PART VII

Building a Real-World Arsenal

TIPS & TECHNIQUES

CHAPTER 19

The 24×7 Toolkit

TIPS & COVERED

Previous chapters discuss various scenarios in which certain critical information is extracted from the database to help alert administrative personnel to impending and/or existing database and application/user process failures. For instance, an important segment inside the database may not have the contiguous space necessary to allocate another extent. Alternatively, an application process may be continuously consuming large amounts of space from the temporary tablespace, causing it and other processes to run out of space. By periodically querying the database for certain irregularities, you can detect them and take preventive action in time. For instance, a script may be deployed to check the database for potential problems every half-hour and page the DBAs if any are detected. This helps greatly in avoiding certain major service disruptions. If some of the scripts are resource consuming, then they can be run less frequently (say, every two hours). The objective is to be alerted to potential problems as early as possible with minimal impact on the production database.

This chapter offers the following tips and techniques:

- Monitor for specific failures by running poll-scripts periodically
- Be familiar with techniques for application failover when calling PL/SQL

Using Scripts to Be Alerted to Disruptive Problems

This chapter presents a series of scripts that may be used to poll the database and report problems. A master shell-script is provided at the very onset that can call a bunch of SQL and PL/SQL (if necessary) scripts. Each of the latter can check for occurrences of a specific error-condition. Note that this chapter does not present a comprehensive set of scripts to check every possible error. Rather, it serves to provide an example of checking for certain commonly encountered “show-stopping” errors and paging the DBAs. Prior to adopting these scripts in your environment, you would need to spend a fair amount of time analyzing the potential problems specific to your site and, if necessary, plug-in additional scripts to specifically check for those. The current set of scripts may not be a complete solution for your site.

Finally, this chapter also presents a shell-script to illustrate application failover when calling PL/SQL code blocks.

Now, let’s look at the various scripts that may be utilized as part of your “24×7 toolkit.” Note that the shell-scripts provided are meant to be used in a UNIX environment. In case your OS is different (Windows NT, VAX VMS, or what have you), then you will need to use the same algorithm illustrated here to code scripts specific to your environment. The SQL and PL/SQL scripts provided here can still be invoked via the new scripts (specific to your OS) that you create.

Monitor for specific failures by running poll-scripts periodically

The Korn shell-script that follows allows specific SQL statements to be run to poll the database and report specific errors. The shell-script may be run every half-hour or so via an OS scheduling utility such as *cron* or any third-party scheduling program.

```
#!/bin/ksh
# Script : ChkErr.ksh
# Functionality : Template to check for errors and page DBAs, if necessary
# Author : Venkat S. Devraj
# Modification history
# 07/1997              VSD             Written
# 07/1997              VSD             Added check for MAXEXTENTS
#

if [[ $# -lt 3 ]]; then
  echo "Usage: $0 <READONLY_MONITORING_USERID> <PASSWORD> <LOGFILE>"
  exit 1
fi

USR_ID=$1
USR_PASS=$2
LOGFIL=$3
NOTIFY_LIST=oncall_dba_pager@att.net

# check 1) verify that no critical segments are reaching
#          MAXEXTENTS. If so, page the DBA
sqlplus -s <<EOF >$LOGFIL 2>/dev/null
$USR_ID/$USR_PASS
    SET PAGES 0
    SELECT DISTINCT 'YES' FROM dba_segments
     WHERE extents >= (max_extents-5);
EOF
grep -i '^ORA-' $LOGFIL >/dev/null
if [ $? -eq 0 ]
then
   mailx -s "${ORACLE_SID} : Script failed" $NOTIFY_LIST <<EOF
      Monitoring script $0 failed.
      Please check $LOGFIL for more details.
EOF
   exit 1
fi

MAXEXTENTS_REACHED=`awk '{ print $1 }' $LOGFIL`
if [ "$MAXEXTENTS_REACHED" = "YES" ]
```

```
then
    mailx -s "${ORACLE_SID} : MAXEXTENTS reached" $NOTIFY_LIST <<EOF
      MAXEXTENTS has been reached for a segment.
      Please run willfail_maxextents.sql in $ORACLE_SID for more details.
EOF
   exit 1
fi

# check 2) Check for “Unable to Allocate Next Extent”
sqlplus -s <<EOF >$LOGFIL 2>/dev/null
$USR_ID/$USR_PASS
    SET PAGES 0
    SELECT DISTINCT 'YES' FROM dba_segments ds
     WHERE next_extent >
       (SELECT MAX(bytes) FROM dba_free_space
         WHERE tablespace_name = ds.tablespace_name);
EOF
grep -i '^ORA-' $LOGFIL >/dev/null
if [ $? -eq 0 ]
then
   mailx -s "${ORACLE_SID} : Script failed" $NOTIFY_LIST <<EOF
      Monitoring script $0 failed.
      Please check $LOGFIL for more details.
EOF
   exit 1
fi

POSSIBLE_NEXTEXT_FAIL=`awk '{ print $1 }' $LOGFIL`
if [ "$POSSIBLE_NEXTEXT_FAIL" = "YES" ]
then
    mailx -s "${ORACLE_SID} : Impending next extent failure" $NOTIFY_LIST <<EOF
      Next extent allocation will fail for a segment.
      Please run willfail_nextextent.sql in $ORACLE_SID for more details.
EOF
   exit 1
fi

# other checks, as necessary
echo “Successful completion of $0 ” `date`
```

