Session ID:
1621

Logging or NoLogging on 12c

That is the question!

Prepared by:
Francisco Munoz Alvarez
Director of Innovation
Data Intensity
@fcomunoz

04/16/2018

Remember to complete your evaluation for this session within the app!

DATA INTENSITY

#C18LV

Francisco Munoz Alvarez

- ✓ Oracle ACE Director
- ✓ 8/9/10g/11g/12c OCP, RAC OCE, AS OCA, E-Business OCP,
- ✓ SQL/PLSQL OCA, Oracle 7 OCM
- ✓ Oracle 7, 11GR2, 12cR1 and OVM 3.1 and 3.2 and 3.3 Beta Tester
- ✓ IOUC LA & APAC Spokesperson, President of APACOUC, AIOUG,CLOUG and NZOUG
- ✓ ITIL Certified
- ✓ Oracle Excellence Award Winner
- ✓ 2010 Oracle ACE Director of the year by Oracle Magazine

Blog: oraclenz.com

Email: fmunozalvarez@dataintensity.com

Twitter : fcomunoz

Director of Innovation

www.dataintensity.com

DATA INTENSITY

Francisco Munoz Alvarez

ORACLE ACE DIRECTOR OF THE YEAR

Oracle ACE director hones skills while helping others.

When it comes to learning something new, Francisco Munoz Alvarez doesn't look for a teacher; he looks for someone who has a problem.

"One of the best ways to learn is by helping others," says Alvarez. "When someone has a problem on an OTN [Oracle Technology Network] forum, I enjoy trying to assist them with it. I've learned a lot by trying to help other Oracle users to solve their problems."

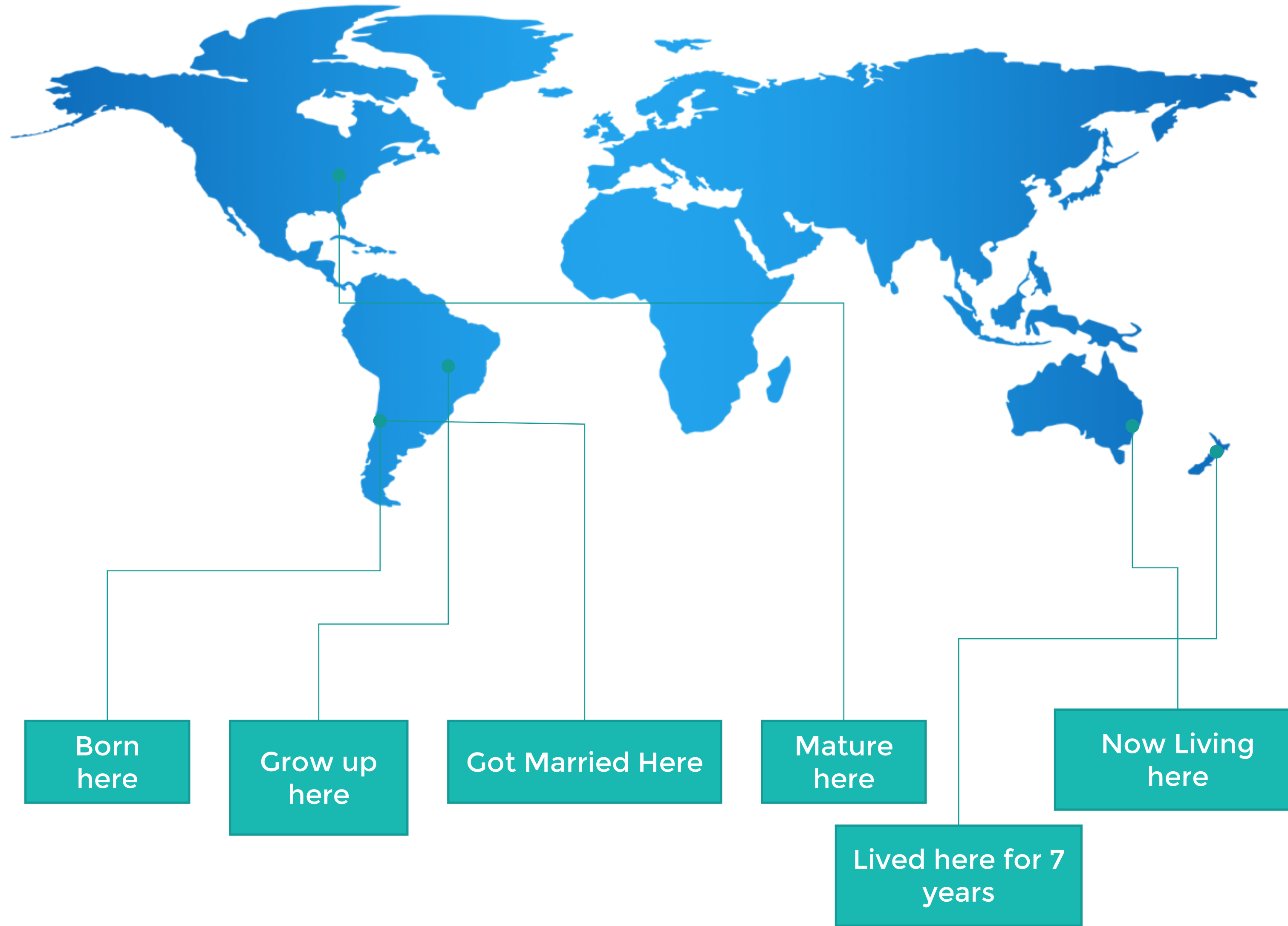
Over the years, he has learned enough to become an Oracle ACE, an Oracle ACE director, and now *Oracle Magazine*'s Oracle ACE Director of the Year. Alvarez is also the founder and CEO of Database Integrated Solutions.

