the rows of a DB2 table based on one or more columns of each row. Each DB2 table defines the columns used as keys to hash partition it.

Using `db2part` can optimize operator execution. For example, if the input data set contains information that updates an existing DB2 table, you can partition the data set so that records are sent to the processing node that contains the corresponding DB2 rows. When you do, the input record and the DB2 row are local to the same processing node, and read and write operations entail no network activity.

### Syntax and Options

The `db2part` operator has the `osh` syntax below. The `-table` option is required; all other options are optional. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose the value in single quotes.

```plaintext
db2part
  -table table_name [-owner owner_name]
  [-client_instance client_instance_name]
  [-client_dbname database]
  [-user user_name]
  [-password password]
  [-db_cs character_set]
  [-dbname database_name]
  [-server server_name]
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-table</code></td>
<td><code>-table table_name [-owner owner_name]</code></td>
</tr>
<tr>
<td></td>
<td>Specify the table name of the table whose partitioning method you want to replicate. Optionally supply the owner of the table.</td>
</tr>
</tbody>
</table>
Specifies the client DB2 instance name. This option is required for a remote connection.

The `-client_dbname` suboption specifies the client database alias name for the remote server database. If you do not specify this option, Orchestrate uses the value of the `-dbname` option, or the value of the `APT_DBNAME` environment variable, or `DB2DBDFT`; in that order.

The required `-user` and `-password` suboptions specify a user name and password for connecting to DB2.

Specify the character set to map between DB2 `char` and `Varchar` values and Orchestrate `ustring` schema types and to map SQL statements for output to DB2. The default character set is `UTF-8` which is compatible with your `osh` jobs that contain 7-bit `US-ASCII` data.

For information on national language support, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:

http://oss.software.ibm.com/icu/charset

Specify the name of the DB2 database to access. By default, the partitioner uses the setting of the environment variable `APT_DBNAME`, if defined, and `DB2DBDFT` otherwise. Specifying `-dbname` overrides `APT_DBNAME` and `DB2DBDFT`.

Type the name of the DB2 instance name for the table name. By default, Orchestrate uses the setting of the `DB2INSTANCE` environment variable. Specifying `-server` overrides `DB2INSTANCE`.

Table 186  dbpart Operator Options (continued)
Example

Following is an example using the **db2part** operator, which partitions the input data set, `inDS.ds`, according to the partitioning method of `table_1` and then inputs the resulting records to a custom operator:

![Diagram showing the flow of data from `inDS.ds` through `db2part` and `custom_operator`]

Here is the **osh** code for this example:

```
$ "osh db2part -table table_1 < inDS.ds | custom_operator ..."
```

See the DB2 Parallel Edition for AIX, Administration Guide and Reference for more information on DB2 partitioning.
The db2lookup Operator

With the db2lookup operator, you can perform a join between one or more DB2 tables and an Orchestrate data set. The resulting output data is an Orchestrate data set containing Orchestrate and DB2 data.

You perform this join by specifying either an SQL SELECT statement, or by specifying one or more DB2 tables and one or more key fields on which to do the lookup.

This operator is particularly useful for sparse lookups, that is, where the Orchestrate data set you are matching is much smaller than the DB2 table. If you expect to match 90% of your data, using the db2read and lookup operators is probably more efficient.

Because db2lookup can do lookups against more than one DB2 table, it is useful for joining multiple DB2 tables in one query.

The -query option command corresponds to an SQL statement of this form:

```
select a,b,c from data.testtbl
where
  Orchestrate.b = data.testtbl.c
and
  data.testtbl.name = "Smith"
```

The operator replaces each orchestrate.fieldname with a field value, submits the statement containing the value to DB2, and outputs a combination of DB2 and Orchestrate data.

Alternatively, you can use the -key/-table options interface to specify one or more key fields and one or more DB2 tables. The following osh options specify two keys and a single table:

```
-key a -key b -table data.testtbl
```

you get the same result as you would by specifying:

```
select * from data.testtbl
where
  Orchestrate.a = data.testtbl.a
and
  Orchestrate.b = data.testtbl.b
```

The resulting Orchestrate output data set includes the Orchestrate records and the corresponding rows from each referenced DB2 table. When an DB2 table has a column name that is the same as an Orchestrate data set field name, the DB2 column is renamed using the following syntax:

```
APT_integer_fieldname
```

An example is `APT_0_lname`. The integer component is incremented when duplicate names are encountered in additional tables.
Chapter 34  The DB2 Interface Library

The db2lookup Operator

Data Flow Diagram

Properties

Table 187  db2lookup Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1; 2 if you include the <code>-ifNotFound reject</code> option</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>determined by the query</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>determined by the SQL query</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>transfers all fields from input to output</td>
</tr>
<tr>
<td>Execution mode</td>
<td>sequential or parallel (default)</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>clear</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

Syntax and Options

The syntax for the db2lookup operator is given below. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose it in single quotes. You must supply either the -table option or the -query option.

Note

If the DB2 table is not indexed on the lookup keys, this operator’s performance is likely to be poor.
db2lookup
   -table table_name [-selectlist selectlist] [-filter filter]
   -key field [-key field ...]
   \[ -table table_name [-selectlist selectlist] [-filter filter] ... \]
   -key field [-key field ...] ...]
   | [-query sql_query]
   [-close close_command]
   [-client_instance client_instance_name
   [-client_dbname database]
   -user user_name
   -password password]
   [-db_cs character_set]
   [-dbname dbname]
   [-ifNotFound fail | drop | reject | continue]
   [-open open_command]
   [-padchar char]
   [-server remote_server_name]

You must specify either the -query option or one or more -table options with one
or more -key fields.

Table 188  db2lookup Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-table</td>
<td>Specify a table and key field(s) from which a SELECT statement is created. Specify either the -table option or the -query option.</td>
</tr>
<tr>
<td>-query</td>
<td>Specify an SQL query, enclosed in single quotes, to read a table. The query specifies the table and the processing that you want to perform on the table as it is read into Orchestrate. This statement can contain joins, views, database links, synonyms, and so on.</td>
</tr>
<tr>
<td>-close</td>
<td>Optionally apply a closing command to execute on a database after Orchestrate execution.</td>
</tr>
</tbody>
</table>
The db2lookup Operator

Specifies the client DB2 instance name. This option is required for a remote connection.

The -client_dbname suboption specifies the client database alias name for the remote server database. If you do not specify this option, Orchestrate uses the value of the -dbname option, or the value of the APT_DBNAME environment variable, or DB2DBDFT; in that order.

The required -user and -password suboptions specify a user name and password for connecting to DB2.

Specify the character set to map between DB2 char and Varchar values and Orchestrate ustring schema types and to map SQL statements for output to DB2. The default character set is UTF-8 which is compatible with your osh jobs that contain 7-bit US-ASCII data.

For information on national language support, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:

http://oss.software.ibm.com/icu/charset

Optionally specify a database name.

Determines what action to take in case of a lookup failure. The default is fail. If you specify reject, you must designate an additional output data set for the rejected records.

Optionally apply an opening command to execute on a database prior to Orchestrate execution.

Specify a string to pad string and ustring fields that are less than the length of the DB2 CHAR column. For more information see "Using the -padchar Option" on page 34-6.
Example

Suppose you want to connect to the APT81 server as user user101, with the password test. You want to perform a lookup between an Orchestrate data set and a table called target, on the key fields lname, fname, and DOB. You can configure `db2lookup` in either of two ways to accomplish this.

Here is the `osh` command using the `-table` and `-key` options:

```
$ osh " db2lookup -server APT81
     -table target -key lname -key fname -key DOB
     < data1.ds > data2.ds "
```

Here is the equivalent `osh` command using the `-query` option:

```
$ osh " db2lookup -server APT81
     -query 'select * from target
     where lname = Orchestrate.lname
     and fname = Orchestrate.fname
     and DOB = Orchestrate.DOB'
     < data1.ds > data2.ds "
```

Orchestrate prints the lname, fname, and DOB column names and values from the Orchestrate input dataset and also the lname, fname, and DOB column names and values from the DB2 table.

If a column name in the DB2 table has the same name as an Orchestrate output data set schema fieldname, the printed output shows the column in the DB2 table renamed using this format:

```
APT_integer_fieldname
```

For example, lname may be renamed to APT_0_lname.
Considerations for Reading and Writing DB2 Tables

Data Translation Anomalies
Translation anomalies exist under the following write and read operations:

Write Operations
An Orchestrate data set is written to a DB2 table in create mode. The table's contents are then read out and reconverted to the Orchestrate format. The final Orchestrate field types differ from the original, as shown in the next table:

Table 189  DB2 Table Write Operations Translation Anomalies

<table>
<thead>
<tr>
<th>Original Orchestrate Data Type</th>
<th>Converted to DB2 Type</th>
<th>Read Back and Reconverted</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixed-length string field, 255 &lt;= fixed length &lt;= 4000 bytes</td>
<td>VARCHAR(n), where n is the length of the string field</td>
<td>string[max=n] and ustring[max=n]</td>
</tr>
<tr>
<td>int8</td>
<td>SMALLINT</td>
<td>int16</td>
</tr>
<tr>
<td>sfloat</td>
<td>FLOAT</td>
<td>dfloat</td>
</tr>
</tbody>
</table>

Invoke the modify operator to convert the data type of the final Orchestrate field to its original data type. See Chapter 13, “The modify Operator”.

Read Operations
A DB2 table is converted to an Orchestrate data set and reconverted to DB2 format. The final DB2 column type differs from the original, as shown in the next table:

Table 190  DB2 Table Read Operations Translation Anomaly

<table>
<thead>
<tr>
<th>Original DB2 Data Type</th>
<th>Converted to Orchestrate Type</th>
<th>Read Back and Reconverted</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE</td>
<td>dfloat</td>
<td>FLOAT (synonymous with DOUBLE)</td>
</tr>
</tbody>
</table>
Using a Node Map

Use an Orchestrate node map to run an operator other than the DB2 read and write operators on only the processing nodes containing partitions of a DB2 table. A node map specifies the processing nodes on which an operator runs. See the chapter on using constraints in the Orchestrate 7.0 User Guide for more information on node maps. This operation is useful when you read a DB2 table, process its contents by means of another operator, and write the results back to DB2.

To constrain an operator to run only on nodes mapped in a node map:

Add this code to the command that runs the operator processing the data that has been read from DB2:

```
[nodemap ('db2nodes -table table_name -dbname db -server s_name')]
```

where:

- `-table table_name` specifies the DB2 table that determines the processing nodes on which the operator runs. Orchestrate runs the operator on all processing nodes containing a partition of `table_name`.

- `-dbname db` optionally specifies the name of the DB2 database to access. By default, Orchestrate uses the setting of `APT_DBNAME`, if it is defined; otherwise, Orchestrate uses the setting of `DB2DBDFT`. Specifying `-dbname` overrides both `APT_DBNAME` and `DB2DBDFT`.

- `-server s_name` optionally specifies the DB2 instance name for `table_name`. By default, Orchestrate uses the setting of the `DB2INSTANCE` environment variable. Specifying `-server` overrides `DB2INSTANCE`.

The next example reads a table from DB2, then uses `db2nodes` to constrain the `statistics` operator to run on the nodes containing a partition of the table:

```
$ osh "db2read -table table_1 |
  statistics -results /stats/results -fields itemNum, price
  [nodemap ('db2nodes -table table_1')] > outDS.ds "
```

Here is the `osh` syntax for this example:
In this case, all data access on each node is local, that is, Orchestrate does not perform any network activity to process the table.
The Informix Interface Library

Describes the INFORMIX interface operators which translate INFORMIX result sets to Orchestrate data sets and write Orchestrate data sets to INFORMIX tables.

Overview of the INFORMIX Interface Operators  35 2
Configuring the INFORMIX User Environment  35 3
Read Operations  35 3
  Data Flow Diagram  35 4
  Read Operator Action  35 4
  Execution Mode  35 4
  Column Name Conversion  35 5
  Data Type Conversion  35 5
  Example 1: Reading All Data from an INFORMIX Table  35 6
Write Operations  35 7
  Data Flow Diagram  35 8
  Execution Mode  35 8
  Column Name Conversion  35 8
  Data Type Conversion  35 9
  Write Modes  35 10
  Writing Fields  35 11
  Limitations  35 11
  Example 2: Appending Data to an Existing INFORMIX Table  35 12
  Example 3: Writing Data to an INFORMIX Table in Truncate Mode  35 13
  Example 4: Handling Unmatched Orchestrate Fields in an INFORMIX Write Operation  35 14
  Example 5: Writing to an INFORMIX Table with an Unmatched Column  35 15
The hplread operator  35 16
  Special Operator Features  35 16
  Data Flow Diagram  35 16
  Properties  35 17
  Syntax and Options  35 17
Overview of the INFORMIX Interface Operators

This chapter of the Orchestrate 7.0 Operators Reference describes how to use the INFORMIX interface operators. To understand the information in this chapter you should be familiar with the operator concepts defined in the Orchestrate 7.0 User Guide.

The INFORMIX interface library contains six operators, grouped in three pairs of read and write operators which share data with specific versions of INFORMIX.
The read operators set up a connection to an INFORMIX database, send it a query, and translate the resulting two-dimensional array (the INFORMIX result set) to an Orchestrate data set. The write operators are a high-speed method for writing to an INFORMIX table. They set up a connection to an INFORMIX database to write data from an Orchestrate data set.

- **hplread** and **hplwrite** share data with version 7.x of INFORMIX. **hplread** is described on page 16, and **hplwrite** is described on page 19.
- **infxread** and **infxwrite** share data with versions 7.x and 8.x of INFORMIX. **infxread** is described on page 23, and **infxwrite** is described on page 26.
- **xpsread** and **xpswrite** share data with version 8.x of INFORMIX. **xpsread** is described on page 29, and **xpswrite** is described on page 31.

### Configuring the INFORMIX User Environment

This section assumes that Orchestrate users have been configured to access INFORMIX using the INFORMIX configuration process outlined in Chapter 8: RDBMS Configuration in the Orchestrate 7.0 Installation and Administration Manual.

User privileges and Orchestrate settings must be correct to ensure INFORMIX access. Users who perform read and write operations must have valid accounts and appropriate privileges on the databases to which they connect. Orchestrate connects to INFORMIX using your system user name. This name must have read and write privileges to the INFORMIX tables you access. In addition, the following activities require Resource privileges:

- In **xpsread**, invoking the **-partition** option
- In write operations, writing in the **create** and **replace** modes

### Read Operations

An Orchestrate operator that reads data from INFORMIX sets up a connection to an INFORMIX server, sends a query, and converts the resulting two-dimensional array (the INFORMIX result set) to an Orchestrate data set.
Read Operator Action

Here are the chief characteristics of the Orchestrate operators that perform read operations of INFORMIX result sets:

- They specify either an INFORMIX table to read (see “Example 1: Reading All Data from an INFORMIX Table” on page 35-6) or an SQL query to carry out. A table specification is translated to an SQL `select * from table` statement.
- They translate the query’s result set, which is a two-dimensional array, row by row, to an Orchestrate data set, as described in “Column Name Conversion” on page 35-5.
- The translation includes the conversion of INFORMIX data types to Orchestrate data types, as listed in “Data Type Conversion” on page 35-5.

Operator output is an Orchestrate data set that you can use as input to a subsequent Orchestrate operator.

Read operations must specify the database on which the operation is performed and must also indicate either an SQL query of the database or a table to read. Read operations can, optionally, specify INFORMIX SQL statements to be parsed and executed on all processing nodes before the read operation is prepared and executed, or after the selection or query is completed.

The discussion of each operator in subsequent sections includes an explanation of properties and options.

Execution Mode

You can control the processing nodes, corresponding to INFORMIX servers, on which the read operators are run. You do so using the `-part` suboption of the `-table` or `-query` option.
Column Name Conversion

An INFORMIX result set is defined by a collection of rows and columns. Here is how an INFORMIX query result set is converted to an Orchestrate data set:

- The rows of an INFORMIX result set correspond to the records of an Orchestrate data set.
- The columns of an INFORMIX row correspond to the fields of an Orchestrate record, and the name and data type of an INFORMIX column correspond to the name and data type of an Orchestrate field.
- Both INFORMIX columns and Orchestrate fields support nulls, and a null contained in an INFORMIX column is stored as a null in the corresponding Orchestrate field.

Data Type Conversion

The INFORMIX read operators convert INFORMIX data types to Orchestrate data types, as shown in the following table:

Table 191  INFORMIX Interface Read Operators Data-Type Conversion

<table>
<thead>
<tr>
<th>INFORMIX Data Type</th>
<th>Orchestrate Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)</td>
<td>string[n]</td>
</tr>
<tr>
<td>CHARACTER VARYING(n,r)</td>
<td>string[max = n]</td>
</tr>
<tr>
<td>DATE</td>
<td>date</td>
</tr>
<tr>
<td>DATETIME</td>
<td>time or timestamp with corresponding fractional precision for time If the DATETIME starts with a year component, the result is a timestamp field. If the DATETIME starts with an hour, the result is a time field.</td>
</tr>
<tr>
<td>DECIMAL[p,s]</td>
<td>decimal(p,s) where p is precision and s is the scale The maximum precision is 32. A decimal with floating scale is converted to a dfloat.</td>
</tr>
<tr>
<td>DOUBLE-PRECISION</td>
<td>dfloat</td>
</tr>
<tr>
<td>FLOAT</td>
<td>dfloat</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int32</td>
</tr>
<tr>
<td>MONEY</td>
<td>decimal</td>
</tr>
<tr>
<td>NCHAR(n,r)</td>
<td>string[n]</td>
</tr>
</tbody>
</table>
Table 191  **INFORMIX Interface Read Operators Data-Type Conversion**

<table>
<thead>
<tr>
<th>INFORMIX Data Type</th>
<th>Orchestrate Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVARCHAR(n,r)</td>
<td>string[max =n]</td>
</tr>
<tr>
<td>REAL</td>
<td>sfloat</td>
</tr>
<tr>
<td>SERIAL</td>
<td>int32</td>
</tr>
<tr>
<td>SMALLFLOAT</td>
<td>sfloat</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>int16</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>string[max =n]</td>
</tr>
</tbody>
</table>

**Example 1: Reading All Data from an INFORMIX Table**

The following figure shows an example of a read operation of an INFORMIX result set. The operator specifies the `table` option, with no further qualification, so that a single table is read in its entirety from INFORMIX. The table is read into a result set and the operator translates the result set into an Orchestrate data set.

For the purposes of the example, the table contains three rows. All rows are read.
Here are the schemas of both the input INFORMIX table rows and output Orchestrate records:

### Table 192  Schemas for Orchestrate Read of INFORMIX Result Set

<table>
<thead>
<tr>
<th>Input INFORMIX Table Row</th>
<th>Output Orchestrate Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemNum INTEGER not null,</td>
<td>itemNum:int32;</td>
</tr>
<tr>
<td>price DECIMAL(3,2) not null,</td>
<td>price:decimal[3,2];</td>
</tr>
<tr>
<td>storeID SMALLINT not null</td>
<td>storeID:int16;</td>
</tr>
</tbody>
</table>

In the example:

- The Orchestrate operator converts the INFORMIX result set, which is a two-dimensional array, to an Orchestrate data set.
- The schema of the INFORMIX result set corresponds to the record schema of the Orchestrate data set, as in the table shown above.
- Each row of the INFORMIX result set corresponds to a record of the Orchestrate data set; each column of each row corresponds to a field of the Orchestrate record.
- The Orchestrate field names are the same as the column names of the INFORMIX table.

Here is the `osh` command for this example:

