

# RESUMING, RESTARTING, RESTORING: THREE R'S OF DATA WAREHOUSE FAULT TOLERANCE

*Stewart Bryson, Rittman Mead America*

## INTRODUCTION

Fault tolerance is the ability to recover from errors, regardless of whether those errors resulted from hardware issues, software issues, general systems issues (network latency, out-of-space errors), or human mistakes. The main point is that BI/DW environments present unique challenges, both for the operations and development teams. I'm not preposing that the divide between transactional and reporting systems is necessarily vast; we still need redundant storage systems and dependable backup strategies. I am preposing, however, that one-size-fits-all approaches to fault-tolerance is problematic, and applying standards that evolved in support of transactional systems may not provide the best protection for BI/DW environments.

The operational teams (DBAs, Unix Admins, Storage Admins, etc.) and the development teams (source system extraction, ETL) have to work closer in a BI/DW than perhaps they do in OLTP environments. Of course, OLTP developers have to write scalable code, but I think that's within their control for the most part. ETL developers are thrashing around millions or billions of rows of data, and because of this, everything needs to be well-oiled: undo spaces need to be available, temp space needs to be plentiful, standard operational jobs such as backup and recovery or statistics gathering need to keep the batch load window in mind, etc. Whereas OLTP code is exclusively SQL... ETL code is packed full of DDL: partition-exchange loads, index and constraint maintenance, table truncates, the whole gamut.

I'm proposing constructing a triage process where we can catch and correct errors early so that their damage is minimal. In essence: don't let simple errors turn into weekend-long data correction issues, where millions of rows need to be updated or deleted. Let's work smarter, not harder, and this means maximizing our technology investment by using Oracle Database features instead of custom development whenever possible.

## RESUMING

Resuming is the ability to continue effortlessly after an error, without causing an "aftermath" full of data corrections and true ups as described above. The Oracle Database has offered out of the box functionality for resuming since version 9i with Resumable Space Allocation. Resumable operations are supported for SELECT queries, DML and DDL, and can be enabled at either the system or the session level. To enable at the system level, the *resumable\_timeout* database parameter should have a nonzero value:

```
SQL> alter system set resumable_timeout=3600;
```

```
System altered.
```

```
SQL>
```

To enable resumable operations at the session level, the statement follows this basic syntax, with the *timeout* and *name* clauses being optional:

```
ALTER SESSION ENABLE RESUMABLE <TIMEOUT n> <NAME string>;
```

The *timeout* value is specified in seconds, and if omitted, the default value of 7200 is used, or 2 hours. The *name* clause gives the resumable session a user-friendly name to make finding information about in the Oracle dictionary much easier (as we will see later). Enabling resumable operations for the session level requires that the *resumable* privilege has been granted:

```
SQL> grant resumable to stewart;
```

```
Grant succeeded.
```

```
SQL>
```

Resumable operations can also be enabled with the Oracle utilities SQL-Loader, Export/Import and Datapump. The command-line parameters *resumable*, *resumable\_name* and *resumable\_timeout* exist to mimic the functionality mentioned above.

To demonstrate Resumable Space Allocation, I'll create a situation that is ripe for a space allocation error: I'll put an empty copy of the SALES fact table from the SH schema in a tablespace with only 250K of space:

```
SQL> create tablespace target datafile '/oracle/oradata/bidw1/target01.dbf' size 250K;
```

```
Tablespace created.
```

```
SQL> create table target.sales tablespace target as select * from sh.sales where 1=0;
```

```
Table created.
```

```
SQL>
```

I'll enable my session for resumable operations, and then load some records into the table, which should cause it to suspend. Since I always instrument my code, I'll first register my code with the database using DBMS\_APPLICATION\_INFO so I have an easy way to guarantee consistency when referring to processes:

```
SQL> exec dbms_application_info.set_module('SALES fact load','insert some rows');
```

```
PL/SQL procedure successfully completed.
```

```
SQL>
```

```
SQL> DECLARE
```

```
2     l_module VARCHAR2(48) := sys_context('USERENV','MODULE');
```

```
3 BEGIN
```

```
4     EXECUTE IMMEDIATE
```

```
5     'alter session enable resumable timeout 18000 name ''||l_module||''';
```

```
6 END;
```

```
7 /
```

```
PL/SQL procedure successfully completed.
```

```
SQL> insert into target.sales select * from sh.sales;
```

I open up a second session and start another transaction against the TARGET.SALES table to demonstrate that multiple sessions can be suspended at the same time waiting for the same space error to be corrected:

```
SQL> exec dbms_application_info.set_module('SALES fact load2','insert more rows');
```

