LOGGING OR NOLOGGING
THAT IS THE QUESTION

By Francisco Munoz Alvarez – May 2008

Table of Contents:

Introduction

The main question about *NOLOGGING* I hear all the time is: does creating a table with the *NOLOGGING* option means there is "no generation of redo ever", or just that the initial creation operation has no redo generation, but that DML down the road generates redo? How and when can the *NOLOGGING* option be employed?

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. Excessive redo generation is the result of excessive work on the database.

This paper covers the subject of reducing redo generation using *LOGGING* and *NOLOGGING* options, the differences between them, how it happens, how to reduce it and when to use. Also, you will find examples and tips regarding each one of them.

The main benefits of the *NOLOGGING* option suggested by the Oracle® Database Administrator's Guide 10g Release 2 are:

- ✓ Space is saved in the redo log files
- ✓ The time it takes to create the table is decreased
- ✓ Performance improves for parallel creation of large tables

"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 a Redo?

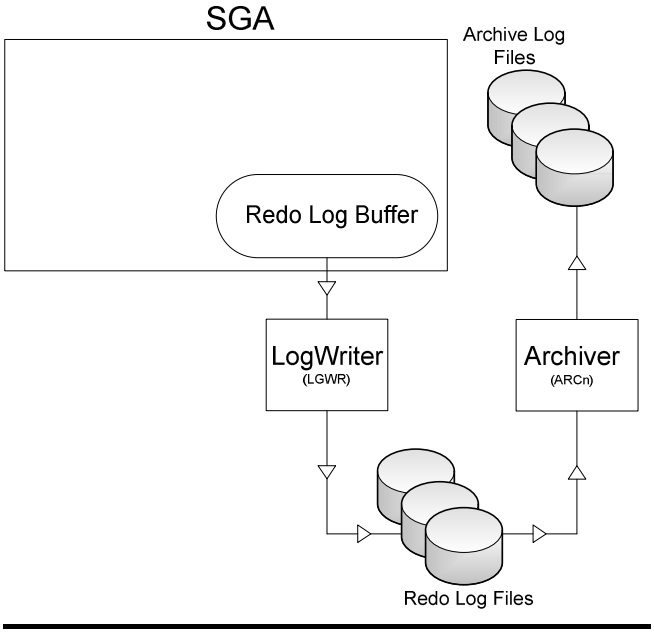

Figure 1

Let's conduct a brief summary about the redo process. 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. (See Figure 1)

The redo logs are written by the LGWR when:

- 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.

"If the log buffer is too small, then log buffer space waits will be seen during bursts of redo generation. LGWR may not begin to write redo until the *_log_io_size* threshold (by default, $^{1}/3$ of the log buffer or 1M whichever is less) has been exceeded, and the remainder of the log buffer may be filled before LGWR can complete its writes and free some space in the log buffer.

Ideally, the log buffer should be large enough to cope with all bursts of redo generation, without any log buffer space waits.

Commonly, the most severe bursts of redo generation occur immediately after a log switch, when redo generation has been disabled for some time, and there is a backlog of demand for log buffer space" by *Steve Adams.*

Redo log files record changes to the database as a result of transactions and internal Oracle server actions. (A transaction is a logical unit of work, consisting of one or more SQL statements run by a user.) Redo log files protect the database from the loss of integrity because of system failures caused by power outages, disk failures, and so on. Redo log files must be multiplexed to ensure that the information stored in them is not lost in the event of a disk failure. The redo log consists of groups of redo log files. A group consists of a redo log file and its multiplexed copies. Each identical copy is said to be a member of that group, and each group is identified by a number. The Log Writer (LGWR) process writes redo records from the redo log buffer to all members of a redo log group until the file is filled or a log switch operation is requested. Then, it switches and writes to the files in the next group. Redo log groups are used in a circular fashion. (See Figure 2)

Best practice tip:

Oracle recommends that redo log groups have at least two files per group, with the files distributed on separate disks or controllers so that no single equipment failure destroys an entire log group.

The loss of an entire log group is one of the most serious possible media failures because it can result in loss of data. The loss of a single member within a multiple-member log group is trivial and does not affect database operation, other than causing an alert to be published in the alert log.

Remember that redo logs heavily influence database performance because a commit cannot complete until the transaction information has been written to the logs. You must place your redo log files on your fastest disks served by your fastest controllers. If possible, do not place any other database files on the same disks as your redo log files. Because only one group is written to at a given time, there is no harm in having members from several groups on the same disk.

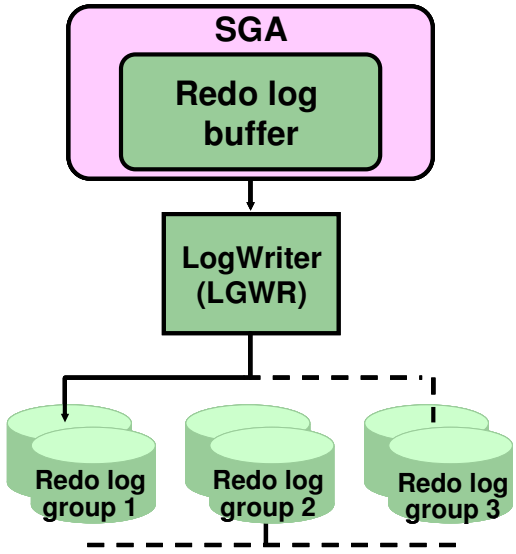

Figure 2

To avoid losing information that could be required to recover the database at some point, Oracle has an archive (ARCn) background process that archives redo log files when they become filled. However, it's important to note not all Oracle Databases will have the archive process enabled. An instance with archiving enabled, is said to operate in ARCHIVELOG mode and an instance with archiving disabled is said to operate in NO ARCHIVELOG mode.