```bash
$ osh '...read_operator -dbname my_db -server IXServer -table table_2 ...'
```

## Write Operations

An Orchestrate operator that writes to INFORMIX sets up a connection to an INFORMIX database and writes data from an Orchestrate data set to a database table or view.
Here are the chief characteristics of the Orchestrate operators that write Orchestrate data sets to INFORMIX tables:

- They specify the name of the database to write to.
- They translate the Orchestrate data set record-by-record to INFORMIX table rows, as discussed in “Column Name Conversion” on page 35-8.
- The translation includes the conversion of INFORMIX data types to Orchestrate data types, as listed in “Data Type Conversion” on page 35-9.
- They append records to an existing table, unless another mode of writing has been set. “Write Modes” on page 35-10 discusses these modes.

Operators optionally specify such information as the mode of the write operation and INFORMIX commands to be parsed and executed on all processing nodes before the write operation is executed or after it has completed.

The discussion of each operator in subsequent sections includes an explanation of properties and options.

**Execution Mode**

The `infxwrite` operator executes sequentially, writing to one server. The coserver in turn partitions the data if necessary. The `xpswrite` and `hplwrite` operators execute in parallel.

**Column Name Conversion**

Orchestrate data sets are converted to INFORMIX tables when:

- An Orchestrate data set corresponds to an INFORMIX table.
- The records of the Orchestrate data set correspond to the rows of the table.
- The fields of the Orchestrate record correspond to the columns of each row.
• The name and data type of an Orchestrate field correspond to those of the corresponding INFORMIX column.

• Both Orchestrate fields and INFORMIX columns support nulls, and a null in an Orchestrate field is stored as a null in the corresponding INFORMIX column.

Data Type Conversion

The INFORMIX interface write operators convert Orchestrate data types to INFORMIX data types. The following table shows how they are converted:

Table 193  INFORMIX Interface Write Operators Data Conversion

<table>
<thead>
<tr>
<th>Orchestrate Data Type</th>
<th>INFORMIX Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>DATE</td>
</tr>
<tr>
<td>decimal[p, s]</td>
<td>DECIMAL[p, s]</td>
</tr>
<tr>
<td>dfloat</td>
<td>FLOAT</td>
</tr>
<tr>
<td>int8</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>int16</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>int32</td>
<td>INTEGER</td>
</tr>
<tr>
<td>raw[n] (fixed length)</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>raw[max] (variable length)</td>
<td>VARCHAR(n) (default is 32 bytes; max &lt;= 255)</td>
</tr>
<tr>
<td>sfloat</td>
<td>FLOAT</td>
</tr>
<tr>
<td>string[n] (fixed length)</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>string[max] (variable length)</td>
<td>VARCHAR(n) (default is 32 bytes; max &lt;= 255)</td>
</tr>
<tr>
<td>subrec</td>
<td>Unsupported</td>
</tr>
</tbody>
</table>
Write Modes

The write mode of the operator determines how the records of the data set are inserted into the destination table. The write mode can have one of the following values:

- **append**: The table must exist and the record schema of the data set must be compatible with the table. A write operator appends new rows to the table, and the final order of rows in the table is determined by INFORMIX. This is the default mode. See “Example 2: Appending Data to an Existing INFORMIX Table” on page 35-12.

- **create**: The operator creates a new table. If a table exists with the same name as the one you want to create, the step that contains the operator terminates with an error. You must specify either create mode or replace mode if the table does not exist.

- **replace**: The operator drops the existing table and creates a new one in its place. If a table exists with the same name as the one you want to create, it is overwritten.

- **truncate**: The operator retains the table attributes but discards existing records and appends new ones. See “Example 4: Handling Unmatched Orchestrate Fields in an INFORMIX Write Operation” on page 35-14.

Each mode requires specific user privileges. These are documented in the next sections as part of the discussion of operator options. You can override the default mode, append, with the -mode option.
Writing Fields

The names and data types of fields in the Orchestrate data set must match those of the columns of the INFORMIX table. However, the fields do not have to appear in the same order. See “Example 2: Appending Data to an Existing INFORMIX Table” on page 35-12.

The following rules determine which fields in an Orchestrate data set are written to an INFORMIX table:

• If the Orchestrate data set contains fields for which there are no matching columns in the INFORMIX table, the step containing the operator returns an error and terminates. However, you can remove an unmatched field from processing either by specifying the -drop option or by using a modify operator to drop the extra field or fields. For an example of using the -drop option, see “Example 4: Handling Unmatched Orchestrate Fields in an INFORMIX Write Operation” on page 35-14.

• If the INFORMIX table contains a column that does not have a corresponding field in the Orchestrate data set, the write operator writes a null if the column is nullable. Otherwise, Orchestrate issues an error and terminates the step. See “Example 5: Writing to an INFORMIX Table with an Unmatched Column” on page 35-15.

Note The default length of Orchestrate variable-length strings is 32 bytes, so all records of the INFORMIX table will have 32 bytes allocated for each variable-length string field in the input data set. If a variable-length field is longer than 32 bytes, the write operator issues an error, but you can use the -stringlength option to modify the default length, up to a maximum of 255 bytes.

Limitations

Write operations have the following limitations:

• While the names of Orchestrate fields can be of any length, the names of INFORMIX columns cannot exceed 18 characters. If you write Orchestrate fields that exceed this limit, the operator issues an error and the step terminates.

• The Orchestrate data set cannot contain fields of these types: raw, tagged aggregate, subrecord, or unsigned integer (of any size).

If an INFORMIX write operator tries to write a data set whose fields contain a data type listed above, the write operation terminates and the step containing the operator returns an error. You can convert unsupported data types by means of the modify operator. See Chapter 13, “The modify Operator” for information.
Example 2: Appending Data to an Existing INFORMIX Table

In this example, an INFORMIX write operator appends the contents of an Orchestrate data set to an existing INFORMIX table. This is the default action of INFORMIX write operators.

The record schema of the Orchestrate data set and the row schema of the INFORMIX table correspond to one another, and field and column names are identical. Here are the input Orchestrate record schema and output INFORMIX row schema:

Table 194  Schemas for Orchestrate Append to INFORMIX Table

<table>
<thead>
<tr>
<th>Input Orchestrate Record</th>
<th>Output INFORMIX Table Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemNum:int32; price:decimal[3,2]; storeID:int16;</td>
<td>price DECIMAL, itemNum INTEGER, store SMALLINT</td>
</tr>
</tbody>
</table>

The following diagram shows the operator action. Columns that the operator appends to the INFORMIX table are shown in boldface type. Note that the order of fields of the Orchestrate record and columns of the INFORMIX rows are not the same but are nonetheless written successfully.
Here is the `osh` command for this example:

```
$ osh '... write_operator -table table_1
    -server IXServer
    -dbname my_db ... '
```

**Example 3: Writing Data to an INFORMIX Table in Truncate Mode**

In this example, the INFORMIX write operator writes entirely new data to a table (\texttt{table}_5) whose schema and column names it retains. It does so by specifying a write mode of \texttt{truncate}. The operator deletes old data and writes new data under the old column names and data types.

As in “Example 2: Appending Data to an Existing INFORMIX Table” on page 35-12”, the record schema of the Orchestrate data set and the row schema of the INFORMIX table correspond to one another, and field and column names are identical. The attributes of old and new INFORMIX tables are likewise identical. Among table attributes are indexes and fragmentation.

![Diagram showing flow of data from Orchestrate Data Set to INFORMIX table with truncate mode]

Here is the `osh` command for this example.

```
$ osh '... write_operator -table table_5
    -server IXServer
    -dbname my_db -mode truncate ... '
```
Example 4: Handling Unmatched Orchestrate Fields in an INFORMIX Write Operation

In this example, the records of the Orchestrate data set contain a field that has no equivalent column in the rows of the INFORMIX table. If the operator attempted to write a data set with such an unmatched field, the corresponding step would return an error and terminate. The `-drop` option of the operator removes the unmatched field from processing and writes a warning to that effect to the message stream, as follows:

```
Input field field_name dropped because it does not exist in table table_name
```

where:

- `field_name` is the name of the Orchestrate field that is dropped from the operation
- `table_name` is the name of the INFORMIX table to which data is being written

In the example, which operates in `truncate` write mode, the field named `code` is dropped and a message to that effect is displayed.

```
"Input field code dropped because it does not exist in table table_5"
```
Here is the `osh` command for this example:

```bash
$ osh "... write_operator -table table_5
   -server IXserver
   -dbname my_db -mode truncate ...
"
```

### Example 5: Writing to an INFORMIX Table with an Unmatched Column

In this example, the INFORMIX column does not have a corresponding field in the Orchestrate data set, and the operator writes the INFORMIX default to the unmatched column. It does so without user intervention. The operator functions in its default write mode of `append`.

This table lists the Orchestrate and INFORMIX schemas:

**Table 195  Schemas for Orchestrate Write to Table with Unmatched Column**

<table>
<thead>
<tr>
<th>Input Orchestrate Record</th>
<th>Output INFORMIX Table Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemNum: int32;</td>
<td>code: SMALLINT,</td>
</tr>
<tr>
<td>price: decimal[3,2];</td>
<td>price: DECIMAL,</td>
</tr>
<tr>
<td>storeID: int16;</td>
<td>itemNum: INTEGER,</td>
</tr>
<tr>
<td></td>
<td>storeID: SMALLINT</td>
</tr>
</tbody>
</table>

The Orchestrate data set and INFORMIX table are shown in the diagram below.
Here is the `osh` command for this example:

```bash
$ osh '...write_operator -dbname my_db
  -server IXserver
  -table table_6 ...'
```

### The `hplread` operator

The `hplread` operator sets up a connection to an INFORMIX database, sends it a query, and translates the resulting two-dimensional array representing the INFORMIX result set to an Orchestrate data set.

**Special Operator Features**

- For small numbers of records, `infxread` (“The `infxread` Operator” on page 35-23) may perform better than `hplread`. In any case, there is no performance benefit from using `hplread` for very small data sets.

- For the High Performance Loader to run (and thus for `hplread` to run), the INFORMIX `onpload` database must exist and be set up. This can be assured by running the INFORMIX `ipload` utility once and then exiting it. An appropriate warning appears if the database is not set up properly.

- The High Performance Loader uses more shared memory, and therefore more semaphores, than INFORMIX does in general. If the HPL is unable to allocate enough shared memory or semaphores, `hplread` may not work. For more information about shared memory limits, contact your system administrator.

### Data Flow Diagram

```
INFORMIX query result

hplread

output data set
```
Properties

Table 196  **hplread Operator Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>0</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>none</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>determined by the SQL query</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>none</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>not applicable</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>clear</td>
</tr>
<tr>
<td>Composite operator</td>
<td>yes</td>
</tr>
</tbody>
</table>

Syntax and Options

The **osh** syntax of the **hplread** operator follows. You must specify either the **-query** or the **-table** option. You must also specify the **-dbname** option.

Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose it in single quotes.

```
hplread
   -table table_name [-filter filter][-selectlist list]
| -query sql_query
| -dbname database_name
| [-close close_command]
| [-dboptions smalldata]
| [-open open_command]
| [-part part]
| [-server server_name]
```
### hplread Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-close</td>
<td>-close close_command</td>
</tr>
<tr>
<td></td>
<td>Optionally specify an INFORMIX SQL statement to be parsed and executed on all processing nodes after the table selection or query is completed.</td>
</tr>
<tr>
<td>-dbname</td>
<td>-dbname database_name</td>
</tr>
<tr>
<td></td>
<td>Specify the name of the INFORMIX database to read from.</td>
</tr>
<tr>
<td>-dboptions</td>
<td>-dboptions smalldata</td>
</tr>
<tr>
<td></td>
<td>Set this option if the number of records is less than about 10 * number of processing nodes.</td>
</tr>
<tr>
<td>-open</td>
<td>-open open_command</td>
</tr>
<tr>
<td></td>
<td>Optionally specify an INFORMIX SQL statement to be parsed and executed by the database on all processing nodes before the read query is prepared and executed.</td>
</tr>
<tr>
<td>-query</td>
<td>-query sql_query</td>
</tr>
<tr>
<td></td>
<td>Specify a valid SQL query that will be submitted to INFORMIX to read the data.</td>
</tr>
<tr>
<td></td>
<td>The sql_query specifies the processing that INFORMIX performs as data is read into Orchestrate. The query can contain joins, views, synonyms, and so on.</td>
</tr>
<tr>
<td>-part</td>
<td>-part table_name</td>
</tr>
<tr>
<td></td>
<td>If the table is fragmented, optionally specify this option. This improves performance by creating one instance of the operator per table fragment. If the table is fragmented across nodes, this option creates one instance of the operator per fragment per node. If the table is fragmented and you do not specify -part, the operator nonetheless functions successfully, if more slowly. You must have Resource privilege to invoke this option.</td>
</tr>
<tr>
<td></td>
<td>When used with the -table option, the table_name given must be the same name as that specified for the -table option.</td>
</tr>
<tr>
<td>-server</td>
<td>-server server_name</td>
</tr>
<tr>
<td></td>
<td>Set the name of the INFORMIX server you want to connect to. If no server is specified, the value of the INFORMIXSERVER environment variable is used.</td>
</tr>
</tbody>
</table>
The hplwrite Operator

The hplwrite operator is a high-speed way to write an INFORMIX table; it sets up a connection to an INFORMIX database to write data to it from an Orchestrate data set. It has essentially the same outward behavior as infxwrite, but works much more quickly, by making use of the INFORMIX High Performance Loader. In addition, there is an -express option to hplwrite that significantly improves performance. If the destination INFORMIX table does not exist, it is created by the operator. This operator runs with INFORMIX Versions 7.2 and 7.3.

Special Operator Features

1. If your step is processing a few records, you may use infxwrite rather than hplwrite. In any case, there is no performance benefit from using hplwrite for very small data sets. If your data set size is less than about 10 times the number of processing nodes, you must set -dboptions smalldata on the hplwrite operator.

2. For the High Performance Loader and hplwrite to run, the INFORMIX onload database must exist and be set up. You do this by running the INFORMIX onload utility once and exiting it. An appropriate warning appears if the database is not set up properly.
The High Performance Loader uses more shared memory, and therefore more semaphores, than INFORMIX does in general. If the HPL is unable to allocate enough shared memory or semaphores, `hplwrite` may not work. For more information about shared memory limits, contact your system administrator.

**Data Flow Diagram**

![Data Flow Diagram]

**Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>0</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>derived from the input data set</td>
</tr>
<tr>
<td>Execution mode</td>
<td>sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>same</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

**Syntax and Options**

The `osh` syntax of the `hplwrite` operator follows. You must specify exactly one occurrence of the `-table` option; all other options are optional. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose it in single quotes.

```
hplwrite
  -table table_name [selectlist list]
  [-close close_command]
  [-dbname database_name]
```
[-dboptions smalldata]  
[-drop]  
[-express]  
[-mode append | create | replace | truncate]  
[-open open_command]  
[-server server_name]  
[-stringlength length]  

Table 199  hplwrite Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-close</td>
<td>-close close_command</td>
</tr>
<tr>
<td></td>
<td>Specify an INFORMIX SQL statement to be parsed and executed by INFORMIX on all processing nodes after the table has been populated.</td>
</tr>
<tr>
<td>-dbname</td>
<td>-dbname database_name</td>
</tr>
<tr>
<td></td>
<td>Specify the name of the INFORMIX database containing the table specified by -table.</td>
</tr>
<tr>
<td>-dboptions</td>
<td>-dboptions smalldata</td>
</tr>
<tr>
<td></td>
<td>Set this option if the number of records is less than 10 times the number of processing nodes.</td>
</tr>
<tr>
<td>-drop</td>
<td>-drop</td>
</tr>
<tr>
<td></td>
<td>Use this option to cause the operator to drop, with a warning, all input fields that do not correspond to columns of an existing table. If you do not specify -drop, an unmatched field generates an error and the associated step terminates.</td>
</tr>
<tr>
<td>-express</td>
<td>-express</td>
</tr>
<tr>
<td></td>
<td>This option significantly improves the performance of the operator. However, we do not recommend using it unless speed is of utmost importance and you have previously written similar data, since there are several potential issues with its use:</td>
</tr>
<tr>
<td></td>
<td>1 Either transaction logging must be turned off, or, after the hplwrite operation has finished, a level-0 backup must be performed in order to make the database writable by other sources.</td>
</tr>
<tr>
<td></td>
<td>2 Normal constraints on writing to an INFORMIX table are turned off. This means that invalid data can be written to the table, and existing data in the table may be corrupted.</td>
</tr>
<tr>
<td></td>
<td>3 It does not invoke triggers on the data being written.</td>
</tr>
<tr>
<td></td>
<td>4 It cannot write any row larger than the system page size.</td>
</tr>
</tbody>
</table>
Table 199  hplwrite Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-mode</td>
<td>- mode append</td>
</tr>
<tr>
<td>hplwrite</td>
<td>appends new records to the table. The database user who writes in</td>
</tr>
<tr>
<td></td>
<td>this mode must have Resource privileges. This is the default mode.</td>
</tr>
<tr>
<td>-mode</td>
<td>- mode create</td>
</tr>
<tr>
<td>hplwrite</td>
<td>creates a new table. The database user who writes in this mode must</td>
</tr>
<tr>
<td></td>
<td>have Resource privileges. Orchestrate returns an error if the table</td>
</tr>
<tr>
<td></td>
<td>already exists.</td>
</tr>
<tr>
<td>-mode</td>
<td>- mode replace</td>
</tr>
<tr>
<td>hplwrite</td>
<td>deletes the existing table and creates a new one in its place. The</td>
</tr>
<tr>
<td></td>
<td>database user who writes in this mode must have Resource privileges.</td>
</tr>
<tr>
<td>-mode</td>
<td>- mode truncate</td>
</tr>
<tr>
<td>hplwrite</td>
<td>retains the table attributes but discards existing records and</td>
</tr>
<tr>
<td></td>
<td>appends new ones. The operator runs more slowly in this mode if the</td>
</tr>
<tr>
<td></td>
<td>user does not have Resource privileges.</td>
</tr>
<tr>
<td>-open</td>
<td>- open open_command</td>
</tr>
<tr>
<td></td>
<td>Specify an INFORMIX SQL statement to be parsed and executed by</td>
</tr>
<tr>
<td></td>
<td>INFORMIX on all processing nodes before opening the table.</td>
</tr>
<tr>
<td>-server</td>
<td>- server server_name</td>
</tr>
<tr>
<td></td>
<td>Specify the name of the INFORMIX server you want to connect to.</td>
</tr>
<tr>
<td>-stringlength</td>
<td>- stringlength length</td>
</tr>
<tr>
<td></td>
<td>Set the default length of variable-length raw or string fields. If</td>
</tr>
<tr>
<td></td>
<td>you do not specify a length, the default is 32 bytes. You can specify</td>
</tr>
<tr>
<td></td>
<td>a length up to 255 bytes.</td>
</tr>
<tr>
<td>-table</td>
<td>- table table_name[.selectlist selectlist]</td>
</tr>
<tr>
<td></td>
<td>Specifies the name of the INFORMIX table to write to. See the mode</td>
</tr>
<tr>
<td></td>
<td>options for constraints on the existence of the table and the user</td>
</tr>
<tr>
<td></td>
<td>privileges required.</td>
</tr>
<tr>
<td></td>
<td>-selectlist specifies an SQL list that determines which fields are</td>
</tr>
<tr>
<td></td>
<td>written. If you do not supply the list, hplwrite writes all fields</td>
</tr>
<tr>
<td></td>
<td>to the table.</td>
</tr>
</tbody>
</table>

Examples

Refer to the following sections for examples of INFORMIX write operator use:

- “Example 2: Appending Data to an Existing INFORMIX Table” on page 35-12
- “Example 3: Writing Data to an INFORMIX Table in Truncate Mode” on page 35-13
• “Example 4: Handling Unmatched Orchestrate Fields in an INFORMIX Write Operation” on page 35-14
• “Example 5: Writing to an INFORMIX Table with an Unmatched Column” on page 35-15

The **infxread** Operator

The **infxread** operator sets up a connection to an INFORMIX database, sends it a query, and translates the resulting two-dimensional array representing the INFORMIX result set to an Orchestrate data set.

**Data Flow Diagram**

```
INFORMIX query result

infxread

output data set
```

**Properties**

Table 200 **infxread Operator Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>0</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>none</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>determined by the SQL query.</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>none</td>
</tr>
</tbody>
</table>
Table 200  infxread Operator Properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution mode</td>
<td>parallel</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>not applicable</td>
</tr>
<tr>
<td>Preserve-partitioning flag in</td>
<td>clear</td>
</tr>
<tr>
<td>output data set</td>
<td></td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

Syntax and Options

The osh syntax for the infxread operator follows. You must specify either the -query or the -table option.

Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose it in single quotes.

infxread
  -table table_name [-filter filter] [-selectlist list]
  | -query sql_query
  | [-close close_command]
  | [-dbname database_name]
  | [-open open_command]
  | [-part table_name]
  | [-server server_name]

Table 201  infxread Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-close</td>
<td>-close close_command</td>
</tr>
<tr>
<td></td>
<td>Optionally specify an INFORMIX SQL statement to be parsed and executed on all processing nodes after the table selection or query is completed.</td>
</tr>
<tr>
<td>-dbname</td>
<td>-dbname database_name</td>
</tr>
<tr>
<td></td>
<td>Optionally specify the name of the INFORMIX database to read from.</td>
</tr>
<tr>
<td>-dboptions</td>
<td>This is a deprecated option and is retained only for backward compatibility.</td>
</tr>
<tr>
<td>-open</td>
<td>-open open_command</td>
</tr>
<tr>
<td></td>
<td>Optionally specify an INFORMIX SQL statement to be parsed and executed by the database on all processing nodes before the read query is prepared and executed.</td>
</tr>
</tbody>
</table>
Table 201  infxread Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
</table>
| -part  | \(-part\) table_name  
If the table is fragmented, optionally specify this option. This improves performance by creating one instance of the operator per table fragment. If the table is fragmented across nodes, this option creates one instance of the operator per fragment per node. If the table is fragmented and you do not specify -part, the operator nonetheless functions successfully, if more slowly. You must have Resource privilege to invoke this option.  
When used with the -table option, the table_name given must be the same name as that specified for the -table option. |
| -query | \(-query\) sql_query  
Specify a valid SQL query that is submitted to INFORMIX to read the data. The sql_query specifies the processing that INFORMIX performs as data is read into Orchestrate. The query can contain joins, views, synonyms, and so on. |
| -server | \(-server\) server_name  
This option is maintained for compatibility with the xpsread operator. It is ignored. |
| -table | \(-table\) table_name [-filter filter] [-selectlist selectlist]  
Specify the name of the INFORMIX table to read from. The table must exist. You can prefix table_name with a table owner in the form:  
\(table\_owner.table\_name\)  
The operator reads the entire table unless you limit its scope by means of -filter or -selectlist, or both.  
With -filter optionally specify selection criteria to be used as part of an SQL statement's WHERE clause, to specify the rows of the table to include in or exclude from the Orchestrate data set.  
With -selectlist, optionally specify a list of columns to read, if you do not want all columns to be read. |

Example

Refer to “Example 1: Reading All Data from an INFORMIX Table” on page 35-6.
The infxwrite Operator

The **infxwrite** operator sets up a connection to INFORMIX and inserts records into a table. The operator takes a single input data set. The record schema of the data set and the write mode of the operator determine how the records of a data set are inserted into the table.

Data Flow Diagram

```
input Orchestrate data set

infxwrite

output INFORMIX table or view
```

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>0</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>derived from the input data set</td>
</tr>
<tr>
<td>Execution mode</td>
<td>sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>same</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

Note

This operator does not function in parallel. For parallel operation, use the **xpswrite** or **hplwrite** operator.
Syntax and Options

The *osh* syntax for the `infxwrite` operator follows. You must specify exactly one occurrence of the `-table` option; all other options are optional.

Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose it in single quotes.

