

COMMON ORACLE DATABASE MISCONCEPTIONS

Jeremiah Wilton, Amazon.com

<u>INTRODUCTION</u>	2
<u>BACKUP AND RECOVERY</u>	2
<u>MISCONCEPTION: HOT BACKUP MODE STOPS WRITING TO THE DATAFILES</u>	2
<u>MISCONCEPTION: MEDIA RECOVERY WILL BE REQUIRED IN THE CASE OF INSTANCE FAILURE DURING BACKUP MODE</u>	4
<u>MISCONCEPTION: ALWAYS ‘SWITCH LOGFILE’ BEFORE/AFTER ONLINE BACKUPS</u>	5
<u>MISCONCEPTION: COLD BACKUP ONCE A WEEK AS A “BASELINE”</u>	5
<u>MISCONCEPTION: EXPORT IS A GOOD WAY TO BACK UP THE DATABASE</u>	6
<u>REDO AND ROLLBACK</u>	7
<u>MISCONCEPTION: ‘SHUTDOWN ABORT’ IS BAD</u>	7
<u>MISCONCEPTION: “SNAPSHOT TOO OLD” CAN BE AVOIDED BY USING “SET TRANSACTION USE ROLLBACK SEGMENT.”</u>	7
<u>MISCONCEPTION: BIG BATCH JOBS SHOULD USE A ‘DEDICATED’ ROLLBACK SEGMENT</u>	8
<u>MISCONCEPTION: CHECKPOINT NOT COMPLETE – MISGUIDED SOLUTIONS</u>	9
<u>MISCONCEPTION: NOLOGGING TURNS OFF REDO LOGGING</u>	10
<u>MISCONCEPTION: AFTER ACTIVATING A STANDBY, THE PRIMARY REQUIRES RE-COPYING TO BECOME THE STANDBY</u>	11
<u>GENERAL PRACTICES</u>	11
<u>MISCONCEPTION: LOTS OF EXTENTS ARE BAD</u>	11
<u>MISCONCEPTION: LISTENERS MUST BE STARTED BEFORE THE INSTANCE</u>	13
<u>MISCONCEPTION: COUNT (1) IS BETTER THAN COUNT (*)</u>	13

INTRODUCTION

The halls of the companies and institutions using Oracle's database echo with the slightly raised voices of DBAs arguing over the finer points of Oracle administration. Disputes arise because there are two kinds of DBAs. Some DBAs base their methods on recommendations they read or heard somewhere, without completely understanding the reasons why. Other DBAs will do nothing without hard facts to support their methods. These latter DBAs believe that absent proof positive, even Oracle's own documentation is not to be entirely trusted. Many of the standard practices, blanket recommendations, warnings, admonitions and advice included in the bulky third-party literature surrounding Oracle, and even in Oracle's documentation, are based on opinions formed in a narrow environment, limited in scope and scale. Such advice may be heeded, but only upon understanding the reasoning behind the advice, and its applicability to the user's environment.

The consequence of large numbers of Oracle professionals taking advice without understanding it has been a great profusion of theories and ideas with no basis in either fact or experience. These theories have a way of spreading like wildfire, in the form of "common wisdom" and "best practices." Unsound ideas abound only because not enough people challenge them.

This document is a collection of short articles, some accompanied by proofs, that attempt to point out, then debunk, a number of common misconceptions and misunderstandings, as well as just plain bad advice, that few people have stepped forward to challenge.

BACKUP AND RECOVERY

MISCONCEPTION: HOT BACKUP MODE STOPS WRITING TO THE DATAFILES

During an Oracle tablespace hot backup, a script or program puts a tablespace into backup mode, then copies the datafiles to disk or tape, then takes the tablespace out of backup mode. Although these steps are widely understood, the idea that datafiles are not written during backup mode is probably the most alarming and widely held misconception about Oracle. So many people think this is true, that it is actually asserted as fact in the backup/recovery sections of a number of major published books on Oracle. Numerous websites on the topic also make the assertion that datafiles are not writable during backup. Many people have reported that Oracle Education DBA classes and third-party classes teach this fallacy as fact.

The erroneous descriptions have several permutations. Most claim that while the datafiles are allegedly not writable, changes are stored somewhere in the SGA, the redologs, the rollback segments or some combination thereof, then written back into the datafile when the tablespace is taken out of backup mode. The passage from *Oracle Unleashed* describing this supposed mechanism is representative.

When you place a tablespace in backup mode, the Oracle instance notes that a backup is being performed and internally compensates for it. As you know, it is impossible to make an authentic copy of a database file that is being written to. On receipt of the command to begin the backup, however, Oracle ceases to make direct changes to the database file. It uses a complex combination of rollback segments, buffers, redo logs, and archive logs to store the data until the end backup command is received and the database files are brought back in sync.

Simplifying a hot backup in this way is tantamount to classifying the USS Nimitz as a boat. The complexity of the actions taken by the Oracle RDBMS under a hot backup could consume an entire chapter and is beyond the scope of this book. What you should understand is the trade-off for taking a hot backup is increased use of rollback segments, redo logs, archive logs, and internal buffer areas within the SGA.

(From *Oracle Unleashed*, Copyright © SAMS/Macmillan, Inc. 1997, chapter 14)

In fact, Oracle's tablespace hot backup does not work this way at all. Rather, it is a simple and failure-resistant mechanism. It absolutely does not stop writing the datafiles, and actually allows continued operation of the database

almost exactly as during normal operation. Contrary to the characterization as “complex” in *SAMS Oracle Unleashed*, it can be almost completely summarized in one sentence.