You can determine with mode or if archiving is been used in your instance either by checking the value for the LOG_ARCHIVE_START parameter in your instance startup parameter file (pfile or spfile – This **parameter is deprecated on version 10g**), by issuing an SQL query to the v$database ("ARCHIVELOG" indicates archiving is enabled, and "NOARCHIVELOG" indicates that archiving is not enabled) or by issuing the SQL ARCHIVE LOG LIST command.

SQL> Select log_mode from v$database;

LOG_MODE

ARCHIVELOG

SQL> archive log list
Database log mode *Archive Mode*
Automatic archival *Enabled*
Archive destination *USE_DB_RECOVERY_FILE_DEST*
Oldest online log sequence *8*
Next log sequence to archive *10*
Current log sequence *10*

Redo Generation and Recoverability

The purpose of redo generation is to ensure recoverability. This is the reason why, Oracle does not give the DBA 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. The cost of maintaining the redolog records is an expensive operation involving latch management operations (CPU) and frequent write access to the redolog files (I/O). You can avoid redo logging for certain operations using the *NOLOGGING* feature.

Regarding redo generation, I saw all the times two questions in the OTN Forums:

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 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.

As long as the table space is in backup mode Oracle will write the entire block is dumped to redo when the ALTER TABLESPACE TBSNAME BEGIN BACKUP MODE is entered but later it generates the same redo. This is done due to the reason Oracle can not guaranty that a block was not copied while it was updating as part of the backup.

Let's go explain better this part:

In hot backup mode only 2 things are different:

- The first time a block is changed in a datafile that is in hot backup mode, the entire block is written to the redo log files, not just the changed bytes. Normally only the changed bytes (a redo vector) are written. In hot backup mode, the entire block is logged the first time. This is because you can get into a situation where the process copying the datafile and DBWR are working on the same block simultaneously. Let's say they are and the OS blocking read factor is 512bytes (the OS reads 512 bytes from disk at a time). The backup program goes to read an 8k Oracle block. The OS gives it 4k. Meanwhile the DBWR has asked to rewrite this block. The OS schedules the DBWR write to occur right now. The entire 8k block is rewritten. The backup program starts running again (multi-tasking OS here) and reads the last 4k of the block. The backup program has now gotten an impossible block -- the head and tail are from two points in time. We cannot deal with that during recovery. Hence, we log the entire block image so that during recovery, this block is totally rewritten from redo and is consistent with itself at least. We can recover it from there.

- 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 this file.

To limit the effect of this additional logging, you should ensure you only place one tablespace at a time in backup mode and bring the tablespace out of backup mode as soon as you have backed it up. This will reduce the number of blocks that may have to be logged to the minimum possible.

What are the differences between REDO and UNDO?

To clear this question we have this table:

	UNDO	**REDO**
Record of	How to undo a change	How to reproduce a change
Used for	Rollback, Read-Consistency	Rolling forward DB Changes
Stored in	Undo segments	Redo log files
Protect Against	Inconsistent reads in multiuser systems	Data loss

Important points about LOGGING and NOLOGGING

Despite the importance of the redo entries, Oracle gave users the ability to limit redo generation on tables and indexes by setting them in *NOLOGGING* mode.

NOLOGGING affect the recoverability. 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, this are some points regarding it:

- ✓ *NOLOGGING* is designed to handle bulk inserts of data which can be easy re-produced.
- ✓ Regardless of *LOGGING* status, writing to undo blocks causes 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 in Oracle 9i R2. (Means that the *NOLOGGING* attribute will not have any effect on the segments) If the database is in *FORCE LOGGING MODE*. *NOLOGGING* can be also override at tablespace level using *ALTER TABLESPACE … FORCE LOGGING.*
- ✓ Any change to the database dictionary will cause redo generation. This will happen to protect the data dictionary. An example: if we allocated a space above the HWM for a table, and the system fail in the middle of one *INSERT /*+ APPEND */* , the Oracle will need to rollback that data dictionary update. There will be redo generated but it is to protect the data dictionary, not your

newly inserted data (Oracle will undo the space allocation if it fails, where as your data will disappear).

- ✓ The data which are not logged will not be able to recover. The data should be backed up after the modification.
- ✓ Tables and indexes should be set back to *LOGGING* mode when the *NOLOGGING* is no longer needed.
- ✓ *NOLOGGING* is not needed for Direct Path Insert if the database is in *NO ARCHIVE LOG MODE. (See table 1.1)*

Table Mode	Insert Mode	ArchiveLog Mode	Result
LOGGING	APPEND	ARCHIVE LOG	REDO GENERATED
NOLOGGING	APPEND	ARCHIVE LOG	NO REDO
LOGGING	NO APPEND	ARCHIVE LOG	REDO GENERATED
NOLOGGING	NO APPEND	ARCHIVE LOG	REDO GENERATED
LOGGING	APPEND	NO ARCHIVE LOG	NO REDO
NOLOGGING	APPEND	NO ARCHIVE LOG	NO REDO
LOGGING	NO APPEND	NO ARCHIVE LOG	REDO GENERATED
NOLOGGING	NO APPEND	NO ARCHIVE LOG	REDO GENERATED

Table 1.1

- ✓ The data which is not able to reproduce should not use the *NOLOGGING* mode. If data which can not be reloaded was loaded using *NOLOGGING.* The data cannot be recovered when the database crashes before backing the data.
- ✓ *NOLOGGING* does not apply to *UPDATE*, *DELETE*, and *INSERT*.
- ✓ *NOLOGGING* will work during certain situations but subsequent DML will generate redo. Some of these situations are:
  - o direct load *INSERT* (using *APPEND* hint),
  - o *CREATE TABLE ... AS SELECT,*
  - o *CREATE INDEX.*
- ✓ 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 tablespace in which it resides.

