Designing Fast and Supportable Applications

CHAPTER 6

Designing Supportable Applications

What exactly does making an application supportable mean? It means that when a running application encounters a problem, the exact location in the code can be located immediately, and the root cause identified as quickly as possible. It means that an application reports back on its status in a format that can be easily assimilated by support staff and automated monitoring processes. It means that an application needs to be written in a way that is robust against various types of failures and takes advantage of available features to mitigate the effects of those failures. Many factors influence supportability, and this chapter covers the following topics to address them:

- Tips for supportable SQL
- How to provide tracing facilities
- How to enable error reporting and logging
- Run-time application configuration
- The importance of restartability
- How to use resumable operations in Oracle9*i*

This chapter is intended for both the database administrator (DBA) and developer. If you're a developer, consider implementing the suggestions to aid supportability. Supportability translates directly to increased availability through reductions in outages and faster problem resolution. If you're a DBA, then you can put forward the information in this chapter as a blueprint for the developers in your organization, with a goal of reducing support costs.

Creating Supportable SQL

This section contains four simple tips for SQL layout and naming that are frequently missing from Oracle code yet can provide significant benefits to supportability with minimal effort.

SQL Layout for Readability

Professional DBAs can spend a significant amount of time inspecting resource-intensive SQL statements and investigating ways to improve them in order to improve application response times for end users. It might surprise developers how much this process can be expedited if SQL is written in a way that makes the SELECT list columns, tables in the FROM clause, and WHERE predicates clear in the statement. This is easily seen with an example. Consider this free formatted SQL statement:

```
SELECT Deal_Type, Deal_Num, Thin_Pack FROM TT_FX_OTC d WHERE
(((DEAL_STATE not in ('DLTD', 'MTRD', 'EXCD','ABND') or
EOD_REALISED_PREMIUM <> 0.0 or EOD_REALISED_PREMIUM_REVERSED <> 0.0) and
ALLOCATION_STATUS<>'ALLOC') or (DEAL_STATE in ('DLTD', 'MTRD','MTDL')
and d.DEAL_NUM in (select dt_vals.DEAL_NUM from DT_VALUES dt_vals where
dt_vals.DEAL_NUM = d.DEAL_NUM and dt_vals.PL_INC_SUR <> 0.0))) and
DEAL_ROLE <> 'BACK'
```

A DBA attempting to make sense of this SQL has a real challenge on his hands. As a contrast, consider the same SQL formatted for readability:

```
SELECT Deal_Type, Deal_Num, Thin_Pack
FROM TT_FX_OTC d
WHERE
(
  (
    (DEAL_STATE not in ('DLTD', 'MTRD', 'EXCD','ABND')
     or EOD_REALISED_PREMIUM <> 0.0
     or EOD_REALISED_PREMIUM_REVERSED <> 0.0
     ) and ALLOCATION_STATUS<>'ALLOC'
  )
 or
  (DEAL_STATE in ('DLTD', 'MTRD','MTDL')
   and d.DEAL_NUM in
     (select dt_vals.DEAL_NUM
      from DT_VALUES dt_vals
      where dt_vals.DEAL_NUM = d.DEAL_NUM
      and dt_vals.PL_INC_SUR <> 0.0
     )
   )
)
and DEAL_ROLE <> 'BACK'
```

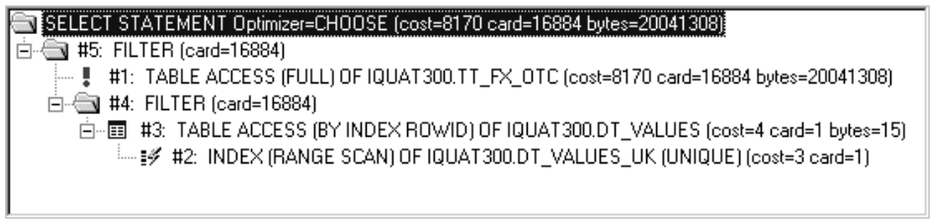

Figure 6.1 Explain plan structure.