- **The tablespace is checkpointed, the checkpoint SCN marker in the datafile headers cease to increment with checkpoints, and full images of changed DB blocks are written to the redologs.**

Those three actions are all that is required to guarantee consistency once the file is restored and recovery is applied. By freezing the checkpoint SCN in the file headers, any subsequent recovery on that backup copy of the file will know that it must commence at that SCN. Having an old SCN in the file header tells recovery that the file is an old one, and that it should look for the redolog file containing that SCN, and apply recovery starting there. Note that checkpoints to datafiles in hot backup mode are not suppressed during the backup, only the incrementing of the main checkpoint SCN flag. A “hot backup checkpoint” SCN marker in the file header continues to increment as periodic or incremental checkpoints progress normally.

There is a confusing side effect of having the checkpoint SCN artificially frozen at an SCN earlier than the true checkpointed SCN of the datafile. In the event of a system crash or a `shutdown abort` during hot backup of a tablespace, the automatic crash recovery routine at startup will think that the files for that tablespace are quite out of date, and will actually suggest the application of old archived redologs in order to bring them back into sync with the rest of the database. It is unnecessary, in this case, to heed Oracle’s suggestion. With the database started up in mount mode, simply check `v$backup` and `v$datafile` to determine which datafiles were in backup mode at the time the database crashed. For each file in backup mode, issue an `alter database datafile '<file name>' end backup;` This action will update the checkpoint SCN in the file headers to be the same as the hot backup checkpoint SCN (which is a true representation of the last SCN to which the datafile is truly checkpointed). Once this action is taken, it allows normal crash recovery to proceed during the `alter database open;` command.

By initially checkpointing the datafiles that comprise the tablespace and logging full block images to redo, Oracle guarantees that any blocks changed in the datafile while in hot backup mode will *also* be available in the redologs in case they are ever used for a recovery.

It is well understood by much of the Oracle user community that during hot backup mode, a greater volume of redo is generated for changes to the tablespace being backed up than when the tablespace is not in backup mode. This is the result of the logging of full images of changed blocks in these tablespaces to the redologs. Normally, Oracle logs an entry in the redologs for every change in the database, but it does not log the whole image of the database block. By logging full images of changed DB blocks to the redologs during backup mode, Oracle eliminates the possibility of the backup containing irresolvable *split blocks*. To understand this reasoning, you must first understand what a split block is.

Typically, Oracle database blocks are a multiple of O/S blocks. For instance, most Unix filesystems have a default block size of 512 bytes, while Oracle’s default block size is 2k. This means that the filesystem stores data in 512 byte chunks, while Oracle performs reads and writes in 2k chunks, or multiples thereof. While backing up a datafile, your backup script makes a copy of the datafile from the filesystem, using O/S utilities such as `copy`, `dd`, `cpio`, or `ocopy`. As it is making this copy, it is reading in O/S-block sized increments. If the database writer happens to be writing a DB block into the datafile at the same time that your script is reading that block’s constituent O/S blocks, your backup copy of the DB block could contain some O/S blocks from before the database performed the write, and some from after. This would be a split block.

By logging the full block image of the changed block to the redologs, it guarantees that in the event of a recovery, any split blocks that might be in the backup copy of the datafile will be resolved by overlaying them with the full legitimate image of the block from the redologs. Upon completion of a recovery, any blocks that got copied in a split state into the backup will have been resolved through application of full block images from the redologs.

It is important to remember that all of these mechanisms exist for the benefit of the backup copy of the files and the recovery process, and have very little effect on the datafiles of the database being backed up. The database files are read by server processes and written by the database writer throughout the backup, as they are when a backup is not taking place. The only difference manifested in the open database files is the freezing of the checkpoint SCN, and the incrementing of the hot-backup SCN.

To demonstrate the principle, we can formulate a simple proof.

- A table is created and a row of data inserted.

```
SQL> create table fruit (name varchar2(32)) tablespace administrator;
Table created.
SQL> insert into fruit values ('orange');
1 row created.
SQL> commit;
Commit complete.
```
- A checkpoint is forced, to flush dirty DB block buffers to the datafiles.

```
SQL> alter system checkpoint;
System altered.
```
- The block number in the datafile where the data resides is determined.

```
SQL> select dbms_rowid.rowid_block_number(rowid) blk, name from fruit;
   BLK NAME
-----
      3 orange
```
- The DB block containing an identifiable piece of the data is excised from a datafile using the Unix dd command, allowing 9 DB blocks for the raw partition header and datafile header, plus the block offset of three.

```
unixhost% dd ibs=8192 skip=11 count=1 \
> if=/raw-01/databases/toy/administrator-01.dbf | strings
1+0 records in
16+0 records out
orange
```
- The tablespace is placed into hot backup mode.

```
SQL> alter tablespace administrator begin backup;
Tablespace altered.
```
- The row is updated, committed, and a global checkpoint forced on the database.

```
SQL> update fruit set name = 'plum';
1 row updated
SQL> commit;
Commit complete.
SQL> alter system checkpoint;
System altered.
```
- The same block is extracted, showing that the DB block has been written on disk.

```
unixhost% dd ibs=8192 skip=11 count=1 \
> if=/raw-01/databases/toy/administrator-01.dbf | strings
1+0 records in
16+0 records out
orange,
plum
```
- The tablespace is taken out of backup mode

```
SQL> alter tablespace administrator end backup;
```

It is quite clear from this demonstration that datafiles receive writes even during hot backup mode.

MISCONCEPTION: MEDIA RECOVERY WILL BE REQUIRED IN THE CASE OF INSTANCE FAILURE DURING BACKUP MODE

If an instance crashes or is aborted while one of the tablespaces is in backup mode, the tablespace(s) will appear to need media recovery upon startup. In normal instance failure situations, automatic crash recovery using the online redologs will be automatically performed, with no intervention required.

