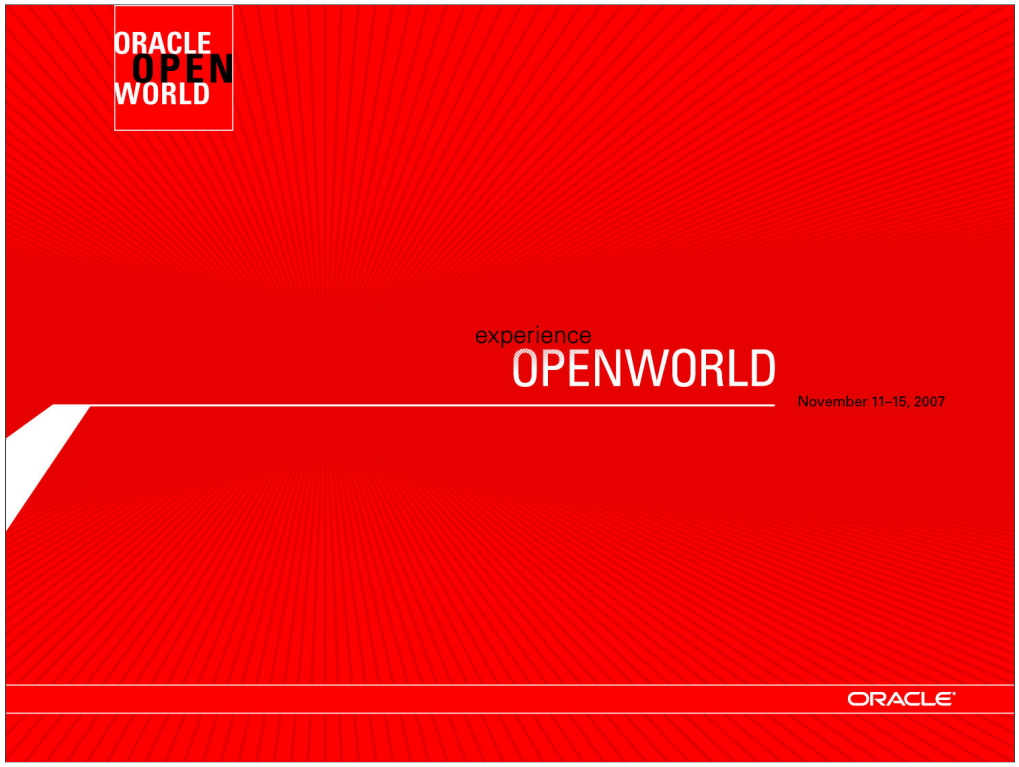

ORACLE®


ORACLE
OPEN
WORLD
experience
OPENWORLD
November 11–15, 2007
ORACLE

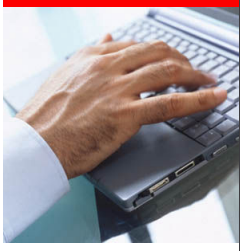
ORACLE

ORACLE THIRTY YEARS 30

When to use the Appropriate Database Technology

Andrew Holdsworth
Senior Director Real World Performance Server Technologies

Topics for Discussion



- Parallel Query, Partitions and Hashing
- Optimizer
- Session and Connection Management

ORACLE

Parallel Query
History and Background

- PQ has existed since Oracle7.1
- Since then a number of things have happened
  - The code has matured and been enhanced considerably to ensure better and more efficient parallelism of queries
  - Individual CPU speeds are over 100 times faster
  - Queries are being required to run over clusters which may mean Degrees of Parallelism > 100
  - Database designers are moving to indexless designs and preferring to solve problems by use of massive amounts of CPU horsepower rather than design of indexes, aggregates etc.
- For Parallel Query it is definitely Brawn over Brains

ORACLE

Parallel Query

Implications of Today's Requirements

- World class query performance becomes a matter of bringing massive amounts of system resources to the query for a short period of time.
- To get good performance these resources need to be sized correctly and be available to run the query.
- In many ways this in analogous to OLTP where to get good performance you need to be able to schedule your transaction immediately and let it run uninterrupted and with no delays for system resources e.g. I/O
- However with parallel query the resources required are on a much bigger scale e.g Multiple CPU Minutes etc
- Time slicing and delays for I/O are death to OLTP response time. This applies equally to DSS environments
- If Queries complete very quickly the concurrency issue becomes moot.