The reformatted version shows the tables involved clearly and more importantly shows that the query result set depends on two OR clauses, where the second has a dependency on another table. The structure of the query often relates directly to the appearance of the query explain plan. The more closely the two match, the easier it is to identify the part of the query on which to concentrate tuning efforts. The query explain plan for the previous query is given in Figure 6.1 and shows that the first part of the query requires a full table scan of the TT_FX_OTC table, identified by the exclamation mark. Tuning efforts could therefore concentrate on that part. Using the unformatted statement, the relationship between the query plan and the SQL is not evident.

Most developers would never consider laying out code—be it Java, C, or PL/SQL—in an unformatted way. The same rule should apply to SQL statements.

Use Table Aliases

Another simple SQL fix that can make tuning efforts easier is to always use table aliases in SQL statements, in order to make explicit the table from which a SELECT list column originates. Identification of the underlying table for each SELECT list column is required during SQL tuning in order to check whether appropriate indexes on the table are being used. The process can be appreciably slower when a query contains one or more joins, and columns in the SELECT list don't identify the table in the join. The following SQL contains a SELECT list column that could originate from any of three underlying tables:

```
SELECT SUM(PREMIUM_REVAL)
FROM TT_FX TT,DT_VALUES DT,SD_LIVE_DEAL_STATES LDS
WHERE DT.DEAL_NUM = TT.DEAL_NUM
AND TT.TRADING_BOOK = :b2 AND TT.DEAL_STATE = LDS.NAME  AND
LDS.LIVE = 'Y'
```

In this case, a simple change to the SELECT list to include the table alias, DT, means the DBA no longer needs to query the Oracle dictionary to identify the underlying table, as shown:

```
SELECT SUM(DT.PREMIUM_REVAL)
FROM TT_FX TT,DT_VALUES DT,SD_LIVE_DEAL_STATES LDS
```

```
WHERE DT.DEAL_NUM = TT.DEAL_NUM
AND TT.TRADING_BOOK = :b2 AND TT.DEAL_STATE = LDS.NAME  AND
LDS.LIVE = 'Y'
```

When used together with the previous tip on layout, the speed with which DBAs can analyze queries can be increased significantly, even for the simple examples shown. The gains are much higher for longer and more complicated SQL.

Use Explicit Constraint Names

Explicit names should be used for Oracle constraints in DDL statements, rather than allowing Oracle to generate them. Oracle-generated names always begin with the prefix SYS_C. Constraint names are used in error messages generated by Oracle when constraints are violated. The more meaningful the name, the quicker the DBA can identify the cause of the underlying problem. The following example shows the Oracle-generated constraint names for a primary key and foreign key on the EMP table:

```
create table emp
       (empno number(4) primary key,
        ename varchar2(10),
        deptno number(2) references dept);

select constraint_name,constraint_type
from user_constraints where table_name='EMP';

CONSTRAINT_NAME   CONSTRAINT_TYPE
----------------- -----------------
SYS_C002402       P
SYS_C002403       R
```

The existence of system-generated constraint names can be avoided by explicit naming of the constraints. The previous example can be rewritten using the following SQL:

```
create table emp
       (empno number(4) constraint pk_emp primary key,
        ename varchar2(10),
        job varchar2(9)
        deptno number(2) constraint fk_deptno references dept);
```

Using explicit names has an extra benefit when the DBA needs to compare schema objects during schema upgrade procedures, such as using Oracle Change Manager. If you allow Oracle to choose the names, the chances are that a constraint with the same purpose will have different names in different databases. Choosing explicit names avoids that possibility and makes change management less complicated. Reduction in complexity for any process generally leads to higher availability.

In Oracle9*i*, the data dictionary views that display constraint information include an extra column named GENERATED to make it easy to identify constraints that use system-generated (as opposed to user-generated) names, as shown in the following example:

```
select constraint_name,constraint_type,generated
from all_constraints
where table_name like 'EMP%' and constraint_type='P';


CONSTRAINT_NAME   CONSTRAINT_TYPE   GENERATED
----------------- ----------------- ---------------
PK_EMP            P                 USER NAME
SYS_C001898       P                 GENERATED NAME
```

Use Meaningful Object Names

A consistent naming scheme for objects helps the DBA to identify the types of objects used in SQL statements more quickly by enabling the types to be identified from the name. For example, many development teams use a V_ prefix or _V suffix to identify views, an SP_ prefix to identify stored procedures, and _SEQ to identify sequences. The use of IX prefixes or suffixes for indexes also helps to make sense of explain plans.