Because hot backup mode artificially freezes the checkpoint SCN in the datafile header, it will appear on startup after a crash to be a file restored from backup. Consequently, Oracle will suggest that you apply archived redologs from the time that the tablespace was placed in backup mode. As we learned in the previous section on hot backup mode, the only difference between a datafile in backup mode and not in backup mode is the checkpoint SCN in the header. Because there is no data missing from the datafile, there is no need to apply recovery to it. For this reason, Oracle provides a way to clear the hot backup flags in the datafile header before you open the database.

If an instance will not start up automatically after a failure, and the reason is that a datafile needs recovery,

then the `v$backup` view may be queried from mount mode. This view will show which datafiles are in backup mode. Based on this information, the following command may be used to take the files out of backup mode:

- `alter database datafile <filenum> end backup;`

Many DBAs create their own startup and shutdown scripts that improve on Oracle's `dbstart` and `dbshut` scripts by introducing exception handling and other important features. A good feature to add to any database startup script is the ability to take files out of backup mode. By having scripts that handle simple common complications, overall stability may be improved.

MISCONCEPTION: ALWAYS 'SWITCH LOGFILE' BEFORE/AFTER ONLINE BACKUPS

In order to be sure that all changes generated during a hot backup are available for a recovery, Oracle recommends that the current online log be archived after the last tablespace in a hot backup is taken out of backup mode. A similar recommendation stands for after a recovery manager online database backup is taken. A wide variety of literature on backup and recovery, including the oracle documentation, suggests using the `'alter system switch logfile'` command to achieve these ends.

The use of `'switch logfile'` to obtain archives of the current logs is ill advised, because the command returns success as soon as the log writer has moved to the next log, but before the previous log has been completely archived. This could allow a backup script to begin copying that archived redolog before it is completely archived, resulting in an incomplete copy of the log in the backup.

A better command to use for archiving the current log is `'alter system archive log current.'` This command will not return until the current log is completely archived.

For the same reasons outlined above, a backup script should never just back up "all the archived redologs." If a script does not restrict itself only to those logs that it knows to be archived, it may improperly try to back up an archived redolog that is not yet completely archived. To determine which logs are archived, a backup script should query the `v$archived_log` view to obtain a file list for copying.

MISCONCEPTION: COLD BACKUP ONCE A WEEK AS A "BASELINE"

A number of books and manuals suggest a backup schedule for Oracle databases in archive mode something like, "daily hot (online) backups and weekly cold backups." What is strange about this recommendation is that the weekly cold backups serve no apparent purpose, other than to needlessly remove the instance or instances from service for some period of time. Shutting the database down for a cold backup also has the side effect of requiring the buffer cache be reloaded all over again after startup, which degrades performance while all reads are initially physical for a period of time.

Hot backups, used exclusively as a backup method, provide protection that is equal in effectiveness to cold backups. No additional protection can be gained by adding periodic cold backups to a backup regime. In an environment requiring a high degree of availability from the Oracle instance, there is only one scenario in which a cold backup could be considered appropriate. In the event that the database is opened with the `'resetlogs'` option, such as during a major (7.x→8.x) version upgrade, or after incomplete media recovery, new logs generated after the database is open cannot be applied to a previous backup. If there is no external way that new transactions could be re-created in the event of a failure, a cold backup may be taken prior to making the database available for general service. This provides a backup to which the new archived redologs may be applied in case a recovery is required very soon after the `resetlogs`. Alternatively the DBA could begin an online backup as soon as the database is opened, allowing users to perform transactions while the backup is taken, hoping that no media failures occur before the initial backup completes. In such a case, the database is briefly prone to an unrecoverable media failure, in which a backup prior to the `resetlogs` would have to be restored, and rolled forward only up to the time of the `resetlogs`. As soon as the initial online backup completes, however, the database is in the clear, and all subsequent transactions can be applied to the initial backup. No incomplete point-in-time recovery between the `resetlogs` and completion of the initial backup will

be possible.

In the unusual event that recovery past a resetlogs is necessary, Oracle Support can guide users through a procedure for doing so that is not generally supported but that works fine.

One concern often voiced in support of periodic cold backup is the possibility of corruption in the archived redologs necessary to recover an online backup. This valid concern may be mitigated in several ways:

- Maintain a site repository for redundant storage of archived redologs from several instances
- Reduce backup time using parallelism and fast media. This decreases the statistical probability that any redo entry necessary to bring a database consistent will be corrupt
- Use disk mirroring in addition to Oracle log multiplexing
- Monitor the O/S system logs for I/O-related messages
- Apply the logs to a standby to prove their applicability

Understand that the likelihood of encountering redo corruption in the relatively small set of logs that are necessary to bring an online backup consistent, and that such complications may be overcome in many circumstances using undocumented methods under the guidance of Oracle Support.

MISCONCEPTION: EXPORT IS A GOOD WAY TO BACK UP THE DATABASE

Many guides to backup and recovery give the Export and Import utilities billing equal to physical backup methods. As a method, Export and Import provide uniformly inferior reliability on backups, as well as recoveries. In addition to being failure prone, mean time to recovery with Import is much longer than comparative recoveries using physical backups. Export creates “dump” files that contain the DDL for all objects and the table data in the database.

Much of Export’s reliability problem stems from the fact that it extracts data using SQL and read consistency (like a query), making exports prone to “ORA-01555 Snapshot too old: Rollback segment with name <RBS> too small.” Export stands the highest probability of encountering ORA-01555 when using the `CONSISTENT=Y` argument. This feature forces the export to extract all table data consistent as of the point in time that the export is started. If the export takes an hour to complete, it must generate consistent reads an hour old. As even moderately utilized databases scale, the likelihood of successfully completing the consistent export becomes slimmer.