PL/SQL procedure successfully completed.

```
SQL>
```

```
SQL> DECLARE
```

```
 2     l_module VARCHAR2(48) := sys_context('USERENV','MODULE');
 3 BEGIN
 4     EXECUTE IMMEDIATE
 5     'alter session enable resumable timeout 18000 name ''||l_module||''';
 6 END;
 7 /
```

PL/SQL procedure successfully completed.

```
SQL> insert into target.sales select * from sh.sales;
```

I'll have a look at the DBA\_RESUMABLE view (there is also a USER\_RESUMABLE version) for the suspended sessions. Even though I could get all the following information with a single SQL statement, I broke then up for enhanced visibility:

```
SQL> select name, start_time, suspend_time, status from dba_resumable;
```

NAME	START_TIME	SUSPEND_TIME	STATUS
SALES fact load2	02/06/10 10:33:33	02/06/10 10:33:33	SUSPENDED
SALES fact load	02/06/10 10:29:03	02/06/10 10:29:03	SUSPENDED

2 rows selected.

```
SQL> select name, sql_text from dba_resumable;
```

NAME	SQL_TEXT
SALES fact load2	insert into target.sales select * from sh.sales
SALES fact load	insert into target.sales select * from sh.sales

2 rows selected.

```
SQL> select name, error_msg from dba_resumable;
```

NAME	ERROR_MSG
SALES fact load2	ORA-01653: unable to extend table TARGET.SALES by 8 in tablespace TARGET
SALES fact load	ORA-01653: unable to extend table TARGET.SALES by 8 in tablespace TARGET

2 rows selected.

```
SQL>
```

The Oracle Database also publishes server alerts about suspended transactions using the Server Generated Alerts infrastructure. This infrastructure uses the AWR toolset, the server package DBMS\_SERVER\_ALERT for getting and setting metric thresholds, and the queue table ALERT\_QUEUE to hold alerts that have been published from AWR. We could write custom processes to mine ALERT\_QUEUE for these alerts, but the easiest way to configure and view server alerts is using Oracle Enterprise Manager (OEM). On the Alerts section of the main OEM page, we can see three different alerts generated by the Oracle Database:

Alerts

Severity	Category	Name	Impact	Message	Alert Triggered
Warning	Invalid Objects by Schema	Owner's Invalid Object Count		208 object(s) are invalid in the PUBLIC schema.	Feb 6, 2010 12:51:07 AM
Warning	Invalid Objects by Schema	Owner's Invalid Object Count		12 object(s) are invalid in the OWBSYS_AUDIT schema.	Feb 6, 2010 12:51:07 AM
Warning	User Audit	Audited User		User SYS logged on from localhost.localdomain.	Feb 6, 2010 10:16:54 AM
Warning	Session Suspended	Session Suspended by Tablespace Limitation	→	Operation on resumable session SALES fact load session id 42 suspended because of errors in tablespace TARGET. Error message is ORA-01653: unable to extend table TARGET.SALES by 8 in tablespace TARGET	Feb 6, 2010 10:29:03 AM
Warning	Waits by Wait Class	Database Time Spent Waiting (%)	→	Metrics "Database Time Spent Waiting (%)" is at 98.96692 for event class "Configuration"	Feb 6, 2010 10:32:26 AM
Critical	Tablespaces Full	Tablespace Space Used (%)	→	Tablespace TARGET is 100 percent full	Feb 6, 2010 10:37:42 AM

Figure 1: OEM Alerts Screen

If we click on the “Session Suspended” link, we can see the multiple alerts generated in this category:

Alert History

Severity	Timestamp	Message	Last Comment	Details
Warning	Feb 6, 2010 2:05:35 AM	Operation on resumable session SALES fact load session id 32 suspended because of errors i...	→	-
Warning	Feb 6, 2010 10:29:03 AM	Operation on resumable session SALES fact load session id 42 suspended because of errors i...	→	-

Figure 2: OEM Alert History

Another alert generated indirectly by the suspended transaction is the “Configuration” class event caused by our session “waiting” to continue. The Oracle wait event interface can show us information about the suspend waits on the system:

```
SQL> SELECT event,
2      SUM(time_waited) time_waited,
3      SUM(total_waits) total_waits,
4      AVG(average_wait) average_wait
5  FROM gv$session_event
6  WHERE lower(event) LIKE '%suspend%'
7  GROUP BY event
8  ORDER BY time_waited ASC
9  /
```

EVENT	TIME_WAITED	TOTAL_WAITS	AVERAGE_WAIT
statement suspended, wait error to be cleared	305373	1377	221.78

1 row selected.

```
SQL>
```

To resolve the space issue, I'll enable AUTOEXTEND on the TARGET tablespace, and have a look around to see if anything changed:

```
SQL> alter database datafile '/oracle/oradata/bidw1/target01.dbf'
  2 autoextend on next 10M maxsize 1000M;
```

Database altered.

```
SQL> select status, resume_time, name from dba_resumable;
```

