

Table Backup and Recovery using SQL*Plus

Peter G Robson

British Geological Survey

Abstract

A technique has been developed whereby a complete auditing system can be quickly established on any selected table or suite of tables, using a combination of SQL*Plus scripts and table triggers. The system has been successfully used for several years, proving to be totally reliable. It has now been extended to permit selective recovery of previously committed transactions, without the inconvenience of invoking the conventional Oracle systems-level restore procedures. Auditing and recovery can be carried out in an individual schema, or across multiple shared schemas.

The procedure requires the addition of some extra audit fields to be added to each table to be audited, as well as the creation of a history table for each audited table, to store pre-change rows, as well as the creation of a database trigger for each data table. All these components are created using automated SQL*Plus code generation features as previously described at this Conference.

The recovery process operates by identifying either an individual transaction to reverse, or a suite of transactions which form a continuous series up to the present table status. The presence of foreign key constraint relationships determine the minimum number of recovery transactions. This process is controlled by a suite of SQL*Plus scripts which again make use of code generation functionality.

There are several major benefits to this approach - firstly, audit control of any controlled table is total, and can extend back in time for as long as disk space is available to hold the history data. Secondly, the master table can be 'rolled back' to any chosen point in time in the past, so long as the history audit table contains a complete transaction record. Thirdly, it does not require dba-level permissions to function, but can operate on an individual (private) schema. Fourthly, it is immune to changes between different versions of Oracle.

Background

The British Geological Survey (BGS) has been collecting considerable quantities of data for over 160 years. Over the last 20+ years, an increasing proportion of this data has been stored in digital form. This data comes in two distinct forms.

Firstly, there is data which consists of unambiguous field measurements, such as the results of a geophysical survey, or the depth to the base of a borehole.

Secondly, data is collected which has been processed, if even to a small degree, by the scientist who collected that data. Therefore, some element of interpretation is being applied to what may be considered raw data. For example, the identification of a rock may be subject to the identification of palaeontological (fossil) evidence. But if that rock material is re-examined several years later in the light of the recognition of new fossil species, the identification of the rock may change. Thus the data in the digital copy must be changed too.

A further example of this subsequent amendment to data happens when old geological records, perhaps collected during the 19th Century, come up for review. The positional information on these old records may lack precision, and so a more precise expression of place may be defined. Some BGS data has been variously defined spatially by both the British national grid system, and then by latitude and longitude, each time with a different precision.

This sort of subsequent interpretation and re-interpretation goes on all the time in geology and indeed in many other disciplines which rely on the data they collect. Often, the same raw material can be subject to two entirely consistent interpretations, depending on the stand-point of the scientist carrying out the analysis.

This situation requires an ability to monitor all changes to the collected raw data. Of critical importance is to know when a record is inserted into the database, and by whom. Interestingly, the credibility of a geological interpretation varies from one scientist to another, thus the importance of knowing who was responsible for any particular re-interpretation!

Data Table Auditing

As a minimum all BGS data tables contain four auditing fields to cover the basic requirement of knowing who did what to what. These are:

<code>user_entered</code>	<code>varchar2(10)</code>
<code>date_entered</code>	<code>date</code>
<code>user_updated</code>	<code>varchar2(10)</code>
<code>date_updated</code>	<code>date</code>

These audit attributes are shown according to their current Oracle definitions. The user field never exceeds ten characters, as a codified expression of persons names is used, expanded through a dictionary look-up system.

With some records subjected to frequent modifications, it was realised that for a full audit trail on these modifications, a record of the changed records should be kept. This was done by creating a duplicate table, in every respect identical to the master table being audited, but with an additional three audit fields added to it. Every time a change was applied to the master table, the pre-change

row of that table was copied into the history audit table, with the appropriate audit attributes being populated. Thus, the history audit table contains a full and complete inventory of every change made to the data table since the creation of the history audit table.

