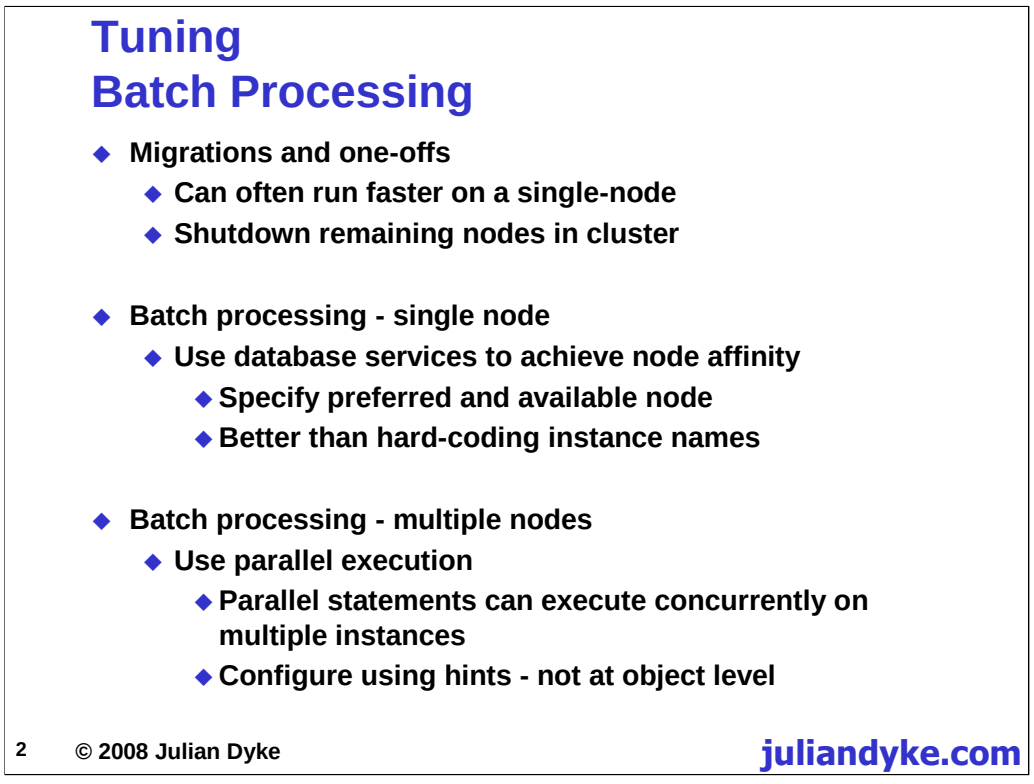
Session 8

Performance Tuning

Advanced RAC
Auckland - May 2008

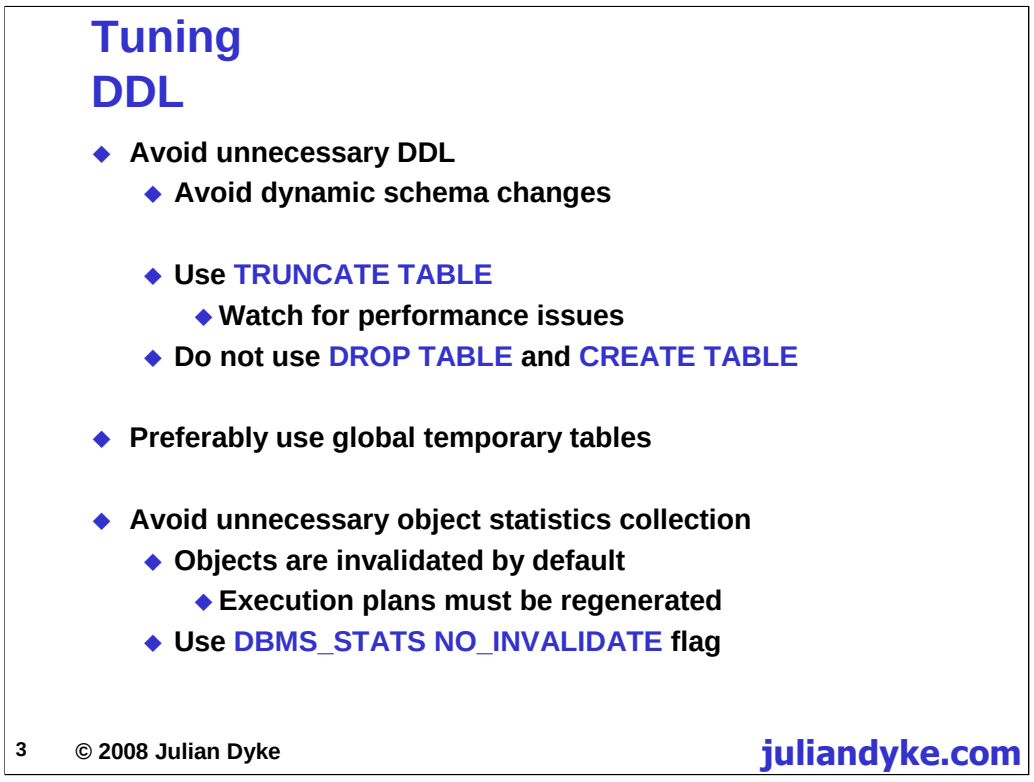


Although RAC can provide good load balancing during normal OLTP operations, the RAC overhead can adversely impact performance during exceptional processing such as data migrations and application upgrades. In these cases where processing is often sequential, it can be more efficient to run the workload on a single node in the cluster and to shutdown the remaining nodes in a cluster. If you are developing a migration procedure I recommend that you test each step of the migration with one node and with multiple nodes to discover the fastest overall time.

Batch processing often has similar characteristics and it can be more efficient to perform this on a single node. For E-Business suite users running on RAC, Oracle often recommends that the Concurrent Manager (batch processor) runs on a separate instance to minimize the impact on other users.

If you need to spread batch processing across multiple nodes, then parallel execution is RAC-aware and can distribute statements, where appropriate, across all available nodes in the cluster.

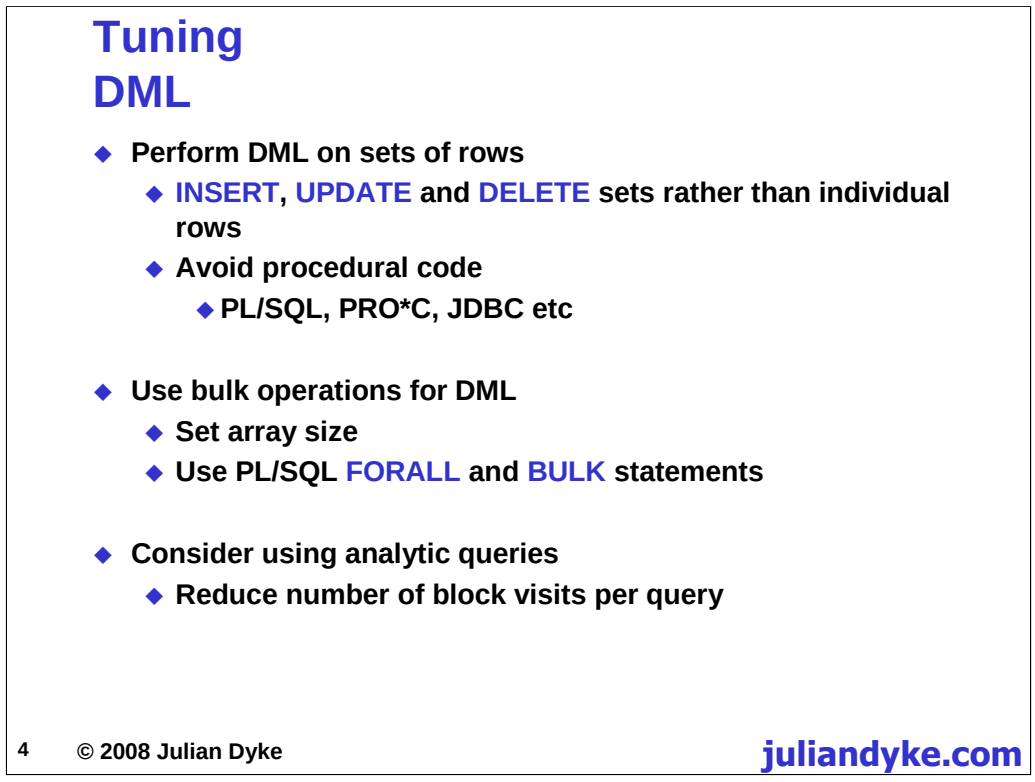
If you wish to use parallel execution, I strongly recommend that you enable it using hints in the SQL statements rather than setting a degree of parallelism for each database object. Although hints are not always regarded as best practise, with parallel execution they allow much granular control of performance.



DDL statements generally require exclusive global locks. These can be expensive to acquire both for the requesting instance and for existing instances holding the locks. It is preferable to avoid unnecessary DDL. For example if you use DDL to create and delete temporary tables for reporting, consider either truncating existing tables using TRUNCATE TABLE or even better using global temporary tables. Global temporary tables are very efficient, but have the disadvantage than they make debugging more complex.

Gathering database, schema or object statistics can also impact performance as by default all affected database objects will be invalidated. New execution plans will need to be generated for all statements referencing the invalidated objects.

Recent versions of DBMS_STATS have included a NO_INVALIDATE flag for GATHER_DATABASE_STATS, GATHER_SCHEMA_STATS and GATHER_TABLES_STATS. If this flag is set then existing execution plans will not be invalidated, but new execution plans will use the new statistics.



There are several standard techniques for improving database performance.

For DML statements it is usually more efficient to execute a single statement against a set of rows than against individual rows. This reduces network round trips for SQL statements and context switches for PL/SQL packages executing SQL statements.

Statements executed against sets of rows are generally more efficient than statements executed procedurally using PL/SQL, Pro*C or JDBC. However it is very easy to make statements over-complicated making them more difficult to maintain and also more prone to errors.