The ability to directly identify the underlying objects in SQL speeds up the tuning process. Proponents of naming standards fall into two camps, those who use prefixes and those who prefer suffixes. Prefixes are easier to identify in SQL because they appear on the front of names, whereas suffixes make for easier identification of groups of related objects by enabling the use of a wildcard on the end of the base object name during queries of the Oracle data dictionary tables. The exact details of the standard are not as important as having one and adhering to it at a company level.

Trace Facilities

All applications—whether interactive graphical user interface (GUI) or batch—should provide built-in features for enabling and disabling Oracle SQL trace, including standard SQL tracing, tracing with bind variables, and tracing with event waits.

NOTE SQL tracing is used for the performance profiling of SQL statements submitted to the database server and is covered in more detail in Chapters 9 and 28.

The options can be set using the SET_EV procedure, as shown in the following examples for a session identified by SID=8 and SERIAL=149:

```
REM identical to ALTER SESSION SET SQL_TRACE TRUE, level 1
begin SYS.DBMS_SYSTEM.SET_EV(SI=>8,SE=>149,EV=>10046,LE=>1,NM=>'');end;

REM trace SQL with bind variables, level 5
begin SYS.DBMS_SYSTEM.SET_EV(SI=>8,SE=>149,EV=>10046,LE=>5,NM=>'');end;

REM trace SQL with event waits, level 9
begin SYS.DBMS_SYSTEM.SET_EV(SI=>8,SE=>149,EV=>10046,LE=>9,NM=>'');end;
```

```
REM trace SQL with bind variables, event waits, level 13
begin SYS.DBMS_SYSTEM.SET_EV(SI=>8,SE=>149,EV=>10046,LE=>13,NM=>'');end;

REM trace off for one session, level 0
begin SYS.DBMS_SYSTEM.SET_EV(SI=>8,SE=>149,EV=>10046,LE=>0,NM=>'');end;
```

Although the DBA can set SQL trace for any session, it's better for developers to provide facilities to set the trace within applications themselves, as this provides finer granularity over the traced sections of code. For example, it's possible to create a mapping table of procedure names and trace levels in a table, and have the procedure read and set the trace settings at the top of the procedure, and unset them at the end. That enables tracing to be turned on and off for individual procedures. In general, it's better to concentrate tracing efforts on the smallest code section possible because tracing can generate massive amounts of trace information in a short time. In the case of batch applications, tracing may need to be turned on at the start of processing, in which case the DBA will not be able to allow tracing early enough during execution by calling SET_EV from a separate session. Command-line utilities should enable tracing to be set through command-line arguments.

The ability to trace the values of bind variables and values is especially important when diagnosing the causes of obscure Oracle error messages in PL/SQL code, especially triggers. It's surprising how often code fails with incorrect values that, according to the developer, couldn't possibly be passed into subroutines. By building extensive tracing facilities into an application, the causes of such problems can be definitively identified more quickly. The inclusion of tracing facilities in code adds an overhead to the software development process. It usually pays off quickly. Chapter 28 shows more examples.

It's necessary for the application code to have access to the System ID (SID) and SERIAL# values that identify the current session in order to pass the values to the SET_EV procedure parameters SI and SE. One way to facilitate that is for the DBA to provide a wrapper around the SET_EV procedure that has the relevant privileges required to access the session settings. Chapter 25 on auditing shows three different ways for identifying the SID and SERIAL# for the current session.

Error Reporting and Logging

All application error messages should provide sufficient information to identify unambiguously the exact location at which an error occurred in code and the cause. Too often, applications use a single error number as a cover-all for several possible causes, and this makes root cause diagnosis more difficult than it needs to be for support staff. Oracle itself has been guilty of this. If you've ever reported an error message to Oracle worldwide support (WWS) and it has taken a long time to identify the root cause, that's probably because the developer of the underlying code could have provided a more specific cause for the error but chose not to in order to get the code completed quicker. Error-handling code is tedious for the developer to implement, but that's a poor excuse for not implementing in a way that can minimize support requirements. If error handling is not complete, then the onus is on the customer and WWS to try and

work out which of the range of possible causes is the real one. In such cases, most of the effort to resolve the problem needs to be made by the customer.