The following operations are a few that cannot make use of *NOLOGGING* mode:

- ✓ Table redefinition cannot be done *NOLOGGING*.
- ✓ Temp files are always set to *NOLOGGING* mode.

The FORCE LOGGING mode is a persistent attribute of the database. That is, if the database is shut down and restarted, it remains in the same logging mode state. FORCE LOGGING must be configured again after recreating the control file.

If the database has a physical standby database, then *NOLOGGING* operations will render data blocks in the standby "logically corrupt" because of the missing redo log entries. If the standby database ever switches to the primary role, errors will occur when

trying to access objects that were previously written with the *NOLOGGING* option, you will an error like this:

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"

That doesn't sound good, and certainly I can't imagine a happy DBA called at 3:00 AM to recover a database and that error message comes up.

The options *UNRECOVERABLE* (introduced in Oracle7) and *NOLOGGING* (introduced in Oracle8) can be used to avoid the redolog entries generation for certain operations that can be easily recovered without using the database recovery mechanism. This option sends the actual DDL statements to the redo logs (this information is needed in the data dictionary) but all data loaded, modified or deleted are not sent to the redo logs.

Even though you can set the *NOLOGGING* attribute for a table, partition, index, or tablespace, this mode does not apply to every operation performed on the schema object for which you set the *NOLOGGING* attribute. See more details on which operations are supported to be executed in this mode in the following topics.

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 (see below regarding temporary tables).

While the RDBMS logically would only need to write UNDO and REDO into the transaction log, the UNDO portion must also be kept online (on disk and accessible to the RDBMS engine) to enable rollback of failed transactions. If UNDO data was only stored in the transaction log, the log could get archived and the RDBMS would have to try and read it from tape. On some platforms, the tape could be sitting in the DBA's desk drawer, so there are practical problems with this solution. Every RDBMS must meet the basic requirement of online access to undo data, and Oracle does this by storing UNDO data in what we call Rollback Segments (rollback = undo).

Because Oracle places UNDO data into a rollback segment and also must (logically) place this data into the transaction log, it is simpler to just treat rollback tablespaces like any other tablespace from a log generation perspective. That is, Oracle generates REDO for a Rollback Segment, which is logically the same as UNDO for a data block (i.e. your table, index, etc.).

Oracle's transaction log is really called the REDO log because it only contains redo records. There logically MUST be UNDO records stored in the log, but they are stored in the form of redo for rollback segments.

For temporary tables, Oracle will need to do things like facilitate rollback, but it is not necessary to bring back temporary tables following media failure.

The undo data is also needed for things like rollback to save point and read consistency, not just to reclaim space that was used by that temporary table.

Amount of redo generated for temporary tables
Metalink Note 94402.1

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.

Can Redo Generation Be Disabled During Materialized View Refresh?
Metalink Note 334878.1

There is no way to turn off redo generation when refreshing materialized views.

Setting the NOLOGGING option during the materialized view creation does not affect this, 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 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 can be reduced by setting ATOMIC_REFRESH=FALSE in the DBMS_MVIEW.REFRESH options.

Flashback and NOLOGGING

When using Flashback Database with a target time at which a *NOLOGGING* operation was in progress, block corruption is likely in the database objects and data files affected by the *NOLOGGING* operation. For example, if you perform a direct-path INSERT operation in *NOLOGGING* mode, and that operation runs from 8:00 to 8:15 on April 7, 2008, and you later need to use Flashback Database to return to the target time 08:07 on that date, the objects and data files updated by the direct-path INSERT may be left with block corruption after the Flashback Database operation completes.

If possible, avoid using Flashback Database with a target time or SCN that coincides with a *NOLOGGING* operation. Also, perform a full or incremental backup of the affected data files immediately after any *NOLOGGING* operation to ensure recoverability to points in time after the operation. If you expect to use Flashback Database to return to a point in time during an operation such as a direct-path INSERT, consider performing the operation in *LOGGING* mode.

Performance and Recovery considerations

The *NOLOGGING* mode improves performance because it generates much less log data in the redo log files helping in eliminating the time needed to execute the redo generation (latch acquisition, redolog writing, etc.). The user is responsible for backing up the data after a *NOLOGGING* insert operation in order to be able to perform media recovery.

Be aware that this feature disables the recover mechanisms for this transaction: It will be required to repeat the process from the very beginning in case of a database or instance failure.

Disabling Redo Generation (NOLOGGING)

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.