Now, the preceding script may be invoked via *cron* from an administrative user-account, so that the Oracle user-name and password are not visible to non-administrative users. Also, the output from the script needs to be redirected (appended via >>) to a log-file, which can be scanned on a daily basis for errors from the execution of this shell-script. This log-file is a permanent one (storing results of multiple runs) and is different from the temporary log-file specified via $LOGFIL (which stores results of only the current run).

Note that the preceding script template only checks for two specific failure scenarios: impending failure due to MAXEXTENTS being reached or nearly reached (five more extents to go) and inability to allocate the next extent due to lack of adequate free-space in the tablespace. These two failure scenarios are provided as an example for checking potentially disruptive events. You will need to add all possible disruptive events specific to your environment prior to using this script. Following are some additional failure scenarios that need to be checked. The check for each scenario is provided as a separate script to allow them to be run independently, as well as be called from the preceding shell-script (in fact, the preceding shell-script makes references to the names of the following SQL scripts in the alerts [pages] sent to the administrative personnel). The names are something I use as a handy reference for these scripts (they can be named just about anything). Prior to your calling these scripts from the preceding shell script, they may have to be changed to return a single flag (such as 'YES') rather than the multiple columns they currently retrieve. The scripts that follow are ones that I use most frequently at various client sites. Some of them have been originally written by me. Some have been adapted from the source-code for the DBA_* and *v$* views in *SQL.BSQ* and *catalog.sql*. Yet others have been collected over the years from various Oracle and third-party books, magazines, articles, white-papers, and so on.

- MAXEXTENTS being reached (*willfail_maxextents.sql*)

  ```
  COLUMN segment_name FORMAT a30
  SELECT owner, segment_name, segment_type, extents
        ,max_extents, tablespace_name
    FROM dba_segments
   WHERE extents >= (max_extents-5);
  ```

- Unable to allocate next extent (*willfail_nextextent.sql*)

  ```
  COLUMN segment_name FORMAT a30
  SELECT owner, tablespace_name, segment_name, next_extent
    FROM dba_segments ds
   WHERE next_extent > (SELECT MAX(bytes)
                          FROM dba_free_space
                         WHERE tablespace_name = ds.tablespace_name);
  ```

- Lack of space in the temporary tablespace (*willfail_tempextent.sql*). This script needs to have the name of the temporary tablespace hard-coded. You could alternatively check the TEMPORARY column to find whether any of the tablespaces are explicitly marked as temporary to avoid the hard-coding. However, in case you do not explicitly mark a tablespace as temporary, that strategy will fail. For this reason, here I'm playing it safe and requesting you to include the temporary tablespace name in the script itself.

```
SELECT 'Inadequate space in temporary tablespace to allocate INITIAL/NEXT'
  FROM dba_tablespaces dt, (SELECT MAX(bytes) max_size, tablespace_name
                             FROM dba_free_space
                            WHERE tablespace_name = ‘TEMP’
                            GROUP BY tablespace_name) fs
 WHERE dt.tablespace_name = fs.tablespace_name
   AND (dt.initial_extent > fs.max_size OR dt.next_extent > fs.max_size);
```