For the developer, incomplete error handling makes application delivery slightly quicker, but it's a completely false economy from a business point of view. For example, an extra couple of minutes spent by a developer adding code to identify the location of an error and to specify the exact cause can translate into savings in terms of hours when an error manifests itself in the code at run time. It's not necessary to report locations in a way that is meaningful to users but to report information in a way that is meaningful to support. The following is a PL/SQL code fragment showing the use of a simple numeric variable whereami and string, the_location, that can be used to identify the precise code location of errors in error messages:

```
...
the_location:='update_procedure';
whereami := 6;
cursor_name := dbms_sql.open_cursor;
whereami := 7;
dbms_sql.parse(cursor_name,update_sql,dbms_sql.v7);
whereami := 8;
ret := dbms_sql.execute(cursor_name);
whereami := 9;
dbms_sql.close_cursor(cursor_name);
whereami := 10;
cursor_name := dbms_sql.open_cursor;
whereami := 11;
dbms_sql.parse(cursor_name,update_last_check_sql,dbms_sql.v7);
whereami := 12;
ret := dbms_sql.execute(cursor_name);
whereami := 13;
dbms_sql.close_cursor(cursor_name);
EXCEPTION
when others then
         dbms_output.put_line(location||':'
                              whereami||':'||
                              sqlerrm||':'||sqlcode);
```

An example of error message reporting in one of Oracle's own products is quite enlightening. If you've set up Oracle Real Application Clusters (RAC), as described in Chapter 22, you'll be aware that Oracle uses the UNIX remote copy (rcp) command to enable the delivery of the Oracle software to each node in the cluster during installation using the Oracle Installer. Chapter 1 covers the basic configuration requirements for rcp. These days, other Oracle products use rcp also.

If you've failed to configure rcp correctly, then the Oracle Installer will report on a failure to connect to the other nodes in the cluster at install time. The developer could have taken the trouble to report that an rcp connection failed but chose not to. If he or she had, you could have addressed the problem in a couple of minutes. Instead, you, as the customer, are left to work out what failure to connect to the other nodes means. Eventually, through a process of trial and error, you'll probably discover that the

remote shell (rsh) configuration is incorrect and that resolving this fixes the install problem, but you shouldn't have to. The developer, with a little extra diligence, could have saved you the effort by being more specific on the cause of the problem when reporting the error. Who knows how many other problems will result in that same error message and cause the whole process to be repeated?

The mechanism used to report errors is just as important as the content of the messages themselves. Three main techniques are typically used for reporting errors, in addition to message dialogs typically returned by interactive GUI applications: files, tables, and email. Use of email is covered in Chapter 24 on monitoring and health checks. Files and tables have an advantage in that records of problems and status information are stored persistently to enable historical information to be searched easily.

The format used to log information should be designed to be easy for search tools to scan and follow a standard format. For example, log output might contain a fixed format date as the first field, then a severity indication in the second field, then the database instance in the third, and so on. If logging information has a poorly thought out format, it can add complexity to the processing performed by monitoring tools that need to raise alerts based on the logged information. Oracle's own alert log information does not follow a standard published format. As a result, it's sometimes necessary to join lines together to pull out the times that events occurred. Performing pattern matching on various parts of the message information is less complex if all messages follow a standard, predefined format and fit into a single line.

Error Logging Using Files

Oracle provides the UTL_FILE package to provide developers with a facility to log messages in server side files. In order to use the facilities of the UTL_FILE package, the DBA needs to make sure the UTL_FILE_DIR initialization parameter is set to the directories in which the file creation is to be enabled.

NOTE Sometimes DBAs choose to use the wildcard "*" as the UTL_FILE_DIR parameter. This requires less effort than choosing an explicit list of directories but represents a major security loophole because it enables any file owned by the UNIX oracle account to be accessed by UTL_FILE, including files that are part of the database. As a result, * should never be used for a production system.