For bulk DML operations ensure that you are setting an array fetch size so that multiple rows can be processed during the same network round trip.

Array sizes can be set for DML and SELECT statements in OCI, JDBC, Pro*C and PL/SQL (Oracle 8.1.5 and above)

Analytic queries can reduce the number of block visits (logical I/Os) made by a query, usually at the cost of increased memory for sorting.

Tuning
Array Fetch

- Use array operations for SELECT statements
  - Reduces number of network packets required to FETCH data
- In SQL*Plus array size defaults to 15
  - To set the array size to 50 use

```
SET ARRAYSIZE 50
```

- In JDBC array size defaults to 10
  - To set the array size for a connection to 20 rows use:

```
((OracleConnection)conn).setDefaultRowPrefetch (20);
```

  - To set the array size for an individual statement to 50 rows use:

```
((OracleStatement)statement).setRowPrefetch (50);
```


By default Oracle fetches 10 rows at a time. This can be increased or decreased either for a connection or for an individual statement

For a connection import the following:

```
import oracle.jdbc.driver.*;
```

To get the default value of the prefetch count use:

```
((OracleConnection)conn).getDefaultRowPrefetch ();
```

The default value is 10

To set the default value of the prefetch count use:

```
((OracleConnection)conn).setDefaultRowPrefetch (<numberOfRows>);
```

e.g.

```
((OracleConnection)conn).setDefaultRowPrefetch (20);
```

For a statement import the following:

```
import oracle.jdbc.driver.*;
```

To get the current value of the prefetch count use:

```
((OracleStatement)statement).getRowPrefetch ();
```

To set the current value of the prefetch count use:

```
((OracleStatement)statement).setRowPrefetch (<numberOfRows>);
```

e.g.

```
((OracleStatement)statement).setRowPrefetch (50);
```

Note that both the mutators throw SQLException.

Tuning
Batch Updates

- Use batch updates for DML statements (INSERT, UPDATE, and DELETE)
- In JDBC, default batch size is 1 row
  - To set the batch size for a connection to 20 rows use:

```
((OracleConnection)conn).setDefaultExecuteBatch (20);
```

  - To set the batch size for an individual statement to 50 rows use:

```
((OraclePreparedStatement)statement).setExecuteBatch (20);
```

  - To send a batch of rows at any time use:

```
((OraclePreparedStatement)statement).sendBatch ();
```


By default Oracle updates 1 row at a time. This can be increased either for a connection or for an individual statement.

For a connection import the following

```
import oracle.jdbc.driver.*;
```

To get the default value of the execute batch for a connection use:

```
((OracleConnection)conn).getDefaultExecuteBatch ();
```

To set the default value of the execute batch for a connection use:

```
((OracleConnection)conn).setDefaultExecuteBatch (<numberOfRows>);
```

e.g.

```
((OracleConnection)conn).setDefaultExecuteBatch (1);
```

For a statement import the following:

```
import oracle.jdbc.driver.*;
```

To get the current value of the batch size for an individual statement use:

```
((OraclePreparedStatement)statement).getExecuteBatch ();
```

To set the current value of the batch size for an individual statement use:

```
((OraclePreparedStatement)statement).setExecuteBatch (<numberOfRows>);
```

e.g.

```
((OraclePreparedStatement)statement).setExecuteBatch (20);
```

Note that both the mutators throw SQLException.

Use the sendBatch () method to send a batch of rows before the array size is reached:

```
((OraclePreparedStatement)statement).sendBatch ();
```

sendBatch returns the number of updated rows

Note also that a Connection.commit(), PreparedStatement.close () and Connection.close () all call sendBatch () to flush the final batch of updates

Tuning
PL/SQL Bulk Collect

```
DECLARE                                          -- 100000 row table
    l_c3 NUMBER;
    CURSOR c1 IS SELECT c3 FROM t1;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO l_c3;                      -- 3.052 seconds
        EXIT WHEN c1%NOTFOUND;
    END LOOP;
    CLOSE c1;
END;
```

```
DECLARE                                          -- 100000 row table
    TYPE NUMTYPE IS TABLE OF NUMBER(6) INDEX BY BINARY_INTEGER;
    l_c3 NUMTYPE;
    CURSOR c1 IS SELECT c3 FROM t1;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 BULK COLLECT INTO l_c3;         -- 0.119 seconds
        EXIT WHEN c1%NOTFOUND;
    END LOOP;
    CLOSE c1;
END;
```

7 © 2008 Julian Dyke juliandyke.com

The first example shows a PL/SQL cursor loop in which each row is fetched individually. On a test system, this loop took over three seconds to execute.

The second example shows a PL/SQL cursor loop which returns the same result set, but which executes in approximately 0.12 seconds. The second loop uses the BULK COLLECT structure which effectively implements an array fetch.

Bulk collect performance improves as optimum result set size is approached

Thereafter bulk collect performance degrades as result set grows

In Oracle 8.1.6 and above the number of rows returned by FETCH INTO can be restricted using the LIMIT clause:

```
FETCH c1 BULK COLLECT INTO l_c3 LIMIT 1000;
```

Tuning
PL/SQL FORALL

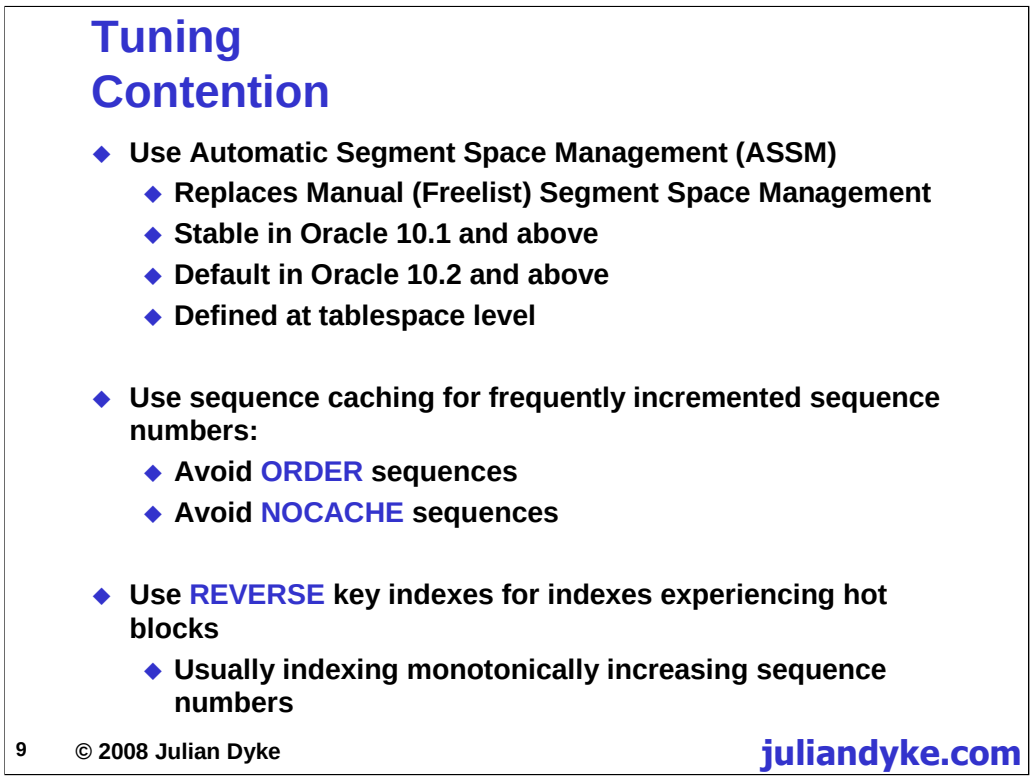
```
DECLARE
    TYPE NUMTYPE IS TABLE OF NUMBER(6) INDEX BY BINARY_INTEGER;
    TYPE NAMETYPE IS TABLE OF VARCHAR2(30) INDEX BY BINARY_INTEGER;
    l_c1 NUMTYPE;
    l_c2 NAMETYPE;
    l_c3 NUMTYPE;

BEGIN
    FOR i IN 1..100000 LOOP
        l_c1(i) := i;
        l_c2(i) := LPAD (TO_CHAR (i),30,'0');
        l_c3(i) := MOD (i, 100);
    END LOOP;

    FOR i IN 1..100000 LOOP            -- FOR Loop – 28 seconds
        INSERT INTO t1 VALUES (l_c1 (i), l_c2 (i), l_c3(i));
    END LOOP;

    FORALL f IN 1..100000 LOOP         -- FORALL Loop – 4 seconds
        INSERT INTO t1 VALUES (l_c1 (i), l_c2 (i), l_c3(i));
END;
```

8 © 2008 Julian Dyke juliandyke.com

The PL/SQL FORALL statement was introduced in Oracle 8.1.5. It sends INSERT, UPDATE or DELETE statements in batches. FORALL has limited functionality in that it can only repeat a single DML statement

It does, however, work with PL/SQL collections including TABLE, VARRAY, NESTED TABLE etc.

FORALL is much faster than equivalent for-loop



There are several techniques to reduce contention in RAC.

Automatic Segment Space Management (ASSM) manages free space in objects using bitmaps rather than chains. Each instance is allocated a different bitmap block from which to manage free space and therefore there is less contention.

ASSM is defined at tablespace level. It was a little unstable in Oracle 9.0.1. However it stabilized in Oracle 10.1 and in Oracle 10.2 and above is the default for newly created tablespaces.

Sequence numbers are a well-known issue with RAC databases. They are only really a problem if they are incremented rapidly. The usual advice is to avoid sequences specifying the ORDER clause or the NOCACHE clause, both of which require a visit to the data dictionary every time a sequence number is generated. If possible use a large cache size though you should remember that sequence numbers will be lost forever when an instance is restarted or if the library cache is so busy that the sequence object is aged out.