- Detect locking problems and deadlocks (*lock_problem.sql*). In some Oracle versions, you may have problems querying DBA_WAITERS and DBA_BLOCKERS directly. Chapter 11 discusses some workarounds.

```
SELECT * FROM dba_waiters;
```

or

```
SELECT * FROM dba_blockers;
```

- Supplementary script to check current locks in the database (*what_locks.sql*):

```
SELECT SUBSTR(v$lock.sid,1,4) "SID",
  SUBSTR(username, 1, 12) "UserName",
  SUBSTR(object_name, 1, 25) "ObjectName",
  v$lock.type “LockType”,
  DECODE(RTRIM(SUBSTR(lmode,1,4)),
         '2', 'Row-S (SS)', '3', 'Row-X (SX)',
         '4', 'Share',      '5', 'S/Row-X (SSX)',
         '6', 'Exclusive',  'Other' ) "LockMode",
  SUBSTR(v$session.program, 1, 25) "ProgramName"
  FROM v$lock, sys.dba_objects, v$session
 WHERE (object_id = v$lock.id1
   AND v$lock.sid = v$session.sid
   AND username IS NOT NULL
   AND username NOT IN ('SYS', 'SYSTEM')
   AND serial# != 1);
```

Other Useful Scripts

Following are some other scripts that provide useful information on the functioning of the database. They draw upon our discussions in the earlier chapters.

- Find total allocated size for each tablespace/data-file (*find_alloc_space.sql*).

```
SELECT tablespace_name "Tablespace Name",
       file_name “DataFile Name”,
       SUM(bytes)/1024/1024 "Allocated Space (MB)"
   FROM dba_data_files
 GROUP BY tablespace_name, file_name;
```

- Find current free-space in all tablespaces (*find_free_space.sql*).

```
SELECT tablespace_name "Tablespace Name",
       SUM(bytes)/1024/1024 "Free Space (MB)",
       MAX(bytes)/1024/1024 "Largest chunk (MB)"
  FROM dba_free_space
 GROUP BY tablespace_name;
```

- Find current sessions logged on (*find_sessions.sql*). When run over a period of time, this script provides a good feel for the overall database activity occurring at various times. This may be used in conjunction with *find_curr_sql.sql* (listed next) to drill down on the SQL statements actually being executed by an ACTIVE session.

```
SELECT username, osuser, server, status, count(*)
  FROM v$session
 WHERE username IS NOT NULL
   AND username NOT IN ('SYS', 'SYSTEM')
 GROUP BY username, osuser, server, status;
```

- Find the SQL statements being executed by a particular session (*find_curr_sql.sql*).

```
SELECT sql_text FROM v$sqltext_with_newlines
 WHERE (hash_value, address) IN
  (SELECT sql_hash_value, sql_address FROM v$session
    WHERE username = UPPER('&Oracle_username'))
  ORDER by address, piece;
```

- Detect badly written SQL preventing sharing of SQL statements in the library cache (*SQL_high_loads.sql*).

```
SELECT sql_text, loaded_versions, version_count, sharable_mem
  FROM v$sqlarea
 WHERE loaded_versions > 5
 ORDER by sharable_mem;
```

- Detect high SQL statement parses occurring in the database (*find_parses.sql*). If the value here continuously increases, especially during peak-hours, at a fast rate (say more than ten per second), then it may be indicative of a SQL problem in the library cache.

```
SELECT name, value FROM v$sysstat
 WHERE name LIKE 'parse count%';
```

- Find which process is using which rollback segment (*find_which_rbs.sql*).

```
COLUMN "Oracle UserName" FORMAT a15
COLUMN "RBS Name" FORMAT a15
SELECT r.name "RBS Name", p.spid, l.sid "ORACLE PID",
       s.username "Oracle UserName"
  FROM v$lock l, v$process p, v$rollname r, v$session s
 WHERE s.sid = l.sid AND l.sid = p.pid(+)
   AND r.usn = TRUNC(l.id1(+)/65536)
   AND l.type(+) = 'TX' AND l.lmode(+) = 6
 ORDER BY r.name;
```