For Alvarez, learning—and helping others—takes legwork. This year, as an Oracle ACE director, he has appeared at 22 conferences in 18 countries, where he not only gives talks but takes the time to answer questions and assist users with complex Oracle challenges. In addition, he frequently helps his blog visitors (about 25,000 per month) solve difficult Oracle-related problems.

Given that he's also president of the New Zealand and Chilean Oracle user groups and the Latin American Oracle User Group, Alvarez' energy seems to have no limits when it comes to the Oracle community. He himself puts it best: "I never get tired of helping people learn how to share their knowledge and experience."

WINNER STATS

Name: Francisco Munoz Alvarez
Job title: Founder and CEO
Company: Database Integrated Solutions
Location: Auckland, New Zealand
Award: Oracle ACE Director of the Year 2010



DATA INTENSITY

DATA INTENSITY

Disclaimer

This views/content in this document are those of the author(s)/presenter(s) and do not necessarily reflect that of Oracle Corporation and/or its affiliates/subsidiaries. The material in this document is for informational purposes only and is published with no guarantee or warranty, express or implied.

“Redo generation is a vital part of the Oracle recovery mechanism. Without it, an instance will not recover when it crashes and will not start in a consistent state. By other side, excessive redo generation is the result of excessive work on the database.“

DATA INTENSITY

NOLOGGING operations do not generate redo records in the redo log files (only a notification that a NOLOGGING operation was made is registered, but not the changes). Consequently, such operations are a very helpful option to reduce the amount of redo to be generated in a transaction, which might make the transaction to run faster, and also reducing any unnecessary stress to the database.

You need to understand that the NOLOGGING operations are direct path—they bypass the buffer cache. If you direct path load say 100 MB of data and that all fits in the buffer cache—a conventional path load might be much faster than a non-logged direct path load (you don't want for the blocks to be written to disk and the redo is streamed to disk in the background by the LGWR (log writer) process.

On the other hand if you are loading gigabytes of data, more than what can be buffered in the cache, then you might benefit from direct path writes since you'd be waiting for the DBWR (database writer) process to empty the cache.

You always need to remember that redo generation is a crucial part of the Oracle recovery mechanism. NOLOGGING operations only affect the recovery from media failure perspective (due that you then will need to recover from a backup and apply all the available archive logs in the recover process), but will not affect a database in case of an instance failure. On the other hand, excessive generation of redo is the result of an excessive workload of update, insert, and delete (DML) operations in the database.

TIP

"A very important rule with respect to data is to never put yourself into an unrecoverable situation. The importance of this guideline cannot be stressed enough, but it does not mean that you can never use time saving or performance enhancing options."

What is Redo? (Long Answer)

When Oracle blocks are changed, including undo blocks, oracle records the changes in a form of vector changes which are referred to as redo entries or redo records. The changes are written by the server process to the redo log buffer in the SGA. The redo log buffer is then flushed into the online redo logs in near real time fashion by the log writer (LGWR).

Short Answer?

In other words:

Redo = Transactions

When Redo is flushed?

The redo are flushed from Log Buffer by the LGWR:

- ✓ When a user issue a commit.
- ✓ When the Log Buffer is 1/3 full.
- ✓ When the amount of redo entries is 1MB.
- ✓ Every three seconds
- ✓ When a database checkpoint takes place.

The redo entries are written before the checkpoint to ensure recoverability.

Redo Generation and Recoverability

“The main purpose of redo generation is to ensure recoverability. “

This is the reason why, Oracle does not give the DBA or the Developer a lot of control over redo generation. If the instance crashes, then all the changes within SGA will be lost. Oracle will then use the redo entries in the online redo files to bring the database to a consistent state.

LOG STATUS (V$LOG)

Log status:

- **UNUSED** – Online redo log has never been written to. This is the state of a redo log that was just added, or just after a RESETLOGS, when it is not the current redo log.
- **CURRENT** – Current redo log. This implies that the redo log is active. The redo log could be open or closed.
- **ACTIVE** – Log is active but is not the current log. It is needed for crash recovery. It may be in use for block recovery. It may or may not be archived.
- **CLEARING** – Log is being re-created as an empty log after an ALTER DATABASE CLEAR LOGFILE statement. After the log is cleared, the status changes to UNUSED.
- **CLEARING_CURRENT** – Current log is being cleared of a closed thread. The log can stay in this status if there is some failure in the switch such as an I/O error writing the new log header.
- **INACTIVE** – Log is no longer needed for instance recovery. It may be in use for media recovery. It might or might not be archived.

Recovering from loss of REDO is completely dependent on the STATUS of the members that are corrupted or lost.