Reverse indexes provide options for distributing I/O over a number of blocks where the index has hot blocks. This usually affects indexes with monotonically increasing sequence numbers. Reverse key indexes do not work will for indexes which are accessed using range scans, though they will still work for index range scans on multi-column indexes where equality predicates are used for some or all of the columns on the left-hand side

Tuning
Automatic Undo Management

- Introduced in Oracle 9.0.1
- Stable in Oracle 9.2 and above
- Easier to administer than manual rollback segments
- Cannot be tuned as efficiently as manual rollback segments

- In RAC each instance has a dedicated undo tablespace
- Only the owning instance can write to its undo tablespace
- All instances can read the undo tablespace
  - Required for read-consistency with long-running transactions
  - Required during failover to rollback uncommitted transactions

Although I state in the slide than Automatic Undo Management is stable in Oracle 9.2 and above, we did manage to break it repeatedly in an early version (possibly 9.2.0.3) when stress testing a high volume telecommunications application on a four node RAC cluster. We eliminated the problem by reverting to manual rollback segment management. At the time one or two other Oak Table members had similar experiences. I have not heard of similar problems in later versions of 9.2 or in Oracle 10.1 and above.

Tuning
Locally Managed Tablespaces

- Dictionary Managed Tablespaces
  - Manage extent allocations within data dictionary tables
  - Require ST enqueue for relatively long period
  - Cause contention for the data dictionary
  - Cause heavy interconnect traffic in RAC

- Locally Managed Tablespaces
  - Introduced in Oracle 8.1.5
  - An instant success
  - Manage extent allocations within bitmap in tablespace
  - Uses TT enqueue which is tablespace-specific
  - Lower contention for data dictionary
  - Relatively low interconnect traffic in RAC

Locally managed tablespaces were introduced in Oracle 8.1.5 and were instantly successful. They are the default for new tablespaces in Oracle 9.0.1 and above. LMTs manage extent allocations within bitmaps in the same tablespace. These use the TT enqueue which is tablespace-specific.

Prior to the introduction of locally managed tablespaces, extent allocations were managed using dictionary managed tablespaces. These required the ST enqueue for a relatively long period, thereby serializing access to the data dictionary. On busy systems, this could cause significant contention for the data dictionary. Note that the ST enqueue has disappeared in Oracle 10.1, but you should still try to migrate any legacy dictionary-managed tablespaces to locally managed tablespaces.

Most sites have migrated to locally-managed tablespaces already.

Tuning
Locally Managed Tablespaces

- Locally-managed tablespaces have been the default for new tablespaces since Oracle 9.0.1
  - Use LMTs wherever practical
- Consider upgrading dictionary managed tablespaces using DBMS_SPACE_ADMIN.TABLESPACE_MIGRATE_TO_LOCAL procedure
- System defined extents can lead to fragmentation
  - Sizes are 65536 and 1048576 (Oracle 10.2)
- Specify user-defined (uniform) extent sizes
  - e.g, small (S), medium (M), large (L) and extra large (XL)

In a test on Oracle 10.2.0.1 (Linux 32 bit) using ALTER TABLE t1 ALLOCATE EXTENT in a locally managed tablespace using system allocation I created 16 65536 byte extents and then 32766 1048576 byte extents. The total segment size was 34,358,689,792. Attempting to allocate the next extent resulted in ORA-01653: unable to extend table <table_name> by 128 in tablespace <tablespace_name>

Tuning
Automatic Segment Space Management

- Introduced in Oracle 9.0.1
- Works in Oracle 9.2 with some exceptions
- Stable in Oracle 10.1
- Default in Oracle 10.2

- Uses bitmaps to manage segment space allocation
- Recommended for RAC databases
  - Each instance uses a separate bitmap block
  - Eliminates contention for freelists on extent header

Automatic Segment Space Management was introduced in Oracle 9.0.1. The basic principle is that segment space allocations are stored in bitmaps within the tablespace. However ASSM has been slightly less successful than locally managed tablespaces because:

• The benefits are less obvious

• There were a few compatibility issues with other Oracle features in Oracle 9.0.1 and Oracle 9.2

In Oracle 10.1 and above ASSM seems to work correctly and in Oracle 10.2 and above it is the default for new tablespaces.

ASSM is recommended for RAC databases. Each instance allocates segments from a separate bitmap block, thus minimizing contention for freelists on the extent header.

Tuning Sequences

- Use sequences instead of tables to generate sequential numbers
- For example if the existing code is:

```
SELECT MAX (c1) INTO :b1 FROM t1;
UPDATE t1 SET c1 = c1 + 1;

INSERT INTO t2  VALUES (:b1,....);
```

- Create a sequence

```
CREATE SEQUENCE s1;
```

- The code can then be rewritten as:

```
INSERT INTO t2 (s1.NEXTVAL,....);
```

14 © 2008 Julian Dyke juliandyke.com

Use sequences instead of tables to generate sequential numbers.

For example if the existing code is:

```
SELECT MAX (c1) INTO k1FROM t1;
UPDATE t1 SET c1 = c1 + 1;
INSERT INTO t2 (k1,...);
```

you can create a sequence e.g.

```
CREATE SEQUENCE t1;
```

The code can then be rewritten as:

```
INSERT INTO t2 (s1.NEXTVAL,....);
```

The sequence should be more efficient than the table, because

• the sequence numbers can be cached thus requiring less I/O

• when a new set of sequence numbers is required they are allocated immediately in a recursive transaction. Therefore contention is minimized.

If a sequence is not used then the single row in t1 will be locked until the transaction commits or rolls back.

Tuning Sequences

- By default sequences have a cache size of 20

```
CREATE SEQUENCE s1;
```

- Each instance will allocate a batch of 20 sequence numbers the first time the sequence is accessed after instance startup
- Allocating a new batch of sequence numbers requires the SQ enqueue
- In databases with high insertion rates, it may be necessary to increase the sequence cache size to reduce contention for the SQ enqueue
- For example:

```
CREATE SEQUENCE s1 CACHE 1000;
```


Sequence numbers are discarded

• when the instance is shutdown

• when the shared pool is flushed

Oracle sequences do not guarantee contiguous numbers will be allocated. In addition, RAC instances will not even necessarily allocate numbers sequentially. For example, if instance 1 caches sequence numbers 21-40 and instance 2 caches sequence numbers 41-60, it is entirely possible that instance 2 will use all of its cached sequence numbers before any are allocated by instance 1.

Tuning
Reverse Key Indexes

- Introduced in Oracle 8.0
- Designed to reduce contention for index insertions
- Column order is preserved
- Contents of each column are reversed
- Useful in RAC to reduce inter-instance contention
  - However, no node affinity unless leading column(s) are node specific
- Only useful where rows are accessed using equality predicates

Reverse key index were introduced in Oracle 8.0. and are designed to reduce contention for index insertions, particularly for time-sequenced or numerically sequenced indexes.

In a reverse key index the column order is preserved, but the contents of each column are reversed. Reverse key indexes are only really useful for equality predicates, though there is some scope for using index range scans if the left hand column(s) in the key are known. For example you might have a reverse key index on ORDER_NUMBER, ITEM_NUMBER. If you have millions of orders each of which only includes a handful of items, then a reverse key index might be sufficient to identify all items for a specific order.

Reverse key indexes are useful in RAC to reduce inter-instance contention. However, unless the leading column(s) are node specific, reverse key indexes cannot have node affinity.

Tuning
Reverse Key Indexes

- Consider the following

```
CREATE TABLE t1 (firstname VARCHAR2(30), surname VARCHAR2(30);

INSERT INTO t1 VALUES ('Fernando','Alonso');
INSERT INTO t1 VALUES ('Felipe','Massa');
INSERT INTO t1 VALUES ('Kimi','Raikkonen');
INSERT INTO t1 VALUES ('Lewis','Hamilton');
INSERT INTO t1 VALUES ('Mark','Webber');

CREATE INDEX i1 ON t1 (firstname,surname) REVERSE;
```

- The names are stored in the index as below:

FIRSTNAME	SURNAME
epileF	assaM
imiK	nenokkiaR
kraM	rebbeW
odnanreF	osnolA
siweL	notilmaH

To illustrate how reverse key indexes are stored, consider the example in the slide.

The reverse key index i1 contains two columns, first name and surname. The data is stored in reverse order for each column, so Fernando Alonso is stored as odnanreF osnolA.

Tuning
Missing Indexes

- In Oracle 10.1 and above, DBMS_ADVISOR can be used to identify missing indexes
- DBMS_ADVISOR can be used with
  - a representative workload
  - a single statement
- DBMS_ADVISOR can recommend
  - indexes
  - materialized views

In Oracle 10.1 and above, the DBMS_ADVISOR package can be used to identify missing indexes. This package is the basis of the SQL Access Advisor which can also be executed within Enterprise Manager.

DBMS_ADVISOR can be used with either a representative workload or a single statement. Although the example on the following slides uses a single statement, this is simply for the sake of brevity. In reality you should always use a representative workload when using this tool. In addition, as the technology is still in its infancy, you should manually check any recommendations generated by the tool.

Tuning
Missing Indexes

- To use the quick tune feature of the DBMS_ADVISOR, first create a directory for the results:

```
CREATE OR REPLACE DIRECTORY advisor
AS '/u01/app/oracle/advisor';

GRANT READ,WRITE ON DIRECTORY advisor TO PUBLIC;
```

- Execute the advisor against the target SQL statement:

```
EXECUTE dbms_advisor.quick_tune ( -
     DBMS_ADVISOR.SQLACCESS_ADVISOR, -
     task_name => 'TASK1', -
     attr1 => 'SELECT c2 FROM t1 WHERE c1 = :b1');
```

- Generate the advice script

```
EXECUTE dbms_advisor.create_file ( -
     buffer => dbms_advisor.get_task_script (task_name => 'TASK1'),
     location => 'ADVISOR', -
     filename => 'task1.sql');
```

19 © 2008 Julian Dyke juliandyke.com

The QUICK_TUNE procedure actually performs all the steps that are performed by the full advisor, but requires much less user input.

Before generating any scripts, it is necessary to create an Oracle directory for the output files.