- Map of the contents of a tablespace (*tlbspc_map.sql*, original source [to the best of my knowledge]: *DBA Handbook* by Kevin Loney, Oracle Press/Osborne/McGraw Hill). Given the name of a (highly used) tablespace, this script prints out the map indicating used-space, free-space, and fragmentation within the tablespace. A sample output of this script is provided in Chapter 12.

```
SET LINES 132 PAGES 1000
COLUMN “Fil_ID” FORMAT 999 HEADING “Fil|ID”
COLUMN “Fil” FORMAT A55 HEADING “Fil-name”
COLUMN “Segment” FORMAT A55
COLUMN “Start blk” FORMAT 999999 HEADING “Strt|blk”
COLUMN “# blocks” FORMAT 999,999 HEADING “#|blks”
SELECT d.file_id "Fil_ID", d.file_name "Fil", segment_type || ' ' ||
       owner || '.' || segment_name "Segment",
       e.block_id "Start blk", e.blocks "# blocks"
  FROM dba_extents e, dba_data_files d
 WHERE e.tablespace_name = UPPER('&&tblspc_name')
   AND d.tablespace_name = e.tablespace_name
   AND d.file_id = e.file_id
UNION
SELECT s.file_id "Fil_ID", d.file_name "Fil", 'Free chunk' "Segment",
       s.block_id "Start blk", s.blocks "# blocks"
  FROM  dba_free_space s, dba_data_files d
 WHERE s.tablespace_name = UPPER('&&tblspc_name')
   AND d.tablespace_name = s.tablespace_name
   AND d.file_id = s.file_id
 ORDER BY 1, 4, 5;
```

- Current waits being encountered by all sessions (*find_sess_waits.sql*). If necessary, you can filter out certain harmless waits (normally) such as “SQL*Net message from client,” “SQL*Net message to client,” and so forth.

```
COLUMN event FORMAT a25
SELECT event,
       SUM(DECODE(wait_time, 0, 1, 0)) "Currently Waiting",
       COUNT(*) "Total Waits"
  FROM v$session_wait
 GROUP BY event
 ORDER BY 3;
```

- Current waits being encountered by a specific session (*find_user_waits.sql*).

```
COLUMN event FORMAT a25
SELECT event,
       SUM(DECODE(wait_time, 0, 1, 0)) "Currently Waiting",
       COUNT(*) "Total Waits"
  FROM v$session_wait
 WHERE sid IN (SELECT sid FROM v$session
               WHERE username = '&OracleUsrName'
                 AND osuser = '&OSUsrName')
 GROUP BY event
 ORDER BY 3;
```

- Waits encountered so far in the database (*find_db_waits.sql*).

```
SELECT event, total_waits FROM v$system_event
 ORDER BY 2;
```

- Monitor temporary segment usage (*monitor_sorts.sql*).

```
SELECT tablespace_name, added_extents, free_extents
  FROM v$sort_segment;
```

- Detect waits for rollback segment header blocks.

```
SELECT class, count FROM v$waitstat
 WHERE class IN ('undo header', 'system undo header');
```

- Track segment-growth in relevant schemas by examining the number of extents they have allocated. This script should be run daily. Examining the results over a period of time will give good insight into segment growth patterns and appropriate proactive action can be taken prior to the segments facing growth problems (such as running out of free space in the tablespace or reaching MAXEXTENTS).

```
SET LINESIZE 132
COLUMN owner FORMAT a15
COLUMN segment_name FORMAT a30
COLUMN segment_type FORMAT a5 HEADING “Type”
SELECT owner, segment_name, segment_type,
       initial_extent/1024 “Initial (KB)”,
       next_extent/1024 “Next (KB)”, extents “# extents”
  FROM dba_segments
```

```
 WHERE segment_type IN ('TABLE', 'INDEX')
   AND owner IN ('relevant-schema1', 'relevant-schema2')
 ORDER BY owner, segment_type, extents DESC;
```

- Acquire inter-table relationship information (somewhat of an ERD of the physical database model). These scripts assume that all referential integrity is declared within the database.
  - Find all parent tables for a given table. Run this script as the owner of the tables. If not the owner, use ALL_CONSTRAINTS or DBA_CONSTRAINTS views, instead of USER_CONSTRAINTS.

```
SELECT prnt.table_name "Parent tables"
  FROM user_constraints prnt, user_constraints chld
 WHERE chld.table_name = UPPER('&child_tbl_nm')
   AND prnt.constraint_type = 'P'
   AND chld.r_constraint_name = prnt.constraint_name;
```