Whenever you use UTL_FILE, great care needs to be taken to ensure that exceptions raised during logging via UTL_FILE don't affect the behavior of the application. In general, error and status logging failures should be transparent to the application. For example, you wouldn't expect an Oracle application to stop working if the disk on which the Oracle alert log is located became filled. The following example shows a procedure that can be used to log errors to a file /tmp/logfile.txt:

```
procedure sp_error_log(vtext in varchar2) is
       fhandle utl_file.file_type;
       location varchar2(16) := 'sp_error_log';
```

```
begin

  fhandle := utl_file.fopen('/tmp','logfile.txt','a');
  utl_file.put_line(fhandle,vtext);
  utl_file.fclose(fhandle);

exception
  when utl_file.invalid_path then
    sp_mail_log_error(location||': invalid path');
  when utl_file.invalid_mode then
    sp_mail_log_error(location||': invalid mode');
  when utl_file.invalid_operation then
    sp_mail_log_error(location||': invalid operation');
  when utl_file.invalid_filehandle then
    sp_mail_log_error(location||': invalid filehandle');
  when utl_file.write_error then
    sp_mail_log_error(location||': write error');
  when utl_file.read_error then
    sp_mail_log_error(location||': read error');
  when utl_file.internal_error then
    sp_mail_log_error(location||': internal error');
  when others then
    utl_file.fclose(fhandle);
end;
```

Errors during logging itself still need to be notified because valuable information is potentially lost while logging is failing. In this case, SP_MAIL_LOG_ERROR uses SMTP mail to notify the DBA team of logging failures. Chapter 24 contains a procedure SP_SENDMAIL that could be used as the basis for such a procedure.

It's worth pointing out that the procedure doesn't cache the file handle between calls and instead opens it every time. This means that the log file can be removed or compressed between calls and the logging will continue to work. The Oracle alert log behaves in a similar way. Opening files in append mode, as indicated by "a," creates the named file if it doesn't exist already. The procedure also reports all the possible causes of failures to write to the log file. It would be easier to simply use a single WHEN OTHERS exception to handle all the possible errors, but, as explained earlier, this makes it more difficult to identify the actual cause of the problem if logging fails. So the error handler mails the specific cause of any problem with logging so it can be resolved more quickly.

Error Logging Using Tables

Error logging using tables rather than files makes it significantly easier to report on error and status information because SQL can be used to perform the process and present the information. However, the need to commit error and status information in log tables in order to view it from other sessions can interfere with the transaction units of processing that failed. Oracle provides a feature known as autonomous (or nested) transactions to enable persistent logging to tables to be performed in a way that doesn't

have side effects on the transactions from which the log message is generated. Autonomous transactions are enabled using the AUTONOMOUS_TRANSACTION pragma in a PL/SQL procedure as shown in the following example:

```
create table log_table(msg varchar2(10));

create procedure log_record(p_msg in varchar2) is
  pragma autonomous_transaction;
begin
  insert into log_table values(p_msg);
  commit;
end;
```

You can demonstrate the behavior by inserting rows into LOG_TABLE from within the same session both directly via SQL INSERTS and using the autonomous transaction. Subsequent viewing of the contents of LOG_TABLE from another session will confirm that inserts performed via the autonomous transaction will be present, and those performed via INSERTS won't be visible until you commit them. This shows that the autonomous transaction has taken place without side effects on the main transaction in the other session.

Run Time Configuration

Application performance and supportability can benefit from the capability to set session attributes at run time. The attributes might include the optimizer goal for the session, the sort area size, trace settings, and resumable space allocation, among others. The following example shows a database logon trigger, which is an appropriate point at which to configure session-specific parameters:

```
create trigger session_config after logon on database
declare
begin

   configure_session_for_user(user);

exception
  when others then
   null;
end;
```

The session-specific settings might be held in a table containing name value pairs for each username and parameter, and be activated through the configure_session_for_user procedure. Some possible settings are shown in the following:

```
select * from user_parameters;

USERNAME   NAME            VALUE
---------- --------------- ---------
BATCH      optimizer goal  ALL ROWS
```

```
BATCH      resumable       YES
BATCH      sort area       10000000


ONLINE     optimizer goal  FIRST ROWS
ONLINE     sort area       65536
ONLINE     resumable       NO
```

By providing mechanisms for influencing session behavior through data held in tables, the developer makes it possible for code behavior to be enhanced without requiring more risky changes to application code.