This practice was first begun in the 1980's, when the main application system for entering and modifying Oracle data was SQL*Forms. In those days, the database could not support triggers, so all audit processing had to be done by triggers built into the various forms applications which were used. A suite of templates were developed which could be pasted into each new forms application (the text version of the Oracle .inp form was being used). It was vitally important that no modifications should be applied to any of these tables except through trigger-protected forms applications.

The advent of Oracle v7 introduced a major benefit in terms of trigger control, as all such controls could now be migrated into the database kernel, ensuring that all applications were automatically subjected to trigger control.

With kernel-based audit control triggers, any modifications to an audited table can be tracked for as long as the history audit table is retained. Consider the following example of a row being inserted into a table, modified several times, and then being finally deleted from that table. Of course there will be no evidence of this activity in the master data table, but only in the history audit table. These are the audit rows, presented in chronological order (eg sorted on Audit Date), but for reasons of clarity without the identity of the person applying the modifications:

Primary Key	Data	Date Entered	Date Updated	Audit Date	Audit Function
101	abc	12-Dec-2001		14-Dec-2001	UPDATE
101	abd	12-Dec-2001	14-Dec-2001	18-Dec-2001	UPDATE
101	abe	12-Dec-2001	18-Dec-2001	20-Dec-2001	UPDATE
101	abf	12-Dec-2001	20-Dec-2001	25-Dec-2001	DELETE

-- Table 1 The History Audit table for Master table PLOUG --

These few records, all relating to the same data row (as defined by primary key '101'), tells the full life history of this particular row of data. It was first inserted into the table on 12th December 2001 as indicated by 'Date_Entered' column. This date repeats for every instance of this row in the audit table.

The row was then updated successively on the 14th, 18th and 20th December, indicated by the Date Updated, the Audit Date and the Audit Function columns.

On the 25th December, the row was finally deleted. The final status of this row of data is recorded in the last Data field, which in its turn reflects all the various values held in that field during the life time of that data row.

Notice how the Audit Date is reflected in the Date Updated field in the 'next' appearance of the row. Before the row is deleted, the maximum value for the Audit Date in the history audit table is the same value as the Date Updated in the actual master data table. However, it is only the history audit table information which is being examined here.

This auditing is all carried out by pre-change database triggers firing in the background. Their construct is unique for each table, and as auditing applies to a very large number of database tables, some automated means of creating these control triggers was required. This was achieved by using SQL*Plus to generate each trigger. In fact, a whole suite of supporting scripts have been created, to add the audit fields to each table where they are missing, to create the history audit table for each

master data table, and then to create the audit control trigger itself. Additionally, a trigger is automatically generated to protect the table primary key from update modification.

An example table ('PLOUG') will be discussed to demonstrate how this auditing takes place.

```
SQL>desc PLOUG
Name                                Null?    Type
-----
PK                                    NUMBER
DATA                                  VARCHAR2(10)
DATE_ENTERED                          DATE
USER_ENTERED                           VARCHAR2(10)
DATE_UPDATED                           DATE
USER_UPDATED                           VARCHAR2(10)
```

The associated history audit table looks like this:

```
desc PLOUG_HIST
Name                                Null?    Type
-----
PK                                    NUMBER
DATA                                  VARCHAR2(10)
DATE_ENTERED                          DATE
USER_ENTERED                           VARCHAR2(10)
DATE_UPDATED                           DATE
USER_UPDATED                           VARCHAR2(10)
THEUSER                                NOT NULL VARCHAR2(10)
THEDATE                                NOT NULL DATE
THEFUNC                                NOT NULL CHAR(1)
```

Note that this audit table takes the name of the master table, and appends the suffix '_HIST' to that name to identify the history table. This is a BGS standard, and is automatically implemented by the script used to build this table. Note also the additional three fields in the audit table. This table can be automatically generated by use of a simple SQL script.