LOGGING vs NOLOGGING

Redo Generation and Recoverability

Despite the importance of the redo entries, Oracle gives users the ability to limit redo generation on tables, partitions, tablespaces, and indexes by setting them in the NOLOGGING mode. NOLOGGING affects the recoverability of a database and before going into how to limit the redo generation, it is important to clear the misunderstanding that NOLOGGING is the way out of redo generation

Some Interesting points…

- NOLOGGING is designed to handle bulk inserts of data which can be easily reproduced. (Remember that the `UPDATE` and `DELETE` operations will always be logged.)
- Regardless of LOGGING status, writing to the UNDO blocks will always cause generation of redo.
- LOGGING should not be disabled on a primary database if it has one or more standby databases. For this reason, Oracle introduced the `ALTER DATABASE FORCE LOGGING` command to place the database in the `FORCE LOGGING` mode—meaning that the NOLOGGING attribute will not have any effect on the segments. The `FORCE LOGGING` mode can also be used at the tablespace level using the `ALTER TABLESPACE <Tablespace_Name> FORCE LOGGING` command. Use of this option results in some performance degradation, but ensures the recoverability of your primary database and the integrity of your standby database.

DATA INTENSITY

NOTE

"Using FORCE LOGGING in the initialization parameter file in a Multitenant Container Database will always place all the pluggable databases using that CDB in the FORCE LOGGING mode."

Some Interesting points… (2)

- Any change to the database dictionary will always cause redo generation. This will happen to protect the data dictionary integrity. As per example, if Oracle allocates a space above the high water mark (HWM) for a table, and the system fails in the middle of an `INSERT /*+ APPEND */` command, then Oracle will need to rollback that data dictionary change that was made. There will be a redo generated, but it is only to protect the data dictionary, not your newly inserted data (Oracle will undo the space allocation if it fails, and your newly inserted data will also disappear).
- Objects should be set back to the LOGGING mode when the NOLOGGING mode is no longer required.

Some Interesting points… (3)

- NOLOGGING is unnecessary for direct path inserts if the database is in the NOARCHIVELOG mode. (see the following table)
- Operations involving data that could not be easily reproduced should always use LOGGING operations; avoid NOLOGGING in such cases! If data is loaded using NOLOGGING, the data will not be able to be recovered when the database crashes if no backup is made after the load.
- NOLOGGING does not apply to normal `UPDATE`, `DELETE`, and `INSERT` operations.

Some Interesting points… (4)

- NOLOGGING will work during specific situations only, but subsequent DML operations over the data will always generate redo (we will see a list of the specific commands that will work in the NOLOGGING mode a little bit later in this chapter).
- If the LOGGING or NOLOGGING clause is not specified when creating a table, partition, or index, the default to the LOGGING attribute will be the LOGGING attribute of the database, or if not set, the tablespace in which it resides

Some Interesting points… (5)

Table Mode	Insert Mode	Archive Log Mode	Result
LOGGING	APPEND	ARCHIVELOG	REDO GENERATED
NOLOGGING	APPEND	ARCHIVELOG	NO REDO
LOGGING	NO APPEND	ARCHIVELOG	REDO GENERATED
NOLOGGING	NO APPEND	ARCHIVELOG	REDO GENERATED
LOGGING	APPEND	NOARCHIVELOG	NO REDO
NOLOGGING	APPEND	NOARCHIVELOG	NO REDO
LOGGING	NOAPPEND	NOARCHIVELOG	REDO GENERATED
NOLOGGING	NOAPPEND	NOARCHIVELOG	REDO GENERATED

NOTE

“When doing the insert mode APPEND, it isn't append really, it is the fact that you are doing a direct path operation that will bypass undo (hence reducing redo) and may bypass redo when in NOLOGGING.“

FREQUENT QUESTIONS

Some Frequent Questions

- Does creating a table with the NOLOGGING option mean there is no generation of redo ever, or just that the initial creation operation has no redo generation, but does that DML down the road generates redo?
- How and when can the NOLOGGING option be employed?

Some Frequent Questions

- Why I have excessive Redo Generation during an Online Backup?
- Why Oracle generates redo and undo for DML?
- Does temporary tables generate Redo?
- Can Redo Generation be Disabled During Materialized View Refresh?
- Why my table on NOLOGGING still Generating Redo?
- Can Table Redefinition be used with NOLOGGING?

THE ANSWERS

DATA INTENSITY

Why I have excessive Redo Generation during an Online Backup?

When a tablespace is put in backup mode the redo generation behaviour changes but there is not excessive redo being generated, there is additional information logged into the online redo log during a hot backup the first time a block is modified in a tablespace that is in hot backup mode.

The datafile headers which contain the SCN of the last completed checkpoint are NOT updated while a file is in hot backup mode. DBWR constantly write to the datafiles during the hot backup. The SCN recorded in the header tells us how far back in the redo stream one needs to go to recover that file.

Why Oracle generates redo and undo for DML?

When you issue an insert, update or delete, Oracle actually makes the change to the data blocks that contain the affected data even though you have not issued a commit. To ensure database integrity, Oracle must write information necessary to reverse the change (UNDO) into the log to handle transaction failure or rollback. Recovery from media failure is ensured by writing information necessary to re-play database changes (REDO) into the log. So, UNDO and REDO information logically MUST be written into the transaction log of the RDBMS

Does temporary tables generate Redo?

Before 12c

The amount of log generation for temporary tables should be approximately 50% of the log generation for permanent tables.

However, you must consider that an INSERT requires only a small amount of "undo" data, whereas a DELETE requires a small amount of "redo" data.

If you tend to insert data into temporary tables and if you don't delete the data when you're done, the relative log generation rate may be much lower for temporary tables that 50% of the log generation rate for permanent tables.

After 12c

In 12c, temporary tables record their undo into temp removing all redo!