STATUS	RESUME_TIME	NAME
NORMAL	02/06/10 10:56:49	SALES fact load2
NORMAL	02/06/10 10:56:49	SALES fact load

2 rows selected.

```
SQL>
```

The Resumable Space Allocation feature includes the AFTER SUSPEND trigger, which allows the specification of a system-wide trigger that will fire whenever a transaction is suspended. The typical use for this functionality is alerting, even though suspended operations are written to the alert log. There are some features in the DBMS\_RESUMABLE package that may make sense when writing an AFTER SUSPEND trigger:

```
SQL> desc dbms_resumable
```

```
PROCEDURE ABORT
  Argument Name          Type          In/Out Default?
-----
SESSIONID               NUMBER        IN
FUNCTION GET_SESSION_TIMEOUT RETURNS NUMBER
  Argument Name          Type          In/Out Default?
-----
SESSIONID               NUMBER        IN
FUNCTION GET_TIMEOUT RETURNS NUMBER
PROCEDURE SET_SESSION_TIMEOUT
  Argument Name          Type          In/Out Default?
-----
SESSIONID               NUMBER        IN
TIMEOUT                 NUMBER        IN
PROCEDURE SET_TIMEOUT
  Argument Name          Type          In/Out Default?
-----
TIMEOUT                 NUMBER        IN
FUNCTION SPACE_ERROR_INFO RETURNS BOOLEAN
  Argument Name          Type          In/Out Default?
-----
ERROR_TYPE              VARCHAR2      OUT
OBJECT_TYPE              VARCHAR2      OUT
OBJECT_OWNER             VARCHAR2      OUT
TABLE_SPACE_NAME         VARCHAR2      OUT
OBJECT_NAME              VARCHAR2      OUT
SUB_OBJECT_NAME          VARCHAR2      OUT
```

```
SQL>
```

This package adds functionality for writing custom processes to handle suspended sessions. The `SPACE_ERROR_INFO` function returns specifics about the table and tablespace affected by the space error. We could code a series of checks enabling custom actions depending on which objects were affected. A suspended process could be ended prematurely with the `ABORT` procedure, or more time could be added using the `SET_TIMEOUT` procedure. I had one client explain how she had written an `AFTER SUSPEND` trigger that collected information about the tablespace affected so that an `ALTER DATABASE... RESIZE...` command could be issued to add more space to the affected datafile. I didn't have the heart to tell her that she had written a feature that already existed in the database: `AUTOEXTEND`.

All ETL mappings and flows, as well as database maintenance processes, should use Resumable Space Allocation, preferably using the `name` clause in conjunction with `DBMS_APPLICATION_INFO`. Setting a `resumable_timeout` value at the system level can be scary, because a single suspended transaction could cause locks that reverberate all the way through the system. Should this concern us? Are there any processes in our batch load window or with any of our operational maintenance processes that we wouldn't want to suspend instead of fail, no matter how many processes back up waiting for it to be cleared? It could be dangerous to use some synchronous replication technologies to move data to an ODS, but that seems to be the only typical use case in the BI/DW arena that wouldn't benefit from this feature.

I've never found much reason to use the `AFTER SUSPEND` trigger though. Data warehouses should have production-type monitoring running already, just like other production systems. OEM is more than satisfactory for basic monitoring and alerting, and with the Server Generated Alerts introduced in 10g, it forms a complete product for Oracle environments. Regardless of the monitoring solution, it should support the execution of simple queries against the database and alert based on the results of those queries. A select against the `DBA_RESUMABLE` table provides all the information required to send out an alert, and with features such as `AUTOEXTEND`, I just can't see a need for procedural code when a transaction has suspended. At the very least, a standard alert log monitoring process could watch for the suspended sessions and respond when one occurs.

## **RESTARTING**

What happens when the errors are not so simple, and Oracle's built in resuming functionality is of no use? In these cases, the ETL processing will fail, and we will have to correct the original error, and possibly clear up any aftermath before restarting the process.

There are numerous approaches to building a framework for managing our overall load process, storing information about what has completed before an error or failure occurs. I describe this as *code-controlled restarting*: building smart ETL process flows and instrumented mappings so that the exact course of the data is charted from beginning to end. This is a required component, and should be included in any best practices approach to ETL development. The restartability function I'm concerned with here, however, is what I'll call *data management restartability*, and involves correcting data sets after failures to enable processes to be restarted.

Flashback provides the capability to revert the entire database, or smaller portions of it, to a particular point in time. For Oracle, a "point in time" is always referenced by the System Change Number(SCN), an internal clock for the Oracle Database. It auto-increments every time a transaction commits, but other sources, such as the SMON process, can increment the SCN as well. The current SCN can be viewed in many of the data dictionary tables, as well as using the `DBMS_FLASHBACK` server package:

```
SQL> select current_scn,
 2  dbms_flashback.get_system_change_number
 3  from v$database;
```

CURRENT_SCN	GET_SYSTEM_CHANGE_NUMBER
2536238	2536238

1 row selected.