  - Find all child tables for a given table.

```
SELECT table_name "Child tables" FROM user_constraints
 WHERE r_constraint_name IN
    (SELECT constraint_name
       FROM user_constraints
      WHERE table_name = UPPER('&parent_tab_nm')
        AND constraint_type = 'P');
```

- Delete all core-dumps and trace-files older than 14 days via *cron*. Note that the commands that follow do not archive the files prior to removing them. If you need to save the old trace-files to tape, you can do so by adding the *tar* or *cpio* commands here or in a separate script that can be called via *cron*.

```
# cron entries in the "oracle" user-account
# Remove all old core-dumps for PT05 database
0 3 * * *   /usr/bin/find /opt/app/oracle/admin/PT05/cdump -name core\?
-mtime +14 -exec rm -f {} \;
# Remove all old background dumps for PT05 database
0 3 * * *   /usr/bin/find /opt/app/oracle/admin/PT05/bdump -name \*.trc
-mtime +14 -exec rm -f {} \;
# Remove all old user dumps for PT05 database
0 3 * * *   /usr/bin/find /opt/app/oracle/admin/PT05/udump -name \*.trc
-mtime +14 -exec rm -f {} \;
# if necessary, other files (such as audit-files) can be added here
```

- Trim the *alert-log* and *listener-log* files, leaving just the 1,000 most recent lines within the files. This is to control growth of these files (in particular, the *listener-log* can grow very fast in an environment with a large number of database connections, in which case this script can be broken up to trim the *alert-log* and *listener-log* separately and the script to trim the *listener-log* can be run more frequently). Note that the older alert-log is not being archived to tape prior to being trimmed. If the database against which you are deploying this script is a production database, you should add the commands to archive the alert-log prior to trimming it.

```
#!/bin/ksh
# Functionality:        To trim alert-log and listener-log
#

echo "start time `date`" >
/opt/app/oracle/admin/PT05/admin_logs/trim_logs.lst

exec 3< /var/opt/oracle/oratab    #open oratab as file-descriptor 3

while read -r -u3 LINE
do
    case $LINE in
        \#*)            ;;               #Ignore comments in oratab
        *)
            ORACLE_SID=$(print $LINE | awk -F: '{print $1}' -)
            ORACLE_HOME=$(print $LINE | awk -F: '{print $2}' -)

         ALERTLOG=\
/opt/app/oracle/admin/$ORACLE_SID/bdump/alert_$ORACLE_SID.log
          tail -1000 $ALERTLOG > /tmp/$ORACLE_SID.log
        # if necessary, archive $ALERTLOG, prior to overwriting
          cp /tmp/$ORACLE_SID.log $ALERTLOG
          rm /tmp/$ORACLE_SID.log
        ;;
    esac
done

LISTENERLOG=$ORACLE_HOME/network/log/listener.log
if [[ -f $LISTENERLOG ]]; then
    tail -1000 $LISTENERLOG > /tmp/listener.log
    # archive $LISTENERLOG if necessary, prior to being overwritten
    cp /tmp/listener.log $LISTENERLOG
    rm /tmp/listener.log
fi
```