Even if export is performed without `CONSISTENT=Y`, it doesn’t mean that read consistency is “shut off” for the exporting session. It just means that each table will be exported consistent with itself, but not necessarily consistent with other tables. If you have a very large table, the possibility of ORA-01555 may be just as likely. Furthermore, since tables are not exported consistently with each other, it is very likely that during an import, parent-child relationships will be out of sync, and Import will fail to successfully enable foreign key constraints. This makes potential “recoveries” using Import unpredictable and highly prone to failure.

In addition to the above problems, several steps must be taken prior to using Import to restore an entire database. The database must be created using `CREATE DATABASE`. Unless someone has thought to remember to save the original create database command, it will be necessary to fumble around until this step has been completed. If any space has been subsequently added to the `SYSTEM` tablespace since inception, that space must be made available before import, or else the import will fail.

Another reason not to use Import as a backup/recovery method is the poor mean time to recovery, when compared with physical backups/restores. For every block of data that import restores, it must read from disk, process into an insert statement, use rollback and redo to load it, and eventually commit the data. All told, this probably consists of over 100 calls per block, on average. In contrast, for every block restored from a traditional physical backup, it is only necessary to read the block from the backup media, and write it to disk.

Many proponents of using Export as a regular part of their backup routine cite the need to do a single-table restore, or a point-in-time recovery of just one table. Unfortunately, this method does not really allow a point-in-time recovery, unless that point in time happens to be one of the points in time that an export was actually taken. Since media recovery cannot be applied to exports, you are stuck with the data in the table as it was as of the last export, and any

data changed since then is lost, or stale.

The propensity to rely on exports as protection from such single table data loss points out another common misconception, that single-table restores/recoveries cannot be accomplished with physical backups. On the contrary, with very little practice, a DBA can learn to restore a small subset of the physical backup as a clone of the original database (even on the same host!). This clone, consisting only of the tablespace containing the desired data, the `SYSTEM` tablespace, and any tablespaces containing online rollback segments at the time of the backup, can be rolled forward to any point in time. If someone has accidentally deleted data, truncated or dropped a table, then the clone can be recovered to a point just prior to the damage, and opened. Once the clone is opened, the table in question can be exported and imported into the production database.

REDO AND ROLLBACK

MISCONCEPTION: ‘SHUTDOWN ABORT’ IS BAD

The best way to shut down a database is to force a checkpoint and then use the `‘shutdown abort’` command. In the few situations such as cold backups and version upgrades where consistent datafiles are required, the shutdown abort can be followed by a `‘startup restrict’` and a `‘shutdown immediate’`.

A common misconception holds that `‘shutdown abort’` is somehow more dangerous or reckless than the other shutdown modes, and can result in data being lost. This is not the case. `‘shutdown abort’` terminates the Oracle background and shadow processes, and removes the shared memory segments, effectively terminating a running instance. This leaves all committed transactions intact in the online redologs, even if the data associated with them has not been written to the datafiles. Upon startup, recovery is applied from the online logs beginning from the time of the most recent checkpoint. When recovery is complete, the database is opened for use. Transaction rollback occurs in the background after startup, so no user’s time is wasted waiting for all uncommitted transactions to roll back.

A common argument against `‘shutdown abort’` is that because instance recovery is necessary after a `‘shutdown abort,’` the total time down will “take as long as if shutdown immediate had been used on the way down.” This argument can be easily overcome. When starting up after a `‘shutdown abort,’` the amount of time spent in instance recovery depends largely upon how recently the last checkpoint was issued. By forcing a checkpoint immediately prior to issuing `‘shutdown abort,’` the redo required to complete crash recovery and bring the database open will be minimal.

The alternative in an active environment to `‘shutdown abort’` is `‘shutdown immediate,’` but immediate shutdowns take too long, rolling back transactions and performing other tasks while precious seconds pass by.

`‘shutdown abort’` can come in handy for very brief downtimes, such as those required to change a non-dynamic initialization parameter. In practice on Oracle instances with very large SGAs, such quick “bounces” can typically take as little as 25 seconds.

In order to expedite planned shutdowns and startups, the same scripts that are devised for reliable and fast startups at machine boot and shutdown should be used for manual shutdowns and startups. The scripts should be used because they can issue commands faster and more accurately than a human typing, and can be designed to resolve potential complications.

MISCONCEPTION: “SNAPSHOT TOO OLD” CAN BE AVOIDED BY USING “SET TRANSACTION USE ROLLBACK SEGMENT.”

It is often asserted that if you get the ubiquitous `“ORA-01555: Snapshot too old: Rollback segment with name <RBS> too small,”` you are “running out of rollback segment.” This misconception is partially the fault of the misleading error text. The rollback segment mentioned is not necessarily “too small.” This error is actually a failure to obtain read consistency.

When you perform a query, all results must be returned by Oracle consistent with the point in time that the query began. If other sessions change data between the time you begin your query and when you read that data, your session

must obtain a “consistent read” of the data block using the data in the rollback segment that was generated when the other session changed the data. Since rollback segments are reused cyclically, and since Oracle can’t guess what you might need to read in the future, there is no way for Oracle to protect or preserve all the rollback entries your query might need over its lifetime. `ORA-01555` occurs when the rollback entries you need for your query have been overwritten through cyclical reuse, or rollback segment shrinkage.

So, since `ORA-01555` has to do with *reading* from the database, and transactions have to do with *writing* to the database, telling your session to use a particular rollback segment for a *transaction* that you are not even using will have no effect on the success of the query. You will notice that each time `ORA-01555` is returned, it usually specifies a different rollback segment. That is because you have no control over which rollback segments the other sessions in the database have used, and therefore no control over where your select may need to obtain data to construct a consistent read.