SQL>

We can convert from SCN's to timestamps and back again, but this conversion is not exact. The Oracle documentation states that the functions are precise to about 3 seconds, which is evident from this example:

```
SQL> select SCN_TO_TIMESTAMP(2536238) scn
       2 from dual;

SCN
-----
02/09/2010 12.47.26000000000

1 row selected.

SQL> select TIMESTAMP_TO_SCN('02/09/2010 12.47.26000000000') ts
       2 from dual;

       TS
-----
    2536237

1 row selected.

SQL>
```

Even though we access both Flashback Database and Flashback Table with the same general syntax, specifying SCN's for both incarnations, the technical implementations under the hood are drastically different. Flashback Table is completely an UNDO operation, and is really not a new feature at all. Oracle has always used the UNDO space (rollback segments before that) to manage the state of our tables at a particular SCN to allow the robust multi-versioning that keeps reads and writes from blocking one another. Flashback Table is just an “exposing” of the multi-version API, in a manner of speaking, so that any SCN can be viewed as long as sufficient UNDO exists.

Flashback Database doesn't use UNDO at all, instead using new instance file structures called flashback logs in conjunction with a little bit of archived REDO. Flashback logs contain prior versions of changed blocks, and we use the version of the block just before the SCN of interest and put it back into the datafile, followed by REDO log recovery to get the database to the exact point of the SCN. So when certain aspects of the load need to be “undone” before we start, Flashback makes this possible:

I created copies of the CUSTOMERS, PRODUCTS and SALES tables including data from the SH schema. Next, I enable row movement on the new SALES table. This step would need to be implemented for all tables in the data warehouse that are a consideration for Flashback Table:

```
SQL> alter table target.sales enable row movement;

Table altered.

SQL>
SQL> SELECT count(*) FROM target.products;

COUNT(*)
-----
       72

1 row selected.

SQL> SELECT count(*) FROM target.customers;
```

```
      COUNT(*)  
-----  
      55500
```

1 row selected.

```
SQL> SELECT count(*) FROM target.sales;
```

```
      COUNT(*)  
-----  
      918843
```

1 row selected.

```
SQL>
```

Next, I create what's called a *restore point* in the database. This allows me to give an intelligent name to a particular SCN and is similar to tagging a release in Subversion or other source control product. Before each new step in the process, I create a restore point so that each phase has a tagged, referenceable SCN. As I'm using the idea of a unique, sequence-generated *execution ID* for each subsequent batch load to the data warehouse, I'll work that number into the name of my restore points:

```
SQL> create restore point dw_load_1001;
```

Restore point created.

```
SQL>
```

Now imagine that I do the processing that moves the necessary files into place (if any), prepares and loads the ODS tables, etc. After that... I move into the load of the dimensional model itself:

```
SQL> create restore point load_customers_1001;
```

Restore point created.

```
SQL> exec dw_load.load_customers;  
Number of records loaded: 0
```

PL/SQL procedure successfully completed.

```
SQL> create restore point load_products_1001;
```

Restore point created.

```
SQL> exec dw_load.load_products;  
Number of records loaded: 72
```

PL/SQL procedure successfully completed.

```
SQL> create restore point load_sales_1001;
```

Restore point created.

```
SQL> exec dw_load.load_sales;
5 indexes and 0 local index partitions affected on table TARGET.SALES
Number of records loaded: 699999
Rebuild processes for unusable indexes on 28 partitions of table TARGET.SALES executed
No matching unusable global indexes found
```

PL/SQL procedure successfully completed.

SQL>

The data warehouse load ran without error, so I can assume that it was successful, right? In looking back over the log, I see that no rows were actually loaded into the CUSTOMERS table. After researching the issue, I discover that the Change Data Capture process on the source system experienced problems, and there were no rows published to the CUSTOMERS change set. Since the load didn't technically fail, the process continued on to load the SALES table, and it's very likely that many of the rows in the fact table have the wrong surrogate key from the CUSTOMERS table.

In my experience, ETL load failures and the subsequent aftermath (investigations, massive data corrections, etc.) cause more downtime than any other hardware or software related issues. With the approach I've put into place, however, this aftermath shouldn't concern me, because now I can simply "undo" it:

```
SQL> flashback table target.sales to restore point load_sales_1001;
```

Flashback complete.

```
SQL> select count(*) from target.sales;
```

```
   COUNT(*)
-----
   918843
```

1 row selected.

```
SQL> create restore point new_load_customers_1001;
```

Restore point created.

```
SQL> exec dw_load.load_customers;
Number of records loaded: 99
```

PL/SQL procedure successfully completed.

```
SQL> create restore point new_load_sales_1001;
```

Restore point created.

```
SQL> exec dw_load.load_sales;
5 indexes and 0 local index partitions affected on table TARGET.SALES
Number of records loaded: 699999
Rebuild processes for unusable indexes on 28 partitions of table TARGET.SALES executed
No matching unusable global indexes found
```

PL/SQL procedure successfully completed.