- Compress all archived logs except the most current. Here, the older archived logs are being transferred to tape prior to being removed. If necessary, you can modify this script to avoid removing them (if your ARCHIVE_LOG_DEST directory is large enough to accommodate the archived logs). Then a *cron* entry can be set up to remove them once every two days or so (after they have been backed up). You can also change the script to avoid compressing the older archived logs but just copy them directly to tape. Here, it is assumed that no ARCH slaves are configured. However, if multiple ARCH I/O slaves are configured (via the initialization parameter ARCH_IO_SLAVES available from Oracle8 onward), then compress all the archived logs, except the N+1 most current (where, N is equivalent to the number of ARCH I/O slaves configured). This script can be called via *cron* every couple of hours or so.

```
echo "`date` Compressing older archived logs" >
/opt/app/oracle/admin/PT05/admin_logs/compress_archs.lst

# check to see whether process is running already
if [ -f /opt/app/oracle/admin/PT05/arch/proc_currently_running* ]; then
   exit
fi

/bin/touch /opt/app/oracle/admin/PT05/arch/proc_currently_running

# if your ARCH file-names are different, then change below command to
# reflect proper names. Also, if multiple ARCH I/O slaves are configured,
# then change “head -1” to “head –[N+1]”, where [N+1] is number of ARCH
# slaves plus one.
/bin/compress -f `ls -t /opt/app/oracle/admin/PT05/arch/*.arch | grep -v
\`ls -t /opt/app/oracle/admin/PT05/arch/*.arch | head -1\``

# ensure that your tape-device name is set properly
tar cvf /dev/rmt0 /opt/app/oracle/admin/PT05/arch/*Z

# Next step is optional. If your ARCHIVE_LOG_DEST is large
# enough to accommodate at least 2 days worth of archived logs,
# then don’t do this. Instead, set up a cron job to remove all
# compressed archived logs older than 2 days. Keeping required
# archived logs on disk will enhance recovery time (if recovery is required).
rm -f /opt/app/oracle/admin/PT05/arch/*Z

rm /opt/app/oracle/admin/PT05/arch/proc_currently_running*
```

- Script to run an export off a named-pipe and simultaneously compress the export dump-file. This script can be extremely useful when the data being exported is large and disk-space is scarce. This script can be modified to directly write the compressed dump-file to tape (if there is not sufficient disk-space to even accommodate the compressed dump-file). Note that this export script uses a parameter (*.par*) file defining all the export options to be utilized (do not define the LOG option; this is defined separately along with the EXP command). The script pages the DBAs if errors are encountered during the export.

```
#!/bin/ksh
# Functionality : to export the data via a named-pipe and
#                 compress the resultant dump-file.

EXPDIR=/opt/app/oracle/admin/PT05/backs
DUMPFILE=Full_PT05.dmp
LOGDIR=/opt/app/oracle/admin/PT05/admin_logs
LOGFILE=${LOGDIR}/Full_PT05.log
EXPLOG=${EXPDIR}/Full_PT05.log
PARAMFILE=${EXPDIR}/Full_PT05.par
PAGEDBA_FILE=/tmp/pagefile
EXITSTATCODE=0
NOTIFY_LIST=oncall_dba_pager@att.net

# preserve existing dump-files (rather than overwriting them)
if [ -f ${EXPDIR}/${DUMPFILE}.Z ]; then
   mv -f ${EXPDIR}/${DUMPFILE}.Z ${EXPDIR}/${DUMPFILE}.old.Z
fi

# create a named pipe and begin the compress
/sbin/mknod ${EXPDIR}/${DUMPFILE} p
compress < ${EXPDIR}/${DUMPFILE} > ${EXPDIR}/${DUMPFILE}.Z &

# start the export
cd $EXPDIR
echo "Export/compress started at `date`" > $PAGEDBA_FILE > $LOGFILE
$ORACLE_HOME/bin/exp log=$EXPLOG parfile=$PARAMFILE

# check for errors during the export
if egrep 'ORA-|EXP-' ${EXPLOG}   >> $PAGEDBA_FILE >> $LOGFILE
then
   EXITSTATCODE=1
   echo "Export failed at `date`"     >> $PAGEDBA_FILE >> $LOGFILE
fi
echo "============================" >> $PAGEDBA_FILE >> $LOGFILE

tail ${EXPLOG} >> $PAGEDBA_FILE
ls -lt ${EXPDIR}/${DUMPFILE}*   >> $PAGEDBA_FILE >> $LOGFILE
echo "Export/compress ended at `date`" >> $PAGEDBA_FILE >> $LOGFILE

# remove the named pipe
```

```
rm -f ${EXPDIR}/${DUMPFILE}
ls -lt ${EXPDIR}/${DUMPILE}*   >> $PAGEDBA_FILE >> $LOGFILE

mailx -s "Full export of PT05" $NOTIFY_LIST < $PAGEDBA_FILE
exit $EXITSTATCODE
```