Logging can be disabled at the table level or the tablespace level. If it is done at the tablespace level then every newly created index or table in this tablespace will be in *NOLOGGING* mode (You can have logging tables inside a *NOLOGGING* tablespace). A table or an index can be created with *NOLOGGING* mode or it can be altered using *ALTER TABLE/INDEX NOLOGGING*. 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 the following situations and while running one of the following commands but not after that.

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)*
- *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*

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

- *ALTER INDEX new_index NOLOGGING.*

The actual rebuild of the index does not generate redo (all data dictionary changes associated with the rebuild will do) but after that any DML on the index will generate redo this includes direct load insert on the table which the index belongs to.

Here is another example to make this point more clear:

- *CREATE TABLE new_table_nolog_test NOLOGGING*(….);

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

- *INSERT INTO new_table_nolog_test ...,*
- *UPDATE new_table_nolog_test SET …,*
- *DELETE FROM new_table_nolog_test ..*

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

- *INSERT /*+APPEND+/ …*
- *ALTER TABLE new_table_nolog_test MOVE …*
- *ALTER TABLE new_table_nolog_test MOVE PARTITION …*

Consider the following example:

```
SQL> select name,value from v$sysstat where name like '%redo size%';

NAME                                                             VALUE
---------------------------------------------------------------- ----------
redo size                                                         27556720

SQL> insert into scott.redo1 select * from scott.redotesttab;

50000 rows created.


SQL> select name,value from v$sysstat where name like '%redo size%';

NAME                                                             VALUE
---------------------------------------------------------------- ----------
redo size                                                        28536820


SQL> insert /*+ APPEND */ into scott.redo1 select * from scott.redotesttab;

50000 rows created.
```

SQL> select name,value from v$sysstat where name like '%redo size%';

NAME	*VALUE*
redo size	*28539944*

You will notice that the redo generated via the simple insert is "980100" while a direct insert generates only "3124".

To activate the *NOLOGGING* for one of the *ALTER* commands above add the *NOLOGGING* clause after the end of the *ALTER* command.

For example:

- *ALTER TABLE new_table_nolog_test MOVE PARTITION parti_001 TABLESPACE new_ts_001 NOLOGGING;*

The same applies for *CREATE INDEX* but for *CREATE TABLE* the *NOLOGGING* should come after the table name.

Example:

- *CREATE TABLE new_table_nolog_test NOLOGGING AS SELECT * FROM big_table;*

"It is a common mistake to add the NOLOGGING option at the end of the SQL (Because oracle will consider it an alias and the table will generate a lot of logging)."

To user Direct Path Load in SQL*Loader you must run the $ORACLE_HOME/rdbms/admin/catldr.sql script before your first sqlldr is run in direct path mode. To run sqlldr in direct path mode use direct=true.

Note: Even though direct path load reduces the generation of redo, it is not totally eliminated. That's because those inserts still generate undo which in turn generates redo.

REDUCING REDO GENERATION TIPS

While Backing Up

As mentioned in the redo generation and recoverability section, user managed backups could generate a lot of redo. The best way to eliminate this problem is to use RMAN. RMAN does not need to write the entire block to redo because it knows when the block is being copied. If the user needs to use the user managed backup then they can follow these steps to reduce redo generation:

- Do not back up all the tablespaces in one go. This will put every tablespace in backup mode for longer than it needs to be and therefore generates redo for longer than it needs to do.
- Automatic backup on the busy tablespaces
- Backup a tablespace during a time when it is least busy in terms of DML.

Bulk Inserts

By bulk we mean a large percentage compared to the existing data.

To reduce the amount of redo generation in a bulk data load, the user needs to disable the indexes (when making a direct load to a table that have indexes, the indexes will produce redo) before the load then re-build them again as follow:

- *Alter index index_name unusable ; # Do this for every index*
- *Alter session set skip_unusable_indexes=true ; (*)*
- *Insert into table_name select …*
- *Alter index index_name rebuild;*

(*)skip_unusable_indexes is an instance initialization parameter in 10g and it default to true. Before 10g, skip_unusable_indexes needs to be set in a session or the user will get an error. It is a good practice to set it in a session, regardless of the database version, when the above steps is done.

Bulk Delete

1. *Create table new_table with logging*
2. *Insert into new_table select the records you want to keep from current_table.*
3. *Create the indexes on the new_table (*)*
4. *Create constraints, grants etc.*
5. *Drop current_table.*
6. *Rename new_table to current.*

(*) If the data left is so small or there are a lot of dependencies on the table (views, procedures, functions, etc) the following steps can be used instead of 3-6 above:

3. *Disable constrains on current_table.*
4. *Truncate current_table;*
5. *Insert into current_table select * from new_table ;*
6. *commit;*
7. *enable constraints*
8. *drop table new_table;*

Bulk Update

Use this method if indexes are going to be affected by the update. This is because mass updating indexes is more expensive than re-building them. If a small portion of the data is updated then use this method:

1. *Disable constraints.*
2. *Alter index index_name unusable ;*
3. *Alter session set skip_unusable_indexes=true ;*
4. *update the table.*
5. *commit;*
6. *alter index index_name rebuild ;*
7. *Enable constraints.*

If the update causes a good portion of the data to be updated then follow this method:

1. *Create new_table as select (updating statement)*
2. *create indexes on the new_table,*
3. *create grants, constraints etc on the new_table*
4. *Drop current table*
5. *Rename the new_table to current_table.*

Using Partitioning

Table and index partitioning are very useful in reducing redo generation. This is because they divide a table into smaller and manageable units. You can use the partition techniques with a table if you know which partitions will be inserted into, deleted from or updated. This way the redo generation is reduced because only the index partitions need to be built not the full one. Please note that the entire global indexes need to be rebuilt. The examples below handle local indexes.