SQL>

Instead of flashing back, I could try to sort out the issue. For instance, if I'm attaching the unique execution ID to every row in the fact table, either directly, or through an AUDIT dimension table, then I could probably identify the rows for this run. Why would I do this when the Flashback functionality is already available to me?

My test case above was a simple one; I was able to continue just by flashing back a single table before restarting the process. However, in a large enterprise data warehouse, the effort involved in a typical aftermath could be staggering depending on how many fact tables are involved, how many dimension tables track history with SCD Type 2 changes, etc. Combine that with the possible need to flashback ODS tables, history tables, persistent staging tables, etc. I've seen numerous situations where the exact ramifications are impossible to quantify: we know what broke, but we have no idea what needs to be fixed. Perhaps there was a hardware failure in the middle of an ETL load, and it's hard to identify which tables were loaded and which ones weren't. In this case, what I really need is the ability to do a complete "do-over": put everything back the way it was and start from the beginning, and Flashback Database makes this possible. So I'll demonstrate what's required to enable this feature, and then I'll replay the test case above and solve it from this angle.

I first need to put my database in Archive Log Mode, as archived redo is a required component of the feature:

```
SQL> startup mount
ORACLE instance started.

Total System Global Area  422670336 bytes
Fixed Size                 1336960 bytes
Variable Size              335546752 bytes
Database Buffers          79691776 bytes
Redo Buffers               6094848 bytes
Database mounted.
SQL> alter database archivelog;

Database altered.

SQL> archive log list
Database log mode          Archive Mode
Automatic archival        Enabled
Archive destination       USE_DB_RECOVERY_FILE_DEST
Oldest online log sequence 25
Next log sequence to archive 27
Current log sequence      27
SQL>
```

Next, I need to configure the Flash Recovery Area, which is a file system on the server where the database will create the flashback logs:

```
SQL> alter system set db_recovery_file_dest_size=3G;

System altered.

SQL> Alter system set db_recovery_file_dest='/oracle/flash_recovery_area';

System altered.

SQL>
```

Finally, I need to set the *flashback\_retention\_target* parameter, which instructs the Flash Recovery Area on our retention requirements in minutes. After that, I just enable flashback and open the database:

```
SQL> alter system set db_flashback_retention_target=2880;
```

System altered.

```
SQL> alter database flashback on;
```

Database altered.

```
SQL> alter database open;
```

Database altered.

```
SQL>
```

So, Flashback Database should be ready to use. I'll take a quick look and see if the database thinks it's ready:

```
SQL> select oldest_flashback_scn,
           2     oldest_flashback_time,
           3     startup_time
           4   from v$flashback_database_log
           5     cross join v$instance;
```

OLDEST_FLASHBACK_SCN	OLDEST_FLASHBACK_TIME	STARTUP_TIME
2912097	02/10/2010 12:10:30 AM	02/10/2010 12:09:08 AM

1 row selected.

```
SQL>
```

Now I'll flashback the entire database to the very first restore point I created: dw\_load\_1001:

```
SQL> shutdown immediate
Database closed.
Database dismounted.
ORACLE instance shut down.
SQL> startup mount
ORACLE instance started.
```

```
Total System Global Area 422670336 bytes
Fixed Size                 1336960 bytes
Variable Size              343935360 bytes
Database Buffers          71303168 bytes
Redo Buffers               6094848 bytes
Database mounted.
SQL> flashback database to restore point dw_load_1001;
```

Flashback complete.

```
SQL> alter database open resetlogs;
```

Database altered.

```
SQL> SELECT count(*) FROM target.products;
```

```
  COUNT(*)
-----
         72
```

```
1 row selected.
```

```
SQL> SELECT count(*) FROM target.customers;
```

```
  COUNT(*)
-----
    55500
```

```
1 row selected.
```

```
SQL> SELECT count(*) FROM target.sales;
```

```
  COUNT(*)
-----
   918843
```

```
1 row selected.
```

```
SQL>
```

So the immediate downside of this approach is that it requires the involvement of the operations team because the database has to be in MOUNT mode, and the data warehouse is not available during this slight outage. However, when compared with the time it might take to sort out and correct a massive aftermath scenario, this seems the preferable choice. Is the data warehouse really “available” while the investigation and subsequent corrective measures are occurring? I would rather involve the operations team for a quick, concrete fix so the reload can begin immediately.