The audit trigger for table PLOUG after generation is displayed below. It is easier to understand the generator script if one has a clear concept of exactly what task it has to perform. Thus this following script is the required output from the generator script:

```
1  create or replace trigger PLOUG_AUD
2  before insert
3  or delete or update on PLOUG
4  referencing new as n old as o for each row
5  declare
6    func char(1);
7  -----
8  -- Trigger Name: PLOUG_AUD.SQL
9  -- Written by:   BGS
10 -- on date:     26-JUN-02
11 -----
12 begin
13 if inserting then
14  :n.user_entered :=user;
15  :n.date_entered :=sysdate;
```

```
16 elsif updating then
17   :n.user_updated :=user;
18   :n.date_updated :=sysdate;
19   func :='U';
20 elsif deleting then
21   func :='D';
22 end if;
23 --
24 =====
25 -- Auditing commences
26 =====
27 --
28 if updating or deleting then
29   insert into PLOUG_HIST
30   (
31   PK
32   DATA
33   DATE_ENTERED
34   USER_ENTERED
35   DATE_UPDATED
36   USER_UPDATED
37   theuser
38   thedate
39   thefunc
40   ) values (
41   :o.PK
42   :o.DATA
43   :o.DATE_ENTERED
44   :o.USER_ENTERED
45   :o.DATE_UPDATED
46   :o.USER_UPDATED
47   user
48   sysdate
49   func)
50   ;
51 --
52 end if;
53 end;
54 /
```

Line numbers are attached purely for reference purposes within the following text - they are NOT part of the output script.

The script can be broken down into three categories of data.

1 - Lines which contain the name of the Table and History Audit Table. These are obtained by prompting the user at run time. (*Lines 1,3,8,29*)

2 - Lines which pick up system constants (the current date and user) (*Lines 9,10,47-49*)

3 - Lines containing the columns or attributes which the Table comprises. (*Lines 31-39 and 41-46*)

4 - Lines which contain elements common to every audit trigger. (*Lines 2,4-7,11-28,30,40,50-54*)

Broken down in this fashion, the script can be seen to be constructed from a combination of string variables, the use of the string returned by the user when prompted for the name of the table, system variables and the returned attribute names when the Oracle data dictionary 'user_tab_columns' is queried.

Note how proper formatting has been maintained throughout the generated script, together with the insertion of comment lines, as well as the basic documentation providing the name of the trigger, the creator and date of creation. The means by which this has been achieved can be understood by carefully studying the script used to generate the trigger. This is of course a generic script, and will generate the appropriate audit trigger for any table. The trigger will generate with compile errors if the associated history audit table has not been previously created.

The trigger generating script is as follows:

```

set sqlprompt ""
set trims on
set feedback off
set pagesize 0
set verify off
set feedback off
set recsep off
undefine table
---
--- BUILD_TRIG.SQL
---
accept table prompt 'Enter Table_Name to build trigger: '
spool build_t.sql
select
'create or replace trigger '||'&&table'||'_aud '           ||chr(10)||
'before insert '                                           ||chr(10)||
'or delete or update on '||'&&table'                       ||chr(10)||
'referencing new as n old as o for each row '             ||chr(10)||
'declare'                                                  ||chr(10)||
'  func char(1);'                                          ||chr(10)||
'-----'                                                 ||chr(10)||
'-- Trigger Name: '||upper('&&table')||'_AUD'||'.SQL'     ||chr(10)||
'-- Written by:   '||user                                  ||chr(10)||
'-- on date:     '||sysdate                                ||chr(10)||
'-----'                                                 ||chr(10)||
'begin'                                                    ||chr(10)||
'if inserting then '                                       ||chr(10)||
'  :n.user_entered :=user;'                                 ||chr(10)||
'  :n.date_entered :=sysdate;'                             ||chr(10)||
'elsif updating then '                                     ||chr(10)||
'  :n.user_updated :=user;'                                 ||chr(10)||
'  :n.date_updated :=sysdate;'                             ||chr(10)||
'  func :='U'';'                                           ||chr(10)||
'elsif deleting then'                                     ||chr(10)||
'  func :='D'';'                                           ||chr(10)||
'end if;'                                                 ||chr(10)||
'--'                                                      ||chr(10)||
'-----'                                                 ||chr(10)||
'-- Auditing commences'                                    ||chr(10)||
'-----'                                                 ||chr(10)||
'--'                                                      ||chr(10)||
'if updating or deleting then'                             ||chr(10)||
'  insert into '||'&&table'||'_hist '                       ||chr(10)||