The QUICK_TUNE procedure takes the statement as one of its parameters. The results of the analysis are stored in:

USER_ADVISOR_FINDINGS
USER_ADVISOR_RECOMMENDATIONS
USER_ADVISOR_ACTIONS

Of these tables, USER_ADVISOR_ACTIONS is the most informative. You can save these actions to a DDL script using the CREATE_FILE procedure which in turn calls GET_TASK_SCRIPT to generate a CLOB containing the index creation statements.

Tuning
Missing Indexes

- The advice script contains DDL statements to create the missing indexes
- For example:

```
Rem  SQL Access Advisor: Version 10.2.0.1.0 - Production
Rem
Rem  Username:       US01
Rem  Task:           TASK1
Rem  Execution date:  07/01/2007 23:07
Rem

CREATE INDEX "US01"."T1_IDX$$_15180001"
    ON "US01"."T1"
    ("C1")
    COMPUTE STATISTICS;
```


You can modify the index creation script. For example, you might wish to change the index name to comply with local standards. Alternatively if this is a very large table, you may wish to estimate statistics.

At the time of writing there is very little research in the public domain about the accuracy of the DBMS_ADVISOR package. However, when used with the full workload, it does provide a useful sanity check if you are investigating missing indexes.

Tuning
Unused Indexes

- In Oracle 9.0.1 and above, index monitoring can be used to determine which indexes are currently in use
- To enable index monitoring on index1 use

```
ALTER INDEX index1 MONITORING USAGE;
```

- To check which indexes are currently being monitored use:

```
SELECT index_name,monitoring
FROM v$object_usage;
```

- To check which indexes have been used by the optimizer use:

```
SELECT index_name, used
FROM v$object_usage;
```

- To disable index monitoring on index1 use:

```
ALTER INDEX index1 NOMONITORING USAGE;
```

21 © 2008 Julian Dyke juliandyke.com

In Oracle 9.0.1 indexes can be monitored to determine if they are being used

If index monitoring is enabled for an index, then Oracle updates a table in the data dictionary when that index is included in an execution plan by the parser. Indexes are only monitored at parse time Only SELECT statements and sub-queries are monitored. You need to take care if the index is used to support primary or unique constraints.

Index monitoring supports both B*tree and bitmap indexes

Every time an index is used, the SYS.OBJECT_USAGE table is updated. If your database is performing frequent hard parses, then this will have an impact on performance. If you decide to enable index monitoring for all indexes, make sure you disable it again immediately when you detect that a commonly used index has been detected.

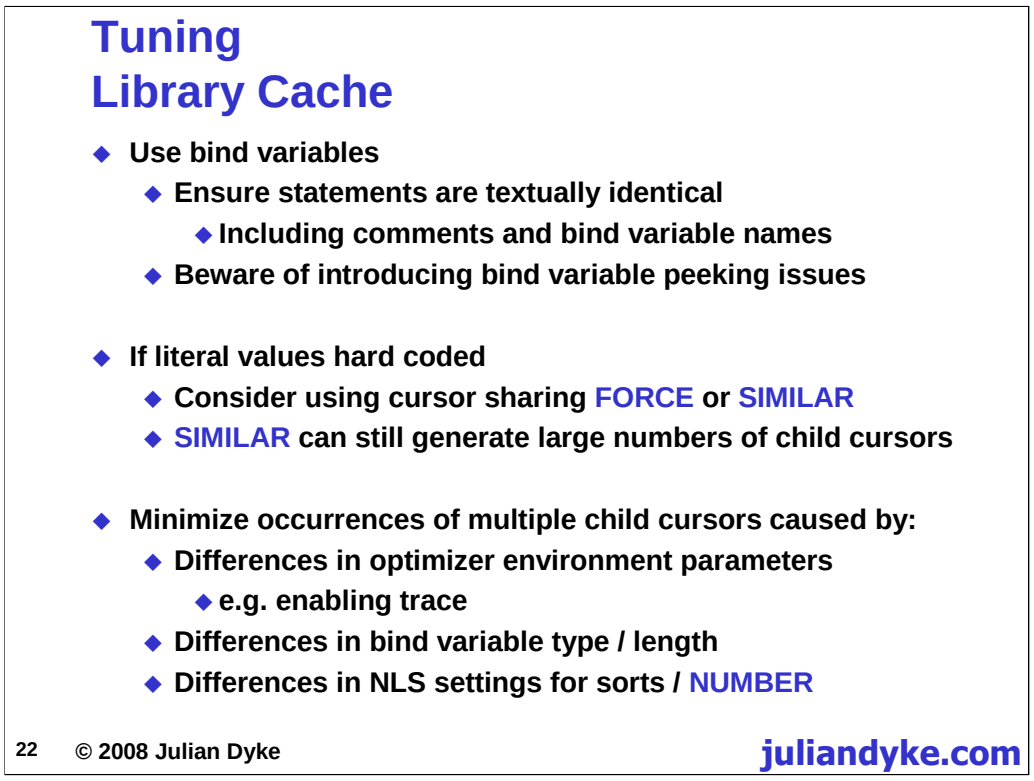
I normally create a table containing all index names and then eliminate all indexes in which I am not interested including

• indexes owned by SYS, SYSTEM, SYSMAN etc

• indexes supporting PRIMARY and UNIQUE constraints

• indexes currently used in execution plans (reported in V$SQL_PLAN)

I then use SQL*Plus to generate the ALTER INDEX MONITORING USAGE statements for the remaining indexes.

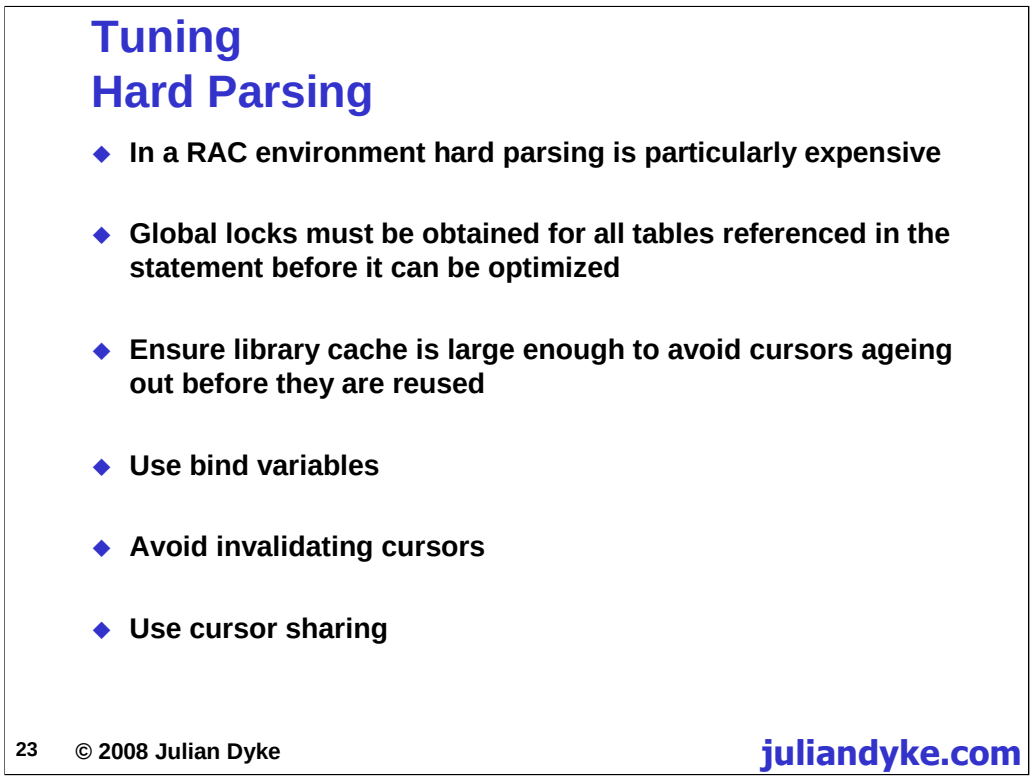


If you still have statements using literal values, consider using bind variables instead. This reduces the amount of hard parsing required which is an even more costly operation in a RAC environment.

Statements must be textually identical; this includes any comments and also the names given to bind variables. In Oracle 9.0.1 and above, introducing bind variables can also introduce issues with bind variable peeking particularly with objects with skewed data using histograms. This can result in some very inefficient execution plans.

If you do not have source code access, then consider using CURSOR_SHARING where Oracle will convert all literal values to bind variables as it executes statements. CURSOR_SHARING can be EXACT (the default), FORCE or SIMILAR. FORCE converts all statements; SIMILAR only converts statements that Oracle considers to be safe. SIMILAR is usually the best option, but sometimes has little positive impact.

Multiple versions of child cursors for the same parent cursor (SQL statement) can consume large amounts of SGA memory. Multiple child cursors can be caused by differences in optimizer environment parameters, differences in bind variable types/length, differences in NLS settings for sorts or NUMBERs in particular, or a large number of other reasons. If possible the cause should be addressed so that each parent cursor only has one or at least < 10 child cursors.