The best way to avoid `ORA-01555` is to tune the query to complete faster, so that the likelihood both of needing to generate CR blocks and of not having necessary rollback entries to do so, are both reduced. There are number of ways to accomplish this, including creating indexes, optimizing for full table scans (`db_file_multiblock_read_count`), and parallel query.

Alternatively, you can increase the size and/or number of rollback segments, which will make reuse of a section of rollback segment less frequent. You can also hunt down the sessions that are imposing a high rate of change on the database, and therefore rapidly causing rollback segments to be reused. These sessions can be tracked down by looking at `v$sesstat` for the largest user of the “redo size” statistic per period of time connected.

Finally, it is important to make sure that rollback entries are not being obliterated through unnecessary shrinks. Looking at `v$rollstat` can reveal the number of times a rollback segment has shrunken since instance startup. If they are shrinking much at all, then the ‘optimal’ size of the rollback segment is probably too low, and should be increased, taking into account available space in the tablespace and appropriate rollback segment size based on your understanding of the system’s workload. If you are frequently manually shrinking rollback segments to free up space in the tablespace, then you yourself may be the cause of the `ORA-01555s`.

In Oracle 9i, internally managed undo (rollback) is available as a new feature. This feature allows a minimum retention time to be specified for undo entries, so that DBAs can set a service level agreement on consistent read capability with users. The only problem with this model is that one job using an extraordinarily large amount of undo within the retention period can use up all available space for undo and ruin the party for everyone.

MISCONCEPTION: BIG BATCH JOBS SHOULD USE A ‘DEDICATED’ ROLLBACK SEGMENT

A common habit among DBAs is to assign large batch jobs that make many changes, or have very large transactions, to a dedicated rollback segment, using the “`set transaction use rollback segment <rbs>;`” command. Many databases have a rollback segment called `rbs_large` or something similar for this purpose.

The best reason not to devote one rollback segment to big batch jobs is that you can’t keep other small transactions out of such a rollback segment. Transactions in other sessions that do not specify a rollback segment can be assigned to any online rollback segment, including the one you have “reserved” for large transactions. This means that the batch jobs that are churning through the big rollback segment are almost certain to rapidly reuse parts of the rollback segment that other sessions will need in order to construct consistent reads. In other words, by specifying one of the rollback segments to be more heavily used than the others, you are basically guaranteeing failure due to `ORA-01555` in other sessions.

The problem that needs to be addressed in cases such as these is the design of the batch processes that require such a huge transaction. Transactions in Oracle should normally be kept fairly short. While it is undesirable to commit for every row processed (which will cause excessive redolog buffer flushing and high waits on “`log file sync`”), it makes sense to have batch processes commit for every few hundred rows processed. In this way, the transactions of a large batch process may be distributed evenly among all available rollback segments, statistically improving the overall ability of the rollback segments to provide data for consistent reads.

Often the greatest barrier to changing batch jobs to commit continuously is failure tolerance. If a batch job that

commits continuously fails part way through, then there must be a way to restart that batch job where it left off, or clean up from the first attempt so that the job can be started over. Whereas before this restart capability was provided by rolling back the large transaction, the proposed rollback-friendly model requires that the appropriate application logic be built into the batch processing software.

Old email on same topic:

Assigning a large rollback segment to a batch job is totally unnecessary, especially if the batch job commits semi-frequently. By assigning large jobs to that one RBS, you virtually guarantee that you will cause people's long-running reports and queries to fail with `ORA-01555`. When you create a large rollback segment and put it online, there is no way for you to prevent other small transactions from also randomly selecting and using that RBS. As your batch job makes massive changes and runs through the big RBS over and over, it obliterates not only its recently committed changes, but also the recently committed changes of the other small transactions going on in the instance. Should any of the reports or queries running against the system need one of those undo entries that your batch job obliterated in order to construct a CR cloned block in the buffer cache, the report will fail with `ORA-01555`.

If you allow your semi-frequently committing batch jobs to randomly select rollback segments like all the rest of the transactions in your system, you will be less likely to overwrite recently committed changes, since the burden of the batch transactions is spread around, rather than concentrated in a single rollback segment.

MISCONCEPTION: CHECKPOINT NOT COMPLETE – MISGUIDED SOLUTIONS

A common problem for people with very active systems who use filesystems to store their datafiles is the error, “Thread <n> cannot allocate new log, sequence <nnn>; Checkpoint not complete” The most commonly recommended remedies for this situation are either to use larger or more online redologs.

Unfortunately, if ‘checkpoint not complete’ is a chronic problem, neither of these solutions will eliminate the problem. They may forestall, or even reduce the frequency of the error, but the problem will not be solved.

‘checkpoint not complete’ is a result of an instance filling and switching through all available online redologs before one checkpoint can complete. Because Oracle must always be able to recover from instance failure from the last checkpoint forward, the rules of recovery prevent an instance from reusing any online redolog that contains changes newer than the last checkpoint.

A checkpoint is the writing of all dirty blocks in the buffer cache to the datafiles as of a particular SCN. In general, if a checkpoint cannot complete in a timely fashion, it is a result of slow I/O to the datafiles. The possible solutions to this problem seek to eliminate the I/O bottleneck, or compensate for a slow I/O subsystem.

The problem with the recommendations to increase the size or number of redologs is that if the rate of DML activity is so high that checkpoints cannot keep up, there is no reason to think that by increasing the amount of online redologs, that it will make the I/O subsystem any more able to keep up. That is to say, it will take a longer time to fill up all the online logs, but the basic problem of not being able to write out dirty blocks as fast as the database is changing will still be there.

The first step in diagnosing ‘checkpoint not complete’ is to determine if the problem is chronic or not. If the error appears in the alert log many times a day, or consistently during peak hours, then the problem is chronic. If the error appears at the same time every day or every week, or if the problem is only occasional, it is not chronic.