ORACLE

Parallel Query

A Premium on Capacity Planning

- Today's CPUs are fast, plentiful and cheap and provide huge scope for processing massive amounts of rows.
- Storage and networking technology has however failed to keep pace with the increase of CPU speed and the ability to place a large number of CPUs in a single box or cluster.
- The storage and networking technology is in many cases now representing 90% of the HW budget and shortfalls of HW prevent adoption of parallel query techniques.

ORACLE

Parallel Query
Some Basic Numbers

- Today's CPUs can consume 100-400 Megabytes of I/O per second per core.
- This means #disks, I/O Fabric and interconnect need appropriate sizing
- For today's Intel 4 CPU x 4 Core in a cluster
  - 4 x 4 x 100MegaByte/s = 1.6 GigaByte/s I/O Fabric BW
  - 4 x 4 x 100MegaByte/s = 1.6 GigaByte/s Interconnect BW
  - #disks = 1.6 GigaByte/s / 20 MegaBytes/s/Disk = 80 Drives
  - (note this does not include redundant disks/network for HA)

ORACLE

These calculations are based on taking the lower value of 100MegaBytes/Sec. The value used is subject to considerable debate. As a general rule narrow tables < 20 columns require more I/O than wide tables > 200 Columns. Likewise tables with more dates and numbers require less I/O to saturate the CPU.

Parallel Query

Why These Numbers May Get Bigger

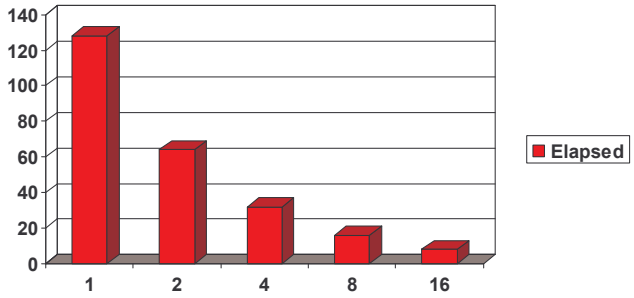
- Compression technology may remove some of the I/O requirements but will increase interconnect requirements because data shipped over the interconnect for join/sort operations is uncompressed
- The desire for higher uptimes will require higher redundancy in the number of drives
- CPUs will continue to speed up !

ORACLE

Parallel Query

Expectations and Reality

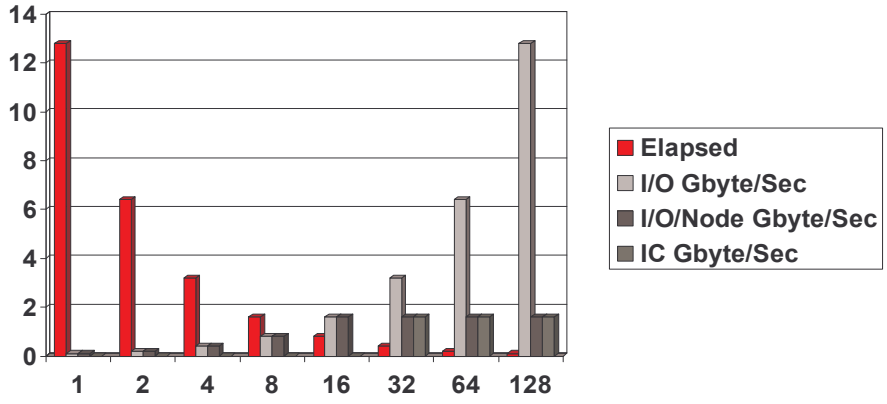
- The expectation is that as you increase the degree of parallelism you should see a proportional decrease in query time.
- In many cases you can simply stockpile equipment until you get the correct performance.