Reporting on Application Status

Oracle provides a package, DBMS_APPLICATION_INFO, that contains procedures to enable developers to build facilities into their applications to report on the status of application processing. Each procedure call updates a related column in the V$SESSION table. The three package procedures and the related column in V$SESSION are shown in Table 6.1.

The DBA can then query the columns in V$SESSION using SQL to determine the application status, for example, if users report that the application appears to be stalled. Although the procedures can be used in any way the application developer chooses, the names are intended to suggest the usage. So the SET_CLIENT_INFO might be called once, at application connection time, to identify the application as follows:

```
begin dbms_application_info.set_client_info('DbCool'); end;
```

The SET_MODULE routine is typically used to identify a business process, which itself might map to a single PL/SQL stored procedure in the application. The ACTION_NAME, used to identify the current action within the module, can be set at the time of the call to SET_MODULE or be passed as an empty string at the top of a procedure and set separately using the SET_ACTION procedure as in the following example:

```
procedure sp_process_trades is
begin
  -- identify the module to V$SESSION
  dbms_application_info.set_module(
             module_name=>'Trade Processing',
             action_name=>'');

  for rec in (select trade_id from all_trades where processed='N') loop

      -- identify the current trade to V$SESSION
      dbms_application_info.set_action('Processing Trade '||trade_id);

      -- process the trade..
      sp_process_one_trade(rec.trade_id);

  end loop;
```

Table 6.1 DBMS_APPLICATION_INFO Procedures

PROCEDURE NAME	V$SESSION COLUMN
SET_CLIENT_INFO	CLIENT_INFO
SET_MODULE	MODULE
SET_ACTION	ACTION

```
   -- MUST UNSET THE VALUES when processing complete
   -- don't forget to unset in exception handlers also
   dbms_application_info.set_module(
              module_name=>'',
              action_name=>'');

end;
```

The use of DBMS_APPLICATION_INFO has a very beneficial side effect on performance management as well as supportability. The ability to identify business transactions is a critical success factor for efficient performance management. As you might expect, the best performance management tools can present information based upon time spent in business transactions, rather than individual microscopic SQL statements. The ability to do this relies on the application setting values for MODULE_NAME and ACTION_NAME at appropriate points in the code and unsetting them when processing is complete.

In effect, DBMS_APPLICATION_INFO provides a facility for developers and designers to instrument the performance of business transactions using a few simple procedure calls. If all Oracle applications were designed up front to include calls to DBMS_APPLICATION_INFO, then performance problems would be identified faster and solved faster. Chapter 16 on using performance management tools shows how the power of DBMS_APPLICATION_INFO can be unleashed using a suitable tool that takes advantage of the information.

It's worth noting that an additional procedure present in DBMS_APPLICATION_INFO, SET_SESSION_LOGOPS, can be used to log status information about long-running operations into the V$SESSION_LONGOPS table. Several of Oracle's own tools, such as RMAN, make use of this feature, and Oracle designers can do the same. The specification of the package is shown in the following code:

```
procedure set_session_longops(  rindex      in out pls_integer,
                                slno        in out pls_integer,
                                op_name     in varchar2 default null,
                                target      in pls_integer default 0,
                                context     in pls_integer default 0,
                                sofar       in number default 0,
                                totalwork   in number default 0,
                                target_desc in varchar2
                                   default 'unknown target',
                                units       in varchar2 default null);
```

Restartability

Restartability is a term I use to define the behavior of applications that can continue to function when Oracle database management system (DBMS) errors occur during processing. For example, if a tablespace space shortage occurs during a batch insert, an application can either exit and report an error, or report an error and attempt to repeat the failed operation on a timer until the underlying problem is fixed. Programmers using Oracle's precompiler interfaces, such as Pro*C, can take advantage of features in the language to identify the array index at which an array insert fails and restart the insert from that point. In general, making programs robust against database errors of any kind adds complexity to the code, and it can be difficult to balance the cost and complexity of extra coding against the benefits that result.

The consequences of aborting a long-running operation, rather than suspending it, can be very significant in terms of resource usage. For example, if a long-running batch job fails due to a space shortage, then the transaction needs to be rolled back and resubmitted. Both operations cause large amounts of redo generation. Prior to Oracle9*i*, it was the responsibility of the programmer to build features to work around space problems. For some situations like rollback segments filling up during a long transaction, there was often no practical alternative other than to roll back the transaction and restart. By their very nature, transactions that cause space problems tend to be long running and costly to repeat.