	Alter index index_name unusable ;
Becomes	*Alter index index_name partition partition_name unusable;*
	Alter index index_name rebuild ;
Becomes	*Alter index index_name rebuild partition partition_name ;*

You can use the alter table exchange partition command to quickly displays the data you inserted into a staging table into the partition so the command:

- *Insert into current_table select * from new_table is replaced by*
- *Alter table current_table exchange partition partition_name with new_table …*

Partitioning is very useful in archiving off old historic data. The table to contain historic data is created with a range partition on a date field. When the data becomes old enough to remove, the partition get dropped. This feature is so important that Oracle created a new type of range partitions in Oracle 11g to handle this situation. The new type is called Interval partition. Here is an example for Oracle 8i to 10g:

```
Create table hist_data(
Sample_date date,
….
)
Partition by range( sample_date) (
partition data200503 values less than (to_date('04/01/2005','mm/dd/yyyy')) tablespace
ts200503,
  partition data200504 values less than (to_date('05/01/2005','mm/dd/yyyy')) tablespace
ts200504,
….
) ;
```

A year down the line we want to delete all the data before April 2005. All we need to do is an "*alter table sample_data drop partition data200503*". This is much more efficient and produces far less redo than "*delete from sample_data where sample_date < to_date('04/01/2005', 'mm/dd/yyyy')"* .

Tips For Developers

The points discussed here are a sample of how to reduce the workload on your database:

1. **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.

   That's how it should be done:

   SQL> set autotrace on statistics
   SQL> insert into test select rownum from dba_objects;

   93244 rows created.

   Statistics
   --
   ... 912326 redo size
   … 93244 rows processed

   That's how it should NOT be done:

   SQL> set autotrace on statistics
   SQL> declare
   2 cursor c1 is
   3 select rownum r from dba_objects;
   4 begin
   5 for v in c1
   6 loop
   7 insert into test values(v.r) ;
   8 end loop ;
   9 end;
   10 /

   PL/SQL procedure successfully completed.

   Statistics
   --
   ... 16112247 redo size

2. **Do not commit more than you need.** By issuing the commit command you are forcing Oracle to do some internal updates which produces redo. I ran the PL/SQL code above with the command COMMIT; inserted after line 7. The redo generated is: 28,642,216. I also ran the script above with the commit at the end followed by a "select * from test" statement to force committed block cleaning the redo generated was 13,216,188. You can see that a lot of committing to insert the same amount of data has produced far more redo. By reducing commits you reduce the strain on the LGWR process.

3. **Set sequences cache correctly.** This is important if the oracle system generates a lot of sequence numbers using oracle sequences. Oracle keeps track of the next sequence number in the SGA but it also keeps the value of the start sequence of the next set of sequences in the data dictionary according to the sequence cache setting. This is needed in case the database crashes. As sequence nextval is acquired the value in the SGA is updated, when this value is the same as the one in the data dictionary the data dictionary is updated producing redo. If sequence cache is small the data dictionary will be updated more often. This is illustrated by the following test:

   create sequence seq2 ***cache 2*** ; /* The data dictionary is updated every second nextval */
   create sequence seq20 ***cache 20****;*
   create sequence seq1000 ***cache 1000****;*

   I created 3 identical tables test2, test20 and test1000. They all have a number column. I inserted into test2 from seq2, into test20 from seq20 and into test1000 from seq1000. Example:

   SQL> insert into test20 select seq20.nextval from dba_objects ;

   68764 rows created.

   The table bellow shows the relation between the redo generated and the sequence cache size:

Cache	Redo
2	21,246,082
20	2,916,868
1000	899,719

Setting the Cache parameter to a high value when a sequence is created will only affect the next sequence value if the database was restarted set it 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 cache as high as you need.

TIPS USING NOLOGGING MODE

DIRECT PATH INSERT

To use Direct Path Insert use the /*+ *APPEND* */ hint as follow:

- *INSERT /*+ APPEND */ into … SELECT …*

When direct path insert is used oracle does the following:

- Format the data to be inserted as oracle blocks.
- Insert the blocks above the High Water Mark (HWM)
- When commit takes place the HWM is moved to the new place (The process is done bypassing the buffer cache).

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

It is very important to understand how Direct Path Inserts affects redo generation. As mentioned above it does not affect indexes but it is affected by the following factors:

- The database Archivelog mode.
- Using the /*+ *APPEND* */ hint.
- The *LOGGING* mode of the table.
- The *FORCE LOGGING* mode of the database (from 9i R2).

If the database is in *FORCE LOGGING* mode then Oracle will treat the table as if it was in *LOGGING* mode regardless of its mode. To find out if the database is in *FORCED LOGGING* or not run:

- *select FORCE_LOGGING from v$database ;*

If the /*+ *APPEND* */ Hint is not used then the insertion will generate the normal amount of redo regardless of the other factors.

This table will show the relation between *ARCHIVELOG* mode and having the table in LOGGING mode when the /*+ *APPEND* */ hint is used. This does not include index and data dictionary changes redo generation.

LOGGING MODE	ARCHIVELOG	NOARCHIVELOG
LOGGING	Redo	No Redo
NOLOGGING	No Redo	No Redo

For Bulk DML

Bulk Inserts

To load bulk data using Direct Path.