```

```

' ( '
from dual
/
---
---
select column_name,','
from user_tab_columns
where table_name=upper('&&table')
order by column_id
/
---
select
' theuser,' ||chr(10)||
' thedate,' ||chr(10)||
' thefunc ' ||chr(10)||
' ) values ( '
from dual
/
---
---
select
' :o.'||column_name,','
from user_tab_columns
where table_name=upper('&&table')
order by column_id
/
---
---
select
' user,' ||chr(10)||
' sysdate,' ||chr(10)||
' func)' ||chr(10)||
' ;' ||chr(10)||
'-- ' ||chr(10)||
'end if;' ||chr(10)||
'end;' ||chr(10)||
'/' ||chr(10)
from dual
/
spool off
set feedback on
set verify on
set heading on
start build_t.sql
undefine table
set pagesize 24
set sqlprompt "SQL>"
set recsep wrap

```

When this script is run from SQL*Plus, this is the result:

```

SQL>@BUILD_TRIG
Enter Table_Name to build trigger: PLOUG

Trigger created.

SQL>

```

For clarity, the user input in the above example is in upper case text.

There are several points to note in the above script.

This script is made up of five sub-scripts, such that the five results, when concatenated together, provides the full script to create the required trigger. The output of the above script is spooled to the file 'build_t.sql', which when run, creates the trigger.

Layout is all important, and therefore carriage-return / line-feed has been frequently forced using the '||chr(10)||' CRFL (ascii 10) control at the end of each line. These components of the script have been set out for maximum clarity.

Documentation is as important as layout, and it will be noted that this script automatically inserts the name of the trigger being built, based on a concatenation of the table_name (prompted for) and the string '_AUD', the schema userid of the person running the script, and the current date.

Examine the SQL*Plus 'set' commands at the top and base of the script. These are vital to get correct, as everything which spools to the output file must only be legal SQL*Plus commands in their own right. Thus headings are turned off, and feedback is switched off to prevent the number of rows being returned as a numerical score. A more subtle setting is 'recsep off', which ensures no blank line appears between the results of each individual script. This is not such an important feature when building a trigger - SQL*Plus will tolerate blank lines in this situation, but other script constructs will not.

Although this is a simple example of a dynamically generated SQL script, it contains a number of important construct details. It forms the basis for the development of several other similar scripts.

Auditing Tables and Database Transactions

Once the audit table is in place and the trigger has been constructed and compiled, all changes to the master data table will then be captured into the audit table. It will then be possible to track all changes applied to every single row within the whole table. The next stage in the process is to collect information regarding multi-table transactions. This is done by using a single transaction log table into which information is collected for all nominated table transactions. The situation can be illustrated diagrammatically as follows (figure 1):