```
infxwrite
  -table table_name[-selectlist list]
  [-close close_command]
  [-dbname database_name]
  [-drop]
  [-mode append | create | replace | truncate]
  [-open open_command]
  [-server server_name]
  [-stringlength length]
```

Table 203  `infxwrite` Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-close</code></td>
<td><code>-close close_command</code>&lt;br&gt;Specify an INFORMIX SQL statement to be parsed and executed by INFORMIX on all processing nodes after the table has been populated.</td>
</tr>
<tr>
<td><code>-dbname</code></td>
<td><code>-dbname database_name</code>&lt;br&gt;Specify the name of the INFORMIX database containing the table specified by <code>table</code>.</td>
</tr>
<tr>
<td><code>-dboptions</code></td>
<td>This is a deprecated option and is retained only for backward compatibility.</td>
</tr>
<tr>
<td><code>-drop</code></td>
<td><code>-drop</code>&lt;br&gt;Use this option to cause the operator to drop, with a warning, all input fields that do not correspond to columns of an existing table. If you do not specify <code>drop</code>, an unmatched field generates an error and the associated step terminates.</td>
</tr>
</tbody>
</table>
Chapter 35  The Informix Interface Library  The infxwrite Operator

Table 203  infxwrite Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-mode</td>
<td>-mode append</td>
</tr>
<tr>
<td>append: infxwrite</td>
<td>appends new records to the table. The database user who writes in this mode must have Resource privileges. This is the default mode.</td>
</tr>
<tr>
<td>create: infxwrite</td>
<td>creates a new table; the database user who writes in this mode must have Resource privileges. Orchestrate returns an error if the table already exists.</td>
</tr>
<tr>
<td>replace: infxwrite</td>
<td>deletes the existing table and creates a new one in its place; the database user who writes in this mode must have Resource privileges.</td>
</tr>
<tr>
<td>truncate: infxwrite</td>
<td>retains the table attributes but discards existing records and appends new ones. The operator runs more slowly in this mode if the user does not have Resource privileges.</td>
</tr>
<tr>
<td>-open</td>
<td>open open_command</td>
</tr>
<tr>
<td></td>
<td>Specify an INFORMIX SQL statement to be parsed and executed by INFORMIX on all processing nodes before opening the table.</td>
</tr>
<tr>
<td>-server</td>
<td>server server_name</td>
</tr>
<tr>
<td></td>
<td>This option is maintained for compatibility with the xpswrite operator. It is ignored.</td>
</tr>
<tr>
<td>-stringlength</td>
<td>stringlength length</td>
</tr>
<tr>
<td></td>
<td>Set the default length of variable-length raw or string fields. If you do not specify a length, the default is 32 bytes. You can specify a length up to 255 bytes.</td>
</tr>
<tr>
<td>-table</td>
<td>table table_name[-selectlist list]</td>
</tr>
<tr>
<td></td>
<td>Specifies the name of the INFORMIX table to write to. See the mode options for constraints on the existence of the table and the user privileges required. The -selectlist option specifies an SQL list that determines which fields are written. If you do not supply the list, infxwrite writes the input field records according to your chosen mode specification.</td>
</tr>
</tbody>
</table>

Examples

Refer to the following sections for examples of INFORMIX write operator use:

- “Example 2: Appending Data to an Existing INFORMIX Table” on page 35-12
- “Example 3: Writing Data to an INFORMIX Table in Truncate Mode” on page 35-13
- “Example 4: Handling Unmatched Orchestrate Fields in an INFORMIX Write Operation” on page 35-14
The xpsread Operator

The xpsread operator sets up a connection to an INFORMIX database, sends it a query, and translates the resulting two-dimensional array representing the INFORMIX result set to an Orchestrate data set.

Data Flow Diagram

![Data Flow Diagram]

Properties

Table 204  xpsread Operator Options

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>0</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>none</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>determined by the SQL query</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>none</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>not applicable</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>clear</td>
</tr>
<tr>
<td>Composite operator</td>
<td>yes</td>
</tr>
</tbody>
</table>
Syntax and Options

The syntax of the `xpsread` operator is below. You must specify either the `-query` or the `-table` option.

Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose it in single quotes.

```
xpsread
  -table table_name [-filter filter] [-selectlist list]
  | -query sql_query
  [-close close_command]
  [-dbname database_name]
  [-open open_command]
  [-part table_name]
  [-server server_name]
```

### Table 205  xpsread Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-close</code></td>
<td><code>-close close_command</code></td>
</tr>
<tr>
<td></td>
<td>Optionally specify an INFORMIX SQL statement to be parsed and executed on all processing nodes after the table selection or query is completed.</td>
</tr>
<tr>
<td><code>-dbname</code></td>
<td><code>-dbname database_name</code></td>
</tr>
<tr>
<td></td>
<td>Specify the name of the INFORMIX database to read from.</td>
</tr>
<tr>
<td><code>-dboptions</code></td>
<td>This is a deprecated option and is retained only for backward compatibility.</td>
</tr>
<tr>
<td><code>-open</code></td>
<td><code>-open open_command</code></td>
</tr>
<tr>
<td></td>
<td>Optionally specify an INFORMIX SQL statement to be parsed and executed by the database on all processing nodes before the read query is prepared and executed.</td>
</tr>
<tr>
<td><code>-part</code></td>
<td><code>-part table_name</code></td>
</tr>
<tr>
<td></td>
<td>Optionally specify this option if the table is fragmented to improve performance by creating one instance of the operator per table fragment. If the table is fragmented across nodes, this option creates one instance of the operator per fragment per node. If the table is fragmented and you do not specify this option, the operator nonetheless functions successfully, if more slowly. You must have Resource privilege to invoke this option.</td>
</tr>
</tbody>
</table>
The xpswrite Operator

The xpswrite operator sets up a connection to INFORMIX and inserts records into a table. The operator takes a single input data set. The record schema of the data set and the write mode of the operator determine how the records of a data set are inserted into the table.

Example

Refer to “Example 1: Reading All Data from an INFORMIX Table” on page 35-6.
Data Flow Diagram

input Orchestrate data set

xpswrite

output INFORMIX table or view

Properties

Table 206  xpswrite Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>0</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>none</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>derived from the input data set’s record schema.</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>none</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>same</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>not applicable</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>

Syntax and Options

You must specify exactly one occurrence of the `table` option; all other options are optional. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose it in single quotes.
**xpswrite**

- `table table_name[-selectlist list]`
- `close close_command`
- `dbname database_name`
- `drop`
- `mode append | create | replace | truncate`
- `open open_command`
- `server server_name`
- `stringlength length`

### Table 207  xpswrite Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-close</td>
<td><code>-close close_command</code></td>
</tr>
<tr>
<td></td>
<td>Specify an INFORMIX SQL statement to be parsed and executed by INFORMIX on all processing nodes after the table has been populated.</td>
</tr>
<tr>
<td>-dbname</td>
<td><code>-dbname database_name</code></td>
</tr>
<tr>
<td></td>
<td>Specify the name of the INFORMIX database containing the table specified by table.</td>
</tr>
<tr>
<td>-dboptions</td>
<td>This is a deprecated option and is retained only for backward compatibility.</td>
</tr>
<tr>
<td>-drop</td>
<td><code>-drop</code></td>
</tr>
<tr>
<td></td>
<td>Use this option to cause the operator to drop, with a warning, all input fields that do not correspond to columns of an existing table. If you do not specify -drop, an unmatched field generates an error and the associated step terminates.</td>
</tr>
<tr>
<td>-mode</td>
<td>`-mode append</td>
</tr>
<tr>
<td></td>
<td><strong>append</strong>: xpswrite appends new records to the table. The database user who writes in this mode must have Resource privileges. This is the default mode.</td>
</tr>
<tr>
<td></td>
<td><strong>create</strong>: xpswrite creates a new table; the database user who writes in this mode must have Resource privileges. Orchestrate returns an error if the table already exists.</td>
</tr>
<tr>
<td></td>
<td><strong>replace</strong>: xpswrite deletes the existing table and creates a new one in its place; the database user who writes in this mode must have Resource privileges.</td>
</tr>
<tr>
<td></td>
<td><strong>truncate</strong>: xpswrite retains the table attributes but discards existing records and appends new ones.</td>
</tr>
<tr>
<td></td>
<td>The operator runs more slowly in this mode if the user does not have Resource privileges.</td>
</tr>
</tbody>
</table>
Examples

Refer to the following for examples of INFORMIX write operator use:

- “Example 2: Appending Data to an Existing INFORMIX Table” on page 35-12
- “Example 3: Writing Data to an INFORMIX Table in Truncate Mode” on page 35-13
- “Example 4: Handling Unmatched Orchestrate Fields in an INFORMIX Write Operation” on page 35-14
- “Example 5: Writing to an INFORMIX Table with an Unmatched Column” on page 35-15
The Oracle Interface Library

Describes the Oracle interface operators which enable sharing of data between Oracle and Orchestrate.

Overview of the Oracle Interface Operators
Introduction
Accessing Oracle from Orchestrate
Changing Library Paths
Preserving Blanks in Fields
Handling # and $ Characters in Oracle Column Names
Using the External Representation
Using the Internal Representation
National Language Support

The oraread Operator
Data Flow Diagram
Properties
Operator Action
Where the oraread Operator Runs
Column Name Conversion
Data Type Conversion
Oracle Record Size
Targeting the Read Operation
Specifying the Oracle Table Name
Specifying an SQL SELECT Statement
Join Operations
Syntax and Options
Example 1: Reading an Oracle Table and Modifying a Field Name
Example 2: Reading from an Oracle Table in Parallel with the query Option

The orawrite Operator
Writing to a Multibyte Database
Specifying chars and varchar
Specifying nchar and nvarchar2 Column Size
Overview of the Oracle Interface Operators

Introduction

This chapter of the Orchestrate 7.0 Operators Reference describes how to use the Oracle Interface operators. To understand the information in this chapter you should be familiar with the operator concepts defined in the chapter on operators in the Orchestrate 7.0 User Guide.
Orchestrate can share data with Oracle, a Relational Database Management System (RDBMS). Four Orchestrate operators provide this access:

- The **oraread** operator reads records from an Oracle table and places them in an Orchestrate data set. This operator is described on page 36-3.

- The **orawrite** operator sets up a connection to Oracle and inserts records into a table. The operator takes a single input data set. The write mode of the operator determines how the records of a data set are inserted into the table. This operator is described on page 36-17.

- The **oraupsert** operator lets you insert into an Oracle table or update an Oracle table with data contained in an Orchestrate data set. You can match records based on field names and then update or insert those records. This operator is described on page 36-32.

- The **oralookup** operator lets you perform a join between an Oracle table and an Orchestrate data set, with the resulting data output as an Orchestrate data set. This operator is described on page 36-39.

### Accessing Oracle from Orchestrate

This section assumes that Orchestrate users have been configured to access Oracle using the Oracle configuration process outlined in Chapter 8: RDBMS Configuration in the Orchestrate 7.0 Installation and Administration Manual.

**To access Oracle from Orchestrate:**

1. Set your **ORACLE_HOME** environment variable to your Oracle client installation.
2. Start Oracle.
3. Add **ORACLE_HOME/bin** to your **PATH** and **ORACLE_HOME/lib** to your **LIBPATH**, **LD_LIBRARY_PATH**, or **SHLIB_PATH**.

**Note:** **APT_ORCHHOME/bin** must appear before **ORACLE_HOME/bin** in your **PATH**.

4. Have login privileges to Oracle using a valid Oracle user name and corresponding password. These must be recognized by Oracle before you attempt to access it.

### Changing Library Paths

Because an Oracle client often cannot connect to a different version of the Oracle database, you may find it necessary to change your Oracle library path.
To change your Oracle library path:

1. Set your `ORACLE_HOME` environment variable to your Oracle client installation which must include the Oracle Database Utilities and the Oracle network software.

2. From your `$APT_ORCHHOME/install` directory, execute this command:
   
   ```bash
   ./install.liborchoracle
   ```

3. Verify that the Oracle library links in your `$APT_ORCHHOME/lib` directory have the correct version. For example, for version 8i on Sun Solaris, the links are:
   
   - `orchoracle.so -> liborchoraclesun4.so`
   - `liborchoraclesun4.so -> liborchoracle8isun4.so`

Preserving Blanks in Fields

Setting the `APT_ORACLE_PRESERVE_BLANKS` environment variable causes the `PRESERVE_BLANKS` option to be set in the control file. This option preserves leading and trailing spaces, and retains fields which contain only spaces. When `PRESERVE_BLANKS` is not set, Oracle removes the spaces and considers fields with only spaces to be NULL values.

Handling # and $ Characters in Oracle Column Names

The Oracle operators accept the `#` and `$` characters for Oracle table column names. Orchestrate converts these two reserved characters into an internal format when they are written to the Oracle database and reconverts them when they are read from Oracle.

The internal representation for the `#` character is `__035__` and the internal representation for the `$` character is `__036__`. You should avoid using these internal strings in your Oracle column names.

Using the External Representation

Use the external representation (with the `#` and `$` characters) when referring to table column names for these options:

- The `selectlist` options of `oraread`
- The `query` option of `oraread` and `oralookup`
- The `insert` and `update` options

Using the Internal Representation

Use the internal representation (with the `__035__` and `__036__` strings) when referring to table column names for these options:
• The `-query` option of `db2lookup` when referring to an Orchestrate name. For example:

```java
-query 'MS##,D#$ FROM tablename WHERE (BS# + ORCHESTRATE.B__036____035)'
```

• The `-insert` and `-update` options when referring to an ORCHESTRATE name. For example:

```java
-insert 'INSERT INTO tablename (A#,B$#)
VALUES (ORCHESTRATE.A__035__, ORCHESTRATE.B__036____035__)'
-update 'UPDATE tablename set B$# = ORCHESTRATE.B__036____035__
WHERE (A# = ORCHESTRATE.A__035__)'
```

### National Language Support

Orchestrate’s National Language Support (NLS) makes it possible for you to process data in international languages using Unicode character sets. Orchestrate uses International Components for Unicode (ICU) libraries to support NLS functionality. For information on National Language Support, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and access the ICU home page:

http://oss.software.ibm.com/developerworks/opensource/icu/project

The Orchestrate Oracle operators support Unicode character data in schema, table, and index names; in user names and passwords; column names; table and column aliases; SQL*Net service names; SQL statements; and file-name and directory paths.

The operators have two options which optionally control character mapping:

• `-db_cs` character_set

  Specifies an ICU character set to map between Oracle `char` and `varchar` data and Orchestrate `ustring` data, and to map SQL statements for output to Oracle.

• `-nchar_cs`

  Specifies an ICU character set to map between Oracle `nchar` and `nvarchar2` values and Orchestrate `ustring` data.
The oraread Operator

The oraread operator sets up a connection to an Oracle table, uses an SQL query to request rows (records) from the table, and outputs the result as an Orchestrate data set.

Data Flow Diagram

![Data Flow Diagram]

Properties

Table 208  oraread Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>0</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>none</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>determined by the SQL query</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>none</td>
</tr>
<tr>
<td>Execution mode</td>
<td>sequential (default) or parallel</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>not applicable</td>
</tr>
<tr>
<td>Collection method</td>
<td>not applicable</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>clear</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>
Operator Action

Here are the chief characteristics of the oraread operator:

• It reads only from non-partitioned and range-partitioned tables when in parallel mode.

• The operator functions sequentially, unless you specify the -part suboption of the -query or -table option.

• You can direct it to run in specific node pools. See “Where the oraread Operator Runs” on page 36-7.

• It translates the query’s result set (a two-dimensional array) row by row to an Orchestrate data set, as discussed in “Column Name Conversion” on page 36-8.

• Its output is an Orchestrate data set that you can use as input to a subsequent Orchestrate operator.

• Its translation includes the conversion of Oracle data types to Orchestrate data types, as listed in Chapter 36, “Data Type Conversion”.

• The size of Oracle rows can be greater than that of Orchestrate records. See “Oracle Record Size” on page 36-9.

• The operator specifies either an Oracle table to read or an SQL query to carry out. See “Specifying the Oracle Table Name” on page 36-10 and “Specifying an SQL SELECT Statement” on page 36-10.

• It optionally specifies commands to be run on all processing nodes before the read operation is performed and after it has completed.

• You can perform a join operation between Orchestrate data sets and Oracle data. See “Join Operations” on page 36-11.

Note
An RDBMS such as Oracle does not guarantee deterministic ordering behavior unless an SQL operation constrains it to do so. Thus, if you read the same Oracle table multiple times, Oracle does not guarantee delivery of the records in the same order every time.

While Oracle allows you to run queries against tables in parallel, not all queries should be run in parallel. For example, some queries perform an operation that must be carried out sequentially, such as a grouping operation or a non-collocated join operation. These types of queries should be executed sequentially in order to guarantee correct results.

Where the oraread Operator Runs

By default the oraread operator runs sequentially. However, you can direct it to read from multiple partitions and run on all processing nodes in the default node pool.
pool defined by your Orchestrate configuration file. You do so by specifying the operator’s `-server` option. The option takes a single argument defining a remote resource pool name. Orchestrate runs the operator on all processing nodes with a resource of type Oracle in the pool specified by `-server`.

For example, the following Orchestrate configuration file defines three processing nodes; two of them define resources of type Oracle:

```json
{
    node "node0"
    {
        fastname "node0_css"
        pool "" "node0" "node0_css" "syncsort"
        resource disk "/orch/s0" {pool "" "export"}
        resource scratchdisk "/scratch" {}
        resource ORACLE node0 {pool "oracle_pool"}
    }

    node "node1"
    {
        fastname "node1_css"
        pool "" "node1" "node1_css" "syncsort"
        resource disk "/orch/s0" {pool "" "export"}
        resource scratchdisk "/scratch" {}
        resource ORACLE node0 {pool "oracle_pool"}
    }

    node "node2"
    {
        fastname "node2_css"
        pool "" "node2" "node2_css"
        resource disk "/orch/s0" {}
        resource scratchdisk "/scratch" {}
    }
}
```

In this case, the option `-server oracle_pool` configures the operator to execute only on `node0` and `node1`.

**Column Name Conversion**

An Oracle result set is defined by a collection of rows and columns. The `oraread` operator translates the query’s result set (a two-dimensional array) to an Orchestrate data set. Here is how an Oracle query result set is converted to an Orchestrate data set:

- The rows of an Oracle result set correspond to the records of an Orchestrate data set.
- The columns of an Oracle row correspond to the fields of an Orchestrate record; the name and data type of an Oracle column correspond to the name and data type of an Orchestrate field.
- Names are translated exactly except when the Oracle column name contains a character that Orchestrate does not support. In that case, the unsupported character is replaced by two underscore characters. See the Orchestrate 7.0 User Guide for a list of unsupported characters.

- Both Oracle columns and Orchestrate fields support nulls, and a null contained in an Oracle column is stored as a null in the corresponding Orchestrate field.

Data Type Conversion

The `oraread` operator converts Oracle data types to Orchestrate data types, as in this table:

<table>
<thead>
<tr>
<th>Oracle Data Type</th>
<th>Corresponding Orchestrate Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)</td>
<td>string[n] or ustring[n], a fixed-length string with length = n</td>
</tr>
<tr>
<td>DATE</td>
<td>timestamp</td>
</tr>
<tr>
<td>NUMBER</td>
<td>decimal[38,10]</td>
</tr>
<tr>
<td>NUMBER[p,s]</td>
<td>int32 if precision (p) &lt; 11 and scale (s) = 0</td>
</tr>
<tr>
<td></td>
<td>decimal[p,s] if precision (p) &gt;= 11 or scale &gt; 0</td>
</tr>
<tr>
<td>RAW(n)</td>
<td>not supported</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>string[max=n] or ustring[max=n], a variable-length string with maximum length = n</td>
</tr>
</tbody>
</table>

Note: Data types that are not listed in the table above generate an error.

Oracle Record Size

The size of an Orchestrate record is limited to 32 KB. However, Oracle records can be longer than 32 KB. If you attempt to read a record longer than 32 KB, Orchestrate returns an error and terminates your application.

Targeting the Read Operation

When reading an Oracle table, you can either specify the table name and allow Orchestrate to generate a default query that reads the table or you can explicitly specify the query.
Specifying the Oracle Table Name

If you choose the -table option, Orchestrate issues the following SQL SELECT statement to read the table:

```sql
select [selectlist]
    from table_name
    and (filter);
```

You can specify optional parameters to narrow the read operation. They are as follows:

- The selectlist specifies the columns of the table to be read; by default, Orchestrate reads all columns.
- The filter specifies the rows of the table to exclude from the read operation; by default, Orchestrate reads all rows.

You can optionally specify -open and -close option commands. These commands are executed by Oracle on every processing node containing a partition of the table before the table is opened and after it is closed.

See “Example 1: Reading an Oracle Table and Modifying a Field Name” on page 36-15 for an example of using the table option.

Specifying an SQL SELECT Statement

If you choose the -query option, you pass an SQL query to the operator. The query specifies the table and the processing that you want to perform on the table as it is read into Orchestrate. The SQL statement can contain joins, views, database links, synonyms, and so on. However, the following restrictions apply to -query:

- The -query may not contain bind variables.
- If you want to include a filter or select list, you must specify them as part of the query.
- The query runs sequentially by default, but can be run in parallel if you specify the -part option.
- You can specify an optional open and close command. Oracle runs these commands immediately before the database connection is opened and after it is closed.

Note | Complex queries executed in parallel may cause the database to consume large amounts of system resources.

See “Example 2: Reading from an Oracle Table in Parallel with the query Option” on page 36-16 for an example of using the -query option.
Join Operations

You can perform a join operation between Orchestrate data sets and Oracle data. First invoke the `oraread` operator and then invoke either the `lookup` operator or a join operator. See Chapter 11, “The lookup Operator” on page 11-1 and Chapter 28, “The Join Library” on page 28-1. Alternatively, you can use the `oralookup` operator described in “The oralookup Operator” on page 36-39.

Syntax and Options

The syntax for `oraread` follows. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose it in single quotes.