- *set table in nologging mode. Alter table table_name nologging;*
- *alter index index_name unusable ;*
- *alter session set skip_unusable_indexes=true ;(*)*
- *Insert /*+ APPEND */ into table_name select …*
- *Alter index index_name rebuild nologging;*
- *Alter table table_name logging ;*
- *Alter index index_name logging ;*
- Backup the data.

(*)skip_unusable_indexes is an instance initialization parameter in 10g and defaulted to true. Before 10g, skip_unusable_indexes needs to be set in a session or the user will get an error. It is a good practice to set it in a session, regardless of the database version, when the above is done.

There is no direct way (at the time of writing this document) of reducing redo generation for bulk update and delete. The user needs to reduce the workload on the database.

Bulk Delete

1. Create a new_table with no logging, *CREATE TABLE table_name NOLOGGING (….);* The NOLOGGING comes after the TABLE_NAME not at the end of the statement**.**
2. *Insert /*+ Append */ into new_table select* the records you want to keep from current_table.
3. Create the indexes on the new table with *NOLOGGING* (*)
4. Create constraints, grants etc.
5. Drop current_table.
6. Rename new_table to current.
7. Alter new_table and indexes logging.
8. Backup the data.

(*) If the data left is so small or there are a lot of dependencies on the table (views, procedures, functions) the following steps can be used instead of 3-6 above:

9. Disable constrains on current_table;
10. Truncate current_table;
11. make indexes unusable;
12. *alter current table NOLOGGING ;*

13. *Insert /*+ APPEND */ into current_table select * from new_table* ;
14. commit;
15. rebuild indexes with *NOLOGGING*;
16. enable constraints
17. Put current table and indexes in *LOGGING* mode
18. backup the data
19. drop table new_table;

Bulk Update

Follow the steps for bulk Delete but integrate the update within the select statement. Lets say that you want to update the value column in the goods table by increasing it by 10% the statement will be like:

1. Create a new_table with no logging, *CREATE TABLE table_name NOLOGGING (….);* (The nologging comes after the table_name, not at the end of the statement.).
2. *Insert /*+ Append */ into new_table select (update statement eg: col1, col2* 1.1,…)*
3. Create the indexes on the new table with *NOLOGGING* (*)
4. Create constraints, grants etc.
5. Drop current_table.
6. Rename new_table to current.
7. Alter new_table and indexes logging.
8. Backup the data.

Backup and Nologging

If required, it is possible that the data loaded using *NOLOGGING* can be loaded again. If the database crashed before backing up the new data then this data can not be recovered.

Here are the two scenarios of backing up:

Export (exp or expdp)

This method will allow you to recover the loaded data up to the point the export was taken but not later.

For customers using 10g Oracle Streams, there is also the option of using Data Pump Export and Import Direct Path API. For more details please refer to the Utilities Guide

Hot Backup

In order to recover any additional data or modification to the table you bulk inserted into using *NOLOGGING* the least you need to do is a hot backup of that tablespace. Remember you still generate redo for DML on the table when it is in *NOLOGGING* mode but you are strongly advised to put it in *LOGGING* mode in case you run one of the operations mentioned in the Disabling Logging section.

Redo Logging I/O-Related Wait Events

There are a number of Wait Events that happen during Redo Logging activities and most of them are I/O-related.

The two most important ones are 'log file parallel write' and 'log file sync'. Oracle foreground processes wait for 'log file sync' whereas the LGWR process waits for 'log file parallel write'.

Although we usually find 'log file sync' in the "Top 5 Wait/Timed Events" section of the Statspack report, in order to understand it we will first look at 'log file parallel write':

'log file parallel write' (See Metalink Note: 34583.1)

The LGWR background process waits for this event while it is copying redo records from the memory Log Buffer cache to the current redo group's member log files on disk.

Asynchronous I/O will be used if available to make the write parallel, otherwise these writes will be done sequentially one member after the other. However, LGWR has to wait until the I/Os to all member log files are complete before the wait is completed. Hence, the factor that determines the length of this wait is the speed with which the I/O subsystem can perform the writes to the log file members.

To reduce the time waited for this event, one approach is to reduce the amount of redo generated by the database:

- Make use of UNRECOVERABLE/NOLOGGING options.
- Reduce the number of redo group members to the minimum necessary to ensure not all members can be lost at the same time.
- Do not leave tablespaces in BACKUP mode for longer than necessary.
- Only use the minimal level of Supplemental Logging required to achieve the required functionality e.g. in LogMiner, Logical Standby or Streams.

Another approach is to tune the I/O itself:

- Place redo group members on storage locations so that parallel writes do not contend with each other.
- Do not uses RAID-5 for redo log files.
- Use Raw Devices for redo log files.
- Use faster disks for redo log files.
- If archiving is being used setup redo storage so that writes for the current redo group members do not contend with reads for the group(s) currently being archived.

'log file sync' (See Metalink Note: 34592.1)

This Wait Event occurs in Oracle foreground processes when they have issued a COMMIT or ROLLBACK operation and are waiting for it to complete.

Part (but not all) of this wait includes waiting for LGWR to copy the redo records for the session's transaction from Log Buffer memory to disk.

So, in the time that a foreground process is waiting for 'log file sync', LGWR will also wait for a portion of this time on 'log file parallel write'.