## **RESTORING**

Now I’ll drill into the backup and recovery aspect of data warehousing fault tolerance, and tackle the age-old question of whether to use Archive Log Mode or No Archive Log Mode in a BI/DW environment. When I engage with clients that have a data warehouse operating in No Archive Log Mode, their usual reasoning for this decision is a perceived performance gain. This makes sense on the surface because we would prevent the generation of all that unwanted and unneeded REDO, right?

Not exactly. There is misconception about what No Archive Log Mode means, and hopefully, I can clear that up with a demonstration. I have a database in No Archive Log Mode, and I’ll test to see whether my statements generate REDO:

```
SQL> SELECT log_mode
       2     FROM v$database;
```

```
LOG_MODE
-----
NOARCHIVELOG
```

```
1 row selected.
```

```
SQL> CREATE TABLE target.sales
  2     AS SELECT *
  3       FROM sh.sales
  4       WHERE 1=0;
```

Table created.

```
SQL> SET autotrace on statistics
SQL>
SQL> INSERT INTO target.sales
  2     SELECT *
  3       FROM sh.sales;
```

918843 rows created.

#### Statistics

```
-----
 1897 recursive calls
40779 db block gets
 7062 consistent gets
 1585 physical reads
38832896 redo size
  742 bytes sent via SQL*Net to client
  958 bytes received via SQL*Net from client
    4 SQL*Net roundtrips to/from client
    2 sorts (memory)
    0 sorts (disk)
918843 rows processed
```

```
SQL>
SQL> ROLLBACK;
```

Rollback complete.

```
SQL>
SQL> INSERT /*+ APPEND */ INTO target.sales
  2     SELECT *
  3       FROM sh.sales;
```

918843 rows created.

#### Statistics

```
-----
 1042 recursive calls
 5581 db block gets
 2874 consistent gets
 1052 physical reads
92108 redo size
  732 bytes sent via SQL*Net to client
  975 bytes received via SQL*Net from client
    4 SQL*Net roundtrips to/from client
    5 sorts (memory)
    0 sorts (disk)
918843 rows processed
```

```
SQL>
SQL> ROLLBACK;
```

Rollback complete.

SQL>

The regular insert statement generated 38M of REDO in No Archive Log Mode, while the direct-path statement generated only 92K. Though it would appear that neither of these statements executed in NOLOGGING mode, the truth is that the direct-path statement did. All statements generate a little bit of REDO, because updates to the data dictionary are always logged.

Choosing No Archive Log Mode doesn't mean that all REDO is suppressed, but instead that we are foregoing the option to use media recovery (restoring datafiles, rolling forward). Think about it: REDO is not simply for media recovery, it's also for crash recovery. If all REDO generation was suspended, Oracle wouldn't be able to open after a simple server crash. In No Archive Log Mode, there are specific operations where the database can suspend most of the REDO because logging those operations wouldn't help us with crash recovery. Direct-path operations write blocks directly into datafiles, bypassing the buffer cache, so if the database went down unexpectedly and needed to perform crash recovery, the blocks from direct-path operations would already exist in the datafiles. We wouldn't have to rely on the online REDO logs to recover those transactions.

So if your database is in No Archive Log Mode for performance reasons, but you are using ETL tools that don't support true direct-path writes on Oracle, or you are using cursor-based, row-by-row load scenarios, the same amount of REDO is generated regardless of which Archive Log Mode is selected. The only thing gained from choosing No Archive Log Mode is the privilege of having to shut down the database every time a backup is needed.

The Oracle Database provides us the best of all possible worlds: we can minimize the amount of REDO generated, we can operate in Archive Log Mode, we can backup our database in online mode, and we can restore from that backup if we need to. The solution: NOLOGGING tables and indexes. I'll put the database in Archive Log Mode and rerun the test case above with a slight caveat: I'll change the table to NOLOGGING:

```
SQL> startup mount
ORACLE instance started.
```

```
Total System Global Area  422670336 bytes
Fixed Size                  1336960 bytes
Variable Size               343935360 bytes
Database Buffers           71303168 bytes
Redo Buffers                 6094848 bytes
Database mounted.
```

```
SQL> alter database
  2  archivelog;
```

Database altered.

```
SQL> alter database
  2  open;
```

Database altered.

```
SQL> SELECT log_mode
  2     FROM v$database;
```

```
LOG_MODE
-----
ARCHIVELOG
```

1 row selected.

```
SQL> ALTER TABLE target.sales
      2      nologging;
```

Table altered.

```
SQL> SET autotrace on statistics
SQL> INSERT INTO target.sales
      2      SELECT *
      3      FROM sh.sales;
```

918843 rows created.