Non-chronic ‘checkpoint not complete’ probably doesn’t require any re-engineering of the systems architecture. It is most likely the result of a single application suddenly making a large amount of DML (inserts, updates, deletes) to the database in a short time. The best way to solve this problem is to find out if the application can reduce its generation of redo by performing its changes ‘nologging.’ Any bulk inserts can be done using append mode unrecoverable, and generate no significant redo. Deletes that clear a whole table or a whole class of records can be converted to truncates of the table or of a partition. It very least, the application can be modified to throttle the rate of change back to a rate that the I/O subsystem can keep up with. Even the crude solution of increasing the number or size of redologs may solve sporadic, non-chronic occurrences of ‘checkpoint not complete.’

Chronic ‘checkpoint not complete’ is a more complicated problem. It means that overall, the rate of DML of the instance is higher than the I/O subsystem can support. In systems with chronically slow I/O, application

performance will be degraded, because the buffer cache is not purged of dirty blocks fast enough or frequently enough. Such systems show relatively long `time_waited` for the “buffer busy wait” and “write complete wait” events in `v$system_event`. The solution to such a problem is either to compensate for the problem by making the checkpoint more aggressive, or to solve the problem by making the I/O more efficient.

To understand the solution to this problem, it is first necessary to understand something about how checkpoints work. When a periodic checkpoint is being performed, a certain portion of the database writer’s capacity, or “write batch,” is made available for the checkpoint to use. If the checkpoint can’t complete in time, it is valid to infer that Oracle is not using enough of the database writer’s write batch for the checkpoint, and that it should probably use a larger portion. Note that none of this has anything to do with the `ckpt` background process. Checkpoints are performed by the database writer. `ckpt` just relieves the log writer from updating file header SCNs when checkpoints complete.

In Oracle8, a new feature, sometimes called “incremental checkpoint” or “fast start instance recovery” was introduced. This feature is enabled with the initialization parameter `FAST_START_MTTR_TARGET` in *9i* (`FAST_START_IO_TARGET` in *8i*), and completely changes the behavior of Oracle checkpointing. Instead of performing large checkpoints at periodic intervals, the database writer tries to keep the number of dirty blocks in the buffer cache low enough to guarantee rapid recovery in the event of a crash. It frequently updates the file headers to reflect the fact that there are not dirty buffers older than a particular SCN. If the number of dirty blocks starts to grow too large, a greater portion of the database writer’s write batch will be given over to writing those blocks out. Using `FAST_START_MTTR_TARGET` is one way to avoid ‘checkpoint not complete’ while living with a chronically slow I/O subsystem.

In Oracle7, although there is no incremental checkpoint feature, there is an “undocumented” initialization parameter that can be set to devote a larger portion of the write batch to checkpoints when they are in progress. The parameter is `_DB_BLOCK_CHECKPOINT_BATCH`, and to set it you need to find out the size in blocks of the write batch and the current checkpoint batch. This can be obtained from the internal memory structure `x$kvii`.

Another way to compensate for slow I/O is to increase the number of database writers. By dedicating more processes to writing the blocks out, it may be possible to allow checkpoints to keep up with the rate of DML activity on the database. Bear in mind that certain filesystems, such as AdvFS on Compaq Tru64 Unix, obtain no benefit, from multiple database writers. Such filesystems exclusively lock a file for write when any block is written to that file. This causes multiple database writers to queue up behind each other waiting to write blocks to a particular file. Oracle has provided notes on Metalink regarding such filesystems.

If you are more inclined to address the root cause of the problem than to compensate for it, then there are a few measures you can take. Oracle supports asynchronous I/O on most platforms, if datafiles are stored in raw or logical volumes. Conversion to raw or LVs requires significant engineering, but is much easier than totally replacing the storage hardware. Using asynchronous I/O also relieves the aforementioned file-locking bottleneck on certain types of filesystems.

I/O hardware upgrade or replacement is the most complex approach to solving the problem of slow I/O. Using RAID disk arrays allows data to be “striped” across many disks, allowing a high rate of write-out. Caching disk controllers add a battery-protected cache for fast write-out of data.

MISCONCEPTION: NOLOGGING TURNS OFF REDO LOGGING

Many people believe that by setting a segment to `NOLOGGING` mode, no redo will be generated for DML performed against that segment. In fact, only a limited number of operations can be performed `NOLOGGING`. According to the Oracle *8i* Concepts Guide, only the following operations may take advantage of the `NOLOGGING` feature:

- direct load (SQL*Loader)
- direct-load INSERT
- CREATE TABLE ... AS SELECT
- CREATE INDEX

- ALTER TABLE ... MOVE PARTITION
- ALTER TABLE ... SPLIT PARTITION
- ALTER INDEX ... SPLIT PARTITION
- ALTER INDEX ... REBUILD
- ALTER INDEX ... REBUILD PARTITION
- INSERT, UPDATE, and DELETE on LOBs in NOCACHE NOLOGGING mode stored out of line

MISCONCEPTION: AFTER ACTIVATING A STANDBY, THE PRIMARY REQUIRES RE-COPYING TO BECOME THE STANDBY

When switching to a standby database, there are two basic scenarios. In one scenario, the switchover is undertaken in an emergency, and due to the unavailability of the primary host or the storage, the final one or two redologs are not available, and therefore cannot be applied to the standby. In such a case, the standby must be activated after incomplete recovery, without applying the last changes from the primary. This scenario is extremely rare.

The second, and much more likely scenario, is that the switchover is undertaken as a planned event, to perform maintenance on the primary host, or in an emergency event that does not preclude access to the most recent couple redologs. In such a scenario, complete recovery of the standby up to the point that the primary was stopped is both possible and advisable. By opening the standby without resetlogs as of that point in time, it is possible to immediately press the primary database into service as a standby without re-copying the database.