- Script to run an import off a compressed dump-file via a named-pipe. This script is a companion script of the previous one (where the export and compress were done via a named pipe). Note that this import script uses a parameter (*.par*) file defining all the import options to be utilized (do not define the LOG option; this is defined separately along with the IMP command). The script pages the DBAs if errors are encountered during the import.

```
#!/bin/ksh
# Functionality : to import data off a compressed dump-file
#                 via a named-pipe.

IMPDIR=/opt/app/oracle/admin/PT05/backs
DUMPFILE=Full_PT05.dmp
LOGDIR=/opt/app/oracle/admin/PT05/admin_logs
LOGFILE=${LOGDIR}/Full_PT05.log
IMPLOG=${IMPDIR}/Full_PT05.log
PARAMFILE=${IMPDIR}/Full_PT05.par
PAGEDBA_FILE=/tmp/pagefile
EXITSTATCODE=0
NOTIFY_LIST=oncall_dba_pager@att.net

# create the named pipe and start the uncompress of the compressed
# dump-file
/sbin/mknod ${IMPDIR}/${DUMPFILE} p
uncompress < ${IMPDIR}/${DUMPFILE}.Z > ${IMPDIR}/${DUMPFILE} &

# start the import
echo "Uncompress/import started at `date`" > $PAGEDBA_FILE > $LOGFILE
cd $IMPDIR
$ORACLE_HOME/bin/imp log=$IMPLOG parfile=$PARAMFILE

# check for errors during the import
if egrep 'ORA-|IMP-' ${IMPLOG}   >> $PAGEDBA_FILE >> $LOGFILE
then
   EXITSTATCODE=1
   echo "Import failed at `date`"     >> $PAGEDBA_FILE >> $LOGFILE
fi
echo "============================" >> $PAGEDBA_FILE >> $LOGFILE
```

```
tail ${IMPLOG} >> $PAGEDBA_FILE
ls -lt ${IMPDIR}/${DUMPFILE}*   >> $PAGEDBA_FILE >> $LOGFILE
echo "Uncompress/import ended at `date`" >> $PAGEDBA_FILE >> $LOGFILE

# remove the named pipe
rm -f ${IMPDIR}/${DUMPFILE}
ls -lt ${IMPDIR}/${DUMPILE}*   >> $PAGEDBA_FILE >> $LOGFILE

mailx -s "Import of PT05" $NOTIFY_LIST < $PAGEDBA_FILE
exit $EXITSTATCODE
```