#### Statistics

```
-----
15560 recursive calls
33573 db block gets
13861 consistent gets
6260 physical reads
38289752 redo size
740 bytes sent via SQL*Net to client
958 bytes received via SQL*Net from client
4 SQL*Net roundtrips to/from client
154 sorts (memory)
0 sorts (disk)
918843 rows processed
```

```
SQL> ROLLBACK;
```

Rollback complete.

```
SQL>
SQL> INSERT /*+ APPEND */ INTO target.sales
      2      SELECT *
      3      FROM sh.sales;
```

918843 rows created.

#### Statistics

```
-----
1 recursive calls
4628 db block gets
1718 consistent gets
59 physical reads
8072 redo size
732 bytes sent via SQL*Net to client
975 bytes received via SQL*Net from client
4 SQL*Net roundtrips to/from client
1 sorts (memory)
0 sorts (disk)
918843 rows processed
```

```
SQL> ROLLBACK;
```

Rollback complete.

```
SQL>
```

We get the same behavior with a NOLOGGING table in Archive Log Mode than we did with No Archive Log Mode. However, this transaction is lost to us if we ever needed to perform media recovery. To correct this, we need one small change to our backup strategy: a well-placed incremental backup. To increase the performance of our incremental backup, we need to create a block change tracking file. The database keeps a list of all changed blocks so that RMAN will know exactly what to backup during an incremental:

```
SQL> alter database enable block change tracking
  2 using file '/oracle/oradata/bidw1/change_blocks.bct';
```

Database altered.

```
SQL> select * from
  2 v$block_change_tracking;
```

STATUS	FILENAME	BYTES
ENABLED	/oracle/oradata/bidw1/change_blocks.bct	11599872

1 row selected.

```
SQL>
```

We start by taking the initial incremental level 0 backup. Some of the output from the RMAN command has been truncated for the sake of brevity:

```
RMAN> backup incremental
  2> level 0 database
  3> plus archivelog;
```

```
Starting backup at 11-FEB-10
current log archived
using target database control file instead of recovery catalog
allocated channel: ORA_DISK_1
channel ORA_DISK_1: SID=45 device type=DISK
channel ORA_DISK_1: starting archived log backup set
[Section Omitted]
channel ORA_DISK_1: finished piece 1 at 11-FEB-10
piece handle=/oracle/flash_recovery_area/BIDW1/backupset/2010_02_11/
o1_mf_annnn_TAG20100211T020143_5q7btr8x_.bkp tag=TAG20100211T020143 comment=NONE
channel ORA_DISK_1: backup set complete, elapsed time: 00:00:03
Finished backup at 11-FEB-10
```

```
RMAN>
```

Now I'll load the SALES table with another INSERT /\*+ APPEND \*/ to make sure we have a NOLOGGING operation since our last backup.

```
SQL> insert /*+ APPEND */
  2 into target.sales
  3 select * from
  4 sh.sales;
```

918843 rows created.

#### Statistics

```
-----
 2780 recursive calls
 6081 db block gets
 2434 consistent gets
 5442 physical reads
136036 redo size
 1536 bytes sent via SQL*Net to client
 1155 bytes received via SQL*Net from client
    6 SQL*Net roundtrips to/from client
   10 sorts (memory)
    0 sorts (disk)
918843 rows processed
```

```
SQL> commit;
```

Commit complete.

```
SQL>
```

This is the step in our process that requires a slight change to our backup and recovery strategy: we should get an incremental level 1 backup as soon as the load is complete. This will physically backup all blocks that have been affected by the load, and we wouldn't need to logically apply the REDO logs that are missing the NOLOGGING operations. Since we have block change tracking enabled, this step will be extremely fast, and I recommend that the ETL process flow or main driving script execute the backup as the very last step in the batch load.

```
RMAN> backup incremental
2> level 1 database
3> plus archivelog;
```

```
Starting backup at 11-FEB-10
current log archived
using target database control file instead of recovery catalog
allocated channel: ORA_DISK_1
channel ORA_DISK_1: SID=30 device type=DISK
[Section Omitted]
channel ORA_DISK_1: starting piece 1 at 11-FEB-10
channel ORA_DISK_1: finished piece 1 at 11-FEB-10
piece handle=/oracle/flash_recovery_area/BIDW1/backupset/2010_02_11/
o1_mf_annnn_TAG20100211T022515_5q7d6vg7_.bkp tag=TAG20100211T022515 comment=NONE
channel ORA_DISK_1: backup set complete, elapsed time: 00:00:01
Finished backup at 11-FEB-10
```

```
RMAN>
```

Now, let's see if we can restore:

```

RMAN> startup mount
Oracle instance started
database mounted

```

```

Total System Global Area      422670336 bytes

Fixed Size                    1336960 bytes
Variable Size                 356518272 bytes
Database Buffers             58720256 bytes
Redo Buffers                  6094848 bytes