```bash
oraread
  -query sql_query
  | -table table_name [-filter filter] [-selectlist list]

  -dboptions '{user = username, password = password}'
  | '{user = @file_name}'

  [-close close_command]
  [-db_cs character_set]
  [-nchar_cs character_set]
  [-open open_command]
  [-ora8partition partition_name]
  [-part table_name]
  [-server remote_server_name]
```

You must specify either the `-query` or `-table` option. You must also specify `-dboptions` and supply the necessary information. The next table lists the options:
### Table 210  **oraread Operator Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-close</code></td>
<td><code>-close close_command</code></td>
</tr>
<tr>
<td></td>
<td>Specify a command, enclosed in single quotes, to be parsed and executed by Oracle on all processing nodes after Orchestrate completes processing the Oracle table and before it disconnects from Oracle.</td>
</tr>
<tr>
<td></td>
<td>If you do not specify a close_command, Orchestrate terminates its connection to Oracle. There is no default close_command.</td>
</tr>
<tr>
<td></td>
<td>You can include an Oracle stored procedure as part of close_command, in the form:</td>
</tr>
<tr>
<td></td>
<td>'execute procedureName;'</td>
</tr>
<tr>
<td><code>-db_cs</code></td>
<td><code>-db_cs character_set</code></td>
</tr>
<tr>
<td></td>
<td>Specify an ICU character set to map between Oracle <code>char</code> and <code>varchar</code> data and Orchestrate <code>ustring</code> data, and to map SQL statements for output to Oracle. The default character set is <code>UTF-8</code> which is compatible with your <code>osh</code> jobs that contain 7-bit <code>US-ASCII</code> data. If this option is specified, the <code>-nchar</code> option must also be specified.</td>
</tr>
<tr>
<td></td>
<td>For information on national language support, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:</td>
</tr>
<tr>
<td></td>
<td><a href="http://oss.software.ibm.com/icu/charset">http://oss.software.ibm.com/icu/charset</a></td>
</tr>
<tr>
<td><code>-nchar_cs</code></td>
<td><code>-nchar_cs character_set</code></td>
</tr>
<tr>
<td></td>
<td>Specify an ICU character set to map between Oracle <code>nchar</code> and <code>nvarchar2</code> values and Orchestrate <code>ustring</code> data. The default character set is <code>UTF-8</code> which is compatible with your <code>osh</code> jobs that contain 7-bit <code>US-ASCII</code> data. If this option is specified, the <code>-db_cs</code> option must also be specified.</td>
</tr>
<tr>
<td></td>
<td>For information on national language support, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:</td>
</tr>
<tr>
<td></td>
<td><a href="http://oss.software.ibm.com/icu/charset">http://oss.software.ibm.com/icu/charset</a></td>
</tr>
</tbody>
</table>
Specify either a user name and password for connecting to Oracle or a file containing the user name and password.

You can optionally use `-arraysize` to specify the number of records in each block of data read from Oracle. Orchestrate reads records in as many blocks as required to read all source data from Oracle.

By default, the array size is 1000 records. You can modify this parameter to tune the performance of your application.

Specify a command, enclosed in single quotes, to be parsed and executed by Oracle on all processing nodes before the table is opened.

Specify the name of the specific Oracle table partition that contains the rows you want to read.

Specifies running the read operator in parallel on the processing nodes in the default node pool. The default execution mode is sequential.

The table name must be the name of the table specified by `-table`.

Specify an SQL query, enclosed in single quotes, to read a table. The query specifies the table and the processing that you want to perform on the table as it is read into Orchestrate. This statement can contain joins, views, database links, synonyms, and so on.

remote_server_name must specify a remote connection. To specify a local connection, set your ORACLE_SID environment variable to a local server.

See “Where the oraread Operator Runs” on page 36-7 for more information.
Specify the name of the Oracle table. The table must exist and you must have SELECT privileges on the table. The table name may contain a view name only if the operator executes sequentially.

If your Oracle user name does not correspond to that of the owner of the table, prefix the table name with that of the table owner in the form:

```
tableowner.table_name
```

The `-filter` suboption optionally specifies a conjunction, enclosed in single quotes, to the WHERE clause of the SELECT statement to specify the rows of the table to include or exclude from reading into Orchestrate. See “Targeting the Read Operation” on page 36-9 for more information.

The suboption `-selectlist` optionally specifies an SQL select list, enclosed in single quotes, that can be used to determine which fields are read. You must specify the fields in `list` in the same order as the fields are defined in the record schema of the input table.
**Example 1: Reading an Oracle Table and Modifying a Field Name**

The following figure shows an Oracle table used as input to an Orchestrate operator:

The Oracle table contains three columns whose data types the operator converts as follows:

- *itemNum* of type `NUMBER[3,0]` is converted to `int32`
- *price* of type `NUMBER[6,2]` is converted to `decimal[6,2]`
- *storeID* of type `NUMBER[2,0]` is converted to `int32`

The schema of the Orchestrate data set created from the table is also shown in this figure. Note that the Orchestrate field names are the same as the column names of the Oracle table.

However, the operator to which the data set is passed has an input interface schema containing the 32-bit integer field `field1`, while the data set created from the Oracle table does not contain a field of the same name. For this reason, the *modify* operator must be placed between *oraread* and *sampleOperator* to translate the name of the field, *itemNum*, to the name `field1`. See Chapter 13, “The modify Operator” for more information.

Here is the *osh* syntax for this example:

```bash
$ modifySpec="field1 = itemNum,;"
```
$ osh 'oraread -t table 'table_1' -dboptions {'user = user101, password = userPword'} | modify '$modifySpec' | ...'

Example 2: Reading from an Oracle Table in Parallel with the query Option

The following query configures oraread to read the columns itemNum, price, and storeID from table_1 in parallel. One instance of the operator runs for each partition of the table.

Here is the osh syntax for this example:

$ osh 'oraread -query 'select itemNum, price, storeID from table_1' -part 'table_1' -dboptions {'user = user101, password = userPword'} ...'}
The orawrite Operator

The orawrite operator sets up a connection to Oracle and inserts records into a table. The operator takes a single input data set. The write mode of the operator determines how the records of a data set are inserted into the table.

Writing to a Multibyte Database

Specifying chars and varchars

Specify chars and varchars in bytes, with two bytes for each character. This example specifies 10 characters:

```
create table orch_data(col_a varchar(20));
```

Specifying nchar and nvarchar2 Column Size

Specify nchar and nvarchar2 columns in characters. For example, this example specifies 10 characters:

```
create table orch_data(col_a nvarchar2(10));
```

Data Flow Diagram

```
input data set

orawrite

output Oracle table
```

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>0</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>none</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>derived from the input data set.</td>
</tr>
</tbody>
</table>
Operator Action

Here are the main characteristics of the **orawrite** operator:

- It translates an Orchestrate data set record by record to an Oracle table using Oracle’s SQL*Loader Parallel Direct Path Load method.
- Translation includes the conversion of Orchestrate data types to Oracle data types. See “Data Type Conversion” on page 36-20.
- The operator appends records to an existing table, unless you set another mode of writing. See "Write Modes" on page 36-21.
- When you write to an existing table, the schema of the table determines the operator’s input interface, and the input data set schema must be compatible with the table’s schema. See “Matched and Unmatched Fields” on page 36-22.
- Each instance of a parallel write operator running on a processing node writes its partition of the data set to the Oracle table. If any instance fails, all fail.

You can optionally specify Oracle commands to be parsed and executed on all processing nodes before the write operation runs or after it completes.

Indexed Tables

Because the Oracle write operator writes to a table using the Parallel Direct Path Load method, the operator cannot write to a table that has indexes defined on it unless you include either the -index rebuild or the -index maintenance option.

If you want to write to such a table and not use an -index option you must first delete the indexes and recreate them after orawrite has finished processing. The operator checks for indexes and returns an error if any are found. You can also write to an indexed table if the operator is run in sequential mode and the...
environment variable APT_ORACLE_LOAD_OPTIONS is set to 'OPTIONS (DIRECT=TRUE, PARALLEL=FALSE)'.

Note that if you define the environment variable APT_ORACLE_LOAD_OPTIONS, Orchestrate allows you to attempt to write to an indexed table, regardless of how the variable is defined.

**Where the Write Operator Runs**

The default execution mode of the orawrite operator is parallel. The default is that the number of processing nodes is based on the configuration file. See the Orchestrate 7.0 Installation and Administration Manual for information about configuration files. However, if the environment variable APTORAIGNORE_CONFIG_FILE_PARALLELISM is set, the number of players is set to the number of data files in the table’s tablespace.

- **To direct the operator to run sequentially:**
  - Specify the [seq] framework argument.

You can optionally specify the resource pool in which the operator runs by choosing the operator’s -server option, which takes a single argument defining a resource pool name. Orchestrate then runs the operator on all processing nodes with a resource of type Oracle in the pool specified by server. See “Where the Write Operator Runs” on page 36-19 for further details.

**Data Conventions on Write Operations to Oracle**

Oracle columns are named identically as Orchestrate fields, with these restrictions:

- Oracle column names are limited to 30 characters. If an Orchestrate field name is longer, you can do one of the following:
  - Choose the -trunc option to configure the operator to truncate Orchestrate field names to 30 characters.
  - Use the modify operator to modify the Orchestrate field name. See Chapter 13, “The modify Operator” for information.
- An Orchestrate data set written to Oracle may not contain fields of certain types. If it does, an error occurs and the corresponding step terminates.
However Orchestrate offers operators that modify certain data types to ones Oracle accepts, as shown in Table 212.

Table 212  Operators for Oracle Data Type Compatibility

<table>
<thead>
<tr>
<th>Incompatible Type</th>
<th>Operator Used to Change It</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strings, fixed or variable length, longer than 2000 bytes</td>
<td>modify</td>
</tr>
<tr>
<td>See Chapter 13, “The modify Operator”.</td>
<td></td>
</tr>
<tr>
<td>Subrecord</td>
<td>promotesubrec</td>
</tr>
<tr>
<td>See “The promotesubrec Operator” on page 30-25.</td>
<td></td>
</tr>
<tr>
<td>Tagged aggregate</td>
<td>tagbatch</td>
</tr>
<tr>
<td>See “The tagbatch Operator” on page 30-33.</td>
<td></td>
</tr>
</tbody>
</table>

Data Type Conversion

Orchestrate data types are converted to Oracle data types as listed in the next table:

Table 213  Orchestrate to Oracle Data Type Conversion

<table>
<thead>
<tr>
<th>Orchestrate Data Type</th>
<th>Oracle Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>DATE</td>
</tr>
<tr>
<td>decimal[p,s]</td>
<td>NUMBER[p,s]</td>
</tr>
<tr>
<td>p is decimal’s precision and s is decimal’s scale</td>
<td></td>
</tr>
<tr>
<td>int8/ uint8</td>
<td>NUMBER[3,0]</td>
</tr>
<tr>
<td>int16/ uint16</td>
<td>NUMBER[5,0]</td>
</tr>
<tr>
<td>int32/ uint32</td>
<td>NUMBER[10,0]</td>
</tr>
<tr>
<td>int64</td>
<td>NUMBER[19,0]</td>
</tr>
<tr>
<td>uint64</td>
<td>NUMBER[20,0]</td>
</tr>
<tr>
<td>sfloat</td>
<td>NUMBER</td>
</tr>
<tr>
<td>dfloat</td>
<td>NUMBER</td>
</tr>
<tr>
<td>raw</td>
<td>not supported</td>
</tr>
<tr>
<td>fixed-length string, in the form string[n] or ustring[n], length ≤ 255 bytes</td>
<td>CHAR(n) where n is the string length</td>
</tr>
</tbody>
</table>
Orchestrate data types not listed in this table generate an error. Invoke the modify operator to perform type conversions. See Chapter 13, “The modify Operator”. All Orchestrate and Oracle integer and floating-point data types have the same range and precision, and you need not worry about numeric overflow or underflow.

**Write Modes**

The write mode of the operator determines how the records of the data set are inserted into the destination table. The write mode can have one of the following values:

- **append**: This is the default mode. The table must exist and the record schema of the data set must be compatible with the table. The write operator appends new rows to the table. The schema of the existing table determines the input interface of the operator.

- **create**: The operator creates a new table. If a table exists with the same name as the one you want to create, the step that contains the operator terminates with an error. The schema of the new table is determined by the schema of the Orchestrate data set. The table is created with simple default properties. To create a table that is partitioned, indexed, in a non-default table space, or in some other non-standard way, you can use the `-createstmt` option with your own create table statement.

<table>
<thead>
<tr>
<th>Orchestrate Data Type</th>
<th>Oracle Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable-length string, in the form string[max=n] or ustring[max=n] maximum length &lt;= 2096 bytes</td>
<td>VARCHAR(maximum =n) where n is the maximum string length</td>
</tr>
<tr>
<td>variable-length string in the form string or ustring</td>
<td>VARCHAR(32)*</td>
</tr>
</tbody>
</table>

* The default length of VARCHAR is 32 bytes. This means all records of the table have 32 bytes allocated for each variable-length string field in the input data set. If an input field is longer than 32 bytes, the operator issues a warning. The `-stringlength` option modifies the default length.

string, 2096 bytes <length not supported

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Oracle Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>DATE (does not support microsecond resolution)</td>
</tr>
<tr>
<td>timestamp</td>
<td>DATE (does not support microsecond resolution)</td>
</tr>
</tbody>
</table>

Table 213 *Orchestrate to Oracle Data Type Conversion (continued)*
• **replace**: The operator drops the existing table and creates a new one in its place. If a table exists of the same name as the one you want to create, it is overwritten. The schema of the new table is determined by the schema of the Orchestrate data set.

• **truncate**: The operator retains the table attributes but discards existing records and appends new ones. The schema of the existing table determines the input interface of the operator.

**Note**

If a previous write operation fails, you can retry your application specifying a write mode of **replace** to delete any information in the output table that may have been written by the previous attempt to run your program.

Each mode requires the specific user privileges shown in the table below:

<table>
<thead>
<tr>
<th>Write Mode</th>
<th>Required Privileges</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>INSERT on existing table</td>
</tr>
<tr>
<td>create</td>
<td>TABLE CREATE</td>
</tr>
<tr>
<td>replace</td>
<td>INSERT and TABLE CREATE on existing table</td>
</tr>
<tr>
<td>truncate</td>
<td>INSERT on existing table</td>
</tr>
</tbody>
</table>

**Matched and Unmatched Fields**

The schema of the Oracle table determines the operator’s interface schema. Once the operator determines this, it applies the following rules to determine which data set fields are written to the table:

1. Fields of the input data set are matched by name with fields in the input interface schema.

   Orchestrate performs default data type conversions to match the input data set fields with the input interface schema. See the chapter on using Orchestrate operators in the Orchestrate 7.0 User Guide for more information on these conversions.

   You can also use the **modify** operator to perform explicit data type conversions. See Chapter 36, “The Oracle Interface Library,” for more information.

2. If the input data set contains fields that do not have matching components in the table, the operator generates an error and terminates the step.

   This rule means that Orchestrate does not add new columns to an existing table if the data set contains fields that are not defined in the table. Note that you can use either the **orawrite-drop** option or the **modify** operator to drop
extra fields from the data set. See Chapter 36, “The Oracle Interface Library,” for more information.

3 Columns in the Oracle table that do not have corresponding fields in the input data set are set to their default value, if one is specified in the Oracle table. If no default value is defined for the Oracle column and it supports nulls, it is set to null. Otherwise, Orchestrate issues an error and terminates the step.

4 Orchestrate data sets support nullable fields. If you write a data set to an existing table and a field contains a null, the Oracle column must also support nulls. If not, Orchestrate issues an error message and terminates the step. However, you can use the modify operator to convert a null in an input field to another value. See the chapter on the modify operator for more information.

Syntax and Options

Syntax for the orawrite operator is given below. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose it in single quotes. Exactly one occurrence of the -dboptions option and the -table option are required.

```
orawrite
  -table table_name
  -dboptions
    '{user = username, password = password}'
    | '{user = '@file_name'}'
  [-close close_command]
  [-computeStats]
  [-createstmt create_statement]
  [-drop]
  [-db_cs character_set]
  [-disableConstraints]
  [-exceptionsTable exceptionsTableName]
  [-index rebuild]
    [-computeStats]
    [-nologging]
    | [-index maintenance]
  [-mode create | replace | append | truncate]
  [-nchar_cs character_set]
  [-open open_command]
  [-ora8partition ora8part_name]
  [-primaryKeys fieldname_1, fieldname_2, ... fieldname_n]
  [-server remote_server_name]
  [-strlen length]
  [-truncate]
  [-useNchar]
```
<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-close</td>
<td>close close_command</td>
</tr>
<tr>
<td></td>
<td>Specify any command, enclosed in single quotes, to be parsed and run by the Oracle database on all processing nodes when Orchestrate completes the processing of the Oracle table.</td>
</tr>
<tr>
<td></td>
<td>You can include an Oracle stored procedure as part of close_command, in the form:</td>
</tr>
<tr>
<td></td>
<td>&quot;execute procedureName;&quot;</td>
</tr>
<tr>
<td>-computeStats</td>
<td>computeStats</td>
</tr>
<tr>
<td></td>
<td>This option adds Oracle's COMPUTE STATISTICS clause to the -index rebuild command.</td>
</tr>
<tr>
<td></td>
<td>The -computeStatus and the -nologging options must be used in combination with the -index rebuild option as shown below:</td>
</tr>
<tr>
<td></td>
<td>-index rebuild</td>
</tr>
<tr>
<td></td>
<td>-computeStats</td>
</tr>
<tr>
<td></td>
<td>-nologging</td>
</tr>
<tr>
<td>-createstmt</td>
<td>create_statement</td>
</tr>
<tr>
<td></td>
<td>Creates a table. You use this option only in create and replace modes. You must supply the create table statement, otherwise Orchestrate attempts to create it based on simple defaults.</td>
</tr>
<tr>
<td></td>
<td>Here is an example create statement:</td>
</tr>
<tr>
<td></td>
<td>'create table test1 (A number, B char(10), primary key (A))'</td>
</tr>
<tr>
<td></td>
<td>You do not add the ending semicolon, as would normally be acceptable with an SQL statement.</td>
</tr>
<tr>
<td></td>
<td>When writing to a multibyte database, specify chars and varchars in bytes, with two bytes for each character. This example specifies 10 characters:</td>
</tr>
<tr>
<td></td>
<td>'create table orch_data(col_a varchar(20))'</td>
</tr>
<tr>
<td></td>
<td>When specifying nchar and nvarchar2 column types, specify the size in characters: This example specifies 10 characters:</td>
</tr>
<tr>
<td></td>
<td>'create table orch_data(col_a nvarchar2(10))'</td>
</tr>
<tr>
<td></td>
<td>This option is mutually exclusive with the -primaryKeys option.</td>
</tr>
</tbody>
</table>
Specify an ICU character set to map between Oracle char and varchar data and Orchestrate ustring data, and to map SQL statements for output to Oracle. The default character set is UTF-8 which is compatible with your osh jobs that contain 7-bit US-ASCII data. If this option is specified, you must also specify the -nchar option.

For information on national language support, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:

http://oss.software.ibm.com/icu/charset

Specify an ICU character set to map between Oracle nchar and nvarchar2 values and Orchestrate ustring data. The default character set is UTF-8 which is compatible with your osh jobs that contain 7-bit US-ASCII data. If this option is specified, you must also specify the -db_cs option.

For information on national language support, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site:

http://oss.software.ibm.com/icu/charset

Specify either a user name and password for connecting to Oracle or a file containing the user name and password. These options are required by the operator.

Specify either a user name and password for connecting to Oracle or a file containing the user name and password. These options are required by the operator.

Disables all enabled constraints on a table, and then attempts to enable them again at the end of a load.

When disabling the constraints, the cascade option is included. If Oracle cannot enable all the constraints, warning messages are displayed.

If the -exceptionsTable option is included, ROWID information on rows that violate constraints are inserted into an exceptions table. In all cases, the status of all constraints on the table are displayed at the end of the operator run.

This option applies to all write modes.
Table 215  **orawrite Operator Options (continued)**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>-drop</strong></td>
<td>-drop</td>
</tr>
<tr>
<td></td>
<td>Specify dropping unmatched fields of the Orchestrate the data set. An</td>
</tr>
<tr>
<td></td>
<td>unmatched field is a field for which there is no identically named</td>
</tr>
<tr>
<td></td>
<td>field in the Oracle table.</td>
</tr>
<tr>
<td><strong>-exceptionsTable</strong></td>
<td>exceptionsTableName</td>
</tr>
<tr>
<td></td>
<td>Specify an exceptions table. exceptionsTableName is the name of a</td>
</tr>
<tr>
<td></td>
<td>table where record ROWID information is inserted if a record violates</td>
</tr>
<tr>
<td></td>
<td>a table constraint when an attempt is made to enable the constraint.</td>
</tr>
<tr>
<td></td>
<td>This table must already exist. It is not created by the operator.</td>
</tr>
<tr>
<td></td>
<td>Your Oracle installation should contain a script that can be</td>
</tr>
<tr>
<td></td>
<td>executed to create an exceptions table named exceptions.</td>
</tr>
<tr>
<td></td>
<td>This option can be included only in conjunction with the</td>
</tr>
<tr>
<td></td>
<td>-disableConstraints option.</td>
</tr>
<tr>
<td><strong>-index</strong></td>
<td>-index rebuild -computeStatus -nologging</td>
</tr>
<tr>
<td></td>
<td>Lets you perform a direct parallel load on an indexed table without</td>
</tr>
<tr>
<td></td>
<td>first dropping the index. You can choose either the -maintenance or</td>
</tr>
<tr>
<td></td>
<td>-rebuild option, although special rules apply to each (see below).</td>
</tr>
<tr>
<td></td>
<td>The -index option is applicable only in append and truncate write</td>
</tr>
<tr>
<td></td>
<td>modes, and in create mode only if a -createmnt option is provided.</td>
</tr>
<tr>
<td></td>
<td><strong>rebuild</strong>: Skips index updates during table load and instead</td>
</tr>
<tr>
<td></td>
<td>rebuilds the indexes after the load is complete using the Oracle</td>
</tr>
<tr>
<td></td>
<td>alter index rebuild command. The table must contain an index, and</td>
</tr>
<tr>
<td></td>
<td>the indexes on the table must not be partitioned.</td>
</tr>
<tr>
<td></td>
<td>The -computeStatus and the -nologging options can be used only with</td>
</tr>
<tr>
<td></td>
<td>the -index rebuild command. The -computeStatus option adds Oracle's</td>
</tr>
<tr>
<td></td>
<td>COMPUTE STATISTICS clause to the -index rebuild command; and the -</td>
</tr>
<tr>
<td></td>
<td>nologging option adds Oracle's NOLOGGING clause to the -index</td>
</tr>
<tr>
<td></td>
<td>rebuild command.</td>
</tr>
<tr>
<td></td>
<td><strong>maintenance</strong>: Results in each table partition being loaded</td>
</tr>
<tr>
<td></td>
<td>sequentially. Because of the sequential load, the table index that</td>
</tr>
<tr>
<td></td>
<td>exists before the table is loaded is maintained after the table is</td>
</tr>
<tr>
<td></td>
<td>loaded. The table must contain an index and be partitioned, and the</td>
</tr>
<tr>
<td></td>
<td>index on the table must be a local range-partitioned index that is</td>
</tr>
<tr>
<td></td>
<td>partitioned according to the same range values that were used to</td>
</tr>
<tr>
<td></td>
<td>partition the table. Note that in this case sequential means</td>
</tr>
<tr>
<td></td>
<td>sequential per partition, that is, the degree of parallelism is</td>
</tr>
<tr>
<td></td>
<td>equal to the number of partitions.</td>
</tr>
<tr>
<td></td>
<td>This option is mutually exclusive with the -primaryKeys option.</td>
</tr>
</tbody>
</table>
Table 215  orawrite Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>-mode</td>
<td>**mode create</td>
</tr>
</tbody>
</table>

Specify the write mode of the operator.

- **append** (default): New records are appended to the table.
- **create**: Create a new table. Orchestrate reports an error if the Oracle table already exists. You must specify this mode if the Oracle table does not exist.
- **truncate**: The existing table attributes (including schema) and the Oracle partitioning keys are retained but existing records are discarded. New records are then appended to the table.
- **replace**: The existing table is dropped and a new table is created in its place. Oracle uses the default partitioning method for the new table.

See “Write Modes” on page 36-21 for a table of required privileges for each mode.

- **-nologging**

This option adds Oracle’s NOLOGGING clause to the **-index rebuild** command.

This option and the **-computeStats** option must be used in combination with the **-index rebuild** option as shown below:

```
-index rebuild
-computeStats
-nologging
```

- **-open**

Specifies any command, enclosed in single quotes, to be parsed and run by the Oracle database on all processing nodes. Orchestrate causes Oracle to run this command before opening the table. There is no default open_command.

You can include an Oracle stored procedure as part of open_command in the form:

```
"execute procedureName;"
```

- **-ora8partition**

Name of the Oracle 8 table partition that records are written to. The operator assumes that the data provided is for the partition specified.
Table 215  orawrite Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-primaryKeys</code></td>
<td><code>-primaryKeys</code> <code>fieldname_1</code>, <code>fieldname_2</code>, ... <code>fieldname_n</code></td>
</tr>
<tr>
<td></td>
<td>This option can be used only with the <code>create</code> and <code>replace</code> write modes. The <code>-createstmt</code> option should not be included.</td>
</tr>
<tr>
<td></td>
<td>Use the external column names for this option.</td>
</tr>
<tr>
<td></td>
<td>By default, no columns are primary keys.</td>
</tr>
<tr>
<td></td>
<td>This clause is added to the create table statement:</td>
</tr>
<tr>
<td></td>
<td><code>constraint</code> <code>tablename_PK</code> <code>primary key</code> (<code>fieldname_1</code>, <code>fieldname_2</code>, .. <code>fieldname_n</code>)</td>
</tr>
<tr>
<td></td>
<td>The operator names the primary key constraint &quot;tablename PK&quot;, where <code>tablename</code> is the name of the table. If the <code>-disableConstraints</code> option is not also included, the primary key index will automatically be rebuilt at the end of the load.</td>
</tr>
<tr>
<td></td>
<td>This option is mutually exclusive with the <code>-index</code> and <code>-createstmt</code> option.</td>
</tr>
<tr>
<td><code>-server</code></td>
<td><code>-server</code> <code>remote_server_name</code></td>
</tr>
<tr>
<td></td>
<td><code>remote_server_name</code> must specify a remote connection.</td>
</tr>
<tr>
<td></td>
<td>To specify a local connection, set your ORACLE_SID environment variable to a local server.</td>
</tr>
<tr>
<td><code>-stringlength</code></td>
<td><code>-stringlength</code> <code>string_len</code></td>
</tr>
<tr>
<td></td>
<td>Set the default string length for variable-length strings written to an Oracle table. If you do not specify a length, Orchestrate uses a default size of 32 bytes. Variable-length strings longer than the set length cause an error.</td>
</tr>
<tr>
<td></td>
<td>You can set the maximum length up to 2000 bytes.</td>
</tr>
<tr>
<td></td>
<td>Note that the operator always allocates <code>string_len</code> bytes for a variable-length string. In this case, setting <code>string_len</code> to 2000 allocates 2000 bytes. Set <code>string_len</code> to the expected maximum length of your longest string.</td>
</tr>
<tr>
<td><code>-table</code></td>
<td><code>-table</code> <code>table_name</code></td>
</tr>
<tr>
<td></td>
<td>Specify the name of the Oracle table.</td>
</tr>
<tr>
<td></td>
<td>If you are writing to the table, and the output mode is <code>create</code>, the table must not exist. If the output mode is <code>append</code>, <code>replace</code>, or <code>truncate</code>, the table must exist.</td>
</tr>
<tr>
<td></td>
<td>The Oracle write operator cannot write to a table that has indexes defined on it. If you want to write to such a table, you must first delete the index(es) and recreate them after <code>orawrite</code> completes. The operator checks for indexes and returns an error if one is found.</td>
</tr>
</tbody>
</table>
Example 1: Writing to an Existing Oracle Table

When an existing Oracle table is written to:

- The column names and data types of the Oracle table determine the input interface schema of the write operator.
- This input interface schema then determines the fields of the input data set that is written to the table.

For example, the following figure shows the `orawrite` operator writing to an existing table:

```
| itemNum: int32; |
| price: decimal[6,2]; |
| storeID: int16; |

orawrite
```

Oracle table

```
itemNum: int32;
price: decimal[6,2];
storeID: int16;
```

**Table 215  orawrite Operator Options (continued)**

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-truncate</code></td>
<td>Configure the operator to truncate Orchestrate field names to 30 characters. Oracle has a limit of 30 characters for column names. By default, if an Orchestrate field name is too long, Orchestrate issues an error. The <code>-truncate</code> option overrides this default.</td>
</tr>
<tr>
<td><code>-useNchar</code></td>
<td>Allows the creation of tables with <code>nchar</code> and <code>nvarchar2</code> fields. This option only has effect when the <code>-mode</code> option is <code>create</code>. If you do not specify the <code>-useNchar</code> option, the table is created with <code>char</code> and <code>varchar</code> fields.</td>
</tr>
</tbody>
</table>

Orchestrate 7.0 Operators Reference
The record schema of the Orchestrate data set and the row schema of the Oracle table correspond to one another, and field and column names are identical. Here are the input Orchestrate record schema and output Oracle row schema:

<table>
<thead>
<tr>
<th>Input Orchestrate Record</th>
<th>Output Oracle Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemNum:int32;</td>
<td>price NUMBER[10,0]</td>
</tr>
<tr>
<td>price:decimal[3,2];</td>
<td>itemNum NUMBER[6,2]</td>
</tr>
<tr>
<td>storeID:int16;</td>
<td>storeD NUMBER[5,0]</td>
</tr>
</tbody>
</table>

Here is the osh syntax for this example:

```
$ osh "... op1 | orawrite -table 'table_2'
   -dboptions {'user = user101, password = userPword'}"
```

Note that since the write mode defaults to append, the mode option does not appear in the command.

**Example 2: Creating an Oracle Table**

To create a table, specify a write mode of either create or replace. The next figure is a conceptual diagram of the create operation.

Here is the osh syntax for this operator:

```
$ osh "... orawrite -table table_2
   -mode create
   -dboptions {'user = user101, password = userPword'} ..."
```

The orawrite operator creates the table, giving the Oracle columns the same names as the fields of the input Orchestrate data set and converting the Orchestrate data types to Oracle data types. See “Data Type Conversion” on page 36-20 for a list of Orchestrate-to-Oracle conversions.
Example 3: Writing to an Oracle Table Using the modify Operator

The modify operator allows you to drop unwanted fields from the write operation and to translate the name and/or data type of a field of the input data set to match the input interface schema of the operator.

The next example uses the modify operator:

```
modify
('itemNum=skewNum,
 storeID=store;
drop rating')
```

In this example, you use the modify operator to:

- Translate field names of the input data set to the names of corresponding fields of the operator’s input interface schema, that is skewNum to itemNum and store to storeID.

- Drop the unmatched rating field, so that no error occurs.

Note that Orchestrate performs automatic type conversion of store, promoting its int8 data type in the input data set to int16 in the orawrite input interface.

Here is the osh syntax for this operator:

```
$ modifySpec="itemNum=skewNum, storeID=store; drop rating"
$ osh "... op1 \modify \$modifySpec \orawrite -table table_2
\dboptions {"user = user101, password = userPword"}
```

See Chapter 13, “The modify Operator” information on using that operator.
The oraupsert Operator

The oraupsert operator inserts and updates Oracle table records with data contained in an Orchestrate data set. You provide the insert statement using the -insert option and provide the update and delete SQL statements using the -update option.

An example -update delete statement is:

```
UPDATE 'DELETE FROM tablename WHERE A = ORCHESTRATE.A'
```

This operator receives a single data set as input and writes its output to an Oracle table. You can request an optional output data set that contains the records that fail to be inserted or updated.

By default, oraupsert uses Oracle host-array processing to optimize the performance of inserting records.

Data Flow Diagram

```
Data Flow Diagram

input data set
oraupsert
optional reject dataset
output Oracle table
```

Properties

Table 216 oraupsert Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>by default, none; 1 when you select the -reject option</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>derived from your insert and update statements</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>Rejected update records are transferred to an output data set when you select the -reject option.</td>
</tr>
</tbody>
</table>
Here are the main characteristics of `oraupsert`:

- An `-update` statement is required. The `-insert` and `-reject` options are mutually exclusive with using an `-update` option that issues a delete statement.

- If an `-insert` statement is included, the insert is executed first. Any records that fail to be inserted because of a unique-constraint violation are then used in the execution of the update statement.

- Orchestrate uses host-array processing by default to enhance the performance of insert array processing. Each insert array is executed with a single SQL statement. Update records are processed individually.

- You use the `-insertArraySize` option to specify the size of the insert array. For example:

  ```
  -insertArraySize 250
  ```

  The default length of the insert array is 500. To direct Orchestrate to process your insert statements individually, set the insert array size to 1:

  ```
  -insertArraySize 1
  ```

- Your record fields can be variable-length strings. You can specify a maximum length or use the default maximum length of 80 characters.

  This example specifies a maximum length of 50 characters:

  ```
  record(field1:string[max=50])
  ```

  The maximum length in this example is, by default, 80 characters:

  ```
  record(field1:string)
  ```

- When an insert statement is included and host array processing is specified, an Orchestrate update field must also be an Orchestrate insert field.

- The `oraupsert` operator converts all values to strings before passing them to Oracle. The following Orchestrate data types are supported:

---

### Table 216  `oraupsert` Operator Properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution mode</td>
<td>parallel by default, or sequential</td>
</tr>
</tbody>
</table>
| Partitioning method | same  
You can override this partitioning method; however, a partitioning method of `entire` cannot be used. |
| Collection method | any                                                                  |
| Combinable operator | yes                                                                  |
- int8, uint8, int16, uint16, int32, uint32, int64, and uint64
- dfloat and sfloat
- decimal
- strings of fixed and variable length
- timestamp
- date

- By default, **oraupsert** produces no output data set. By using the **-reject** option, you can specify an optional output data set containing the records that fail to be inserted or updated. It’s syntax is:

  ```
  -reject filename
  ```

  For a failed insert record, these sqlcodes cause the record to be transferred to your reject dataset:
  
  -1400: cannot insert NULL
  -1401: inserted value too large for column
  -1438: value larger than specified precision allows for this column
  -1480: trailing null missing from string bind value

- **Note** An insert record that fails because of a unique constraint violation (sqlcode of -1) is used for updating.

  For a failed update record, these sqlcodes cause the record to be transferred to your reject dataset:

  -1: unique constraint violation
  -1401: inserted value too large for column
  -1403: update record not found
  -1407: cannot update to null
  -1438: value larger than specified precision allows for this column
  -1480: trailing null missing from string bind value

- A -1480 error occurs when a variable length string is truncated because its length is not specified in the input dataset schema and it is longer than the default maximum length of 80 characters.

- **Caution** When a record fails with an sqlcode other than those listed above, **oraupsert** also fails. Therefore, we recommend that you backup your Oracle table before running your data flow.

**Associated Environment Variables**

There are two environment variables associated with the **oraupsert** operator. They are:
• **APT_ORAUPSER_CXXMIT_TIME_INTERVAL**: You can reset this variable to change the time interval between Oracle commits; the default value is 2 seconds.

• **APT_ORAUPSER_CXXMIT_ROW_INTERVAL**: You can reset this variable to change the record interval between Oracle commits; the default value is 5000 records.

Commits are made whenever the time interval period has passed or the row interval is reached, whichever occurs first.

**Syntax and Options**

The syntax for `oraupsert` is shown below. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose it in single quotes.

```
oraupsert
  -dboptions
    `{user =username, password =password}`
    | `{user =\@file_name\}`
  -update update_or_delete_statement
    [-db_cs character_set]
    [-nchar_cs character_set]
    [-insert insert_statement]
    [-insertArraySize n]
    [-reject]
    [-server remote_server_name]
```

Exactly one occurrence of the `-dboptions` option and exactly one occurrence of the `-update` are required. All other options are optional.

<table>
<thead>
<tr>
<th>Options</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-dboptions</td>
<td>-dboptions</td>
</tr>
<tr>
<td></td>
<td><code>{user =username, password =password}</code></td>
</tr>
<tr>
<td></td>
<td><code>{user =\@filename\}</code></td>
</tr>
</tbody>
</table>

Table 217  **oraupsert Operator Options**
### oraupsert Operator Options (continued)

<table>
<thead>
<tr>
<th>Options</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-db_cs</td>
<td><code>db_cs</code> character set</td>
</tr>
<tr>
<td></td>
<td>Specify an ICU character set to map between Oracle <code>char</code> and <code>varchar</code></td>
</tr>
<tr>
<td></td>
<td>data and Orchestrate <code>ustring</code> data, and to map SQL statements for output</td>
</tr>
<tr>
<td></td>
<td>to Oracle. The default character set is UTF-8 which is compatible with</td>
</tr>
<tr>
<td></td>
<td>your <code>osh</code> jobs that contain 7-bit US-ASCII data. If this option is</td>
</tr>
<tr>
<td></td>
<td>specified, the <code>-nchar</code> option must also be specified.</td>
</tr>
<tr>
<td></td>
<td>For information on national language support, see Chapter 7: National</td>
</tr>
<tr>
<td></td>
<td>Language Support in the Orchestrate 7.0 User Guide; and reference this IBM</td>
</tr>
<tr>
<td></td>
<td>ICU site:</td>
</tr>
<tr>
<td>-nchar_cs</td>
<td><code>nchar_cs</code> character set</td>
</tr>
<tr>
<td></td>
<td>Specify an ICU character set to map between Oracle <code>nchar</code> and <code>nvarchar2</code></td>
</tr>
<tr>
<td></td>
<td>values and Orchestrate <code>ustring</code> data. The default character set is UTF-8</td>
</tr>
<tr>
<td></td>
<td>which is compatible with your <code>osh</code> jobs that contain 7-bit US-ASCII</td>
</tr>
<tr>
<td></td>
<td>data. If this option is specified, the <code>-db_cs</code> option must also be</td>
</tr>
<tr>
<td></td>
<td>specified.</td>
</tr>
<tr>
<td></td>
<td>For information on national language support, see Chapter 7: National</td>
</tr>
<tr>
<td></td>
<td>Language Support in the Orchestrate 7.0 User Guide; and reference this IBM</td>
</tr>
<tr>
<td></td>
<td>ICU site:</td>
</tr>
<tr>
<td>-insert</td>
<td><code>insert</code> insert_statement</td>
</tr>
<tr>
<td></td>
<td>Optionally specify the insert statement to be executed.</td>
</tr>
<tr>
<td></td>
<td>This option is mutually exclusive with using an <code>-update</code> option that</td>
</tr>
<tr>
<td></td>
<td>issues a delete statement.</td>
</tr>
<tr>
<td>-insertArraySize</td>
<td><code>insertArraySize</code></td>
</tr>
<tr>
<td></td>
<td>Specify the size of the insert host array. The default size is 500</td>
</tr>
<tr>
<td></td>
<td>records. If you want each insert statement to be executed individually,</td>
</tr>
<tr>
<td></td>
<td>specify 1 for this option.</td>
</tr>
<tr>
<td>-reject</td>
<td><code>reject</code> filename</td>
</tr>
<tr>
<td></td>
<td>If this option is set, records that fail are written to a reject data</td>
</tr>
<tr>
<td></td>
<td>set. You must designate an output data set for this purpose.</td>
</tr>
<tr>
<td></td>
<td>This option is mutually exclusive with using an <code>-update</code> option that</td>
</tr>
<tr>
<td></td>
<td>issues a delete statement.</td>
</tr>
</tbody>
</table>
Example

This example updates an Oracle table that has two columns: acct_id and acct_balance, where acct_id is the primary key. Input is from a text file.

Note

The records are hashed and sorted on the unique key, acct_id. This ensures that records with the same account ID are in the same partition which avoids non-deterministic processing behavior that can lead to deadlocks.

Two of the records cannot be inserted because of unique key constraints; instead, they are used to update existing records. One record is transferred to the reject dataset because its acct_id generates an -1438 sqlcode due to its length exceeding the schema-specified length.

Summarized below are the state of the Oracle table before the dataflow is run, the contents of the input file, and the action Orchestrate performs for each record in the input file.

Table before dataflow                   Input file contents                    Orchestrate action
acct_id     acct_balance   873092  67.23   865544  8569.23   566678  2008.56   678888  7888.23   073587  82.56   update
073587     45.64           873092  67.23   865544  8569.23   566678  2008.56   678888  7888.23   073587  82.56   insert
873092     2001.89         873092  67.23   865544  8569.23   566678  2008.56   678888  7888.23   073587  82.56   update
675066     3523.62         873092  67.23   865544  8569.23   566678  2008.56   678888  7888.23   073587  82.56   insert
566678     89.72           873092  67.23   865544  8569.23   566678  2008.56   678888  7888.23   073587  82.56   update
995666     75.72

osh Syntax

$ osh 'import -schema record(acct_id:string[6]; acct_balance:dfloat;)' -file input.txt |
hash -key acct_id |
tsort -key acct_id |
oraupsert -dboptions '{user=apt, password=test}'
  -insert 'insert into accounts
     values(ORCHESTRATE.acct_id,
             ORCHESTRATE.acct_balance)'
  -update 'update accounts
    set acct_balance = ORCHESTRATE.acct_balance
    where acct_id = ORCHESTRATE.acct_id'
  -reject '/user/home/reject/reject.ds''

**Table after dataflow**

<table>
<thead>
<tr>
<th>acct_id</th>
<th>acct_balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>073587</td>
<td>82.56</td>
</tr>
<tr>
<td>873092</td>
<td>67.23</td>
</tr>
<tr>
<td>675066</td>
<td>3523.62</td>
</tr>
<tr>
<td>566678</td>
<td>2008.56</td>
</tr>
<tr>
<td>865544</td>
<td>8569.23</td>
</tr>
<tr>
<td>678888</td>
<td>7888.23</td>
</tr>
<tr>
<td>995666</td>
<td>75.72</td>
</tr>
</tbody>
</table>
The oralookup Operator

With the `oralookup` operator, you can perform a join between one or more Oracle tables and an Orchestrate data set. The resulting output data is an Orchestrate data set containing Orchestrate and Oracle data.

You perform this join by specifying either an SQL SELECT statement, or by specifying one or more Oracle tables and one or more key fields on which to do the lookup.

This operator is particularly useful for sparse lookups, that is, where the Orchestrate data set you are matching is much smaller than the Oracle table. If you expect to match 90% of your data, using the `oraread` and `lookup` operators is probably more efficient.

Because `oralookup` can do lookups against more than one Oracle table, it is useful for joining multiple Oracle tables in one query.

The `-statement` option command corresponds to an SQL statement of this form:

```sql
select a, b, c from data.testtbl
where
  orchestrate.b = data.testtbl.c
  and
  orchestrate.name = "Smith"
```

The operator replaces each `orchestrate.fieldname` with a field value, submits the statement containing the value to Oracle, and outputs a combination of Oracle and Orchestrate data.

Alternatively, you can use the `-key/-table` options interface to specify one or more key fields and one or more Oracle tables. The following `osh` options specify two keys and a single table:

```bash
-key a -key b -table data.testtbl
```

you get the same result as you would by specifying:

```sql
select * from data.testtbl
where
  orchestrate.a = data.testtbl.a
  and
  orchestrate.b = data.testtbl.b
```

The resulting Orchestrate output data set includes the Orchestrate records and the corresponding rows from each referenced Oracle table. When an Oracle table has a column name that is the same as an Orchestrate data set field name, the Oracle column is renamed using the following syntax:

```
APT_integer_fieldname
```

An example is `APT_0_lname`. The integer component is incremented when duplicate names are encountered in additional tables.
Note: If the Oracle table is not indexed on the lookup keys, the performance of this operator is likely to be poor.

Data Flow Diagram

Properties

Table 218  oralookup Operator Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Number of output data sets</td>
<td>1; 2 if you include the <em>-ifNotFound reject</em> option</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>determined by the query</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>determined by the SQL query</td>
</tr>
<tr>
<td>Transfer behavior</td>
<td>transfers all fields from input to output</td>
</tr>
<tr>
<td>Execution mode</td>
<td>sequential or parallel (default)</td>
</tr>
<tr>
<td>Preserve-partitioning flag in output data set</td>
<td>clear</td>
</tr>
<tr>
<td>Composite operator</td>
<td>no</td>
</tr>
</tbody>
</table>
Syntax and Options

The syntax for the `oralookup` operator is given below. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose it in single quotes.