ORACLE

Parallel Query

System Requirements on a 4 x 4 x 4 Cluster



ORACLE

Parallel Query

What did we learn from the Previous Graph

- It is really easy to generate requests of 1.6 GigaBytes/s of I/O or Interconnect.
- Most of our customer base has machines with 1 or 2 100 MegaByte/s HBAs.
- This effectively rules out using Parallel query.
- This is the first step on the Data Warehouse death spiral !

ORACLE

Parallel SQL Execution

Assuming we have the HW

- The challenge is now to design the database objects and SQL for parallel execution.
- The biggest decision usually comes down to parallel execution vs serial execution. Most people chose/evolve to serial execution because they don't have enough HW.
- User response time degrades further as users try index access rather than a full table scan.
- Indexes are great for OLTP type workloads with well tested predictable SQL statements that reference as small number of rows
- In DSS indexes will kill ETL performance, require maintenance, confuse both developers and the optimizer.
- And who knows what index to build when running ad hoc queries ?

SQL Design For Parallel

Table Scans vs Index access

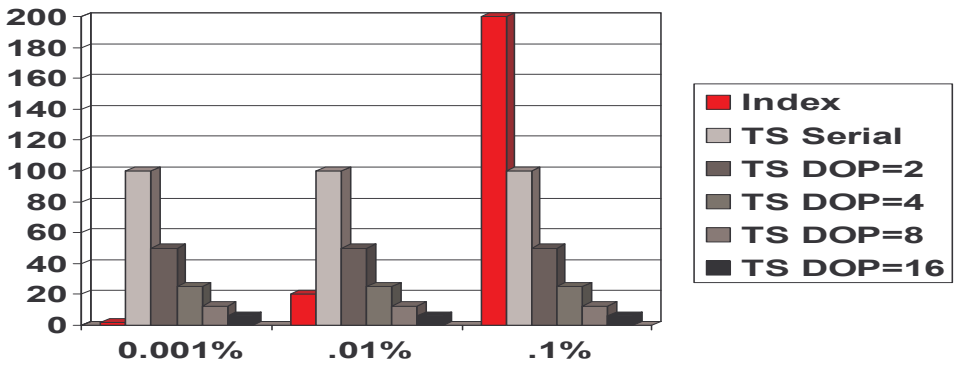
- Index access by its nature is a series of synchronous small random block size I/Os requiring a seek for each I/O
- Table scans are a series of large I/Os ideally done in parallel
- Knowing this it is easy to see why index access is very slow for large datasets and is unable to utilize many system resources. This is another step on the DW death spiral

ORACLE

SQL Design For Parallel Execution

Table Scans vs Index access: some numbers

- Lets assume a 100,000,000 row table of 100 Byte row length, TS I/O is 1M @ 10ms, Index I/O is 8k @ 5ms and assuming 80% cache hit



ORACLE

They point to make here is that the Table is so large it is not in cache. Index traversals via disk are very slow. In most databases this is not the case because the entire index is rapidly cached. In this example it is assumed the index is to big to fit into cache.

SQL Design For Parallel

Table Scans vs Index access

- Hopefully the last slide will explain why Indexes yield low utilization and poor response time.
- To get good response time you need to have plenty of HW available to schedule parallel table scans
- This is ultimately how you deliver predictable response times.
  - Just like OLTP (resources are ready and available)
  - The numbers are just multiple orders of magnitude bigger !

ORACLE

SQL Design For Parallel

Partitioning for Pruning

- To optimize table scans further the I/O requirements can be further reduced by elimination of data by partitioning on one of the where clause predicates in your most frequent queries.
- This is usually a date field but can be other predicates
- This technique can be utilized by both serial and parallel execution plans.
- Pruning on key predicates may mean that a higher % of rows in the scanned partition need not be eliminated by predicate filtering again favoring a table scan approach.