```

```
RMAN> restore database;
```

```

Starting restore at 11-FEB-10
allocated channel: ORA_DISK_1
channel ORA_DISK_1: SID=18 device type=DISK

```

```

channel ORA_DISK_1: starting datafile backup set restore
channel ORA_DISK_1: specifying datafile(s) to restore from backup set
channel ORA_DISK_1: restoring datafile 00001 to /oracle/oradata/bidw1/system01.dbf
channel ORA_DISK_1: restoring datafile 00002 to /oracle/oradata/bidw1/sysaux01.dbf
channel ORA_DISK_1: restoring datafile 00003 to /oracle/oradata/bidw1/undotbs01.dbf
channel ORA_DISK_1: restoring datafile 00004 to /oracle/oradata/bidw1/users01.dbf
channel ORA_DISK_1: restoring datafile 00005 to /oracle/oradata/bidw1/example01.dbf
channel ORA_DISK_1: restoring datafile 00006 to /oracle/oradata/bidw1/tdrep01.dbf
channel ORA_DISK_1: restoring datafile 00007 to /oracle/oradata/bidw1/target01.dbf
channel ORA_DISK_1: reading from backup piece /oracle/flash_recovery_area/BIDW1/backupset/
2010_02_11/o1_mf_nnnd0_TAG20100211T015557_5q7bhz1o_.bkp
channel ORA_DISK_1: piece handle=/oracle/flash_recovery_area/BIDW1/backupset/2010_02_11/
o1_mf_nnnd0_TAG20100211T015557_5q7bhz1o_.bkp tag=TAG20100211T015557
channel ORA_DISK_1: restored backup piece 1
channel ORA_DISK_1: restore complete, elapsed time: 00:06:37
Finished restore at 11-FEB-10

```

```
RMAN> recover database;
```

```

Starting recover at 11-FEB-10
using channel ORA_DISK_1
channel ORA_DISK_1: starting incremental datafile backup set restore
channel ORA_DISK_1: specifying datafile(s) to restore from backup set
destination for restore of datafile 00001: /oracle/oradata/bidw1/system01.dbf
destination for restore of datafile 00002: /oracle/oradata/bidw1/sysaux01.dbf
destination for restore of datafile 00003: /oracle/oradata/bidw1/undotbs01.dbf
destination for restore of datafile 00004: /oracle/oradata/bidw1/users01.dbf
destination for restore of datafile 00005: /oracle/oradata/bidw1/example01.dbf
destination for restore of datafile 00006: /oracle/oradata/bidw1/tdrep01.dbf
destination for restore of datafile 00007: /oracle/oradata/bidw1/target01.dbf
channel ORA_DISK_1: reading from backup piece /oracle/flash_recovery_area/BIDW1/backupset/
2010_02_11/o1_mf_nnnd1_TAG20100211T022457_5q7d6cgv_.bkp
channel ORA_DISK_1: piece handle=/oracle/flash_recovery_area/BIDW1/backupset/2010_02_11/
o1_mf_nnnd1_TAG20100211T022457_5q7d6cgv_.bkp tag=TAG20100211T022457
channel ORA_DISK_1: restored backup piece 1
channel ORA_DISK_1: restore complete, elapsed time: 00:00:15

```

```

starting media recovery
media recovery complete, elapsed time: 00:00:03

```

```
Finished recover at 11-FEB-10
```

```
RMAN> alter database open;
```

```
database opened
```

```
RMAN>
```

We see that No Archive Log Mode really achieves nothing; it doesn't provide exclusive performance gains and really limits what we can do and how we can respond to faults in our environment. Archive Log Mode is the only way to treat a production database, regardless of whether that database is OLTP or BI/DW in nature

## **CONCLUSION**

As we consider what it takes to build a triage process that moves us ever closer to a fault-free data warehouse, the features already existing in the Oracle Database can take us at least as far as fault-tolerance. Relying on these features makes practical and economical sense, and we should keep that in mind before developing comparable custom processes in our ETL, or adopting backup and recovery strategies that preclude these features. Our focus should favor built-in database features: they only have to be implemented, not built, and in most circumstances are turnkey.

## **ABOUT THE AUTHOR**

Stewart Bryson is Managing Director of Rittman Mead America, and has 14 years of experience designing, building and supporting complex database systems and data warehouses. Stewart has an expert-level understanding of the Oracle Database and BI stacks, and is capable of leading a project from initial scope to final delivery. Based in Atlanta, GA, Stewart has delivered projects of varying sizes across multiple verticals, including logistics, supply chain, hospitality, municipal government and utilities. A regular writer and blogger on Oracle data warehousing, including contributing to the Rittman Mead blog at [www.rittmanmead.com/blog](http://www.rittmanmead.com/blog), Stewart has presented at user groups in the US and UK, including UKOUG, ODTUG Kaleidoscope and the upcoming RMOUG.