When you fail over to a standby, you shut down the primary, apply all the archived redologs to the standby, then copy all the online logs and the controlfile from the primary to the standby. People who use incremental checkpoints (`FAST_START_MTTR_TARGET` in 9i, `FAST_START_IO_TARGET` in 8i) must do a `'create controlfile reuse database <blah> noresetlogs'` at this point. Other people don't have to.

Finally, you `'recover database'` to get the last one or two online logs and open the standby "noresetlogs." The standby just picks up the chain of SCNs where the primary left off.

The old primary can be immediately pressed into service as a standby. Just generate a standby controlfile on the new primary, copy it into place on the old primary and start it up as a standby database.

You can go back and forth in this way as many times as you want. One database just picks up the chain of SCNs where the last one left off. You never get a divergence of changes between the two.

It is worth noting that in 9i, they have an "automated" graceful failover mechanism for standby databases. I have not yet tested this feature to determine if it is any different from the method described here.

GENERAL PRACTICES

MISCONCEPTION: LOTS OF EXTENTS ARE BAD

Juan Loaiza's popularization of the uniform extent approach to tablespace management has gone a long way to dispelling the myth that having an object composed of a large number of extents causes "performance problems." Nevertheless, the myth persists enough that I constantly encounter DBAs who believe that one of their responsibilities is to constantly re-import into a single extent any tables that have exceeded some number of extents.

To some degree, the blame for this misconception must be laid at the feet of the Export utility. The notorious and misleadingly-named `COMPRESS=Y` option exists only to mislead DBAs into believing that there might actually be some valid reason to force an object to occupy only one extent.

It must be noted that by "reorganizing" and importing/exporting to reduce extents, DBAs often inadvertently resolve problems unrelated to the number of extents. Tables that have had `PCTFREE` and `PCTUSED` incorrectly set may have block-level space issues such as data sparseness and chained rows through reorganizing or rebuilding the table. Similarly, indexes rebuilt to "reduce extents" will experience improved performance, unrelated to the number of

extents, if they were very unbalanced.

The Great Extents Debate usually boils down to the following common arguments in favor of fewer extents.

- Having larger extents allows sequential disk reads, and therefore improves read performance.
- Having many extents constitutes “fragmentation,” which is a Bad Thing™.
- Having many extents makes it take a long time to truncate or drop an object.

Because index lookups use indexes to quickly find rows, the “sequential read” argument is usually asserted in association with table scan performance. The first thing to remember about table scans is that they cause blocks to be read from disk in chunks of a size specified by `db_file_multiblock_read_count`. In order for a large number of extents to result in more disk reads than necessary, it would have to be a likely scenario for a typical multi-block read to have to cross an extent boundary, and therefore result in two reads where only one should be necessary. Since the upper limit of a read on most operating systems is 256Kb, the segment in question would have to be composed of extents smaller than 256Kb, which is highly improbable.

The common assertion that “head movement” negatively impacts read performance in a many-extents situation is also false. This argument holds that if your extents are in all different parts of a disk, then the disk head will have to “skip around” to find the data you need, whereas if everything is in one extent, then the head can stay relatively still on the platter, resulting in less time lost to seeks. This theory might be more compelling if it were indeed the case that every user in the database is issued his own disk head. Unfortunately, in a multi-user system, many users share a given storage resource, and therefore disk heads will always move all over the platter to fulfill various I/O requests from the many processes on the host machine. This theory also falls completely apart when one notes that some or many of the blocks needed for a read may be fulfilled from the buffer cache, thus needing no disk read, and causing the disk head to have to seek to the part of the disk where the next block is located. Also, any major enterprise implementation of Oracle uses a RAID array with a caching controller, which will also fulfill many of the requests without requiring a physical disk read. Finally, in a RAID configuration, data is simultaneously read from stripes on many disks, and it doesn’t matter where in a tablespace the block you need is located.

The misconception that one should avoid large numbers of extents goes hand in hand with the borderline obsession with the detection and avoidance of “fragmentation.” If one were to judge from the newsgroups and listserves, one might suspect that most of Oracle performance tuning consists of eliminating something called “fragmentation.” Having lots of extents is not fragmentation. Having the extents for a particular object located in many different parts of a tablespace is not fragmentation. Tablespace fragmentation is the condition of having many differently-sized used and free spaces in a tablespace, resulting in the inability of existing segments to extend into any of the free space chunks available. The only detrimental effect of tablespace fragmentation is wasted space, and the whole thing can be avoided by standardizing on a uniform extent size within a tablespace.

Uniform extent sizing can be enforced, so that nobody ever accidentally creates segments with nonstandard-size extents. When creating traditional dictionary-managed tablespaces, the `minimum extent` parameter forces all segments to allocate extents that are equal to, or a multiple of, the value for `minimum extent`. Note that this is not the same thing as the `minextents` storage attribute, but is called outside the storage clause. With locally managed tablespaces, uniform extent management can be specified using the `extent management` parameter.

The one possibly detrimental consequence of having large numbers of extents for one segment is the time required to drop or truncate it in a dictionary-managed tablespace. Oracle’s method of deallocating large numbers of dictionary-managed extents is extremely inefficient. In the data dictionary, free space extents are stored in the `sys.fets` table, and used (occupied) extents are stored in the `sys.uset` table. When you drop or truncate a segment, the entries from `uset` have to be deleted and inserted into `fets`. It seems like this should be a fast and easy operation, even for several tens of thousands of extents. After all, we all know it is possible to delete and insert a few thousand rows in Oracle quickly and without any problems. Alas, Oracle’s routine for deallocating extents is not that simple, and as a result is very slow. If you observe the wait events of a session trying to deallocate many thousands of extents, you will see it spending a great deal of time waiting on IPC with the database writer. This is because for each extent deallocated, the session waits for the database writer to write out the blocks from the object before proceeding. In certain systems,

especially if the database writer is busy, deallocation of several tens of thousands of extents can take hours. For the duration of the extent deallocation, the session performing the DDL holds the `st` enqueue, which prevents any other sessions from allocating or deallocating extents, including sessions that just want to create a sort segment to do a disk sort.