Resumable Operations in Oracle9*i*

Oracle9*i* provides resumable space management features that can be used to suspend sessions at the database level when space problems are encountered, until the DBA adds more space. Oracle's own products take advantage of these features. For example, Oracle's import utility includes a resumable=y option. Using features in the database rather than providing similar features at the application is preferable because it reduces application-coding complexity.

Three classes of spaces errors are resumable: those resulting from out-of-space errors on data segments and rollback segments, those resulting from maximum extents-reached conditions, and those resulting from space quota-exceeded errors. Even long-running queries that perform sorts that exceed temporary space availability can be resumed. Be aware that space allocation errors for rollback segments in dictionary managed tablespaces are not resumable. This should not be an issue, as you should be using the automatic undo features of Oracle9*i* in any case (as covered in Chapter 2). In the simplest case, making an operation resumable means adding the following SQL statement to a section of code:

```
alter session enable resumable;
```

Because suspended statements can lock system resources, possibly for an extended period, the RESUMABLE privilege is required in order to execute resumable operations. The following statement disables resumable operations:

```
alter session disable resumable;
```

Resumable operations that are suspended are shown in the DBA_RESUMABLE view, and if resumable operations are in use, it's essential that the DBA group performs monitoring for the early detection of such errors. Here is an example of a transaction that has suspended due to lack of undo space, which could be resolved by extending the undo tablespace datafile:

```
select error_msg from dba_resumable where status <> 'NORMAL';


ERROR_MSG
--------------------------------------------------------------------------------
ORA-30036: unable to extend segment by 16 in undo tablespace 'UNDOTBS2'
```

After a timeout period, which by default is set to 7,200 seconds and configurable through the ALTER SESSION ENABLE RESUMABLE TIMEOUT *seconds* statement, the suspended statement returns an error to the application. The DBMS_RESUMABLE package contains routines to enable resumable parameters to be set and to read named sessions, and it includes an ABORT procedure to enable a suspended operation to be aborted by a DBA, if necessary.

Constraining Undo Requirements

Long-running batch jobs, such as data load and purge operations, can exhaust the available undo space. Although Oracle9*i* provides resumable operations to provide the potential for space shortages to be fixed, it's not always a good idea to do that. Developers can take steps to constrain undo requirements by performing operations in batches, rather than in a single large transaction. In general, Oracle performs the same bulk DML operation faster when undo requirements are constrained within limits by performing the operation across several transactions. Chapter 19 contains an example showing that a reduction in import time can result from placing an upper limit on transaction size.

For simple purge operations using the DML DELETE operation, the ROWNUM pseudocolumn can be used to constrain transaction size to a fixed number of rows. For example, the following statement has unbounded undo requirements that are determined by the number of rows in the ALL_TRANSACTIONS table in the given state:

```
delete from all_transactions where processed='Y';
```

The following PL/SQL does the same job but commits after each batch of 10,000 rows deleted, which means that the undo requirements are limited to the space required to delete 10,000 rows, independent of the size of the ALL_TRANSACTIONS table:

```
while true loop

  delete from all_transactions where processed='Y'
  and rownum <=10000;
```

```
    exit when SQL%NOTFOUND;

    commit;

  end loop;
```

Summary

In some large organizations, the development and DBA teams often work in isolation without a clear understanding of each other's roles. By making the developer and DBA more aware of the requirements of the other, both performance and availability can be enhanced. Apparently mundane development practices, such as adherence to well-thought-out naming standards, code layout, and error reporting, can pay off significantly in production environments. The use of the procedures in the DBMS_APPLICATION_INFO package systematically throughout the development cycle for all Oracle applications can pay off significantly in terms of earlier problem diagnosis and an enhanced capability for performance management. The DBA can enhance availability by ensuring that organizations make use of the resumable space operations available in Oracle9*i*. The potential for reducing outages through these features is a very compelling reason to upgrade to Oracle9*i*, and the DBA has an important role to play as an evangelist for Oracle9*i* within the development community.