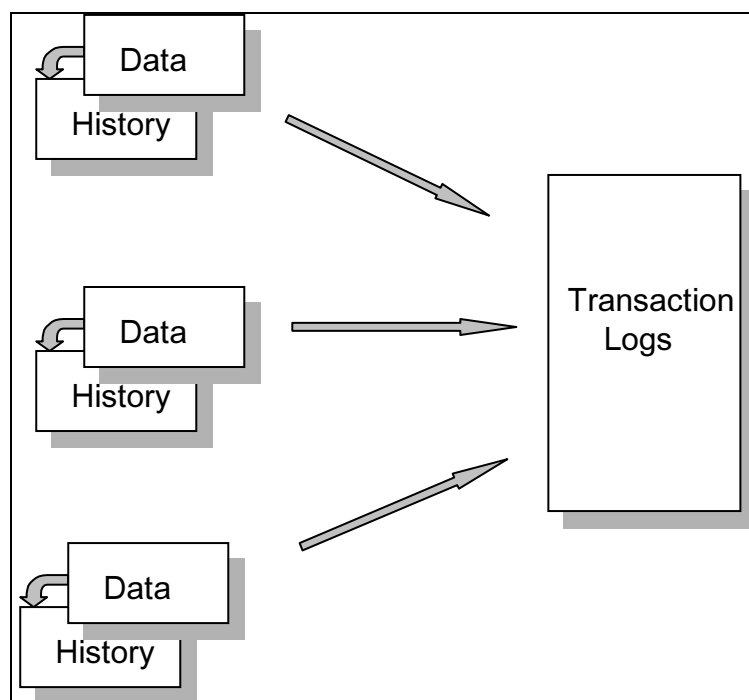


Figure 1

Note how update modifications results in a related transaction from each data table to its associated history table, while every transaction to each individual table results in an entry to the central transaction log table.

There is a subtle point involved here with regard to logging each transaction in unit time, which relates to the inability of SQL*Plus in Oracle to break down time events to less than one second (unless one uses a separate PL/SQL call). Where two or more transactions all commit modifications to the table - whether to the same row or to several rows does not matter - within the same second, it is impossible to track the chronology within that single second.

The answer to this problem is the generally accepted technique of generating an integer sequence number which simply increments for each transaction. Chronology is then based on the sorted order of this integer.

This technique has been used in the replication system built by the author. It is employed in a transaction log table which collects minimal data for each transaction, sufficient to identify the following information:

- The Name of the table being modified (and Owner if multiple schemas are being used)
- The Date of modification
- The Type of modification (Insert, Update or Delete)
- The Identify of the row being subject to modification
- The integer indicator of transaction chronology.

All five parameters are straight forward, with the exception of the row identifier. This can be on the basis of either the primary key of the table in question, or on the basis of the Oracle rowid value. After some examination and testing, the latter option was adopted, for the following reasons:

- Rowid has a format common to all tables
- The Primary Key format varies from table to table
- Rowid provides a very fast access path into the table
- All data tables are in the same tablespace (rowid can be duplicated across tablespaces)

Although rowid can be lost in the event of a tablespace failure, or a total export / import, neither of these events has impacted operating procedures to date. Just in case this should ever happen, the primary key attributes are also collected for each transaction, such that rowid values could be re-constructed should a tablespace require a full export / import.

A 3-table example of Audit and Recovery

It is important at this point to explain why the order in which transactions are logged is critical. This only becomes an issue if the tables being logged take part in formalised referential associations with each other.

Consider a table named `Department`, with an attribute `Location_id`. This attribute forms a foreign key to a table called `Location` (indicating where the company has established offices at various places). Within a table called `Employee` is a foreign key attribute `Department_id` which points to the table `Department`. The business rules of this simple model are that no employee may exist unless assigned to a `Department`, and no `Department` may exist unless assigned to a `Location`.

It is therefore axiomatic that one cannot insert a new employee into the `Employee` table until the `Department` to which the `Employee` is to be assigned has itself been created. Similarly, no new `Department` can be created in isolation of the `Location` of the office to which it belongs.

We therefore have a simple dependency between these three tables. Transactions which created a `Location`, then `Departments` within that `Company`, and then `Employees` within those `Departments`, can be tracked, and the order in which they were created would be reflected in the chronology integer, which order would reflect the constraints of the PK / FK relationships established for this model.