The key to understanding what is delaying 'log file sync' is to compare average times waited for 'log file sync' and 'log file parallel write':

- If they are almost similar, then redo logfile I/O is causing the delay and the guidelines for tuning it should be followed.
- If 'log file parallel write' is significantly different i.e smaller, then the delay is caused by the other parts of the Redo Logging mechanism that occur during a COMMIT/ROLLBACK (and are not I/O-related). Sometimes there will be latch contention on redo latches, evidenced by 'latch free' or 'LGWR wait for redo copy' wait events.

Handling Block Corruptions due to NOLOGGING

If a *NOLOGGING* (or *UNRECOVERABLE*) operation is performed on an object and the datafile containing that object is subsequently recovered then the data blocks affected by the *NOLOGGING* operation are marked as corrupt and will signal an ORA-1578 error when accessed. In Oracle8i an ORA-26040 is also signaled ("ORA-26040: Data block was loaded using the *NOLOGGING* option") which makes the cause fairly obvious, but earlier releases have no additional error message. If a block is corrupt due to recovery through a *NOLOGGING* operation then you can use the Metalink Note 28814.1 onwards but note that:

- ✓ Recovery cannot retrieve the *NOLOGGING* data
- ✓ No data is salvageable from inside the block

At this point basically you can do (Metalink Note 150694.1) :

- ✓ the indexes with corrupt blocks can be dropped and re-created,
- ✓ the corrupt tables can be dropped and built from an alternative data source.
- ✓ the data file(s) impacted by *NOLOGGING* operations can be refreshed from the Primary or backup which completed after *NOLOGGING* operation.
- ✓ Or a combination of the above.

Currently in Oracle 9i and Oracle 10gR1, only the primary's database V$DATAFILE view reflects *NOLOGGING* operations.. In 10gR2, the V$DATAFILE view will be enhanced to include information regarding when an invalidation redo is applied and the aforementioned corrupted blocks are written to the corresponding data file on a Redo Apply (or media recovery or standby) instance

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. This is crucial if you want to preserve the data integrity of your standby databases.

For a physical standby database, Redo Apply will process the invalidation redo and mark the corresponding data blocks corrupt, follow these steps[2] to reinstate the relevant data files:

1. Stop Redo Apply (recover managed standby database cancel)
2. Offline corresponding datafile(s) (alter database datafile <NAME> offline drop ;)
3. Start Redo Apply (recover managed standby database disconnect)

4. Copy the appropriate backup datafiles over from the primary database (e.g. use RMAN to backup datafiles and copy them)
5. Stop Redo Apply (recover managed standby database cancel)
6. Online corresponding data files (alter database datafile <NAME> online ;)
7. Start Redo Apply (recover managed standby database disconnect)

For a logical standby database, SQL Apply skips over the invalidation redo completely; so, 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 Section 9.1.7. Basically, you will be using the DBMS_LOGSTDBY.INSTANTIATE_TABLE procedure.

[1]Invalidation redo containing information about the *NOLOGGING* operation and the range of blocks it affects.

[2]Please also refer to the Data Guard documentation:
http://download-west.oracle.com/docs/cd/B14117_01/server.101/b10823/scenarios.htm#1015741

How to find Sessions Generating Lots of Redo

To find sessions generating lots of redo, you can use either of the following methods. Both methods examine the amount of undo generated. When a transaction generates undo, it will automatically generate redo as well.

The methods are:

1) Query V$SESS_IO. This view contains the column BLOCK_CHANGES which indicates how much blocks have been changed by the session. High values indicate a session generating lots of redo.

The query you can use is:

```
SQL> SELECT s.sid, s.serial#, s.username, s.program,
  2  i.block_changes
  3  FROM v$session s, v$sess_io i
  4  WHERE s.sid = i.sid
  5  ORDER BY 5 desc, 1, 2, 3, 4;
```

Run the query multiple times and examine the delta between each occurrence of BLOCK_CHANGES. Large deltas indicate high redo generation by the session.

2) Query V$TRANSACTION. These view contains information about the amount of undo blocks and undo records accessed by the transaction (as found in the USED_UBLK and USED_UREC columns).

The query you can use is:

```
SQL> SELECT s.sid, s.serial#, s.username, s.program,
  2  t.used_ublk, t.used_urec
  3  FROM v$session s, v$transaction t
  4  WHERE s.taddr = t.addr
  5  ORDER BY 5 desc, 6 desc, 1, 2, 3, 4;
```

Run the query multiple times and examine the delta between each occurrence of USED_UBLK and USED_UREC. Large deltas indicate high redo generation by the session.

You use the first query when you need to check for programs generating lots of redo when these programs activate more than one transaction. The latter query can be used to find out which particular transactions are generating redo.