Note: Temp files are always set to the NOLOGGING mode, but any non-direct path operation on them such as INSERT/UPDATE/DELETE will generate redo since they do generate undo.

Can Redo Generation Be Disabled During Materialized View Refresh?

Setting the NOLOGGING option during the materialized view creation does not affect this fact, as the option only applies during the actual creation and not to any subsequent actions on the materialized view.

Enhancement requests have been raised to be able to turn off redo generation during a refresh, but these were rejected as doing this could put the database into an inconsistent state and affect options such as Data Guard as well as backup and recovery.

The amount of redo generated during a complete refresh can be reduced by setting `ATOMIC_REFRESH=FALSE` in the `DBMS_MVIEW.REFRESH` option. The complete refresh will use a `TRUNCATE+INSERT /*+APPEND*/` command to refresh, and this can skip all undo and redo, all of it.

Why my table on NoLogging mode still Generating Redo?

The NOLOGGING attribute tells the Oracle that the operation being performed does not need to be recoverable in the event of a failure.

In this case Oracle will generate a minimal number of redo log entries in order to protect the data dictionary, and the operation will probably run faster.

Oracle is relying on the user to recover the data manually in the event of a media failure.

NOTE

"It is important to note that just because an index or a table was created with NOLOGGING does not mean that redo generation has been stopped for this table or index. NOLOGGING is active in some situations and while running one of the valid commands but not after that. "

Can Table Redefinition be done using NOLOGGING?

- Table redefinition cannot be done in NOLOGGING, in other words, it will needs to be in the LOGGING mode and will always generate redo.

Direct Path Operations

...

- `ALTER TABLE … MERGE PARTITION`
- `ALTER TABLE … MODIFY PARTITION`
  - *ADD SUBPARTITON*
  - *COALESCE SUBPARTITON*
  - *REBUILD UNUSABLE INDEXES*
- `ALTER INDEX ... SPLIT PARTITION`
- `ALTER INDEX ... REBUILD`
- `ALTER INDEX ... REBUILD PARTITION`

NOLOGGING OPERATIONS

NOLOGGING Operations

The NOLOGGING attribute tells Oracle that the operation being performed does not need to be recoverable in the event of a failure. In this case the database will generate only a small set of metadata that is written to the redo log, and the operation will probably run faster. Oracle is relying on the user to recover the data manually in the event of any failure. In other words, the NOLOGGING option skips the generation of redo for the affected object, but will still log many things such as data dictionary changes caused by space management.

NOTE

“The options UNRECOVERABLE (introduced in Oracle 7) and NOLOGGING (introduced in Oracle 8) can be used to avoid the redo log entries to be generated for certain operations that can be easily recovered without using the database recovery mechanism, but remember that the UNRECOVERABLE option is deprecated and is replaced by the NOLOGGING option.“

Direct Path Operations

This is a partial list:

- `DIRECT LOAD` (SQL*Loader)
- `DIRECT LOAD  INSERT` (using APPEND hint)
- `CREATE TABLE ... AS SELECT`
- `CREATE INDEX`
- `ALTER TABLE MOVE`
- `ALTER TABLE ... MOVE PARTITION`
- `ALTER TABLE ... SPLIT PARTITION`
- `ALTER TABLE … ADD PARTITION` (if HASH partition)

NOLOGGING

Logging is stopped only while one of the commands in the previous slides is running, so if a user runs this:

```
SQL> ALTER INDEX new_index NOLOGGING.
SQL> ALTER INDEX new_index REBUILD;
```

The actual rebuild of the index does not generate redo (only all data dictionary changes associated with the rebuild will do). Afterwards though, any DML operation on the index will generate redo, this includes a direct load insert on the table to which the index belongs.

NOLOGGING

Here is another example to make this point more clear:

```
SQL> CREATE TABLE table_nolog_test (a number) NOLOGGING;
```

All the following statements will generate redo despite the fact the table is in NOLOGGING mode:

```
SQL> INSERT INTO table_nolog_test values (1);
SQL> UPDATE table_nolog_test SET a = 2 WHERE a = 1;
SQL> DELETE FROM table_nolog_test WHERE a = 2;
```

The following will not generate redo (except from dictionary changes and indexes):

- `INSERT /*+APPEND+/ ...`
- `ALTER TABLE table_nolog_test MOVE ...`
- `ALTER TABLE table_nolog_test MOVE PARTITION ...`

SCENARIO

NOARCHIVELOG MODE

DATA INTENSITY

```
SQL> CREATE TABLE test1 AS SELECT *
 2   FROM dba_objects
 3   WHERE rownum=0;

Table created.

SQL> SET AUTOTRACE ON STATISTICS
```

Note: The SET AUTOTRACE ON STATISTICS statement will show only the SQL statement execution statistics after the execution of one SQL DML statement (SELECT, DELETE, UPDATE, or INSERT).

```
SQL> INSERT INTO test1 SELECT * FROM dba_objects;

88012 rows created.

Statistics
----------------------------------------------------------
   11560972  redo size
      88012  rows processed
```

```
SQL> INSERT /*+ APPEND */ INTO test1 SELECT *
  2    FROM dba_objects;

88012 rows created.

Statistics
----------------------------------------------------------
      36772  redo size
      88012  rows processed
```

You can see in the example that the amount of redo generated via the simple insert was 11 MB while a direct insert generates only 36 KB.

TIP

"You never need to set NOLOGGING in the NOARCHIVELOG mode—everything that can skip redo will skip redo already. NOLOGGING doesn't apply in the NOARCHIVELOG mode—it doesn't change any behavior."