```
oralookup
  -dboptions
    '{user = username, password = password}'
    | '{user = '@file_name'}'
  -table table_name -key field [-key field ...]
    | -table table_name -key field [-key field ...]|
  -query sql_query
    | -part table_name
    | -db_cs character_set
    | -nchar_cs character_set
    | -ifNotFound fail | drop | reject filename | continue|
    | -server remote_server
```

You must specify either the `-query` option or one or more `-table` options with one or more `-key` fields. Exactly one occurrence of the `-dboptions` is required.

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
</table>
| -dboptions | `-dboptions
  '{user = username, password = password}'
  | '{user = '@filename'}'`                                          |
| -db_cs     | `-db_cs character_set
  Specify an ICU character set to map between Oracle `CHAR` and `VARCHAR` data and Orchestrate `USTRING` data, and to map SQL statements for output to Oracle. The default character set is `UTF-8` which is compatible with your `OSH` jobs that contain 7-bit `US-ASCII` data. If this option is specified, you must also specify the `-nchar` option. |
|            | For information on national language support, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site: |
|            | http://oss.software.ibm.com/icu/charset |
Example

Suppose you want to connect to the APT81 server as user user101, with the password test. You want to perform a lookup between an Orchestrate data set and a table called target, on the key fields lname, fname, and DOB. You can configure oralookup in either of two ways to accomplish this.

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-nchar_cs</code></td>
<td>Specify an ICU character set to map between Oracle <code>nchar</code> and <code>nvarchar2</code> values and Orchestrate <code>ustring</code> data. The default character set is UTF-8 which is compatible with your osh jobs that contain 7-bit US-ASCII data. If this option is specified, you must also specify the <code>-db_cs</code> option. For information on national language support, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide; and reference this IBM ICU site: <a href="http://oss.software.ibm.com/icu/charset">http://oss.software.ibm.com/icu/charset</a></td>
</tr>
<tr>
<td><code>-ifNotFound</code></td>
<td>Optionally determines what action to take in case of a lookup failure. The default is <code>fail</code>. If you specify <code>reject</code>, you must designate an additional output data set for the rejected records.</td>
</tr>
<tr>
<td><code>-query</code></td>
<td>Specify an SQL query, enclosed in single quotes, to read a table. The query specifies the table and the processing that you want to perform on the table as it is read into Orchestrate. This statement can contain joins, views, database links, synonyms, and so on. A select statement corresponding to this lookup.</td>
</tr>
<tr>
<td><code>-server</code></td>
<td>remote_server_name must specify a remote connection. To specify a local connection, set your ORACLE_SID environment variable to a local server.</td>
</tr>
<tr>
<td><code>-table</code></td>
<td>Specify a table and key field(s) from which a SELECT statement is created. This is equivalent to specifying the SELECT statement directly using the <code>-statement</code> option. You can specify multiple instances of this option.</td>
</tr>
</tbody>
</table>
Here is the `osh` command using the `-table` and `-key` options:

```bash
$ osh ' oralookup -dboptions {'user = user101, password = test'}
          -server APT81
          -table target -key lname -key fname -key DOB
          < data1.ds > data2.ds '
```

Here is the equivalent `osh` command using the `-query` option:

```bash
$ osh ' oralookup -dboptions {'user = user101, password = test'}
          -server APT81
          -query 'select * from target
          where lname = Orchestrate.lname
          and fname = Orchestrate.fname
          and DOB = Orchestrate.DOB'
          < data1.ds > data2.ds '
```

Orchestrate prints the `lname`, `fname`, and `DOB` column names and values from the Orchestrate input dataset and also the `lname`, `fname`, and `DOB` column names and values from the Oracle table.

If a column name in the Oracle table has the same name as an Orchestrate output data set schema field name, the printed output shows the column in the Oracle table renamed using this format:

```
APT_integer_fieldname
```

For example, `lname` may be renamed to `APT_0_lname`. 

---

**Chapter 36** The Oracle Interface Library

**The oralookup Operator**

---

Orchestrate 7.0 Operators Reference
Overview of the Teradata Interface Operators

This chapter of the Orchestrate 7.0 Operators Reference describes how to use the Teradata Interface operators. To understand the information in this chapter you
should be familiar with the operator concepts defined in the chapter on operators in the Orchestrate 7.0 User Guide.

Summary of the Teradata Interface Operators

- The `teraread` operator sets up a connection to a Teradata database and reads the results of a query into an Orchestrate data set. As the operator requests records, the result set is translated row by row into an Orchestrate data set. This operator is described on page 4.

- The `terawrite` operator sets up a connection to a Teradata database to write data to it from an Orchestrate data set. As the operator writes records, the Orchestrate data set is translated row by row to a Teradata table. This operator is described in on page 10.

National Language Support

Orchestrate's National Language Support (NLS) makes it possible for you to process data in international languages using Unicode character sets. NLS is built on IBM's International Components for Unicode (ICU). For information on National Language Support, see Chapter 7: National Language Support in the Orchestrate 7.0 User Guide, and access the ICU home page:

http://oss.software.ibm.com/developerworks/opensource/icu/project

The Orchestrate Teradata operators support Unicode character data in usernames, passwords, column names, table names, and database names.

Teradata Database Character Sets

The Teradata database supports a fixed number of character set types for each char or varchar column in a table. Use this query to get the character set for a Teradata column:

```sql
select 'column_name', chartype from dbc.columns
where tablename = 'table_name'
```

The database character set types are:

- **Latin**: `chartype=1`. The character set for U.S. and European applications which limit character data to the ASCII or ISO 8859 Latin1 character sets. This is the default.

- **Unicode**: `chartype=2`. 16-bit Unicode characters from the ISO 10646 Level 1 character set. This setting supports all of the ICU multi-byte character sets.

- **KANJISJIS**: `chartype=3`. For Japanese third-party tools that rely on the string length or physical space allocation of KANJISJIS.

- **Graphic**: `chartype=4`. Provided for DB2 compatibility.
During the installation of the Teradata server software, you are prompted to specify whether Japanese language support should be enabled. If it is enabled, the Teradata data-dictionary tables use the Kanji1 character set type to support multi-byte code points in database object names such as tables and columns. When not enabled, the Teradata data-dictionary tables use the Latin character set to support single-byte code points for object names.

**Specifying an Orchestrate ustring Character Set**

The Orchestrate Teradata interface operators perform character-set conversions between the Teradata database characters sets and the multi-byte Unicode ustring field type data. You use the `-db_cs` option of the Teradata interface operators to specify a ICU character set for mapping strings between the database and the Teradata operators. The default value is Latin1.

Its syntax is:

```
   teraread | terawrite -db_cs icu_character_set
```

For example:

```
   terawrite -db_cs ISO-8859-5
```

Your database character set specification controls the following conversions:

- SQL statements are mapped to your specified character set before they are sent to the database via the native Teradata APIs.

  If you do not want your SQL statements converted to this character set, set the `APT_TERA_NO_SQL_CONVERSION` environment variable. This variable forces the SQL statements to be converted to Latin1 instead of the character set specified by the `-db_cs` option.

- All char and varchar data read from the Teradata database is mapped from your specified character set to the ustring schema data type (UTF-16). If you do not specify the `-db_cs` option, string data is read into a string schema type without conversion.

  The `teraread` operator converts a varchar(n) field to ustring[n/ min], where min is the minimum size in bytes of the largest codepoint for the character set specified by `-db_cs`.

---

**Note**

The KANJI1:chartype=5 character set is available for Japanese applications that must remain compatible with previous releases; however, this character set will be removed in a subsequent release because it does not support the new string functions and will not support future characters sets. We recommend that you use the set of SQL translation functions provided to convert KANJI1 data to Unicode.
• **ustring** schema type data written to a char or varchar column in the Teradata database is converted to your specified character set.

When writing a varchar(n) field, the **terawrite** operator schema type is **ustring[n * max]** where max is the maximum size in bytes of the largest codepoint for the character set specified by `-db_cs`.

No other environment variables are required to use the Teradata operators. All the required libraries are in `/usr/lib` which should be on your PATH.

To speed up the start time of the load process slightly, you can set the **APT_TERA_NO_PERM_CHECKS** environment variable to bypass permission checking on several system tables that need to be readable during the load process.

### The **teraread** Operator

The **teraread** operator sets up a connection to a Teradata database and reads the results of a query into an Orchestrate data set. As the operator requests records, the result set is translated row by row into the Orchestrate data set. The operator converts the data type of Teradata columns to corresponding Orchestrate data types, as listed in Table 221 on Page 37-6.

#### Data Flow Diagram

```
  +---+     +---+    +---+
  |   |     |   |    |   |
  +---+     +---+    +---+
  Teradata query

  |   |     |   |    |   |
  +---+     +---+    +---+
  | teraread |     |   |
  +----------+     +---+

  |   |     |   |    |   |
  +---+     +---+    +---+
  | output data set |
```

#### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of output data sets</td>
<td>1</td>
</tr>
<tr>
<td>Output interface schema</td>
<td>determined by the SQL query.</td>
</tr>
</tbody>
</table>
By default, Orchestrate reads from the default database of the user who performs the read operation. If this database is not specified in the Teradata profile for the user, the user name is the default database. You can override this behavior by means of the `-dbname` option. You must specify either the `-query` or the `-table` option.

The operator can optionally pass `-open` and `-close` option commands to the database. These commands are run once by Teradata on the conductor node either before a query is executed or after the read operation is completed.

### Specifying the Query

You read the results of a Teradata query by specifying either the `-query` or the `-table` option and the corresponding suboptions.

- The `-query` option defines an SQL query to read Teradata data. Orchestrate evaluates the query and performs the read operation.
- The `-table` option specifies a table, the rows to include in or exclude from a read operation, and the columns to read. These specifications are then used internally to construct a query.

By default, the operator displays a progress message for every 100,000 records per partition it processes, for example:

```
##I TTER000458 16:00:50(009) <teraread,0> 98300000 records processed.
```

However, you can either change the interval or disable the messages by means of the `-progressInterval` option.

### Column Name and Data Type Conversion

A Teradata result set has one or more rows and one or more columns. A Teradata result set is converted to an Orchestrate data set as follows:
• The rows of a Teradata result set correspond to the records of an Orchestrate data set.
• The columns of a Teradata row correspond to the fields of an Orchestrate record.
• The name and data type of a Teradata column determine the name and data type of the corresponding Orchestrate field.
• Both Teradata columns and Orchestrate fields support nulls, and a null contained in a Teradata column is stored as a null in the corresponding Orchestrate field.

Orchestrate gives the same name to its fields as the Teradata column name. However, while Orchestrate field names appear in either upper or lower case, Teradata column names appear only in upper case.

The **teraread** operator automatically converts Teradata data types to Orchestrate data types, as in the following table:

<table>
<thead>
<tr>
<th>Teradata Data Type</th>
<th>Orchestrate Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte(size)</td>
<td>raw[size]</td>
</tr>
<tr>
<td>byteint</td>
<td>int8</td>
</tr>
<tr>
<td>char(size)</td>
<td>string[size]</td>
</tr>
<tr>
<td>date</td>
<td>date</td>
</tr>
<tr>
<td>decimal(m,n)</td>
<td>decimal(m,n)</td>
</tr>
<tr>
<td>double precision</td>
<td>dfloat</td>
</tr>
<tr>
<td>float</td>
<td>dfloat</td>
</tr>
<tr>
<td>graphic(size)</td>
<td>raw[max=size]</td>
</tr>
<tr>
<td>Integer</td>
<td>int32</td>
</tr>
<tr>
<td>long varchar</td>
<td>string[max=size]</td>
</tr>
<tr>
<td>long varchargraphic</td>
<td>raw[max=size]</td>
</tr>
<tr>
<td>numeric(m,n)</td>
<td>decimal(m,n)</td>
</tr>
<tr>
<td>real</td>
<td>dfloat</td>
</tr>
<tr>
<td>smallint</td>
<td>int16</td>
</tr>
<tr>
<td>time</td>
<td>not supported</td>
</tr>
</tbody>
</table>


You can use the `modify` operator to perform explicit data type conversion after the Teradata data type has been converted to an Orchestrate data type.

### teraread Restrictions

The `teraread` operator is a distributed FastExport and is subject to all the restrictions on FastExport. Briefly, these are:

- There is a limit to the number of concurrent FastLoad and FastExport jobs. Each use of the Teradata operators (`teraread` and `terawrite`) counts toward this limit.
- Aggregates and most arithmetic operators in the SELECT statement are not allowed.
- The use of the USING modifier is not allowed.
- Non-data access (that is, pseudo-tables like DATE or USER) is not allowed.
- Single-AMP requests are not allowed. These are SELECTs satisfied by an equality term on the primary index or on a unique secondary index.

### Syntax and Options

The syntax of the `teraread` operator is given below. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose it in single quotes.

```
  teraread
    . table table_name [-filter filter] [-selectlist list]  
    | .query sqlquery
    [ .db_cs character_set]
    . server servername
    . db options
        '{ . user = username . password = password
        [ . sessionsperplayer = nn] [ . requestedsessions = nn]}'
```

<table>
<thead>
<tr>
<th>Teradata Data Type</th>
<th>Orchestrate Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp</td>
<td>not supported</td>
</tr>
<tr>
<td>varbyte(size)</td>
<td>raw[max=size]</td>
</tr>
<tr>
<td>varchar(size)</td>
<td>string[max=size]</td>
</tr>
<tr>
<td>vargraphic(size)</td>
<td>raw[max=size]</td>
</tr>
</tbody>
</table>

You can use the `modify` operator to perform explicit data type conversion after the Teradata data type has been converted to an Orchestrate data type.
You must specify either the `-query` or the `-table` option. You must also specify the `-server` option to supply the server name, and specify the `-dboptions` option to supply connection information to log on to Teradata.

### Table 222: `teraread` Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-close</code></td>
<td><code>-close</code> close_command</td>
</tr>
<tr>
<td></td>
<td>Optionally specifies a Teradata command to be run once by Teradata on</td>
</tr>
<tr>
<td></td>
<td>the conductor node after the query has completed.</td>
</tr>
<tr>
<td><code>-db_cs</code></td>
<td><code>-db_cs</code> character_set</td>
</tr>
<tr>
<td></td>
<td>Optionally specify the character set to be used for mapping strings</td>
</tr>
<tr>
<td></td>
<td>between the database and the Teradata operators. The default value</td>
</tr>
<tr>
<td></td>
<td>is Latin1. See “Specifying an Orchestrate ustring Character Set” on</td>
</tr>
<tr>
<td></td>
<td>page 37-3 for information on the data affected by this setting.</td>
</tr>
<tr>
<td><code>-dbname</code></td>
<td><code>-dbname</code> database_name</td>
</tr>
<tr>
<td></td>
<td>By default, the read operation is carried out in the default database</td>
</tr>
<tr>
<td></td>
<td>of the Teradata user whose profile is used. If no default database</td>
</tr>
<tr>
<td></td>
<td>is specified in that user’s Teradata profile, the user name is the</td>
</tr>
<tr>
<td></td>
<td>default database. This option overrides the default.</td>
</tr>
<tr>
<td></td>
<td>If you supply the database_name, the database to which it refers</td>
</tr>
<tr>
<td></td>
<td>must exist and you must have the necessary privileges.</td>
</tr>
</tbody>
</table>
You must specify both the `username` and `password` with which you connect to Teradata.

The value of `-sessionsperplayer` determines the number of connections each player has to Teradata. Indirectly, it also determines the number of players. The number selected should be such that (`-sessionsperplayer` * number of nodes * number of players per node) equals the total requested sessions. The default is 2.

Setting the value of `-sessionsperplayer` too low on a large system can result in so many players that the step fails due to insufficient resources. In that case, `-sessionsperplayer` should be increased.

The value of the optional `-requestedsessions` is a number between 1 and the number of vprocs in the database. The default is the maximum number of available sessions.

Optionally specifies a Teradata command run once by Teradata on the conductor node before the query is initiated.

By default, the operator displays a progress message for every 100,000 records per partition it processes. Specify this option either to change the interval or disable the messages. To change the interval, specify a new number of records per partition. To disable the messages, specify 0.

Specifies a valid Teradata SQL query in single quotes that is submitted to Teradata. The query must be a valid Teradata SQL query that the user has the privileges to run. Do not include formatting characters in the query.

A number of other restrictions apply; see “teraread Restrictions” on page 37-7.

You must specify a Teradata server.
The terawrite Operator

The `terawrite` operator sets up a connection to a Teradata database to write data to it from an Orchestrate data set. As the operator writes records, the Orchestrate data set is translated row by row to a Teradata table. The operator converts the data type of Orchestrate fields to corresponding Teradata types, as listed in Table 224. The mode of the operator determines how the records of a data set are inserted in the table.

### Table 222  teraread Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-table</code></td>
<td><code>-table table_name [-filter filter] [-selectlist list]</code></td>
</tr>
</tbody>
</table>

Specifies the name of the Teradata table to read from. The table must exist, and the user must have the necessary privileges to read it. The `teraread` operator reads the entire table, unless you limit its scope by means of the `-filter` or `-selectlist` option, or both options.

The `-filter` suboption optionally specifies selection criteria to be used as part of an SQL statement's WHERE clause. Do not include formatting characters in the query. A number of other restrictions apply; see “teraread Restrictions” on page 37-7.

The `-selectlist` suboption optionally specifies a list of columns to read. The items of the list must appear in the same order as the columns of the table.
Data Flow Diagram

By default, Orchestrate writes to the default database of the user name used to connect to Teradata. The `userid` under which the step is running is not the basis of write authorization. If no default database is specified in that user’s Teradata profile, the user name is the default database. You can override this behavior by means of the `-dbname` option. You must specify the table.

By default, the operator displays a progress message for every 100,000 records per partition it processes, for example:

```
##I TTR000458 16:00:50(009) <terawrite,0> 98300000 records processed.
```

The `-progressInterval` option can be used to change the interval or disable the messages.

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input data sets</td>
<td>1</td>
</tr>
<tr>
<td>Input interface schema</td>
<td>derived from the input data set</td>
</tr>
<tr>
<td>Execution mode</td>
<td>parallel (default) or sequential</td>
</tr>
<tr>
<td>Partitioning method</td>
<td>same</td>
</tr>
<tr>
<td></td>
<td>You can override this partitioning method. However, a partitioning method of entire is not allowed.</td>
</tr>
</tbody>
</table>

Note

The `terawrite` operator is not compatible with Parallel CLI, which should not be enabled. In any case, no performance improvements can be obtained by using Parallel CLI with `terawrite`.

By default, Orchestrate writes to the default database of the user name used to connect to Teradata. The `userid` under which the step is running is not the basis of write authorization. If no default database is specified in that user’s Teradata profile, the user name is the default database. You can override this behavior by means of the `-dbname` option. You must specify the table.

By default, the operator displays a progress message for every 100,000 records per partition it processes, for example:

```
##I TTR000458 16:00:50(009) <terawrite,0> 98300000 records processed.
```

The `-progressInterval` option can be used to change the interval or disable the messages.
Column Name and Data Type Conversion

Here is how Orchestrate data sets are converted to Teradata tables:

- An Orchestrate data set corresponds to a Teradata table.
- The records of the Orchestrate data set correspond to the rows of the Teradata table.
- The fields of the Orchestrate record correspond to the columns of the Teradata row.
- The name and data type of an Orchestrate field are related to those of the corresponding Teradata column. However, Orchestrate field names are case sensitive and Teradata column names are not. Make sure that the field names in the data set are unique, regardless of case.
- Both Orchestrate fields and Teradata columns support nulls, and an Orchestrate field that contains a null is stored as a null in the corresponding Teradata column.

The `terawrite` operator automatically converts Orchestrate data types to Teradata data types as shown in the following table:

### Table 224  terawrite Operator Data Type Conversions

<table>
<thead>
<tr>
<th>Orchestrate Data Type</th>
<th>Teradata Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>date</td>
</tr>
<tr>
<td>decimal(p,s)</td>
<td>numeric(p,s)</td>
</tr>
<tr>
<td>dfloat</td>
<td>double precision</td>
</tr>
<tr>
<td>int8</td>
<td>byteint</td>
</tr>
<tr>
<td>int16</td>
<td>smallint</td>
</tr>
<tr>
<td>int32</td>
<td>integer</td>
</tr>
<tr>
<td>int64</td>
<td>unsupported</td>
</tr>
<tr>
<td>raw</td>
<td>varbyte(default)</td>
</tr>
<tr>
<td>raw[fixed_size]</td>
<td>byte(fixed_size)</td>
</tr>
<tr>
<td>raw[max=n]</td>
<td>varbyte(n)</td>
</tr>
<tr>
<td>sfloat</td>
<td>unsupported</td>
</tr>
<tr>
<td>string</td>
<td>varchar(default length)</td>
</tr>
<tr>
<td>string[fixed_size]</td>
<td>char(fixed_size)</td>
</tr>
</tbody>
</table>
When `terawrite` tries to write an unsupported data type to a Teradata table, the operation terminates with an error.

You can use the `modify` operator to perform explicit data type conversions before the write operation is initiated.

**Correcting Load Errors**

The `terawrite` operator presents a summary of the number of rows processed, the number of errors, and the elapsed time. This summary is useful for determining the state of the operator upon completion, since `terawrite` *succeeds* even if errors in the data were encountered.

If there were errors, the summary gives a limited indication of the number and nature of the errors; in particular, the field name and Teradata error code is given.

Since FastLoad creates error tables whenever it is run (see your FastLoad documentation for details), you may use BTEQ to examine the relevant error table and then correct the records that failed to load. FastLoad does not load duplicate records, however, such records are not loaded into the error tables.

The `remdup` operator can be used to remove the duplicate records prior to using `terawrite`.

**Write Modes**

The write mode of the operator determines how the records of the data set are inserted into the destination table. The write mode can have one of the following values:

- **append**: The `terawrite` operator appends new rows to the table; the database user who writes in this mode must have TABLE CREATE privileges and INSERT privileges on the database being written to. This is the default mode.
• **create**: The `terawrite` operator creates a new table. The database user who writes in this mode must have TABLE CREATE privileges. If a table exists with the same name as the one you want to create, the step that contains `terawrite` terminates in error.

• **replace**: The `terawrite` operator drops the existing table and creates a new one in its place. The database user who writes in this mode must have TABLE CREATE and TABLE DELETE privileges. If a table exists with the same name as the one you want to create, it is overwritten.

• **truncate**: The `terawrite` operator retains the table attributes (including the schema) but discards existing records and appends new ones. The database user who writes in this mode must have DELETE and INSERT privileges on that table.

**Note**
The `terawrite` operator cannot write to tables that have indexes other than the primary index defined on them. This applies to the `append` and `truncate` modes. Override the default `append` mode by means of the `-mode` option.

**Writing Fields**

Fields of the Orchestrate data set are matched by name and data type to columns of the Teradata table but do not have to appear in the same order.

The following rules determine which fields of an Orchestrate data set are written to a Teradata table:

• If the Orchestrate data set contains fields for which there are no matching columns in the Teradata table, the step containing the operator terminates. However, you can remove an unmatched field from processing: either specify the `-drop` option and the operator drops any unmatched field, or use the `modify` operator to drop the extra field or fields before the write operation begins.

• If the Teradata table contains a column that does not have a corresponding field in the Orchestrate data set, Teradata writes the column's default value into the field. If no default value is defined for the Teradata column, Teradata writes a null. If the field is not nullable, an error is generated and the step is terminated.

**Limitations**

Write operations have the following limitations:

• A Teradata row may contain a maximum of 256 columns.

• While the names of Orchestrate fields can be of any length, the names of Teradata columns cannot exceed 30 characters. If you write Orchestrate fields
that exceed this limit, use the `modify` operator to change the Orchestrate field name.

- Orchestrate assumes that the `terawrite` operator writes to buffers whose maximum size is 32 KB. However, you can override this and enable the use of 64 KB buffers by setting the environment variable `APT_TERA_64K_BUFFERS`.

- The Orchestrate data set cannot contain fields of the following types:
  - `int64`
  - Unsigned integer of any size
  - String, fixed- or variable-length, longer than 32 KB
  - Raw, fixed- or variable-length, longer than 32 KB
  - Subrecord
  - Tagged aggregate
  - Vectors

  If `terawrite` tries to write a data set whose fields contain a data type listed above, the write operation is not begun and the step containing the operator fails. You can convert unsupported data types by using the `modify` operator.

**Restrictions**

The `terawrite` operator is a distributed FastLoad and is subject to all the restrictions on FastLoad. In particular, there is a limit to the number of concurrent FastLoad and FastExport jobs. Each use of the `teraread` and `terawrite` counts toward this limit.

**Syntax and Options**

You must specify the `-table` option, the `-server` option to supply the server name, and the `-dboptions` option to supply connection information to log on to Teradata. Option values you supply are shown in italic typeface. When your value contains a space or a tab character, you must enclose it in single quotes.