Be familiar with techniques for application failover when calling PL/SQL

Besides checking for error-conditions and possible service disruptions, you also need application failover capabilities built into all mission-critical programs. Chapters 15 to 18 describe database failover and in Chapter 6, a piece of pseudo-code illustrates failover after checking for certain Oracle errors at runtime (such as ORA-1092 and ORA-3113). Such a strategy can be directly applied when using 3GLs such as Pro*C and C/OCI. However, this may be challenging to achieve in 4GL scenarios, such as in PL/SQL. In PL/SQL, direct reconnection cannot be obtained within a code block. Therefore, you need to plug-in the failover code within the external calling routines (that call the code block). Sample code for achieving this is provided in this chapter. Here, a PL/SQL script is called via a Korn shell-script with the failover code embedded within the shell-script. This strategy or a suitable variation needs to be adopted in conjunction with other application failover features (via SQL*Net/Net8, TAF, and so on) to allow seamless reconnection to the standby/parallel database/instance when the current connection is aborted (and retrying the current connection does not result in success).

```
#!/bin/ksh
# Script : Call_SQL.ksh
# Functionality : Sample script to call PL/SQL code-block and illustrate failover
# Author : Venkat S. Devraj
# Modification history
# 07/1997                VSD              Written
#

if [ $# -ne 5 ]
then
   echo “Usage: $0 <script_name> <username> <password> <primary_db> <stndby_db>”
   echo “Where: script_name is the name of the PLSQL script to run”
   echo “Where: username is the Oracle user-id”
   echo “Where: password is the Oracle password”
   echo “Where: primary_db is the connect-string for the primary database”
   echo “Where: stndby_db is the connect-string for the standby database”
   exit 1
fi

PLSCRIPT=$1
USR=$2
PASSWD=$3
PRIM_DB=$4
STND_DB=$5

echo "Start date and time: " date

# connect to primary db to run script
sqlplus -s $USR/$PASSWD@PRIM_DB @$PLSCRIPT
RETVAL=$?
echo "Exit-code from primary database" $RETVAL

# check exit-code returned from script
if  [ $RETVAL -eq -1033 ] || [ $RETVAL -eq -1034 ] || [ $RETVAL -eq -1089 ] ||
    [ $RETVAL -eq -9352 ]
then
     echo "Failure to connect to the primary database " $PRIM_DB
     echo "Now trying to connect to standby" $STND_DB
     sqlplus -s $USR/$PASSWD@STND_DB @$PLSCRIPT
     RETVAL=$?
     echo "Exit-code from secondary database " $RETVAL
     if  [ $RETVAL -eq -1033 ] || [ $RETVAL -eq -1034 ] || [ $RETVAL -eq -1089 ] ||
          [ $RETVAL -eq -9352 ]
     then
        echo “Cannot connect to either primary or secondary database. Aborting”
        exit $RETVAL
     elif  [ $RETVAL -eq -1092 ] || [ $RETVAL -eq -3113 ] || [ $RETVAL -eq -3114 ]
     then
      echo "Script was abruptly terminated from secondary database" $RETVAL
       exit $RETVAL
     fi

elif  [ $RETVAL -eq -1092 ] || [ $RETVAL -eq -3113 ] || [ $RETVAL -eq -3114 ]
then
   echo "Script aborted due to abrupt session termination in primary"
   echo "Now trying to connect to standby (to rerun script) " $STND_DB
   sqlplus -s $USR/$PASSWD@STND_DB @$PLSCRIPT
   RETVAL=$?
   echo "Exit-code from secondary database " $RETVAL
   if  [ $RETVAL -eq -1033 ] || [ $RETVAL -eq -1034 ] || [ $RETVAL -eq -1089 ] ||
      [ $RETVAL -eq -9352 ]
   then
     echo “Cannot connect to secondary database. Aborting”
     exit $RETVAL
   elif  [ $RETVAL -eq -1092 ] || [ $RETVAL -eq -3113 ] || [ $RETVAL -eq -3114 ]
   then
    echo "Script was abruptly terminated from secondary database too" $RETVAL
    exit $RETVAL
   elif [ $RETVAL -ne 0 ]
   then
     echo "Some other error in script" $RETVAL
     exit $RETVAL
   fi

elif [$RETVAL -ne 0 ]
   echo "Some other error in script" $RETVAL
   exit $RETVAL
fi

# No errors.
# Successful completion of script either in primary or secondary db
echo "End date and time: " `date`
```

Now the following sample PL/SQL code may be used with the preceding shell-script:

```
$ cat upd_sal.sql
/* Name : upd_sal.sql
** Functionality : Update salary, give everyone a raise!
** History :
** 06/99Written  VSD
**
*/
SET SERVEROUTPUT ON
SPOOL upd_sal.log
WHENEVER SQLERROR EXIT SQL.SQLCODE ROLLBACK
WHENEVER OSERROR EXIT SQL.OSCODE ROLLBACK

BEGIN
  DBMS_OUTPUT.PUT_LINE('Start time : ' || TO_CHAR(SYSDATE, 'mm/dd/yyyy hh24:mi:ss');
  UPDATE empl SET sal = sal * 1.1;
  COMMIT;
  DBMS_OUTPUT.PUT_LINE('End time : ' || TO_CHAR(SYSDATE, 'mm/dd/yyyy hh24:mi:ss');
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error encountered : ' || SQLCODE || ' : ' ||
                         SQLERRM);
    RETURN SQLCODE;
END;
/
SPOOL OFF
EXIT
```

Now the *Call_SQL.ksh* script can be invoked as follows:

```
$ Call_SQL.ksh  upd_sal.sql  HRD_User  HRD_UsrPasswd  Prim_DB  Stnd_DB
```

Summary

This chapter provided some critical SQL scripts to detect current or impending service disruptions. Common causes of disruption, such as lack of space or locking conflicts, are monitored. The chapter also provided a sample template to use these SQL scripts within a shell-script that can be run via a scheduler at pre-determined intervals. Finally, a shell script to illustrate failover in a PL/SQL environment was provided. These scripts and strategies should help you build your own 24×7 toolkit and, if you already have one, should help you reinforce your arsenal.

The next chapter looks at new high-availability features in Oracle8*i*.