If the transaction log table were examined, it might look something like Table 2:

Chrono Counter	Row Identity	Table Name	Transaction Date	Action
3456	AABBCCDD1	Location	12-October-1998	Insert
3457	AABBCCDD7	Department	12-October-1998	Insert
7653	AABCCCDD3	Department	15-October-1998	Insert
8321	AAGGDDDD6	Employee	16-October-1998	Insert
8537	AAGGGFFDD2	Employee	16-October-1998	Insert
8583	AABBDDDD5	Employee	16-October-1998	Insert

-- Table 2 -- The Transaction Log / Chronology Table --

The row identifier illustrated is not of course accurate, but indicates the type of format that the Oracle rowid follows.

Event 3456 records the creation of a new row in the table `Location`, which established the existence of a new company office at some location.

Events 3457 and 7653 record the creation of two new `Departments` assigned to the `Location` already established by virtue of the first event.

Event 8321, 8537, and 8583 all record new `Employees` being inserted into the `employee` table, and being assigned to one of the new `departments` created above, via the `Department_id` foreign key in the table `Employee`.

Note the similarity between this table and the history audit table previously illustrated (Table 1). The major difference is that the chronology table lists every table which is modified, together with the order in which these modifications are applied. The chronology table, taken together with every individual history audit table, provides a *totally comprehensive record* of every single change which has taken place in the database since auditing commenced, and the order in which those changes took place.

Mistakes can be made, and sometimes errors in data insertion may not be detected immediately. An error may remain undetected until a table data manager examines an audit log in detail, and then flags up the error. Hopefully, the error may be corrected by a simple SQL DML command, but it is possible that the erroneous modification to the table was tightly dependent on a whole suite of other dependent tables, via referential constraints, implemented by primary key / foreign key dependencies.

This is the time when a transaction may have to be reversed out of the database, and done such that the dependent transactions are backed out in the reverse order in which they were applied.

The Transaction Reverse Process in Detail

Three types of transactions exist (insert, update and delete), and all three can be reversed. In the first instance, the principles behind the reverse process will be discussed by examining the reversal of an insert transaction.

Take the example of a department having been created, and populated with several employees, to be subsequently discovered to have been given the wrong name. That department name cannot be corrected until the dependent employees are corrected - but they cannot have their department name changed to a non-existing department. The only answer is to delete the employees entry, and then delete the department entry, and resubmit the entire suite of transactions with corrections.

Note: - 'On Cascade' could be used here, but in principal, this is regarded as a very powerful and therefore potentially dangerous setting, and one that is discouraged from use. If the dependencies are more complex than simply one more dependent table, a 'cascade on delete' action can have totally unforeseen consequences.

In fact, one could solve the problem by locking the tables involved in the transaction, disabling the relevant triggers, correcting the employee and department tables, and then enabling the triggers again. However, this would result in destroying the integrity of the table audit, as no evidence would be held in the audit history table of the correction being made. For this reason, this short-cut solution is not recommended.

If the full auditing system described above is operative, it is possible to simply track back through the chronology table until one discovers the erroneous data modification, and then reverse every one of the dependent transactions from that point forward.

The word 'dependent' is important here, as there will in all probability have been numerous transactions applied to the database since the point of error, but only a small percentage will be affected by the erroneous transaction. So only those dependent transactions are reversed out.

The actual method of doing this is as simple as running a SQL*Plus script, which prompts the user for the transaction number from which all dependent transactions have to be rolled back. This number then identifies the table against which the incorrect transaction was first applied, from which the script determines the dependencies which flow from this table. Every subsequent transaction which was applied to the original table, and all dependent tables, are then reversed, the process completing with a full log presented to the user.

The original transaction can then be corrected, and the log of modifications then rolled forward, to re-create the database system in its latest state, but with the original error corrected.

In the above example, the erroneous transaction is number 3457, where one of the two new departments was created. The reversing script can then examine the Oracle data dictionary to determine all those tables which have foreign key dependencies against the table `Department`. Where this dependent table appears with a chronological count number greater than 3457, the transaction against that table must be reversed. Looking at the transaction log table, it will be evident that event 7653 is unaffected by this reverse process, but events 8321, 8537 and 8583 are all vulnerable, and must be examined further.