To activate the NOLOGGING mode when using an ALTER command, you will need to add the NOLOGGING clause after the end of the ALTER command. For example:

```
SQL> ALTER TABLE test1
  2  MOVE PARTITION parti_001 TABLESPACE new_ts_001 NOLOGGING;
```

The same applies for a `CREATE INDEX` command and the `CREATE TABLE` command. An exception is that if your `CREATE TABLE` command has the clause `AS SELECT`, and you use NOLOGGING at the end of the command, then the operation will not use the NOLOGGING mode, and instead will generate an alias called NOLOGGING.

Note: It is a common mistake to add the NOLOGGING option at the end of the SQL when using the `AS SELECT` statement (if done, Oracle will consider it as an alias and the table will generate normal LOGGING).

```
SQL> CREATE TABLE table_nolog_test2 NOLOGGING AS SELECT *
  2   FROM dba_objects;

Table created.

SQL> CREATE TABLE table_nolog_test3
  2   AS SELECT *
  3   FROM dba_objects NOLOGGING;

Table created.

SQL> SELECT table_name, logging
  2   FROM user_tables;

TABLE_NAME                 LOGGING
------------------         -------
TABLE_NOLOG_TEST           NO
TABLE_NOLOG_TEST2          NO
TABLE_NOLOG_TEST3          YES

3 rows selected.

SQL>
```

Tips when LOGGING is in effect (not using NOLOGGING)

While Backing Up

The best way to eliminate this problem is to use RMAN. RMAN does not need to write entire blocks to redo, because it knows when a block is being copied. If you have the need to use the user-managed backup technique, then you can follow these steps to reduce redo generation:

- Do not back up all the tablespaces at once (using the `ALTER DATABASE BEGIN BACKUP` command). Doing so will put every tablespace into the BACKUP mode for longer than it really needs to be, and therefore generating redo for longer. Instead back up one tablespace at the time using the `ALTER TABLESPACE <Tablespace_name> BEGIN/END BACKUP` command.
- Generate automatic backups on the busy tablespaces during a time when they are least busy in terms of DML.

Bulk Inserts

I use the term bulk inserts in this section to mean loading a large percentage compared to the existing data. To reduce or eliminate the amount of redo generated in a bulk data load, you need first to disable the indexes (when making a direct load to a table that have indexes, the indexes will also produce redo) before the load, and then rebuild them again as follows:

```
SQL> ALTER INDEX index_name UNUSABLE ; # Do this for every index
SQL> INSERT /*+ APPEND */ INTO table_name SELECT …
SQL> ALTER INDEX index_name REBUILD;
```

Note: Please ensure that the initialization parameter `skip_unusable_indexes` is set to TRUE before making an index unusable. If set to FALSE and a user tries to access data from a table with an index unusable, it will return an error to the user session. Also it is important to know, that prior to 10g, the `skip_unusable_indexes` was needed to be set at session level.

Bulk Delete

1. Create a new table with the same structure as the table you want to bulk delete from, with only the rows you want to keep, as in the following example:

```
SQL> CREATE TABLE new_table
  2    AS SELECT *
  3    FROM test1
  4    WHERE … ;
```

2. Create indexes on the new table.
3. Create constraints, grants, and so on.
4. Drop the original table.
5. Rename the new table to the original name.

Bulk Delete (2)

If the data remaining after step 2 is small, or if there are a lot of dependencies on the table in the form of views, procedures, functions, and so on, then following steps can be used after step 1 to move forward:

1. Truncate the original table, thus deleting all its data.
2. Disable all constraints on the original table.
3. Insert back to the original table all data in the new table. For example:

```
SQL> INSERT /*+ APPEND */ INTO test1
 2   SELECT *
 3   FROM new_table;
```

4. Commit your changes.
5. Enable the constraints on the original table.
6. Drop the new table that you created in step 1.

Bulk Update

Use the method in this section if indexes are going to be affected by a bulk update, because a massive update on indexes is more expensive than rebuilding them. If a small portion of the data is updated, then use this first approach:

1. Disable all constraints.
2. Make all indexes associated with the columns to be updated UNUSABLE. For example:
   ```
   SQL> ALTER INDEX index_name UNUSABLE;
   ```
3. Run the update on the table.
4. Commit the update.
5. Rebuild all indexes that you made unusable in step 2. For example:
   ```
   SQL> ALTER INDEX index_name REBUILD;
   ```
6. Enable all constraints you disabled in step 1.

Bulk Update (2)

If the update causes a change to all the data to be updated, then follow this second approach:

1. Create a new table to be used as a holding table and modify the amount in the column value at the same time:

```
SQL> CREATE TABLE new_table AS
 2   SELECT (value*1.10) value, ... FROM goods;
```

2. Create all the same indexes on the new table as exists on the original table. For example:

```
SQL> CREATE INDEX idx_test3 ON test3 (owner);
```

3. Create all grants, constraints, and so on, on the new table.
4. Drop the original table.
5. Finally rename the new table to become the original one.