In a RAC environment hard parsing is particularly expensive. Global locks must be obtained for all tables referenced in the statement before it can be optimized

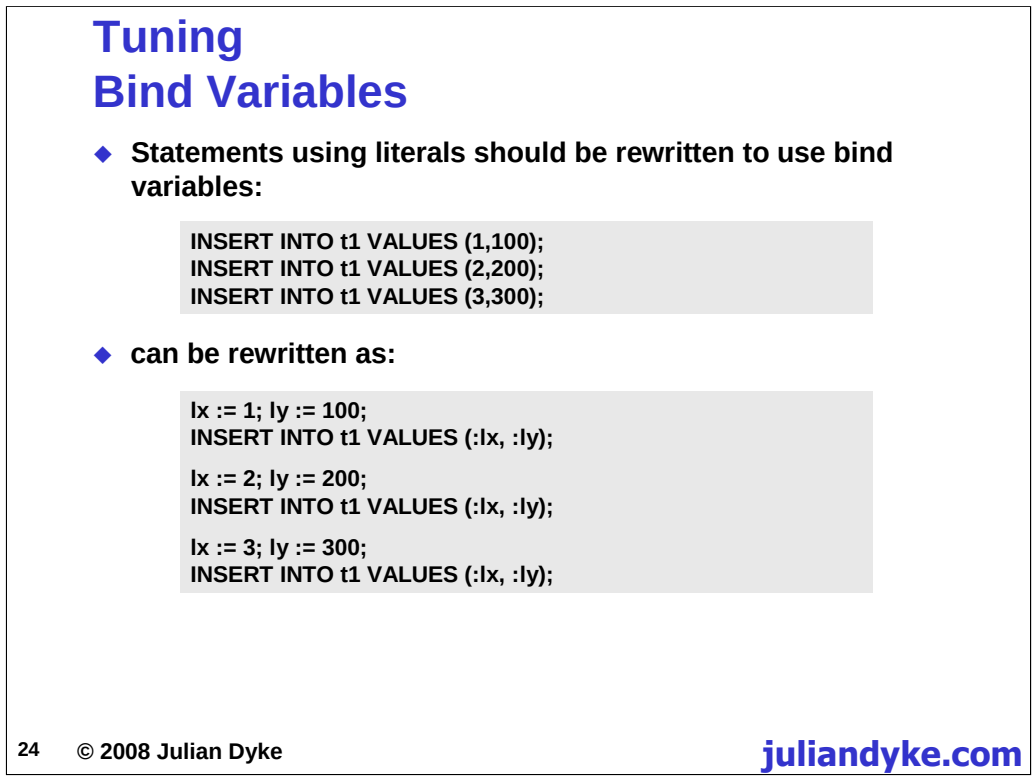
Use the following techniques to minimize hard parsing:

• Ensure library cache is large enough to avoid cursors ageing out before they are reused

• Use bind variables

• Avoid invalidating cursors

• Use cursor sharing



Bind variables allow statements to be reused. Wherever possible statements should be rewritten to use bind variables.

For example the INSERT statements:

```
INSERT INTO t1 VALUES (1,100);
INSERT INTO t1 VALUES (2,200);
INSERT INTO t1 VALUES (3,300);
```

can be rewritten to use bind variables as follows:

```
lx := 1; ly := 100;
INSERT INTO t1 VALUES (:lx, :ly);
lx := 2; ly := 200;
INSERT INTO t1 VALUES (:lx, :ly);
lx := 3; ly := 300;
INSERT INTO t1 VALUES (:lx, :ly);
```

In this example, this change reduces the number of pairs of parent and child cursors required to execute the statements from three to one.

Tuning CURSOR_SHARING parameter

- For applications which do not use binds consider using cursor sharing
- Introduced in Oracle 8.1.6
  - Stable in Oracle 8.1.7 and above
- Values are
  - EXACT - Default - no cursor sharing
  - FORCE - 8.1.6 and above - replace all literal values with pseudo-bind variables
  - SIMILAR - 9.0.1 and above - replace "safe" literal values with pseudo-bind variables

In theory you should never use this parameter as the application should be modified to use bind variables. In practise this may not be possible for a variety of reasons including:

• cost and/or risk of modifying application

• technical difficulty of modifying application

• no access to source code

• application unsupported

• application approaching end-of-life

There is some debate about the usefulness of the SIMILAR option. However, I have proved that this option can reduce the amount of SGA memory required for a statement, even if the amount of parsing is unaffected because the statement includes unsafe literal values.

Tuning
Soft Parsing

- Soft parsing occurs
  - when parent cursor is already in the library cache
  - the first time a session accesses the parent cursor
- Need to check existing cursor is valid for new session
  - parameters
  - object
  - access privileges (security)
- Soft parsing can be reduced
  - Not closing cursors in application
  - Using Session Cursor Cache
  - Using CURSOR_SPACE_FOR_TIME parameter

Soft parsing occurs when the parent cursor is already in the library cache, but it is the first time a session has accessed that parent cursor. Soft parsing also occurs if the session has access the parent cursor before, but has subsequently closed the cursor.

Soft parsing is required to check that the existing child cursor is valid for the new session. This involves checking the parameters, objects and access privileges in each existing cursor.

The amount of soft parsing can be reduced by not closing cursors in the application. You can also use the session cursor cache or set the CURSOR_SPACE_FOR_TIME parameter.

Tuning SESSION_CACHED_CURSORS parameter

- Session cursor cache allows each session to pin multiple statements cached in SGA
  - Reduces amount of hard parsing
- Use parameter SESSION_CACHED_CURSORS to specify number of cached cursors to be pinned
- Default value is 20 in Oracle 10.2.0.1
- Requires small amount of space in PGA
  - Pins cursor space in SGA heaps
  - Can result in ORA-04031 if set too high

The session cursor cache works in the parse..open..execute..close cursor handling model.

The SESSION_CACHED_CURSORS parameter keeps closed cursor address information in session memory, if the statement is parsed three or more times by any session. It also pins the handles (in the SGA) and heap 0 for lookups.

The session cursor cache can help relieve concurrency pressure. A hit in the session cache reduces the number of library cache latch gets.

However, the effectiveness of the session cursor cache is limited if there is space pressure in which case big heaps can age out and be reloaded.

Tuning CURSOR_SPACE_FOR_TIME parameter

- Keeps open cursors pinned between executions
  - Can relieve concurrency pressure
- Increases space required to hold cursors
- Boolean parameter which defaults to FALSE
- Not necessary in Oracle 10.2.0.2 and above
  - Mutexes have replaced library cache latches and pins for relevant cursor operations

The CURSOR_SPACE_FOR_TIME parameter keeps open cursors pinned between executions. It can be used to relieve concurrency pressure in Oracle 10.2.01 and below.

CURSOR_SPACE_FOR_TIME increases the amount of memory required to hold cursors in the SGA, but reduces the amount of parsing. It is a Boolean parameter which defaults to FALSE.

The CURSOR_SPACE_FOR_TIME parameter also works for the parse..open..execute..close cursor handling model.

PL/SQL cursors will remain pinned. Soft closing PL/SQL cursors keeps the cursor open.

Setting CURSOR_SPACE_FOR_TIME is not necessary in Oracle 10.2.0.2 and above where mutexes have replaced library cache latches and pins for relevant cursor operations

Tuning
Row Level Locking

- Avoid pessimistic locking using SELECT FOR UPDATE
  - Setting row lock generates additional undo/redo
  - Requires rollback if transaction uncommitted

- SELECT FOR UPDATE changes cannot be batched
  - Individual undo / redo generated for each row

- If possible lock data when it is updated
  - May require additional error handling

Rows subject to update can be locked pessimistically or optimistically. Pessimistic locking usually involves using the SELECT FOR UPDATE statement which acquires a row lock until the transaction commits or rolls back. SELECT FOR UPDATE is a fairly expensive operation. Row locks are set in the row header on the data block in which the row is stored. Setting the row lock therefore involves generating undo and redo. Rolling back a row lock also generates undo and redo. If the transaction is committed then the row lock will normally be unset during a block cleanout operation by the next session reading the lock.

SELECT FOR UPDATE generates a lot of undo and redo, mainly because separate changes must be generated for each affected row; there is no concept of an array processing of undo/redo for SELECT FOR UPDATE.

If possible, modify the application to use optimistic locking in which the locks will be acquired by the UPDATE statement. This is more efficient in terms of undo/redo generation though it may lead to a less satisfactory user experience if there are a large number of collisions. It may also require additional error handling in the application.

Tuning
Avoid Pessimistic Locking

- Pessimistic locking is usually implemented using SELECT FOR UPDATE statements
- For example:

```
CURSOR c1 IS SELECT x,y FROM t1;

LOOP
    FETCH FROM c1 INTO lx, ly;
    SELECT y,z FOR UPDATE FROM t2 WHERE x = lx;
    UPDATE t2 SET y = ly WHERE x = lx;
    ....
END LOOP;
```

- Might be rewritten without the SELECT FOR UPDATE statement

```
CURSOR c1 IS SELECT x,y FROM t1;

LOOP
    FETCH FROM c1 INTO lx, ly;
    UPDATE t2 SET y = ly WHERE x = lx;
    ....
END LOOP;
```

30 © 2008 Julian Dyke juliandyke.com

Pessimistic locking is usually implemented using SELECT FOR UPDATE statements

SELECT FOR UPDATE statements are fairly expensive in Oracle. A SELECT FOR UPDATE places a lock on the row being selected. The lock is stored in the row header and therefore a SELECT FOR UPDATE statement generates both undo and redo. You may also experience contention if another session needs to perform a SELECT FOR UPDATE on the same row.

Try to eliminate unnecessary SELECT FOR UPDATE statements. Sometimes it is better to allow an UPDATE statement to wait when it attempts to update a row that is currently held by another transaction than to impose the overhead of using SELECT FOR UPDATE statements throughout the application.

Tuning
Avoid DDL Statements

- **Avoid unnecessary DDL Statements**
- **Temporary tables are often created and dropped during report generation**
- **For example:**

```
LOOP
    ...
    CREATE TABLE t1 (...);
     ...
    DROP TABLE t1;
    ...
END LOOP;
```

- **Use global temporary tables or TRUNCATE statement**

31 © 2008 Julian Dyke juliandyke.com

Avoid unnecessary DDL statements. DDL statements are very expensive in a RAC environment as locks must be obtained on all nodes in the cluster.

If your application creates or drops tables dynamically then consider using global temporary tables instead. This reduces the impact on the data dictionary.

For example if you have code such as:

```
LOOP
    ...
    CREATE TABLE t1 (...);
        ...
    DROP TABLE t1;
    ...
END LOOP;
```

consider using a global temporary table or the TRUNCATE statement to eliminate the CREATE TABLE and DROP TABLE statements.