```
terawrite
  -table table_name [-selectlist list]
  -server servername
  -db_cs character_set
  -dboptions
    '{-user = username -password = password
     [-sessionsperplayer = nn] [-requestedsessions = nn]}'
  [-close close_command]
  [-dbname database_name]
```
Table 225  terawrite Operator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage and Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-close</td>
<td>close close_command</td>
</tr>
<tr>
<td></td>
<td>Optionally specify a Teradata command to be parsed and executed by Teradata on all processing nodes after the table has been populated.</td>
</tr>
<tr>
<td>-db_cs</td>
<td>db_cs character_set</td>
</tr>
<tr>
<td></td>
<td>Optionally specify the character set to be used for mapping strings between the database and the Teradata operators. The default value is Latin1. See “Specifying an Orchestrate ustring Character Set” on page 37-3 for information on the data affected by this setting.</td>
</tr>
<tr>
<td>-dbname</td>
<td>dbname database_name</td>
</tr>
<tr>
<td></td>
<td>By default, the write operation is carried out in the default database of the Teradata user whose profile is used. If no default database is specified in that user’s Teradata profile, the user name is the default database. If you supply the database name, the database to which it refers must exist and you must have necessary privileges.</td>
</tr>
<tr>
<td>-dboptions</td>
<td>{ -user = username,</td>
</tr>
<tr>
<td></td>
<td>-password = password</td>
</tr>
<tr>
<td></td>
<td>[ -sessionsperplayer = nn]</td>
</tr>
<tr>
<td></td>
<td>[ -requestedsessions = nn] }</td>
</tr>
<tr>
<td></td>
<td>You must specify both the user name and password with which you connect to Teradata.</td>
</tr>
<tr>
<td></td>
<td>The value of -sessionsperplayer determines the number of connections each player has to Teradata. Indirectly, it also determines the number of players. The number selected should be such that (sessionsperplayer * number of nodes * number of players per node) equals the total requested sessions. The default is 2.</td>
</tr>
<tr>
<td></td>
<td>Setting the value of -sessionsperplayer too low on a large system can result in so many players that the step fails due to insufficient resources. In that case, the value for -sessionsperplayer should be increased.</td>
</tr>
<tr>
<td></td>
<td>The value of the optional -requestedsessions is a number between 1 and the number of vprocs in the database.</td>
</tr>
</tbody>
</table>
The terawrite Operator

Chapter 37

The Teradata Interface Library

- **-drop**
  - This optional flag causes the operator to silently drop all unmatched input fields.

- **-mode**
  - **append**: Specify this option and terawrite appends new records to the table. The database user must have TABLE CREATE privileges and INSERT privileges on the table being written to. This mode is the default.
  - **create**: Specify this option and terawrite creates a new table. The database user must have TABLE CREATE privileges. If a table exists of the same name as the one you want to create, the step that contains terawrite terminates in error.
  - **replace**: Specify this option and terawrite drops the existing table and creates a new one in its place; the database user must have TABLE CREATE and TABLE DELETE privileges. If a table exists of the same name as the one you want to create, it is overwritten.
  - **truncate**: Specify this option and terawrite retains the table attributes, including the schema, but discards existing records and appends new ones. The database user must have DELETE and INSERT privileges on the table.

- **-open**
  - Optionally parse a Teradata command to be parsed and executed by Teradata on all processing nodes before the table is populated.

- **-primaryindex**
  - Optionally specify a comma-separated list of field names that become the primary index for tables. Format the list according to Teradata standards and enclose it in single quotes.

  For performance reasons, the Orchestrate data set should not be sorted on the primary index. The primary index should not be a smallint, or a field with a small number of values, or a high proportion of null values. If no primary index is specified, the first field is used. All the considerations noted above apply to this case as well.

- **-progressInterval**
  - By default, the operator displays a progress message for every 100,000 records per partition it processes. Specify this option either to change the interval or to disable the messages. To change the interval, specify a new number of records per partition. To disable the messages, specify 0.

- **-server**
  - You must specify the name of a Teradata server.

---

Table 225 terawrite Operator Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage and Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-drop</td>
<td>-drop</td>
</tr>
<tr>
<td></td>
<td>This optional flag causes the operator to silently drop all unmatched input fields.</td>
</tr>
<tr>
<td>-mode</td>
<td>-mode append</td>
</tr>
<tr>
<td></td>
<td>append: Specify this option and terawrite appends new records to the table. The database user must have TABLE CREATE privileges and INSERT privileges on the table being written to. This mode is the default.</td>
</tr>
<tr>
<td></td>
<td>create: Specify this option and terawrite creates a new table. The database user must have TABLE CREATE privileges. If a table exists of the same name as the one you want to create, the step that contains terawrite terminates in error.</td>
</tr>
<tr>
<td></td>
<td>replace: Specify this option and terawrite drops the existing table and creates a new one in its place; the database user must have TABLE CREATE and TABLE DELETE privileges. If a table exists of the same name as the one you want to create, it is overwritten.</td>
</tr>
<tr>
<td></td>
<td>truncate: Specify this option and terawrite retains the table attributes, including the schema, but discards existing records and appends new ones. The database user must have DELETE and INSERT privileges on the table.</td>
</tr>
<tr>
<td>-open</td>
<td>-open open_command</td>
</tr>
<tr>
<td></td>
<td>Optionally specify a Teradata command to be parsed and executed by Teradata on all processing nodes before the table is populated.</td>
</tr>
<tr>
<td>-primaryindex</td>
<td>-primaryindex = 'field1, field2, ... fieldn'</td>
</tr>
<tr>
<td></td>
<td>Optionally specify a comma-separated list of field names that become the primary index for tables. Format the list according to Teradata standards and enclose it in single quotes.</td>
</tr>
<tr>
<td></td>
<td>For performance reasons, the Orchestrate data set should not be sorted on the primary index. The primary index should not be a smallint, or a field with a small number of values, or a high proportion of null values. If no primary index is specified, the first field is used. All the considerations noted above apply to this case as well.</td>
</tr>
<tr>
<td>-progressInterval</td>
<td>-progressInterval number</td>
</tr>
<tr>
<td></td>
<td>By default, the operator displays a progress message for every 100,000 records per partition it processes. Specify this option either to change the interval or to disable the messages. To change the interval, specify a new number of records per partition. To disable the messages, specify 0.</td>
</tr>
<tr>
<td>-server</td>
<td>-server servername</td>
</tr>
<tr>
<td></td>
<td>You must specify the name of a Teradata server.</td>
</tr>
<tr>
<td>Option</td>
<td>Usage and Meaning</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| -stringlength | `stringlength length`  
Optionally specify the maximum length of variable-length raw or string fields. The default length is 32 bytes. The upper bound is slightly less than 32 KB. |
| -table   | `tablename [selectlist list]`  
Specify the name of the table to write to. The table name must be a valid Teradata table name.  
`selectlist` optionally specifies a list that determines which fields are written. If you do not supply the list, `terawrite` writes to all fields. Do not include formatting characters in the list. |
Index

A
Adobe Acrobat Reader
  included in the Orchestrate installation xxxix
  searching for text in Orchestrate documentation xxxix
aggtorec restructure operator 30
  example
  with multiple key options 30
  with toplevelkeys option 30
  without toplevelkeys option 30
  properties 30
  syntax and options 30
APT_ORAUPsert_COMMIT_ROW_INTERVAL 36
APT_ORAUPsert_COMMIT_TIME_INTERVAL 36
APT_SYNCSORT_DIR 31
assistance xl
associativity of operators 21

determining differences 23
example
  dropping output results 28
  general 28
  key and value fields 23
  properties 22
  syntax and options 24
  transfer behavior 23
COBOL schema conversion
  see readcobol utility
collecting key 29
collecting keys
  for sortmerge collecting 29
  collection methods
  See collection operators
collection operators
  roundrobin 29
  sortmerge 29
  See also individual operator entries
  collector
  ordered 29
  compare operator 31
  data-flow diagram 32
  example
  sequential execution 37
  properties 32
  restrictions 33
  results field 33
  syntax and options 34
compressing data
  See pcompress operator
copy operator
  data-flow diagram 42
  example
  copying a data set twice 45
  sequential execution 45

changeapply operator 11
  actions on change records 13
  data-flow diagram 12
  example 18
  key comparison fields 14
  properties 12
  schemas 14
  syntax and options 15
  transfer behavior 14
changecapture operator 21
  data-flow diagram 22
Index

preventing duplicate copy operator removal 4 3
properties 4 2
syntax and options 4 3
copying last N records to output
See tail operator
copying multiple data sets to output
See sequence operator
crosstab statistics operator 33 4
aggregation modes 33 11
data-flow diagram 33 7
example
  with clustered keys 33 14
  without clustered keys 33 13
generating statistics 33 13
grouping keys 33 5
grouping overview 33 4
limiting memory consumption 33 13
properties 33 7
syntax and options 33 8
crosstabulation
  See crosstab statistics operator
customer support  xl
customer support services  ii  xl

D

DB2
  configuring Orchestrate access 34 3, 34 53
    APT_DBNAME 34 4
    DB2 required privileges 34 4
    db2nodes.cfg system file 34 4
db2lookup operator 34 47
environment variable
    DB2INSTANCE 34 4
joining DB2 tables and an Orchestrate data set 34 3, 34 47
operators that modify Orchestrate data types 34 23
Orchestrate field conventions 34 22
specifying DB2 table rows 34 13
specifying the SQL SELECT statement 34 13
using a node map 34 53
DB2 interface operators
db2load 34 20
db2lookup 34 47
db2part 34 44
db2read 34 9
db2upsert 34 38
db2write 34 20
establishing a remote connection 34 4
handling # and $ in column names 34 5
national language support 34 7
running multiple operators in a single step 34 7
using the-padchar option 34 6
See also individual operator entries
DB2 partitioning
  db2part operator 34 44
DB2INSTANCE 34 4
db2load operator
data-flow diagram 34 20
DB2 field conventions 34 22
default SQL INSERT statement 34 22
example
  appending data to a DB2 table. 34 33
  DB2 table with an unmatched column 34 36
  unmatched Orchestrate fields 34 35
  writing data in truncate mode 34 34
  matching fields 34 26
  operators that modify Orchestrate data types for DB2 34 23
  options 34 27
  properties 34 20
  requirements for using 34 33
  specifying the select list 34 22
  translation anomalies 34 52
  write modes 34 25
db2lookup operator 34 47
example 34 51
properties 34 48
syntax and options 34 48
db2nodes command syntax 34 53
db2part operator 34 44
example 34 46
syntax and options 34 44
db2read operator
data-flow diagram 34 9
example
  reading a DB2 table sequentially with the query option 34 19
  reading a DB2 table with the table option 34 17
  reading a table in parallel with the query option 34 19
  operator action 34 10

38 2

Orchestrator 7.0 Operators Reference
Orchestrate conversion
  DB2 column names 34 11
  properties 34 9
  specifying DB2 table rows 34 13
  specifying the DB2 table name 34 13
  specifying the SQL SELECT statement 34 13
  syntax and options 34 14
  translation anomaly 34 52
db2upsert operator
  operator action 34 39
  partitioning 34 38
  properties 34 39
  syntax and options 34 40
db2write operator 34 20
  data-flow diagram 34 20
  DB2 field conventions 34 22
  default SQL INSERT statement 34 22
  example
    appending data to a DB2 table 34 33
    DB2 table with an unmatched column 34 36
    unmatched Orchestrate fields 34 35
    writing data in truncate mode 34 34
    matching fields 34 26
  operators that modify Orchestrate data types for DB2 34 23
  properties 34 20
  specifying the select list 34 22
  syntax and options 34 27
  translation anomalies 34 52
  write modes 34 25
diff operator 5 1
  data-flow diagram 5 2
  example
    dropping output results 5 9
    typical 5 9
    properties 5 2
    syntax and options 5 6
displaying records on the screen
  See peek operator
documentation
  searching for text xxxix
  syntax conversions xxxviii
documentation titles for Orchestrate xxxviii
data-flow diagram 6 2
encoded data sets and partitioning 6 4
encoding Orchestrate data sets 6 4
example 6 5
processing encoded data sets 6 4
properties 6 2
encoded data sets and partitioning 6 4
encoded-compressed data sets 6 4
encoding operator
  See encode operator
encoding Orchestrate data sets 6 4
entire partitioner 27 3
  data-flow diagram 27 5
  properties 27 5
  syntax 27 5
  using 27 4
environment variable
  APT_DBNAME 34 4
  APT_ORAUPSERT_COMMIT_ROW_INTERVAL 36 35
  APT_ORAUPSERT_COMMIT_TIME_INTERVAL 36 35
  APT_SYNCSORT_DIR 31 20
  DB2INSTANCE 34 4
export utility operator 24 1
  data set versus export schema 24 9
  data-flow diagram 24 2
  example
    exporting to a single file 24 14
    exporting to multiple files 24 16
  export formats 24 3
  import/ export properties
    See import/ export utility properties
  properties 24 2
  specifying data destination 24 11
  destination program input 24 13
  file sets 24 11
  files and named pipes 24 11
  list of destination programs 24 13
  nodes and directories 24 13
  specifying export schema 24 10
  default exporting of all fields 24 10
  defining properties 24 10
  exporting all fields with overrides and new properties 24 10
  exporting selected fields 24 10
  specifying record and field layout 24 9
  syntax and options 24 4

E
encode operator 6 1
### F

**field transform**
- field naming **33** **48**
- forward **33** **46**
- identity **33** **67**
- inverse **33** **46**
- nominal transform **33** **68**
- ordinal transform **33** **73**
- using **33** **47**
- Z-score transform **33** **57**, **33** **59**
- Z-sigmoid transform **33** **62**
- Z-symmetric transform **33** **62**

**field_export restructure operator**
- data-flow diagram **30** **10**
- example **30** **11**
- properties **30** **10**
- transfer behavior **30** **10**, **30** **13**

**field_import restructure operator** **30** **13**
- data-flow diagram **30** **13**
- example **30** **15**
- properties **30** **14**
- syntax and options **30** **14**
- transfer behavior **30** **14**

**fields**
- changing data type **13** **7**
- data-type conversion errors **13** **9**
- data-type conversions
  - date field **13** **10**, **21** **28**
  - decimal field **13** **14**
  - raw field length extraction **13** **17**
  - string conversions and lookup tables **13** **23**, **21** **41**
  - string field **13** **18**
  - time field **13** **26**
  - timestamp field **13** **29**
- default data-type conversion **13** **7**
- duplicating with a new name **13** **6**
- explicitly converting the data type **13** **10**
- keeping and dropping **13** **6**
- name change example **13** **5**
- renaming **13** **6**
- transfer behavior of the modify operator **13** **4**

**fieldtransform statistics operator** **33** **46**
- configuring Z-score transforms **33** **59**
- data-flow diagram **33** **49**
- general transformation process **33** **47**
- nulls **33** **50**
- performing identity transform **33** **67**
- performing Z-sigmoid and Z-symmetric transforms **33** **62**
- properties **33** **50**
- standard deviation of zero **33** **66**
- syntax and options **33** **50**
- transforming ordinal fields **33** **73**
- Z-score transform
  - performing **33** **57**
  - using **33** **58**

**filter operator** **7** **2**
- data-flow diagram **7** **2**
- example
  - comparing two fields **7** **8**
  - evaluating input records **7** **8**
  - filtering wine catalog recipients **7** **9**
  - testing for a null **7** **8**
- input data types **7** **6**
- Job Monitor information **7** **5**
- customizing messages **7** **6**
- example messages **7** **5**
- properties **7** **2**
- supported boolean expressions **7** **7**

**fullouterjoin operator** **28** **14**
- example **28** **16**
- output data set **28** **15**
- syntax and options **28** **15**

**funnel operator** **8** **1**
- data-flow diagram **8** **2**
- properties **8** **2**
- syntax and options **8** **6**

### G

**general operators**
- changeapply **1** **1**
- changecapture **2** **1**
- compare **3** **1**
- diff **5** **1**
- encode **6** **1**
- filter **7** **2**
- funnel **8** **1**
- generator **9** **1**
- head **10** **1**
- lookup **11** **1**
- merge **12** **2**
- modify **13** **2**
- pcompress **14** **1**
- peek **15** **1**
- remdup **16** **1**
sample 17 1
sequence 18 1
sortfunnel 8 1
switch operator 19 1
tail 20 1
See also individual operator entries
generator operator 9 1
data-flow diagram 9 2
defining record schema 9 7
example
general 9 5
parallel execution 9 6
using with an input data set 9 6
generator options for Orchestrate data types 9 8
properties 9 2
record schema
data fields 9 10
cycle option 9 10
epoch option 9 10
function option 9 10
invalids option 9 10
random option 9 10
decimal fields 9 11
cycle option 9 11
invalids option 9 11
random option 9 11
zeros option 9 11
null fields 9 14
nulls option 9 14
nullseed option 9 15
numeric fields 9 8
cycle option 9 9
random option 9 9
raw fields 9 11
max parameter 9 11
string fields 9 12
alphabet property 9 12
cycle option 9 13
max parameter 9 12
time fields 9 13
cycle option 9 13
invalids option 9 14
random option 9 13
scale option 9 13
timestamp fields 9 14
supported data types 9 4
syntax and options 9 3
usage details 9 4
group statistics operator
creating a summary 33 19
data-flow diagram 33 22
defining a group 33 18
example
creating summary statistics 33 32
regrouping data 33 32
specifying a weight field 33 31
two-phase grouping with hash mode 33 35
using in hash mode 33 29
using in sort mode 33 30
grouping operation 33 17
modes 33 23
properties 33 22
summary statistics 33 33
maximum value 33 33
mean 33 33
minimum value 33 33
number of missing values 33 33
number of records 33 33
record schema 33 33
sum of the corrected squares 33 34
sum of the weights 33 33
syntax and options 33 24
grouping data
grouping keys 33 19
groups defined 33 18
operation 33 17
uses 33 16
with the crosstab statistics operator 33 4
See also group statistics operator
grouping keys 33 19
defined 33 18
examples 33 19
groups 33 18
defined 33 18
properties 33 18
gzip UNIX utility
using with the encode operator 6 5

H
hash partitioner 27 5
data-flow diagram 27 8
example 27 7
Index

properties 27 8
syntax and option 27 9
using 27 8
head operator 10 1
data-flow diagram 10 2
example
default behavior 10 3
extracting records from a large data set 10 4
locating a single record 10 4
properties 10 2
syntax and options 10 2
hplread INFORMIX interface operator 35 16
data-flow diagram 35 16
example 35 19
properties 35 17
special features 35 16
syntax and options 35 17
hplwrite INFORMIX interface operator 35 19
data-flow diagram 35 20
listing of examples 35 22
properties 35 20
special features 35 19
syntax and options 35 20
IBM format files 23 3
exporting data 24 3
identity transform
defined 33 67
forward 33 68
inverse 33 68
import utility operator 23 1
data-flow diagram 23 2
example
importing from a single data file 23
12
importing from multiple data files 23
13
import and export properties
See import/ export utility properties
import source file data types 23 3
properties 23 2
specifying schema 23 9
specifying the layout of imported data 23
9
specifying the source of imported data 23
9
file patterns 23 11
file sets 23 10
files and named pipes 23 10
list of source programs 23 11
nodes and directories 23 11
source program output 23 11
syntax and options 23 3
import/ export utility introduction 22 2–22 22
ASCII and EBCDIC conversion 22 15
ASCII to EBCDIC 22 20
EBCDIC to ASCII 22 16
table irregularities 22 22
error handling 22 14
failure 22 14
warning 22 15
format of imported data 22 2
implicit import and export 22 11
default export schema 22 13
default import schema 22 12
no specified schema 22 11
overriding defaults 22 13
record schema 22 3
comparison of partial and complete 22 7
defining complete schema 22 5
defining partial schema 22 7
export example 22 4
exporting with partial schema 22 9
import example 22 3
See also import operator, export operator, and import/ export utility properties
import/ export utility properties 25 3–25 66
field properties 25 6
date 25 10
decimal 25 9
nullable 25 12
numeric 25 7
raw 25 11
string 25 8
tagged subrecord 25 13
time 25 10