At this point it is not possible to know whether all three or indeed any of those three events were affected by the incorrectly named department. This can be determined by a comparison between the primary key of the department table for that row with the incorrect name, and the foreign key attributes in the employee table. Where they are the same - that employee row must be reversed out of its original transaction, eg deleted.

In a more complex system in which the employee table also had foreign key dependencies, these would have to be identified, and then validated against the primary key of the employee table, to determine if they too should be reversed out. The entire process would thus cycle round until all dependencies had been tested, and the minimum number of rows identified for which a dependency existed.

In summary, therefore, the structure of the insert reverse process follows these stages:

The transaction log table is queried to identify the chronological number of the event from which the reversal is to be commenced.

The hierarchy of dependent tables is established, by querying the Oracle data dictionary constraint table.

The primary key / foreign key values are compared, to determine which of the dependent tables in the transaction log have to be reversed.

A list of all pending transactions to be reversed is created.

The reversal commences and runs to conclusion.

Reversing Inserts, Updates and Deletes

In the above example, the initiating transaction was an erroneous insert, which had to be reversed by physically deleting the row from the database. Before that could be done, a number of dependent transactions had to be identified. These other transactions could have been inserts, updates or deletes. Each type would require a slightly different processing method, but fundamentally the approach is very similar.

In fact, transaction 3457 could be corrected by updating the record in question, once all the dependent rows had been backed out of the database. However, for the sake of this discussion, it is assumed that correction is by deletion followed by insertion.

As the list of pending transactions is scanned, each table transaction is identified as one of the three major types. For an insert - the row simply has to be deleted. This can be done with a statement as simple as 'delete from <table> where rowid = '<rowid>'; as the rowid value is held in the transaction log table.

For a delete, the row has to be inserted back into the master data table. This is achieved by creating an insert script from the history audit table for that row. The identity of the deleted row in the history audit table is known (the transaction log table holds a pointer to the deleted row in the history audit table), and therefore an insert script can be dynamically generated in which all the attribute values are populated from that history audit table row.

The update reverse is slightly more complex than the others. Here, the pre-update version of the row has to be extracted from the history audit table, and manipulated into an update statement based on the rowid of that table as held in the transaction log table. So long as the rowid pointers have not been lost, the rowid is the fastest and easiest way to affect the corrected update. If the rowid pointers have become lost, then the original table primary key will have to be used. The update is then applied to the master data table, thus returning that row to its original pre-update condition.

All three transaction types are reversed using dynamically generated SQL*Plus scripts

Conclusions

The table auditing processes described here have been in reliable operation for more than a decade. There is no reason to doubt the 'industrial strength' of this system. Onto this has been built a replication system, which although not described in this paper, has also proved to be a very reliable means of replicating data in a true master-master asynchronous mode between instances of Oracle on both local and wide area networks. The latest version of this replication software has been running reliably since completion in April 2001, and again has proved highly resilient.

The construction of the transaction reversal process was the logical extension to the above work. It has used exactly the same logic, and the same technical approach, to facilitate the reversal of transactions out of a data system in which referential integrity is fully developed. This is the key issue with both replication and recovery - they must respect the referential integrity constraints built into the data system.

The implementation of all of these solutions has been achieved in a very modular fashion. A whole suite of SQL scripts have been carefully constructed, which integrate one with another to provide a total solution. Many of these scripts are generated by other scripts, thus ensuring uniformity of function. Although the same solution could have been achieved in a 3GL program, using SQL*Plus

avoids the compile/test cycle. It has also proved to be a valuable exercise in exploring the limits of SQL processing. Finally, it also ensures that the functionality of all these modules is totally unaffected by subsequent releases of Oracle, and will continue to be unaffected for as long as SQL underpins the Oracle product.