ORACLE

SQL Design For Parallel

Partitioning for Pruning Example

```
SQL> l
  1  select * from
  2  fact_a a,
  3  fact_b b
  4  where a.part_col1 = b.part_col1
  5  and a.date_key = b.date_key
  6* and a.date_key = 20070621
SQL> /
```

Id	Operation	Name	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT						
1	PX COORDINATOR						
2	PX SEND QC (RANDOM)	:TQ10000			Q1,00	P->S	QC (RAND)
3	PX PARTITION HASH ALL		1	1024	Q1,00	PCWC	
*4	HASH JOIN				Q1,00	PCWP	
5	PX PARTITION RANGE SINGLE		**11**	**11**	Q1,00	PCWC	
*6	TABLE ACCESS FULL	FACT_B	**10241**	11264	Q1,00	PCWP	
7	PX PARTITION RANGE SINGLE		**11**	**11**	Q1,00	PCWC	
*8	TABLE ACCESS FULL	FACT_A	**10241**	**11264**	Q1,00	PCWP	

Blue = Range Partition

Red = Hash Partition

SQL Design For Parallel

Partitioning for Joins and Sorts

- When joining or sorting data we have the ability to take advantage of partition wise operations.
- The goal of these operations is to reduce the system resources involved in join/sort operations
  - CPU
  - Interconnect
  - Memory
  - Sort Space
- These operations are know as partition aware operations and yield large performance gains especially when running at high degrees of parallelism or in a cluster.

ORACLE

SQL Design For Parallel

Partitioning for Joins and Sorts

- Basic rules for partition aware operations
- Achieve equal partition sizes for each partition to ensure uniform execution time for each partition.
- This is best achieved by hash partitioning or sub partitioning the tables
  - You must choose a power of 2 e.g. 64,128,512
  - For partition aware joins the number of hash partitions must equal the degree of parallelism
  - For partition aware sorts the number of hash partitions must be at least twice the degree of parallelism
  - This encourages high numbers of hash partitions when running at high degrees of parallelism.

ORACLE

These heuristics apply to Oracle releases <= 10.2.0.3

SQL Design For Parallel

Examples of Joins(No Partitioning, Note 3 DFOs and Parallelism)

```
SQL> l
  1  select * from
  2  fact_a a,
  3  fact_b b
  4* where a.non_part_col1 = b.non_part_col1
SQL> /
```

Id	Operation	Name	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT						
1	PX COORDINATOR						
2	PX SEND QC (RANDOM)	**:TQ10002**			Q1,02	**P->S**	QC (RAND)
*3	HASH JOIN BUFFERED				Q1,02	PCWP	
4	PX RECEIVE				Q1,02	PCWP	
5	PX SEND HASH	**:TQ10000**			Q1,00	**P->P**	HASH
6	PX BLOCK ITERATOR		1	1024	Q1,00	PCWC	
7	TABLE ACCESS FULL	FACT_B	1	15360	Q1,00	PCWP	
8	PX RECEIVE				Q1,02	PCWP	
9	PX SEND HASH	**:TQ10001**			Q1,01	**P->P**	HASH
10	PX BLOCK ITERATOR		1	1024	Q1,01	PCWC	
11	TABLE ACCESS FULL	FACT_A	1	15360	Q1,01	PCWP	

SQL Design For Parallel

Examples of Joins(Hash Partition on join column, Note 1 DFO)

```
SQL> l
  1  select * from
  2  fact_a a,
  3  fact_a b
  4* where a.part_col1 = b.part_col1
SQL> /
```

Id	Operation	Name	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT						
1	PX COORDINATOR						
2	PX SEND QC (RANDOM)	:TQ10000			Q1,00	P->S	QC (RAND)
3	PX PARTITION HASH ALL		1	1024	Q1,00	PCWC	
*4	HASH JOIN				Q1,00	PCWP	
5	PX PARTITION RANGE ALL		1	15	Q1,00	PCWC	
6	TABLE ACCESS FULL	FACT_B	1	15360	Q1,00	PCWP	
7	PX PARTITION RANGE ALL		1	15	Q1,00	PCWC	
8	TABLE ACCESS FULL	FACT_A	1	15360	Q1,00	PCWP	