timestamp 25 11
ustring 25 9
vector 25 11
property description 25 17
actual_length 25 17
ascii 25 18
big_endian 25 19
binary 25 19
<table>
<thead>
<tr>
<th>Property</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_format</td>
<td>25</td>
</tr>
<tr>
<td>check_intact</td>
<td>25</td>
</tr>
<tr>
<td>date_format</td>
<td>25</td>
</tr>
<tr>
<td>days_since</td>
<td>25</td>
</tr>
<tr>
<td>default</td>
<td>25</td>
</tr>
<tr>
<td>delim</td>
<td>25</td>
</tr>
<tr>
<td>delim_string</td>
<td>25</td>
</tr>
<tr>
<td>drop</td>
<td>25</td>
</tr>
<tr>
<td>ebcdic</td>
<td>25</td>
</tr>
<tr>
<td>export_ebcdic_as_ascii</td>
<td>25</td>
</tr>
<tr>
<td>fill</td>
<td>25</td>
</tr>
<tr>
<td>final_delim</td>
<td>25</td>
</tr>
<tr>
<td>final_delim_string</td>
<td>25</td>
</tr>
<tr>
<td>fix_zero</td>
<td>25</td>
</tr>
<tr>
<td>generate</td>
<td>25</td>
</tr>
<tr>
<td>import_ascii_as_ebcdic</td>
<td>25</td>
</tr>
<tr>
<td>in_format</td>
<td>25</td>
</tr>
<tr>
<td>intact</td>
<td>25</td>
</tr>
<tr>
<td>julian</td>
<td>25</td>
</tr>
<tr>
<td>link</td>
<td>25</td>
</tr>
<tr>
<td>little_endian</td>
<td>25</td>
</tr>
<tr>
<td>max_width</td>
<td>25</td>
</tr>
<tr>
<td>midnight_seconds</td>
<td>25</td>
</tr>
<tr>
<td>native_endian</td>
<td>25</td>
</tr>
<tr>
<td>nofix_zero</td>
<td>25</td>
</tr>
<tr>
<td>null_field</td>
<td>25</td>
</tr>
<tr>
<td>null_length</td>
<td>25</td>
</tr>
<tr>
<td>out_format</td>
<td>25</td>
</tr>
<tr>
<td>packed</td>
<td>25</td>
</tr>
<tr>
<td>padding</td>
<td>25</td>
</tr>
<tr>
<td>position</td>
<td>25</td>
</tr>
<tr>
<td>precision</td>
<td>25</td>
</tr>
<tr>
<td>prefix</td>
<td>25</td>
</tr>
<tr>
<td>print_field</td>
<td>25</td>
</tr>
<tr>
<td>quote</td>
<td>25</td>
</tr>
<tr>
<td>record_delim</td>
<td>25</td>
</tr>
<tr>
<td>record_delim_string</td>
<td>25, 13, 53</td>
</tr>
<tr>
<td>record_format</td>
<td>25</td>
</tr>
<tr>
<td>record_length</td>
<td>25</td>
</tr>
<tr>
<td>record_prefix</td>
<td>25</td>
</tr>
<tr>
<td>reference</td>
<td>25</td>
</tr>
<tr>
<td>round</td>
<td>25</td>
</tr>
<tr>
<td>scale</td>
<td>25</td>
</tr>
<tr>
<td>separate</td>
<td>25</td>
</tr>
<tr>
<td>skip</td>
<td>25</td>
</tr>
<tr>
<td>tagcase</td>
<td>25</td>
</tr>
<tr>
<td>text</td>
<td>25</td>
</tr>
<tr>
<td>time_format</td>
<td>25</td>
</tr>
<tr>
<td>timestamp_format</td>
<td>25</td>
</tr>
<tr>
<td>vector_prefix</td>
<td>25</td>
</tr>
<tr>
<td>width</td>
<td>25, 40, 25, 64</td>
</tr>
<tr>
<td>zoned</td>
<td>25</td>
</tr>
<tr>
<td>record-level properties</td>
<td>25</td>
</tr>
<tr>
<td>setting properties</td>
<td>25</td>
</tr>
<tr>
<td>defining field-level properties</td>
<td>25</td>
</tr>
<tr>
<td>defining record-level properties</td>
<td>25</td>
</tr>
<tr>
<td>table of all record and field properties</td>
<td>25</td>
</tr>
</tbody>
</table>

**INFORMIX**

- accessing: 35, 3
- configuring your environment: 35, 3

**INFORMIX interface operators**

- hplread: 35, 16
- hplwrite: 35, 19
- infxread: 35, 23
- xpsread: 35, 29
- xpswrite: 35, 31

See also individual operator entries

**INFORMIX interface read operators**

- column name conversion: 35, 5
- data type conversion: 35, 5
- data-flow diagram: 35, 3
- example: 35, 6
- operator action: 35, 4

**INFORMIX interface write operators**

- column name conversion: 35, 8
- data type conversion: 35, 9
- data-flow diagram: 35, 7
- dropping unmatched fields: 35, 11
- example:
  - appending data: 35, 12
  - handling unmatched fields: 35, 14
  - table with unmatched column: 35, 15
  - writing in truncate mode: 35, 13
- execution modes: 35, 8
- limitations:
  - column name length: 35, 11
  - field types: 35, 11
- write modes: 35, 10
- writing fields: 35, 11

infxread INFORMIX interface operator: 35, 23

- data-flow diagram: 35, 23
- properties: 35, 23
- syntax and options: 35, 24

infxwrite INFORMIX interface operator: 35, 26

- data-flow diagram: 35, 26
- listing of examples: 35, 28
- properties: 35, 26
syntax and options 35 27
innerjoin operator 28 7
  example 28 9
  syntax and option 28 7
inverse identity transform 33 67
inverse nominal transform 33 70
inverse ordinal transform 33 75
inverse Z-score transform 33 60
inverse Z-sigmoid transform 33 65
inverse Z-symmetric transform 33 65

J
join operators
  fullouterjoin 28 14
  general characteristics
    input data set requirements 28 5
    memory use 28 5
    multiple keys and values 28 5
    properties 28 4
    transfer behavior 28 4
  innerjoin 28 7
  leftouterjoin 28 9
  rightouterjoin 28 12
  See also individual operator entries

L
leftouterjoin operator 28 9
  example 28 11
  syntax and option 28 10
lookup operator 11 1
  create-only mode 11 8
  data-flow diagram 11 3
  dropping a field with the transform operator 11 12
  example 11 12
    handling duplicate fields 11 12
    interest rate lookup 11 11
    handling error conditions 11 11
    lookup with single table record 11 9
    lookup with two table records 11 10
  operator behavior 11 1
  properties 11 4
  renaming duplicate fields 11 12
  syntax and options 11 5
  table characteristics 11 9

M
makerangemap utility
  creating a range map 27 28
  example commands 27 29
  specifying sample size 27 28
  syntax and options 27 26
  using with the psort operator 31 32
  using with the range partitioner 27 28
  using with the tsort operator 31 17
  using with the writerangemap operator 27 26
makeubrec restructure operator 30 17
  data-flow diagram 30 17
  example 30 20
  properties 30 18
  subrecord length 30 19
  syntax and options 30 19
  transfer behavior 30 18
makevect restructure operator 30 21
  data-flow diagram 30 22
  example 30 24
  general 30 24
    missing input fields 30 24
    non-consecutive fields 30 23
  properties 30 22
  syntax and option 30 23
  transfer behavior 30 22
merge operator 12 2
  data-flow diagram 12 2
  example 12 16
    application scenario 12 16
    handling duplicate fields 12 14
      dropping 12 15
      renaming 12 15
    merging operation 12 9
    merging records 12 5
      example diagram 12 10
      master record and update record 12 6
      multiple update data sets 12 7
      missing records 12 18
        handling bad master records 12 18
        handling bad update records 12 19
    properties 12 3
    syntax and options 12 4
    typical data flow 12 8
modify operator 13 2
  aggregate schema components 13 36
  changing field data type 13 7
data-flow diagram  13  3
data-type conversion errors  13  9
data-type conversion table  13  38
data-type conversions
date field  13  10, 21  28
decimal field  13  14
raw field length extraction  13  17
string conversions and lookup tables  13  23, 21  41
string field  13  13
time field  13  26
timestamp field  13  29
default data-type conversion  13  7
duplicating a field with a new name  13  6
explicitly converting the data type  13  10
field name-change example  13  5
keeping and dropping fields  13  6
null support  13  31
partial schemas  13  35
properties  13  3
renaming fields  13  6
syntax and options  13  3
transfer behavior  13  4
vectors  13  35
modulus partitioner  27  10
data-flow diagram  27  11
example  27  12
properties  27  11
syntax and option  27  11

N

NaN values
and the group statistics operator  33  32
node maps
for DB2 interface operators  34  53
nominal transform  33  68
defined  33  68
false value  33  69
forward
example  33  72
false value  33  69
field naming  33  69
true value  33  69
inverse
calculating  33  71
example  33  73
field naming  33  71
performing  33  70
true value  33  69
nulls
field transform statistics operator  33  50
modify operator  13  31

O
operator
associativity  21  24
writerangemap  27  23
operators
collection  29  1–??
        See collection operators
DB2 interface  34  3–34  54
        See DB2 interface operators
general  1  1–21  84
        See general operators
INFORMIX interface  35  2–35  34
        See INFORMIX interface operators
join  28  2–28  17
        See join operators
Oracle interface  36  2–36  43
        See Oracle interface operators
partitioning
        See partitioners
precedence  21  24
restructure  30  3–30  51
sort  31  2–31  33
        See sort operators
statistics  33  3–33  77
        See statistics operators
Teradata interface  37  1–37  18
        See Teradata interface operators
utility  22  2–??
        See utility operators

Oracle
accessing from Orchestrate
PATH and LIBPATH
requirements  36  3
data type restrictions  36  19
indexed tables  36  18
joining Oracle tables and an Orchestrate
data set  36  39
Orchestrate to Oracle data type
conversion  36  20
performing joins between data sets  36  11
reading tables
        SELECT statement  36  10
        specifying rows  36  10
SQL query restrictions 36 10
writing data sets to
   column naming conventions 36 19
dropping data set fields 36 22
Oracle interface operators
   handing # and $ in column names 36 4
   national language support 36 5
orlookup 36 39
oraesert 36 32
orawrite 36 17
   preserving blanks in fields 36 4
See also individual operator entries
orlookup operator 36 39
   example 36 42
   properties 36 40
   syntax and options 36 41
oraesert operator 36 32
   data-flow diagram 36 32
   environment variables
      APT_ORAUPSERT_COMMIT_ROW_INTERVAL 36 35
      APT_ORAUPSERT_COMMIT_TIME_INTERVAL 36 35
   example 36 37
   operator action 36 34
   properties 36 39
   syntax and options 36 40
orawrite operator 36 17
   column naming conventions 36 19
   data-flow diagram 36 17
   data-type restrictions 36 19
   example
      creating an Oracle table 36 30
      using the modify operator 36 31
      writing to an existing table 36 29
   execution modes 36 19
matched and unmatched fields 36 22
operator action 36 18
Orchestrate to Oracle data type
   conversion 36 20
   properties 36 17
   required privileges 36 22
   syntax and options 36 23
   write modes 36 21
writing to indexed tables 36 18
Orchestrated SAS data set format 32 9
ordered collection operator
   properties 29 3
   syntax 29 4
ordinal transform 33 73
   defined 33 73
   false value 33 74
   forward
      false value 33 74
      field naming 33 74
      performing 33 74
      true value 33 74
   inverse 33 75
      example 33 77
      naming 33 76
      performing 33 75
      true value 33 74
Parallel SAS data set format 32 8
partition sorting
   See psort operator
partitioners 27 2–27 32
   entire 27 3
   hash 27 5
   modulus 27 10
   random 27 13
   range 27 15
   roundrobin 27 29
   same 27 31
partitioning methods 27 3
   See partitioners
partitioning operators
   See partitioners
pcompress operator 14 1
   compressed data sets
      partitioning 14 4
      using orchadmin 14 4
   data-flow diagram 14 2
Index

example 14 5
mode
  compress 14 3
  expand 14 3
properties 14 2
syntax and options 14 3
UNIX compress facility 14 3
peek operator 15 1
  data-flow diagram 15 2
  example 15 4
  properties 15 2
  syntax and options 15 3
precedence of operators 21 24
preserve-partitioning flag
  psort operator 31 20
  tsort operator 31 5
promotesubrec restructure operator 30 25
  data-flow diagram 30 27
  example 30 27
  properties 30 26
  syntax and options 30 26
psort operator 31 17
  configuring 31 19
  data-flow diagram 31 22
  example
    parallel execution 31 27
    sequential execution 31 27
execution mode
  parallel 31 18
  sequential 31 18
  performing a total sort 31 29
preserve-partitioning flag 31 20
  properties 31 23
  specifying sorting keys 31 21
  syntax and options 31 24
  using UNIX SyncSort 31 19
algorithm 27 16
data-flow diagram 27 21
example 27 20
properties 27 21
specifying partitioning keys 27 16
syntax and options 27 22
using 27 17, 27 21
range partitioning
  and the psort operator 31 30
reducobol redefines
  for tagbatch restructuring 30 33
reducobol utility
  handling COBOL redefines 26 7
  invoking 26 5
    ASCII input file 26 5
    command-line syntax 26 5
overriding defaults 26 3
record-formatting information 26 2
supported COBOL data storage formats 26 3
supported COBOL standards 26 2
  table of arguments 26 6
  table of COBOL storage format conversion 26 5
  using the generated schema 26 3
reducstats utility 33 41
  syntax and options 33 41
  using 33 41
  using with the statistics operator 33 41
reducers 33 19
regrouping data
  See group statistics operator
remdup operator 16 1
  data-flow diagram 16 2
  data-flow with hashing and sorting 16 4
effects of case-sensitive sorting 16 6
example
  case-insensitive string matching 16 8
  removing all but the first duplicate 16 8
  using two keys 16 9
example osh command with hashing and sorting 16 4
properties 16 2
removing duplicate records 16 4
syntax and options 16 3
usage 16 8
  retaining the first duplicate record 16 5
R
random partitioner 27 13
  data-flow diagram 27 14
  properties 27 14
  syntax 27 15
  using 27 15
range map
  creating with the makerangemap utility 27 28
  definition 27 26
range partitioner 27 15
Index

- retaining the last duplicate record 16
  - using the last option 16
  - removing duplicate records See remdup operator
- structure operators
  - aggtorec 30
  - field_import 30
  - makesubrec 30
  - makevect 30
  - promotesubrec 30
  - splitsubrec 30
  - splitvect 30
  - tagbatch 30
  - tagswitch 30
- restructuring data
  - See structure operators
  - rightouterjoin operator 28
    - example 28
    - syntax and option 28
- roundrobin collection operator 29
  - properties 29
  - syntax 29
- roundrobin partitioner 27
  - data-flow diagram 27
  - properties 27
  - syntax 27
  - using 27
- sample operator 17
  - data-flow diagram 17
  - example 17
  - properties 17
  - syntax and options 17
- SAS DATA steps
  - executing in parallel 32
- SAS interface library
  - configuring your system 32
  - converting between Orchestrate and SAS data types 32
  - data representations 32
  - example data flow 32
- executing DATA steps in parallel 32
- executing PROC steps in parallel 32
- getting input from a SAS data set 32
- getting input from an Orchestrate or Orchestrated SAS data set 32
- Orchestrate example 32
- Orchestrated SAS data set format 32
- overview 32
- Parallel SAS data set format 32
- parallelizing SAS code
  - rules of thumb for parallelizing 32
  - SAS programs that benefit from parallelization 32
- pipeline parallelism and SAS 32
- sequential SAS data set format 32
- using SAS on sequential and parallel systems 32
- writing SAS programs 32
- SAS interface operators
  - doing ETL 32
  - sas 32
  - sascontents 32
  - sasin 32
  - sasout 32
  - specifying a character set 32
- sas interface operators
  - controlling ustring truncation 32
  - determining SAS mode 32
  - environment variables 32
  - generating a Proc Contents report 32
  - long name support 32
  - specifying a character set 32
  - specifying an output schema 32
- sas operator 32
  - data-flow diagram 32
  - properties 32
  - syntax and options 32
- SAS PROC steps
  - executing in parallel 32
- sascontents operator
  - data-flow diagram 32
  - properties 32
  - syntax and options 32
- sasin operator
  - data-flow diagram 32
  - properties 32
  - syntax and options 32
- sasout operator 32
  - data-flow diagram 32
Index

properties 32 52
syntax and options 32 52
sequence operator 18 1
data-flow diagram 18 2
example 18 3
properties 18 2
syntax and options 18 3
sequential SAS data set format 32 8
sort operators
psort 31 17
tsort 31 2
See also individual operator entries
sortfunnel operator 8 1
data-flow diagram 8 2
example 8 8
input requirements 8 5
properties 8 2
setting primary and secondary keys 8 5
syntax and options 8 6
sortmerge collection operator 29 5
collecting keys 29 6
ascending order 29 7
case-sensitive 29 7
data types 29 7
primary 29 7
secondary 29 7
data-flow diagram 29 6
properties 29 8
syntax and options 29 8
splitsubrec restructure operator 30 27
data-flow diagram 30 28
example 30 29
properties 30 28
syntax and option 30 29
splitvect restructure operator 30 30
data-flow diagram 30 31
example 30 32
properties 30 31
syntax and option 30 32
squashing function
Z-sigmoid transform 33 62
Z-symmetric transform 33 62
statistics operator 33 38
calculating the mean 33 39
data-flow diagram 33 40
field types 33 36
properties 33 40
readstats utility 33 41
results storing 33 39
string fields 33 44
syntax and options 33 40
using statistics tools example 33 43
statistics operators
crosstab 33 4
fieldtransform 33 46
group 33 16
statistics 33 38
See also individual operator entries
statistics utility
example 33 43
string fields 33 44
Z-sigmoid transform 33 64
Z-symmetric transform 33 64
summary statistics 33 33
generating 33 33
group statistics operator 33 33
record schema of 33 33
switch operator 19 1
data-flow diagram 19 2
discard value 19 2
example 19 6
Job Monitor information 19 8
customizing messages 19 9
example messages 19 8
properties 19 3
syntax and options 19 3
SyncSort
setting APT_SYNCSORT_DIR 31 20
using with the psort operator 31 19

T
tagbatch restructure operator 30 33
added, missing, and duplicate fields 30 35
data-flow diagram 30 35
example
with missing and duplicate cases 30 40
with multiple keys 30 41
with simple flattening of tag cases 30 38
input data set requirements 30 36
operator action and transfer behavior 30 34
properties 30 35
readcobol redefines 30 33
syntax and options 30 36

tagged fields and operator limitations 30 33
tagswitch restructure operator 30 43
case option 30 44
data-flow diagram 30 43
example
  with default behavior 30 46
  with one case chosen 30 48
input and output interface schemas 30 44
properties 30 43
syntax and options 30 45
using 30 45
tail operator 20 1
data-flow diagram 20 2
example
  default behavior 20 3
  using the nrecs and part options 20 4
properties 20 2
syntax and options 20 2

Teradata
writing data sets to
data type restrictions 37 15
dropping data set fields 37 14
write mode 37 13
Teradata interface operators
teraread operator 37 4
terawrite operator 37 10
See also individual operator entries
teraread operator 37 4
column name and data type conversion 37 5
data-flow diagram 37 4
national language support 36 5, 37 2
properties 37 4
restrictions 37 7
specifying the query 37 5
syntax and options 37 7

terawrite operator 37 10
column name and data type conversion 37 12
correcting load errors 37 13
data-flow diagram 37 11
limitations 37 14
national language support 36 5, 37 2
properties 37 11
restrictions 37 15
syntax and options 37 15
write mode

append 37 13
create 37 14
replace 37 14
truncate 37 14
writing fields 37 14
total sort
See tsort operator
transform operator
data types 21 59
  default data-type conversion 21 59
data-flow diagram 21 4
properties 21 5
syntax and options 21 5
the Transformation Language
see Transformation Language
Transformation Language 21 14-21 57

built-in functions
  bit-manipulation functions 21 54
data conversion and data mapping 21 27
  mathematical functions 21 39
  miscellaneous functions 21 56, 21 57
  null handling functions 21 38
  string functions 21 42
time and timestamp transformations 21 34
  ustring functions 21 48
conditional branching 21 25
  if ... else 21 25
data types and record fields 21 18
differences with the C language 21 57
  C references 21 58
casting 21 58
  flow control 21 58
  labeled statements 21 58
  local variables 21 58
  names and keywords 21 57
  operators 21 58
  pre-processor commands 21 58
expressions 21 21
  language elements 21 21
  operators 21 22
general structure 21 14
  local variables 21 20
  names and keywords 21 14
specifying datasets
  complex field types 21 20
  simple field types 21 19
transforms
See fieldtransform statistics operator
See transform operator
tsort operator
  configuring 31 4
data-flow diagram 31 7
example
  parallel execution 31 12
  sequential execution 31 11
execution modes
  parallel 31 2
  sequential 31 2
performing a total sort 31 13
preserve-partitioning flag 31 5
properties 31 7
sorting keys
  case sensitivity 31 6
  restrictions 31 6
  sort order
  ascending 31 6
  descending 31 6
specifying sorting keys 31 5
syntax and options 31 8
using a sorted data set 31 4
using a sorted data set with a sequential operator 31 5
typographic conventions used in this Reference xxxviii

X
xpsread INFORMIX interface operator 35 29
data-flow diagram 35 29
properties 35 29
syntax and options 35 30
xpswrite INFORMIX interface operator 35 31
data-flow diagram 35 31
listing of examples 35 34
properties 35 32
syntax and options 35 32

Z
Z-score transform 33 57
defined 33 57
fieldtransform statistics operator 33 58
forward
  equation 33 58
  example 33 59
  field naming 33 59
  performing 33 58
  standard deviation of zero 33 61
inverse
  equation 33 58
  example 33 60
  standard deviation of zero 33 61
large values 33 61
Z-sigmoid transform
defined 33 62
forward
  equation 33 62
  example 33 64
  field naming 33 65
  performing 33 64
  standard deviation of zero 33 66
inverse
  equation 33 63
  example 33 65
  field naming 33 65
  performing 33 65
  standard deviation of zero 33 67
maximum value 33 62
minimum value 33 62
statistics utility 33 64
Z-symmetric transform
defined 33 62
forward
  equation 33 62

W
weighting field example 33 37
writerangemap operator 27 23
data-flow diagram 27 23
properties 27 24
syntax and options 27 24
Index

example 33 64
field naming 33 65
performing 33 64
standard deviation of zero 33 66, 33 67
inverse
  equation 33 63
  example 33 65
field naming 33 65
performing 33 65
  standard deviation of zero 33 67
maximum value 33 62
minimum value 33 62
statistics utility 33 64