The extent deallocation problem is a very good reason for using locally managed tablespaces.

MISCONCEPTION: LISTENERS MUST BE STARTED BEFORE THE INSTANCE

The idea that the instance and the listeners must be started in a certain order probably stems from the way that PMON registers MTS dispatchers with listeners. When an instance is started, at the time that the database is opened, PMON attempts to register the MTS dispatchers with any listener on the localhost's port 1521, and/or any listeners specified in the MTS initialization parameters. If a listener is not started, then there is nothing for PMON to register the dispatchers with.

The common misconception is that the only time dispatcher registration can occur is at database open time. In fact, after the database is open, PMON attempts to register dispatchers, or perform a "service update," periodically. This means that if the listener is started after the instance, PMON will soon perform a service update and registration will be successful. This behavior can be observed by looking for the "service update" string in the listener logs. These messages will appear every minute or so when PMON is idle, and less frequently when PMON is busy.

In many environments, DBAs prefer to start listeners after the database instance because their applications bombard the listeners with large numbers of connections, and compete for system resources during database startups. Unfortunately, if such systems rely on MTS to prevent excessive resource consumption during normal operation, they become flooded with dedicated server connection until PMON can get around to performing dispatcher registration.

There are several satisfactory solutions to this problem. Net8 allows the use of a `SERVER=SHARED` attribute in the client's `tnsnames.ora`, which will force connections to establish only if an MTS service handler is available on the listener. With this parameter enabled, the listener will not attempt to establish dedicated server processes for all these incoming connections. From the point of view of the client applications, the database service becomes available only when the dispatchers have been registered with the listeners.

An alternative to using `SERVER=SHARED` is to start the listeners when the database is in mount mode. If a startup script first starts the database up `MOUNT`, then starts the listeners, then opens the database, the instance startup won't have to compete with the listeners for resources as large numbers of shared servers and job queue processes are being forked.

Finally, a new feature introduced in Oracle9i allows on-demand registration of the dispatchers with the listener. This also provides better support for brief listener restarts during normal operations. Because a listener reload is insufficient to make a variety of changes, it is sometimes necessary to fully restart the listener. It is highly preferable to be able to make such a change without causing undue interruption in service. With on-demand dispatcher registration, it is no longer necessary to wait around for PMON to register dispatchers after a listener restart. The lack of such a feature in earlier versions of Oracle has resulted in no small amount of frustration while clients fail on connect to a database whose PMON can't seem to get around to doing the next service update.

MISCONCEPTION: COUNT (1) IS BETTER THAN COUNT (*)

The idea that you get faster results by selecting `count(1)` or `count('x')` or some similar literal value rather than `count(*)` has been popular ever since I began working with Oracle in 1994.

The truth is that Oracle8's cost-based optimizer will do an index fast full scan on the first unique index on a not-null column (usually the primary key) if it is available. If no such index is available, a full table scan will be performed. This behavior is the same under the cost-based and rule-based optimizers no matter what you put in the parentheses of the count function.

Under the rule-based optimizer, `count()` always results in a full table scan.

Here's a proof that demonstrates that `count(*)` is the same as `count(<any literal>)`, using logical reads and explain plan, instead of wall clock time. It shows that the various `count()` theories are, in fact, just legend. Your session

performs the same amount of work no matter which method you use.

Take an imaginary table, foo, with 955190 rows, and a primary key. Rows will be counted by sid 90. In a separate session, I record the starting statistics of the session (read from another session, so stats are not affected by the statistics queries):

```
SQL> select sn.name, ss.value from v$session s, v$sesstat ss, v$statname sn where s.sid = ss.sid
and ss.statistic# = sn.statistic# and sn.name = 'session logical reads' and s.sid = 90;
NAME                                VALUE
-----
session logical reads                130054
```

First, count the usual way:

```
SQL> select count (*) from foo;
COUNT(*)
-----
  955190

select statement [cost = 303, cardinality = 1.00, bytes = 0]
 2  2.1 sort aggregate
 3  3.1 index fast full scan pk_f_foo_id unique [cost = 303, cardinality = 952723.00, bytes =
0]
```

Querying v\$sesstat, I see that the session logical reads have increased to 132006 (an increase of 1952 logical reads).

Then, I count rowids:

```
SQL> select count (rowid) from foo;
COUNT(ROWID)
-----
  955190

select statement [cost = 303, cardinality = 1.00, bytes = 7]
 2  2.1 sort aggregate
 3  3.1 index fast full scan pk_f_foo_id unique [cost = 303, cardinality = 952723.00, bytes =
6669061]
```

The session logical reads increase to 133958 (an increase of 1952 logical reads).

Finally, I count the PK column:

```
SQL> select count (foo_id) from foo;
COUNT(FOO_ID)
-----
  955190

select statement [cost = 303, cardinality = 1.00, bytes = 0]
 2  2.1 sort aggregate
 3  3.1 index fast full scan pk_f_foo_id unique [cost = 303, cardinality = 952723.00, bytes =
0]
```

The session logical reads increase to 135910 (an increase of 1952 logical reads).

So in each case, the number of logical reads and the explain plans are identical. This proof shows that the three methods of counting are identical.