Tuning
Read Consistency

- Avoid reading recently updated blocks on other instances
  - Use Database services to ensure node affinity
  - Minimize transaction lengths
    - Maximize COMMIT frequency
  - Minimize undo generation
    - Drop unused indexes
    - Compress data columns
      - e.g. use D or E instead of DISABLED or ENABLED
    - Use nullable columns
    - Use nullable indexes e.g. NULL / Y
    - Avoid updating columns where data has not changed
      - Before and after values are saved even for unchanged data

In order to minimize the impact of RAC on consistent reads avoid reading recently updated blocks on other instances. This is particularly important with uncommitted transactions.

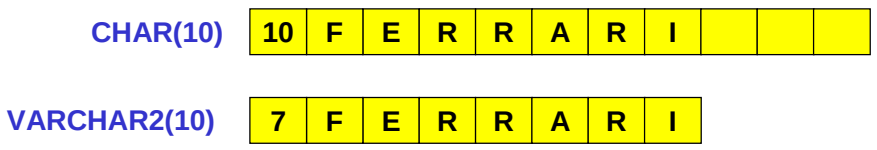
Database services can sometimes be used to ensure node affinity at segment or block level. However some applications cannot avoid reading recently updated blocks. In these cases attempt to minimize transaction lengths; issuing more commits may be preferable to reconstructing read-consistent images of uncommitted blocks for other instances.

Undo generation should also be minimized to reduce the number of blocks that need to be shipped to other instances. There are a number of ways to achieve this, for example, dropping any unused indexes, logically compressing data in columns, using nullable columns where possible and avoiding issuing updates on columns that have not changed.

Conditional indexes (such as marking a row for printing) are best implemented using a combination of NULL values (for the majority of rows) and non-NULL values for the remaining rows. NULL rows are not stored in an index, so this can reduce the size of the index and the amount of undo/redo generated when maintaining the row.

Tuning
CHAR columns

- Avoid CHAR columns where the column length > 1
- Use VARCHAR2 columns instead
- CHAR columns
  - require more disk space
  - take more space in buffer cache
- For example, consider the amount of space required to store 'FERRARI':



- CHAR columns are space-padded to their maximum length
- VARCHAR2 columns are not space-padded

Avoid using the CHAR datatype for columns with a length greater than 1. Use the VARCHAR2 datatype instead.

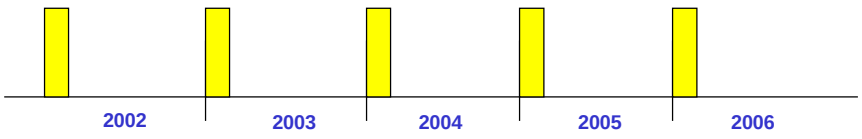
CHAR columns require more disk space and occupy more space in the buffer cache. This is because CHAR columns are space-padded to their maximum length. On the other hand, VARCHAR2 columns only occupy the space required to store their contents.

The example on the slide shows the difference between a CHAR(10) column and a VARCHAR2(10) column. Both have a single byte which stores the length of the data. To store the word "FERRARI", the VARCHAR2 column requires a further 7 bytes. However, the CHAR column requires a further 10 bytes as the data is space padded up to the length of the column.

Tuning
Do not store dates as VARCHAR2

- For example if a table contains 100 values for each day over the past 5 years
- Using a DATE column data is evenly distributed:

- Using a VARCHAR2 column data is bunched:



- Affects CBO estimates for range scans

34 © 2008 Julian Dyke juliandyke.com

Dates are often stored in VARCHAR2 columns. For example '20061201'

This requires more space than DATE column. For example a storing the date 01-DEC-2006 in a DATE column will require 8 bytes (7 bytes for the value plus one for the length byte) . However, storing the same date in the format 20061201 as a VARCHAR2 will require 9 bytes (including a length byte).

Storing dates as VARCHAR2 also affects Cost-Based Optimizer as it limits the information available to the CBO.

Tuning
NULL values

- Use NULL to represent unknown values
- NULL takes a maximum of one byte in row
- Trailing NULLs do not take any space in row
- NULL values are not indexed

- Do not use artificial NULL values such as
  - Space character (0x20) for VARCHAR2
  - -1 for NUMBER
  - 01-JAN-1980 for DATE

- Artificial NULL values
  - Require more disk space
  - Require indexing
  - Skew histograms

Use NULL to represent unknown values. Do not use artificial NULL values such as 0 or -1 for NUMBER data types, space characters for VARCHAR2/CHAR data types or dates such as 01-JAN-1980 or 31-DEC-3999. Artificial NULL values require more disk space, require indexing and can confuse the optimizer because the skew histograms.

NULL values take a maximum of one byte per row. Trailing NULL values in a row do not occupy any space. NULL values are not indexed unless the index contains multiple columns and other indexed column values are not NULL or the index is a bitmap index.

Tuning
Column Values

- Avoid long constant values such as
  - ENABLED and DISABLED
  - TRUE and FALSE
  - YES and NO
- Often found in STATUS columns
- Particularly expensive for very large tables
  - e.g. > 10 million rows
- Consume
  - space on disk
  - space in buffer cache
  - and potentially space in indexes

- Use a numeric constant e.g. 1,2 etc
- If possible use NULL for most popular value

Avoid using long VARCHAR2 values for constants. Examples include:

• ENABLED and DISABLED

• TRUE and FALSE

• YES and NO

Long VARCHAR2 constant values are often found in STATUS columns. They can be particularly expensive if used in very large tables. For example consider a 10 million row table - it would take between 70 and 80 million bytes to represent the values ENABLED and DISABLED on every row. This consumes space on disk, space in the buffer cache and, if indexed, space in the index as well.

Use numeric constants if possible e.g. 1, 2 etc or CHAR constants e.g. 'E', 'D'. If possible use NULL for the most popular value.

Tuning
Column Values

- For example consider a table containing 10 million invoices of which 10,000 are printed each night

```
CREATE TABLE invoice
(
     .....
     print_flag varchar2(1)
     ....
);
CREATE INDEX i_invoice1 ON invoice (print_flag);
```

```
IF print_flag = 'Y' THEN
     -- print invoice
     print_flag := 'N';
END IF;
```

  - If PRINT_FLAG can take the values 'N' or 'Y'
    - I_INVOICE1 will contain 10,000,000 rows
  - If PRINT_FLAG takes the values NULL or 'Y'
    - I_INVOICE1 will only contain 10,000 rows

For example, consider a table containing 10 million invoices of which 10,000 are printed each night.

```
CREATE TABLE invoice
(
            .....
            print_flag varchar2(1)
            ....
);
```

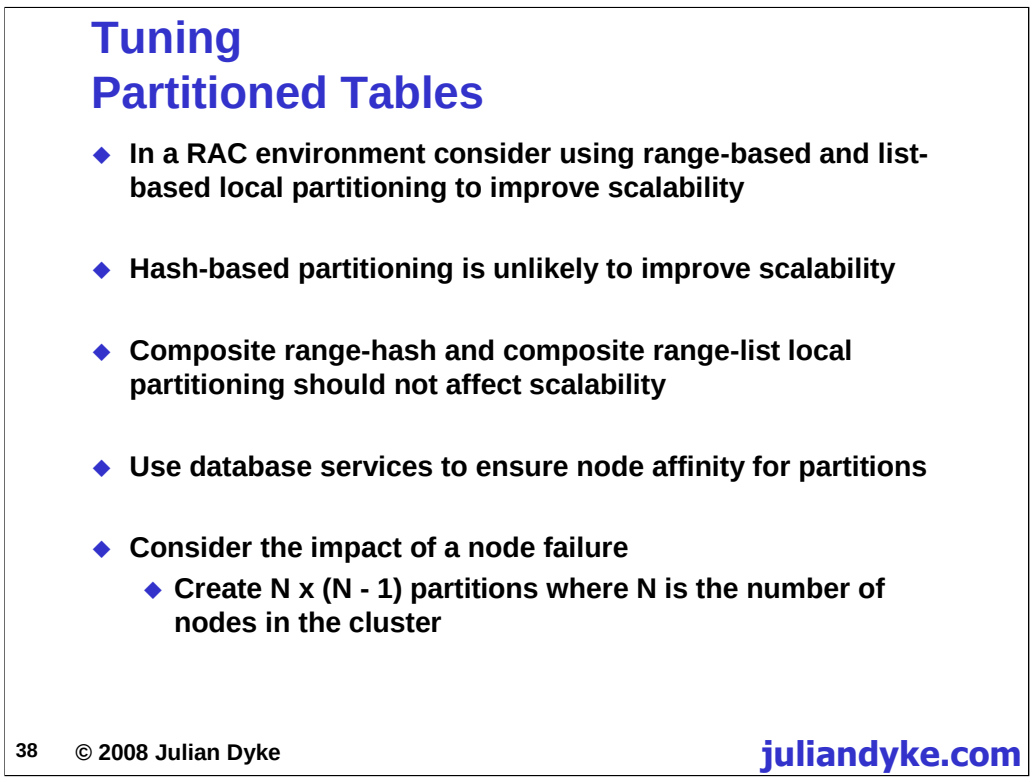
An index has been created on the PRINT_FLAG column of the INVOICE table as follows:

```
CREATE INDEX i_invoice1 ON invoice (print_flag);
```

The following code is executed against each row which needs printing.

```
IF print_flag = 'Y' THEN
            -- print invoice
            print_flag := 'N';
END IF;
```

We are only interested in finding the rows that should be printed tonight. We are not interested in the remaining rows, so there is little point in indexing them. If PRINT_FLAG can take the values 'N' or 'Y' I_INVOICE1 will contain 10,000,000 rows. On the other hand, if PRINT_FLAG takes the values NULL or 'Y' I_INVOICE1 will only contain 10,000 rows



In a RAC environment consider using range-based and list-based local partitioning to improve scalability.

Hash-based partitioning is unlikely to improve scalability as it is difficult to control which hash partitions are used by which index.

Composite range-hash and composite range-list local partitioning can also be used to improve scalability

You can use database services to ensure node affinity for individual partitions

If data is distributed evenly across all partitions (e.g. month of birth, state or country) you may need to consider logical partitioning to optimize performance. In order to determine the number of partitions you might need, consider the impact of a node failure and create N * (N - 1) partitions where N is the number of nodes in the cluster. For example if there are 4 nodes in the cluster then create 4 x (4 - 1) = 4 x 3 = 12 partitions.