SCENARIO 2

Do things the easy way

Tips For Developers

Run the DML in as few SQL statements as you can. This will reduce the generation of undo and block header update and therefore reduces redo generation.

```
SQL> CREATE TABLE test4
 2   AS SELECT owner, object_name, object_type
 3   FROM dba_objects;

Table Created.

SQL> set autotrace on statistics

SQL> INSERT INTO test4
  2  SELECT owner, object_name, object_type
  3  FROM dba_objects;

88019 rows created.

Statistics
------------------------------------------------------
    4660736  redo size
      88019  rows processed
```

Tips For Developers

Run the DML in as few SQL statements as you can. This will reduce the generation of undo and block header update and therefore reduces redo generation.

```
SQL> CREATE TABLE test4
 2   AS SELECT owner, object_name, object_type
 3   FROM dba_objects;

Table Created.

SQL> set autotrace on statistics

SQL> INSERT INTO test4
  2  SELECT owner, object_name, object_type
  3  FROM dba_objects;

88019 rows created.

Statistics
------------------------------------------------------
    4660736  redo size
      88019  rows processed
```

But a Developer will have a tendency to do this…

Tips For Developers

```
SQL> connect /

Connected.

DECLARE
  CURSOR cur_c1 is
     SELECT owner, object_name, object_type FROM dba_objects;
     rec_c1 cur_c1%ROWTYPE;
BEGIN
    OPEN cur_c1;
    FOR rec_c1 in cur_c1
    LOOP
        INSERT INTO test4 VALUES (rec_c1.owner, rec_c1.object_name,rec_c1.object_type);
    END LOOP;
    COMMIT;
    CLOSE cur_c1;
END;
/

PL/SQL procedure successfully completed.
```

Are they different?

Tips For Developers

Then let's check how much redo was generated during this session:

```
SQL> SELECT a.name, b.value
  2     FROM v$statname a, v$mystat b
  3    WHERE a.statistic# = b.statistic#
  4      AND a.name = 'redo size';

NAME                                  VALUE
------------------------------ ----------
redo size                          26776640
```

The amount of redo generated was **26,776,640** bytes. The PL/SQL approach generated over 26 million bytes of redo, whereas the `INSERT` approach generated a mere **4,660,736** bytes. Simpler is better. The simple `INSERT` statement generated far less redo than even that by simple PL/SQL block.

Tips For Developers

Also do not commit more than you need. By issuing the `COMMIT` command you are forcing Oracle to do some internal updates which produce redo. I ran the PL/SQL code from tip 1 with a `COMMIT` command inserted after the `INSERT` command (making a `COMMIT` command after each `INSERT` command, instead that once in the end of the `LOOP`), and the result was even more awful than before. The amount of redo generated increased to **51,917,676** bytes. You can see that excessive committing generates far more redo. By reducing unnecessary commits you will reduce the strain on the log writer (LGWR) .

Want more tips?

Tips For Developers

Set sequences to cache correctly. This is important since your system generates a lot of sequence numbers using the Oracle sequences. The database keeps track of the next sequence number in the SGA, but it also keeps the starting value of the next set of sequence numbers in the data dictionary according to the sequence cache setting. This starting value is needed in case the database crashes. As a sequence nextval is acquired, the value in the SGA is updated. When the value in the SGA is the same as the one in the data dictionary, the data dictionary is updated producing redo. If the sequence cache is small, the data dictionary will be updated more often.

Tips For Developers

`SQL> CREATE SEQUENCE seq2 CACHE 2;`	The data dictionary is updated every second `nextval`.
`SQL> CREATE SEQUENCE seq20 CACHE 20;`	The data dictionary is updated after every twenty `nextval` instances.
`SQL> CREATE SEQUENCE seq1000 CACHE 1000;`	The data dictionary is updated after every thousand `nextval` instances.

Lets' create three identical tables, test_seq_2, test_seq_20, and test_seq_1000. They all have a number column. Then we will insert rows into test_seq_2 using seq2, into test_seq_20 using seq20, and into test_seq_1000 using seq1000.

Tips For Developers

```
SQL> create table test_seq_2 (a number);
SQL> create table test_seq_20 (a number);
SQL> create table test_seq_1000 (a number);
SQL> INSERT INTO test_seq_2 SELECT seq2.nextval FROM dba_objects
;

88025 rows created.

SQL> INSERT INTO test_seq_20 SELECT seq20.nextval FROM
dba_objects ;

88025 rows created.

SQL> INSERT INTO test_seq_1000 SELECT seq1000.nextval FROM
dba_objects ;

88025 rows created.
```

How do you think each one behaved?

Tips For Developers

Cache	Redo (bytes)
2	32,077,760
20	4,494,368
1000	1,492,836

Set the cache to a higher value if the application accesses the sequence a lot. It is wrong to believe that setting cache to 1000 means that the SGA will have 1000 numbers stored for the sequence.

There is only one number stored for the sequence, so do not worry about setting the cache as high as you need.

“Managing the amount of redo generated by a database takes a partnership between DBA and developer. To the extent you can work with your developers to implement the discussed tips, to educate them about the value from implementing those tips, you can reduce redo and make your job easier and your database more efficient.”

Tips when NOLOGGING is in effect

DIRECT PATH INSERT

When using direct path inserts, the database will insert data without generation of redo or undo. Instead, Oracle logs a small number of block range invalidation redo records, and will periodically update the control file with information about the most recent direct writes.

Note: Direct path insert without LOGGING may improve performance, but once again makes the data inserted unrecoverable in case of a media failure.