SQL Design For Parallel

Examples of Joins(Hash Partition on join column, Note DOP != # of Hash Partitions)

```
SQL> l
  1  select /*+ PARALLEL( a 256 ) PARALLEL( b 256 ) */ * from
  2  fact_a a,
  3  fact_a b
  4* where a.part_col1 = b.part_col1
SQL> /
```

Id	Operation	Name	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT						
1	PX COORDINATOR						
2	PX SEND QC (RANDOM)	:TQ10001			Q1,01	P->S	QC (RAND)
*3	HASH JOIN				Q1,01	PCWP	
4	PX BLOCK ITERATOR		1	1024	Q1,01	PCWC	
5	TABLE ACCESS FULL	FACT_B	1	15360	Q1,01	PCWP	
6	BUFFER SORT				Q1,01	PCWC	
7	PX RECEIVE				Q1,01	PCWP	
8	PX SEND BROADCAST LOCAL	**:TQ10000**			Q1,00	P->P	**BCST LOCAL**
9	PX BLOCK ITERATOR		1	1024	Q1,00	PCWC	
10	TABLE ACCESS FULL	FACT_A	1	15360	Q1,00	PCWP	

SQL Design For Parallel

Examples of Sorts (Aggregate group by with no partitioning with 2 DFOs)

```
SQL> l
  1  select non_part_col1, count(*)
  2  from fact_a
  3* group by non_part_col1
SQL> /
```

Id	Operation	Name	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT						
1	PX COORDINATOR						
2	PX SEND QC (RANDOM)	**:TQ10001**			Q1,01	P->S	QC (RAND)
3	HASH GROUP BY				Q1,01	PCWP	
4	PX RECEIVE				Q1,01	PCWP	
5	PX SEND HASH	**:TQ10000**			Q1,00	P->P	HASH
6	HASH GROUP BY				Q1,00	PCWP	
7	PX BLOCK ITERATOR		1	1024	Q1,00	PCWC	
8	TABLE ACCESS FULL	FACT_A	1	15360	Q1,00	PCWP	

SQL Design For Parallel

Examples of Sorts (Aggregate group by with 1024 sub hash partitioning with 1 DFO)

```
SQL> l
  1  select /*+ PARALLEL( a 256 ) */ part_col1, count(*)
  2  from fact_a a
  3* group by part_col1
SQL> /
```

Id	Operation	Name	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT						
1	PX COORDINATOR						
2	PX SEND QC (RANDOM)	:TQ10000			Q1,00	P->S	QC (RAND)
3	PX PARTITION HASH ALL		1	1024	Q1,00	PCWC	
4	HASH GROUP BY				Q1,00	PCWP	
5	PX PARTITION RANGE ALL		1	15	Q1,00	PCWC	
6	TABLE ACCESS FULL	FACT_A	1	15360	Q1,00	PCWP	

SQL Design For Parallel

Examples of Sorts (Aggregate group by with 1024 sub hash partitioning with DOP > 1024/2 hence 2 DFOs)

```
SQL> l
  1  select part_col1, count(*)
  2  from fact_a a
  3* group by part_col1
SQL> /
```

Id	Operation	Name	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT						
1	PX COORDINATOR						
2	PX SEND QC (RANDOM)	:TQ10001			Q1,01	P->S	QC (RAND)
3	HASH GROUP BY				Q1,01	PCWP	
4	PX RECEIVE				Q1,01	PCWP	
5	PX SEND HASH	:TQ10000			Q1,00	P->P	HASH
6	PX BLOCK ITERATOR		1	1024	Q1,00	PCWC	
7	TABLE ACCESS FULL	FACT_A	1	15360	Q1,00	PCWP	