Tuning
Partitioned Tables

- For example consider a four node cluster
- Largest table is partitioned on month of birth
- Can use range or list partitioning

```
CREATE TABLE person
(
    month       NUMBER,
    surname     VARCHAR2(30),
    birth_date  DATE,
    .....
)
PARTITION BY LIST (month)
(
    PARTITION jan VALUES (1),
    PARTITION feb VALUES (2),
    PARTITION mar VALUES (3),
    PARTITION apr VALUES (4),
    .....
    PARTITION dec VALUES (12)
);
```

39 © 2008 Julian Dyke juliandyke.com

For example consider a four-node cluster containing a database in which the largest table is partitioned by month of birth. You can use either range or list partitioning.

```
CREATE TABLE person
(
   month    NUMBER,
   surname  VARCHAR2(30),
   birth_date          DATE,
   .....
)
PARTITION BY LIST (month)
(
   PARTITION jan VALUES (1),
   PARTITION feb VALUES (2),
   PARTITION mar VALUES (3),
   PARTITION apr VALUES (4),
   PARTITION may VALUES (5),
   PARTITION jun VALUES (6),
   PARTITION jul VALUES (7),
   PARTITION aug VALUES (8),
   PARTITION sep VALUES (9),
   PARTITION oct VALUES (10),
   PARTITION nov VALUES (11),
   PARTITION dec VALUES (12)
);
```

Tuning
Partitioned Tables

- **Use database services to assign transactions to instances**

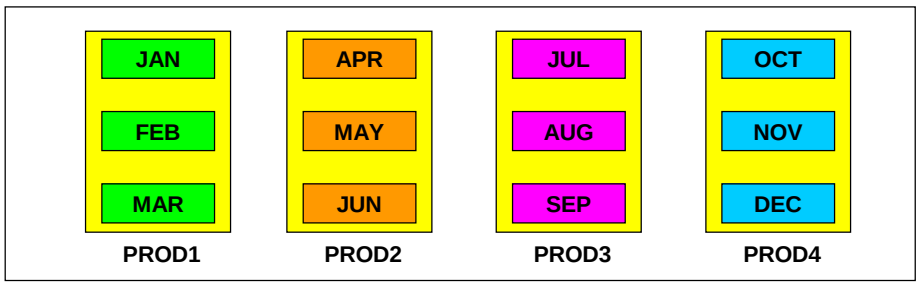
Service Name	Preferred Instance	Available Instance
JAN	PROD1	PROD2
FEB	PROD1	PROD3
MAR	PROD1	PROD4
APR	PROD2	PROD1
MAY	PROD2	PROD3
JUN	PROD2	PROD4
JUL	PROD3	PROD1
AUG	PROD3	PROD2
SEP	PROD3	PROD4
OCT	PROD4	PROD1
NOV	PROD4	PROD2
DEC	PROD4	PROD3

You can use database service to assign transactions to specific instances. Since the service assignment is based on birth date, you must obviously know this information before attempting to connect to the appropriate session. However, this should be relatively straightforward if you are using a connection cache in the application tier.

In the example, imagine an application partitioned on customer date of birth. There are 12 database services corresponding to the 12 months. Each database service has a preferred instance and an available instance. The first three months JAN, FEB and MAR use PROD1 as their preferred instance. However, if PROD1 fails for any reason, each month will failover to a different available instance. So JAN will failover to PROD2, FEB will failover to PROD3 and MAR will failover to PROD4. This ensures that the resulting workload after failover is distributed as equally as possible amongst the remaining instances while at the same time guaranteeing node affinity for individual partitions.

Tuning
Partitioned Tables

- When all four nodes are available the database services and consequently the partitioned tables will be evenly distributed across all nodes



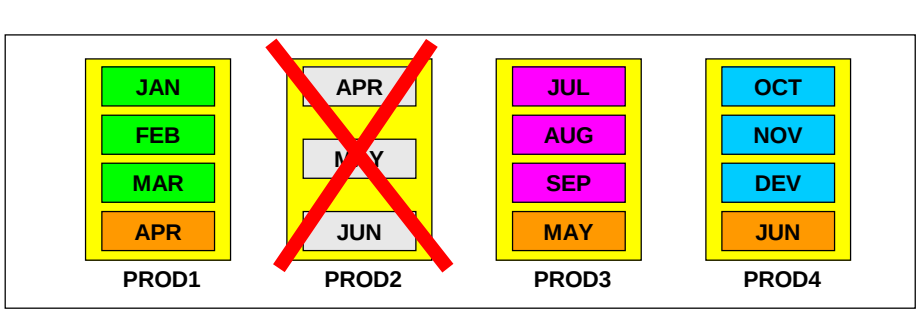
- Partitioned objects have affinity with preferred instance
- Workload is distributed evenly across available nodes

When all four nodes are available the database services and consequently the partitioned tables will be evenly distributed across all nodes

The partitioned objects have affinity with their preferred instances. The workload is distributed evenly across all available nodes.

Tuning
Partitioned Tables

- If PROD2 fails, database services will be failed over to available instances



- Partitioned objects have affinity with available instance
- Workload is still distributed evenly across available nodes

In the event of the failure of the node running instance PROD2, the APR service will be relocated to PROD1, the MAY service will be relocated to PROD3 and the JUN service will be relocated to PROD4.

The partitioned objects still have affinity with the local node and the workload is still distributed evenly across the available nodes.

Tuning
System Statistics

- If your RAC cluster has symmetrical hardware then consider using system statistics
- System statistics are gathered using:

```
EXECUTE DBMS_STATS.GATHER_SYSTEM_STATS;
```

- By default statistics are stored in SYS.AUX_STATS$
- If available system statistics are used by the Cost-Based Optimizer to produce better execution plans
- System statistics are database-specific
  - Do not use in RAC environments with asymmetrical hardware

If your RAC cluster has symmetrical hardware then consider using system statistics

System statistics are gathered using:

```
EXECUTE DBMS_STATS.GATHER_SYSTEM_STATS;
```

By default statistics are stored in SYS.AUX_STATS$

If available system statistics are used by the Cost-Based Optimizer to produce better execution plans

System statistics are database-specific. Therefore you should not use system statistics in RAC environments with asymmetrical hardware or workloads.

Tuning
Avoid Unnecessary Commits

- Avoid unnecessary commits
- For example:

```
LOOP
    ...
    INSERT INTO t1 VALUES (lx,ly,lz);
    COMMIT;
    ...
END LOOP;
```

- The COMMIT can often be moved outside the loop
- For example:

```
LOOP
    ...
    INSERT INTO t1 VALUES (lx,ly,lz);
    ...
END LOOP;
COMMIT;
```

44 © 2008 Julian Dyke juliandyke.com

Avoid unnecessary commits. Commits are expensive for a number of reasons:

• they generate additional undo and redo

• they temporarily lock the undo segment header, potentially causing contention

• they force any redo currently in the log buffer to be written to disk

Check for any unnecessary commits in your application. For example:

```
LOOP
   ...
   INSERT INTO t1 VALUES (lx,ly,lz);
   COMMIT;
   ...
END LOOP;
```

The commit can often be moved outside the loop. For example:

```
LOOP
   ...
   INSERT INTO t1 VALUES (lx,ly,lz);
   ...
END LOOP;
COMMIT;
```

Tuning
Avoid Procedural Code - INSERT

- Avoid code which processes individual rows
- For example:

```
CURSOR c1 IS SELECT x,y,z FROM t1;
LOOP
    FETCH FROM c1 INTO lx, ly, lz;
    ....
    INSERT INTO t2 (x,y,z) VALUES (lx, ly, lz);
    ....
END LOOP;
```

- Might be rewritten as:

```
INSERT INTO t2 (x,y,z) SELECT x,y,z FROM t1;
```

45 © 2008 Julian Dyke juliandyke.com

Avoid code which processes individual rows. For example if you have a cursor loop which inserts one row at a time e.g.:

```
CURSOR c1 IS SELECT x,y,z FROM t1;
LOOP
     FETCH FROM c1 INTO lx, ly, lz;
     ....
     INSERT INTO t2 (x,y,z) VALUES (lx, ly, lz);
     ....
END LOOP;
```

you may be able to rewrite the code as an INSERT..SELECT statement. For example:

```
INSERT INTO t2 (x,y,z) SELECT x,y,z FROM t1;
```

This allows Oracle to use internal array operators to generate the undo and redo operations and is therefore more efficient than the single row version.

Tuning
Avoid Procedural Code - UPDATE

- From the previous example

```
CURSOR c1 IS SELECT x,y FROM t1;

LOOP
    FETCH FROM c1 INTO lx, ly;
    UPDATE t2 SET y = ly WHERE x = lx;
    ....
END LOOP;
```

- Might be rewritten as:

```
UPDATE t2
SET t2.y =
(
  SELECT t1.y
  FROM t1
  WHERE t1.x = t2.x
);
```

46 © 2008 Julian Dyke juliandyke.com

Similarly you can rewrite cursor statements including UPDATE statements to use a subquery instead. For example you can rewrite

```
CURSOR c1 IS SELECT x,y FROM t1;
LOOP
    FETCH FROM c1 INTO lx, ly;
    UPDATE t2 SET y = ly WHERE x = lx;
    ....
END LOOP;
```

to

```
UPDATE t2
SET t2.y =
(
   SELECT t1.y
   FROM t1
   WHERE t1.x = t2.x
);
```

Tuning
Unnecessary Column Updates

- **For example:**

```
SELECT a,b,c,d,e,f,g,h,i,j
INTO la,lb,lc,ld,le,lf,lg,lh,li,lj
FROM t1
WHERE z = :b1
FOR UPDATE;

la = la + 1;

UPDATE t1
SET
     a  = :la;
     b  = :lb;
     c  = :lc;
     d  = :ld;
     e  = :le;
     f  = :lf;
     g  = :lg;
     h  = :lh;
     i  = :li;
     j  = :lj
WHERE z = :b1;
```


Avoid unnecessary column updates.