DIRECT PATH INSERT

Since the release of Oracle Database 11.2.0.2, you can significantly improve the performance of a direct path insert with NOLOGGING by disabling the periodic update of the control files. You can do so by setting the initialization parameter `DB_UNRECOVERABLE_SCN_TRACKING` to FALSE. However the resulting benefit comes at a price. If you perform a direct path insert with NOLOGGING, and the control file is not updated, you will no longer be able to accurately determine whether any datafiles are currently unrecoverable.

To use direct path insert use the `/*+ APPEND */` hint as follows:

```
SQL> INSERT /*+ APPEND */ INTO test1
 2   SELECT *
 3   FROM dba_objects;
```

DIRECT PATH INSERT

When direct path insert is used, Oracle does the following in order to bypass using the buffer cache:

- Formats the data to be inserted as Oracle blocks.
- Inserts the blocks above the high water mark. When the commit takes place, the high water mark is moved to include the newly placed block.

It is clear that direct load is useful for bulk inserts. However using it to insert few hundred records at a time can have bad effect on space and performance.

DIRECT PATH INSERT

The statement `INSERT /*+APPEND*/ INTO <table_name> SELECT…FROM` will use the NOLOGGING option if available and will not produce redo. However, since 11.2 there is a new hint `APPEND_VALUES` designed to be used with bulk (array) inserts.

```
SQL> DROP TABLE t;

Table dropped.

SQL> CREATE TABLE t ( x char(2000) ) nologging;

Table created.

SQL> CONNECT /

Connected.
```

DIRECT PATH INSERT

```
SQL> DECLARE
  2           type array is table of char(2000) index by
binary_integer;
  3           l_data array;
  4   BEGIN
  5           for i in 1 .. 1000
  6           loop
  7                   l_data(i) := 'x';
  8           end loop;
  9           forall i in 1 .. l_data.count
 10                   INSERT INTO t (x) VALUES (l_data(i));
 11   END;
 12   /

PL/SQL procedure successfully completed.
```

DIRECT PATH INSERT

```
SQL> SELECT a.name, b.value
  2    FROM v$statname a, v$mystat b
  3   WHERE a.statistic# = b.statistic#
  4     AND a.name = 'redo size';

NAME                                          VALUE
------------------------------ ----------
redo size                                   2253664

SQL> connect /

Connected.
```

DIRECT PATH INSERT

```
SQL> DECLARE
  2           type array is table of char(2000) index by
binary_integer;
  3           l_data array;
  4   BEGIN
  5           for i in 1 .. 1000
  6           loop
  7                   l_data(i) := 'x';
  8           end loop;
  9           forall i in 1 .. l_data.count
 10                   INSERT /*+ APPEND_VALUES */ INTO t (x)
values (l_data(i));
 11   END;
 12   /

PL/SQL procedure successfully completed.
```

DIRECT PATH INSERT

```
SQL> SELECT a.name, b.value
  2    FROM v$statname a, v$mystat b
  3   WHERE a.statistic# = b.statistic#
  4     AND a.name = 'redo size';

NAME                                    VALUE
------------------------------          ----------
redo size                               7408
```

DATA INTENSITY

DIRECT PATH INSERT

It is very important to understand how direct path inserts affect redo generation. The operation of a direct path insert is affected by the following factors:

- The LOGGING mode (ARCHIVELOG/NOARCHIVELOG) of the database
- Using the `/*+ APPEND */` hint
- The LOGGING mode of the table
- The FORCE LOGGING mode of the database

It is very easy to verify if our `INSERT /*+APPEND*/` statement really did inserted the data after the HWM (high water mark). All you need to do is—without issuing a `COMMIT` command—run a normal `SELECT` command against the data just inserted. If the query works, the data was not appended. If the query returns an `ORA-12838` error, then the data was appended. The error comes about because you cannot read from a table in the same transaction as a direct load was made without issuing a `COMMIT` command.

Bulk Inserts

To further reduce redo generation when doing a bulk insert we will make use of direct patching. Direct path operations help to skip undo generation and maintain indexes in bulk—hence less redo.

1. Set the table in the NOLOGGING mode:
```
SQL> ALTER TABLE table_name NOLOGGING;
```
2. Make all table indexes unusable:
```
SQL> ALTER INDEX index_name UNUSABLE;
```
3. Insert the bulk data using the /*+ APPEND */ hint:
```
SQL> Insert /*+ APPEND */ into table_name select
```
4. Rebuild all indexes that were set as UNUSABLE using the NOLOGGING option:
```
SQL> ALTER INDEX index_name REBUILD NOLOGGING;
```
5. Set your table back to the LOGGING mode:
```
SQL> ALTER TABLE table_name LOGGING;
```
6. Set your indexes back to the LOGGING mode:
```
SQL> ALTER INDEX index_name LOGGING;
```
7. Backup the data.

NOTE

"The creation of an index with NOLOGGING will save space in the redo log files, and decrease the creation time (it will end faster when parallelizing large index creation). But do not forget to backup all affected datafiles and perhaps move them over to a standby database if necessary."

Bulk Delete

Use the following technique to reduce redo generation when doing a bulk delete:

1. Create a new table with all records you want to keep from the original table with the NOLOGGING option:

```
SQL> CREATE TABLE new_table NOLOGGING
  2   AS SELECT *
  3   FROM original_table WHERE ...
```

2. Create the indexes on the new table with the NOLOGGING option.
3. Create all constraints, grants, and so on as the original table.
4. Drop the original table.
5. Rename the new table as the original table.
6. Place the table and all indexes to the LOGGING mode.
7. Backup the data.