Useful Scripts

To see the redo generated since instance started:

```
col name format a30 heading 'Statistic\Name'
col value heading 'Statistic\Value'
start title80 "Redo Log Statistics"
spool rep_out\&db\red_stat
SELECT name, value
FROM v$sysstat
WHERE name like '%redo%'
order by name
/
spool off
pause Press enter to continue
ttitle off
```

The redo generated during my session since the session started:

```
select value redo_size
from v$mystat, v$statname
where v$mystat.STATISTIC# = v$statname.STATISTIC#
and name = 'redo size'
/
```

The redo generated by current user sessions:

```
select v$session.sid, username, value redo_size
from v$sesstat, v$statname, v$session
where v$sesstat.STATISTIC# = v$statname.STATISTIC#
and v$session.sid = v$sesstat.sid
and name = 'redo size'
and value > 0
and username is not null
order by value
/
```

Provide a current status for redo logs:

```
column first_change# format 999,999,999  heading Change#
column group#        format 9,999     heading Grp#
column thread#       format 999       heading Th#
column sequence#     format 999,999   heading Seq#
column members       format 999       heading Mem
column archived      format a4        heading Arc?
column first_time    format a25       heading  First\Time
break on thread#
set pages 60 lines 132 feedback off
start title132 'Current Redo Log Status'
spool rep_out\&db\log_stat
select thread#, group#, sequence#,
bytes, members,archived,status,first_change#,
to_char(first_time,'dd-mon-yyyy hh24:mi') first_time
 from sys.v_$log
order by thread#, group#;
spool off
pause Press Enter to continue
set pages 22 lines 80 feedback on
clear breaks
clear columns
ttitle off
/
```

Provide redo log groups and log switch (archive generation) information:

```
set echo on
set linesize 150
set pagesize 500
column day format a16  heading 'Dia'
column d_0 format a3  heading '00'
column d_1 format a3  heading '01'
column d_2 format a3  heading '02'
column d_3 format a3  heading '03'
column d_4 format a3  heading '04'
column d_5 format a3  heading '05'
column d_6 format a3  heading '06'
column d_7 format a3  heading '07'
column d_8 format a3  heading '08'
column d_9 format a3  heading '09'
column d_10 format a3  heading '10'
column d_11 format a3  heading '11'
column d_12 format a3  heading '12'
```

```
column d_13 format a3  heading '13'
column d_14 format a3  heading '14'
column d_15 format a3  heading '15'
column d_16 format a3  heading '16'
column d_17 format a3  heading '17'
column d_18 format a3  heading '18'
column d_19 format a3  heading '19'
column d_20 format a3  heading '20'
column d_21 format a3  heading '21'
column d_22 format a3  heading '22'
column d_23 format a3  heading '23'
column  Total   format 9999
column status  format a8
column member  format a40
column archived heading 'Archived' format a8
column bytes heading 'Bytes|(MB)' format 9999
Ttitle  'Log Info'  skip 2
select l.group#,f.member,l.archived,l.bytes/1078576 bytes,l.status,f.type
  from v$log l, v$logfile f
 where l.group# = f.group#
/
Ttitle off
prompt
================================================================
================================================================
===============
Ttitle  'Log Switch on hour basis'  skip 2

select to_char(FIRST_TIME,'DY, DD-MON-YYYY') dia,
      decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'00',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'00',1,0))) d_0,
      decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'01',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'01',1,0))) d_1,
      decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'02',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'02',1,0))) d_2,
      decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'03',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'03',1,0))) d_3,
      decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'04',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'04',1,0))) d_4,
      decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'05',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'05',1,0))) d_5,
      decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'06',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'06',1,0))) d_6,
      decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'07',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'07',1,0))) d_7,
```

```
    decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'08',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'08',1,0))) d_5,
    decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'09',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'09',1,0))) d_9,
    decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'10',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'10',1,0))) d_10,
    decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'11',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'11',1,0))) d_11,
    decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'12',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'12',1,0))) d_12,
    decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'13',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'13',1,0))) d_13,
    decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'14',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'14',1,0))) d_14,
    decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'15',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'15',1,0))) d_15,
    decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'16',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'16',1,0))) d_16,
    decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'17',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'17',1,0))) d_17,
    decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'18',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'18',1,0))) d_18,
    decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'19',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'19',1,0))) d_19,
    decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'20',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'20',1,0))) d_20,
    decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'21',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'21',1,0))) d_21,
    decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'22',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'22',1,0))) d_22,
    decode(sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'23',1,0)),0,'-
',sum(decode(substr(to_char(FIRST_TIME,'HH24'),1,2),'23',1,0))) d_23,
    count(trunc(FIRST_TIME)) Total
from v$log_history
group by to_char(FIRST_TIME,'DY, DD-MON-YYYY')
order by to_date(substr(to_char(FIRST_TIME,'DY, DD-MON-YYYY'),5,15) )
/
Ttitle off
```

How to check for LOGGING/NOLOGGING objects in the DB:

Two example methods of querying the database for this information:

```
select owner , table_name, index_name
from dba_indexes
where logging='NO';


select tablespace_name, logging
from dba_tablespaces
/
```

References:

- Oracle® Database Administrator's Guide 10g Release 2
- Oracle® Database Administrator's Guide 10g Release 1
- Oracle® Database Administrator's Guide 9i Release 2

Metalink Notes:

285285.1, 373804.1, 272503.1, 212119.1, 199298.1, 188691.1, 94402.1, 160092.1, 268476.1, 427456.1, 223117.1, 427805.1, 115079.1, 432763.1, 116522.1, 49739.1, 147474.1, 269274.1, 269274.1, 4047608.8, 211125.1, 174951.1, 234096.1, 69739.1, 4047608.8, 234096.1, 28814.1, 150694.1, 167492.1, 268395.1, 334878.1, 118300.1 269274.1, 290161.1, 403747.1, 435998.1

Internet:

- AskTom
- OTN Forum
- Oracle Forensics
- Orafaq/wiki
- psoug.org