For example consider the statement UPDATE t1 SET c1 = c1. If table t1 contains a million rows and C1 is NOT NULL column, then Oracle will generate a million undo records containing the original value of C1 and a million redo records containing the undo (the original value of C1) and the redo (the new value of C1).

The slide shows another example. In this case ten columns have been selected from a single row in table t1 for update.

Only one of the rows has been changed, but then all ten columns have been updated. This means that the undo and redo will be generated for all ten columns including the nine values that have not changed.

Tuning
Unnecessary Column Updates

- When a column is updated with an unchanged value:
  - Undo is generated for the old value
    - Written back to undo block
    - Written to online redo log
  - Redo is generated for the new value
    - Written to online redo log
  - Both values will subsequently be:
    - Copied to archived redo log
    - Transported to standby database (if configured)

- Code on previous slide might be rewritten as:

```
UPDATE t1
SET a = a + 1
WHERE z = :b1;
```


The code on the previous slide could be rewritten as

```
UPDATE t1 SET a = a + 1 WHERE z = :b1;
```

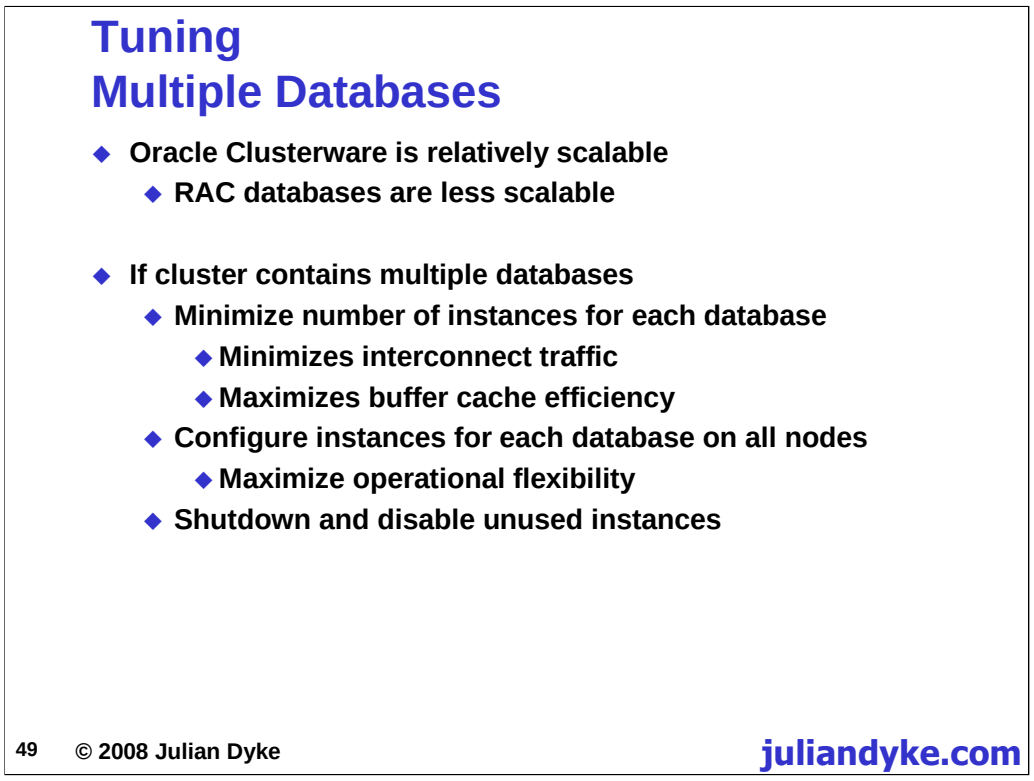
This would significant reduce the undo and redo generation when the code was executed.

However, you need to identify candidate statements carefully. For example if only one column was ever updated at a time, it would be reasonable to code a separate statement for each column e.g.:

```
UPDATE t1 SET a = a + 1 WHERE z = :b1;
UPDATE t1 SET b = b + 1 WHERE z = :b1;
UPDATE t1 SET c = c+ 1 WHERE z = :b1;
```

and so on

However if there were many possible multiple column combinations, then the amount of parsing could become excessive and in this case it is better to list all columns in a single statement to avoid flooding the library cache.



From a scalability perspective we can conclude that Oracle Clusterware is very scalable; the only non-scalable element is the TCP/IP heartbeat message and this will not have a significant impact in clusters of less than, say, 16 nodes. However RAC databases are less scalable. As the number of nodes increases, if each database has a new instance on each new node the database will potentially scale less efficiently. The number of interconnect messages exchanged between nodes will naturally increase although this may be offset to a limited extent by dynamic resource mastering.

I would always recommend creating an instance for each database on each node in the cluster. This provides maximum flexibility during normal operation and increased flexibility during a disaster. However, if there are more than 2 nodes in the cluster, it may not be necessary to start all of the instances for a given database. As long as there is a preferred instance and an available instance, this may be sufficient to support a database. Additional instances can be started if required by the workload; otherwise they can be disabled.

Tuning
Node Affinity

- Attempt to achieve node affinity for individual data blocks
  - Minimizes interconnect traffic
  - Maximizes buffer cache efficiency

- At segment level use:
  - Database services
  - Partitioning (range/list/system)

- At block level use:
  - PCTFREE storage clause
  - ALTER TABLE MINIMIZE RECORDS_PER_BLOCK
  - Filler columns e.g.
    - FILLER CHAR(2000)
  - Multiple block sizes

Ideally an application will have node affinity at segment level; this will allow dynamic remastering to be most effective.

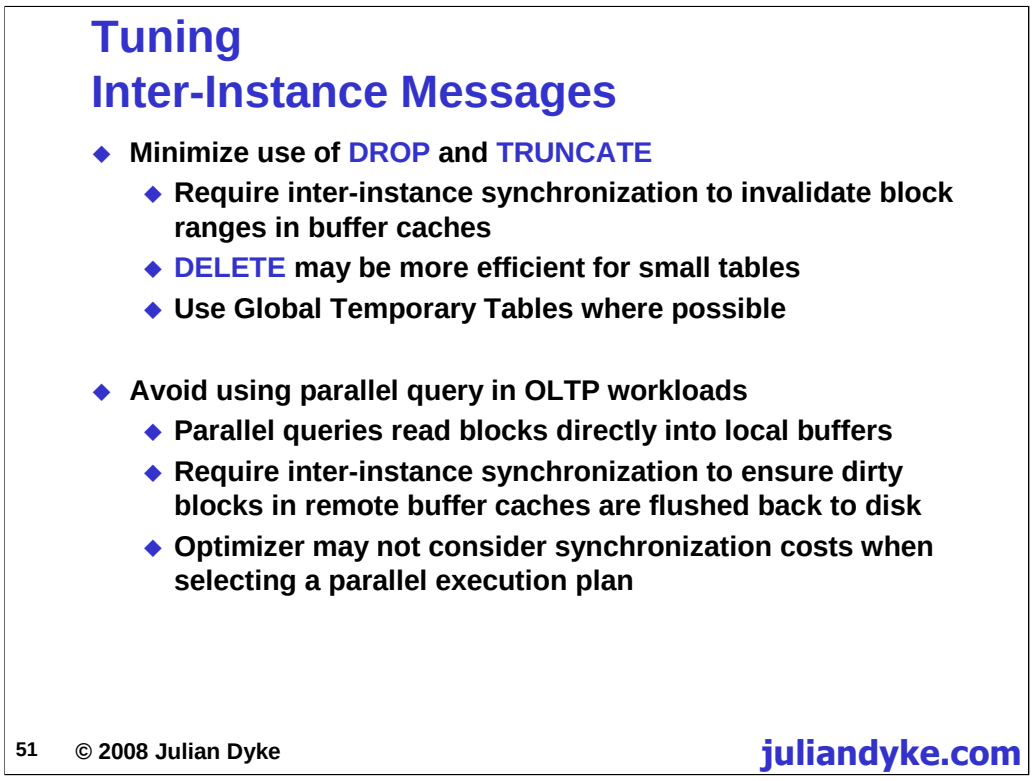
If this is not possible, then try to achieve node affinity at block level. Although this will not facilitate dynamic remastering, it will allow blocks to be maintained on specific nodes thus reducing interconnect traffic.

At segment level node affinity can be achieved using database services to assign objects to specific nodes based on usage or using the Partitioning Option to partition objects. Usually range or list partitioning will be used; in Oracle 11.1 and above you might also use system partitioning to design your own partitioning criteria.

At block level the number of rows in a block can be controlled in a number of ways. Restricting the number of rows may lead to better node affinity for the block or at least reduce inter-instance contention. The number of rows in a block can be restricted using the PCTFREE storage clause. Alternatively you can use the ALTER TABLE MINIMIZE RECORDS_PER_BLOCK statement to specify the exact number of rows per block. (This is messy as you must already have a block with the correct number of rows).

Another alternative is to simply add some filler columns to the table definition. CHAR columns are quite efficient for this purpose. This will slow down INSERT statements, but should not have any significant impact on UPDATE.

Finally you could try using a smaller block size for a limited number of tables using the multiple block size feature introduced for tablespaces in Oracle 9.0.1



Some SQL statements impact other instances.

For example both the DROP and TRUNCATE statements invalidate ranges of blocks. If another instance currently has these blocks in its local buffer cache then they must be flushed to disk and removed.

For small tables, you may find that deleting all of the rows using a DELETE statement is more efficient than issuing a DROP or TRUNCATE statement in a RAC environment.

Parallel queries also have an inter-instance performance impact. Parallel servers read blocks directly into a local buffer from disk. This is generally very efficient, but requires that all blocks have been saved to disk before they are read. Therefore a parallel query sends a message to all instances in the cluster requesting that they flush any affected blocks to disk before the parallel query can start. Whilst parallel execution may be efficient for large complex queries and changes, it may have a negative impact for smaller queries, particularly within an OLTP environment.