Bulk Delete

If the amount of data that will be left is very small compared with the original number of rows, or there are many dependencies on the table (views, procedures, functions, and so on), then the following steps can be used after step 2:

3. Disable constraints on the original table.
4. Truncate the original table.
5. Make indexes related to the original table UNUSABLE.
6. Place the original table in the NOLOGGING mode.
```
SQL> ALTER TABLE original_table NOLOGGING ;
```
7. Do a direct insert of the data in the new_table to the original_table :
```
SQL> INSERT /*+ APPEND */ INTO original_table
 2   SELECT * FROM new_table ;
```
8. Do a commit.
9. Rebuild all indexes using the NOLOGGING option.
10. Enable all constraints that were disabled in step 3.
11. Place the original table and all indexes in the LOGGING mode.
12. Backup the data.
13. Drop the holding table.

Bulk Update

To make a bulk update, follow the same steps you used for the bulk delete in the previous section, but integrate the update within the select statement. Let's say that you want to update the value column in the goods table by increasing it by 10 percent. The steps to achieve this goal are:

1. Create a new table to be used as a holding table and modify the amount in the column value at the same time, specifying the NOLOGGING option:
   ```
   SQL> CREATE TABLE new_table NOLOGGING AS
    2   SELECT (value*1.10) value, ... FROM goods;
   ```
2. Create all indexes on the holding table as they exist in the original table. Specify NOLOGGING.
3. Create all constraints, grants, and so on.
4. Drop the original table.
5. Rename the holding table to the original name.
6. Alter the table and all related indexes to the LOGGING mode.
7. Backup the data.

Some Common Problems

Some Common Problems

- Block Corruption due to NoLogging (Standby DB)
- Recover problems (NoLogging Data)
- Excessive Log Swiches on Bulk Transactions (Logging)
- ***'log file parallel write'***
- ***'log file sync'***

In some SQL statements, the user has the option of specifying the NOLOGGING clause, which indicates that the database operation is not logged in the online redo log file. Even though the user specifies the clause, a redo record is still written to the online redo log file. However, there is no data associated with this record. This can result in log application or data access errors at the standby site and manual recovery might be required to resume applying log files.

ORA-01578: ORACLE data block corrupted (file # 3, block # 2527)

ORA-01110: data file 1: '/u1/oracle/dbs/stdby/tbs_nologging_1.dbf'

ORA-26040: Data block was loaded using the NOLOGGING option"

Or

ORA-16211 unsupported record found in the archived redo log.

DATA INTENSITY

Repair NOLOGGING changes on physical and logical standby databases

After a NOLOGGING operation on the primary is detected, it is recommended to create a backup immediately if you want to recover from this operation in the future. However there are additional steps required if you have an existing physical or logical standby database. Executing these steps is crucial if you want to preserve the data integrity of your standby databases.

For a physical standby database, Data Guard's Redo Apply process will process the invalidation redo and mark the corresponding data blocks corrupted.

Repair NOLOGGING changes on physical and logical standby databases

Follow these steps to reinstate the relevant datafiles:

1. Stop Redo Apply (`RECOVER MANAGED STANDBY DATABASE CANCEL`)
2. Take the corresponding datafile(s) offline (`ALTER DATABASE DATAFILE <datafile_name> OFFLINE DROP;`)
3. Start Redo Apply (`RECOVER MANAGED STANDBY DATABASE DISCONNECT`)
4. Copy the appropriate backup of affected datafiles over from the primary database (for example, use RMAN to backup datafiles and copy them)
5. Stop Redo Apply (`RECOVER MANAGED STANDBY DATABASE CANCEL`)
6. Make the corresponding datafiles online (`ALTER DATABASE DATAFILE <datafile_name> ONLINE;`)
7. Start Redo Apply (`RECOVER MANAGED STANDBY DATABASE DISCONNECT`)

Repair NOLOGGING changes on physical and logical standby databases

For a logical standby database, Data Guard's SQL Apply process skips over the invalidation redo completely; thus, the subsequent corresponding table or index will not be updated. However, future reference to missing data will result in ORA-1403 (no data found). In order to resynchronize the table with the primary table, you need to re-create it from the primary database. Follow the steps described in Oracle Data Guard Concepts and Administration 12c Release 1 Section 11.5.5. Basically, you will be using the `DBMS_LOGSTDBY.INSTANTIATE_TABLE` procedure.

Flashback and NOLOGGING

When using Flashback Database with a target time at which a NOLOGGING operation was made, a block corruption is likely to be produced in the database objects and datafiles affected by the NOLOGGING operation.

For example, if you perform a direct path insert operation in the NOLOGGING mode, and that operation runs from 9:00 A.M. to 9:15 A.M. on July 7, 2015, and later you require to use Flashback Database to return to the target time 09:07 A.M. on that date, the objects and datafiles modified by the direct path insert may leave the database with block corruption after the Flashback Database operation completes.

TIP

“If possible, avoid using Flashback Database with a target time or SCN (System Change Number) that coincides with a NOLOGGING operation. Also, always perform a full or incremental backup of the affected datafiles immediately after any NOLOGGING operation is done, to ensure the recoverability to a given point-in-time . If you expect to use Flashback Database to return to a point-in-time during an operation such as a direct path insert, consider to perform the operation in the LOGGING mode to ensure the recoverability of it.”

fmunozalvarez@dataintensity.com

Session ID:

1621

Remember to complete your evaluation for this session within the app!

DATA INTENSITY

#C18LV