SQL Design For Parallel

Some Ideas for DOP and Hash Partition Sizing

- The Challenge is to select appropriate DOP values and number of Hash Partitions
  - DOP(Degree of Parallelism)
    - Either allow to default or set at a value in phase with partition frequency for joins or sorts.
    - Use DB Resource Manager to define an individual user's DOP. Query optimization will evaluate DOP at default and then allocate slaves according to DBRM scheme.
    - Use of PARALLEL_ADAPTIVE_MULTI_USER may get you into trouble if predictable response time is your goal.
  - # of Hash partitions
    - Must be a power of 2
    - A multiple of the DOP values used on the system. If DOP != power of 2 there will remainder effects hence a multiple of this value.
    - Will impact loading, data dictionary, stats gathering, ability to drop tables
- This process is all a compromise

ORACLE

The Optimizer

- Probably the most maligned, misunderstood and poorly used part of the Oracle Database.
- This is a shame because correct use of the optimizer has the most impact on database performance and reliability.
- Part of this is historical as there has been little guidance on how to best use the optimizer to get good plans, how to maintain good plans.
- This has led to the tuning via Google or Witchcraft approach to the optimizer.
- Most sites we see have set non-default values for init.ora parameters that control the optimizer and are probably reducing their chance of optimal plans and risking a more stressful upgrade process in the future.

The Optimizer

Basics of Optimization

- Run with default init.ora parameters that influence the Optimizer
- Do not make any global changes without testing. Global changes are init.ora parameters and schema statistics.
- Be aware hints do not ensure identical plans over time and hints may result in sub optimal execution plans in the future.
- Determine a stats gathering and refresh strategy applicable to the type of application.
- Accept pragmatically there may be plan changes(good and bad) over time and database releases and follow a logical debugging route to either solve the issue or escalate to development
- Testing is crucial to success.
- You may have to face the reality the reason you got good plans in a prior release is pure luck rather than excellent database engineering/management.

ORACLE

The Optimizer

Basics of Optimization

- Conceptually the optimization process can be broken down into two phases.
  - Cardinality Estimation
    - This where the optimizer computes the number of rows coming out of a row source. This is mainly influenced by schema statistics which is under control of the DBA.
  - Execution Plan Selection
    - This process chooses the access path type, join order, join type, view merging etc etc
    - This is under control of the optimizer developers. This phase can be influenced by init.ora parameters and for this reason it is dangerous to change these without knowing fully what is happening.
    - This part really is the rocket science part !

ORACLE

The Optimizer

Strategy for Gathering Stats

- The goal is to provide the optimizer a mechanism for calculating the correct cardinality of rows coming from a row source.
- Schema Statistics can be broken down into two types of statistics
  - Table level Statistics
    - This tells the optimizer the size of the table, how many rows etc.
  - Column level Statistics(potentially including index statistics and histograms)
    - This tells the optimizer about the distribution of key values within a column and the columns actual size.
    - These are crucial as you select rows by use of where clauses on columns.
  - All statistics should be gathered, set, exported, imported using the dbms_stats package.

ORACLE

The Optimizer

Strategy for Gathering Stats

- The defaults are a good start but are not perfect in specific situations. In these situations the DBA will need to decide when to either gather or remove statistics.
- In reality this may result in less frequent statistics gathering, manual copying/setting of statistics etc.
- In the end the goal is to generate good cardinality estimates to allow the optimizer to do a good job.
- In summary **POOR** statistics yield **POOR** cardinality estimates yield **POOR** execution plans.

ORACLE

The Optimizer

Strategy for Gathering Table Stats

- Table statistics are probably the easiest to gather and probably need to change the least.
  - How often does a table change 10% and would that really influence the execution plans
  - Look for volatile tables where the row counts change by orders of magnitude
  - Temporary and Intermediate tables are classic examples of this.
  - In this case the statistics should be gathered for the worst possible case and locked e.g. the highest row population

ORACLE

The Optimizer

Strategy for Gathering Table Stats

- Partitioned tables are very vulnerable to issues here especially when using daily partitions. A classic case would be a trading application which starts from an empty partition each day and inserts millions of rows. The settlement transactions at the end of the day would run sub optimally if care is not taken to reflect the new rows in the table statistics.
- The DBA needs to decide if to build partition and/or global table level statistics. There are pros and cons to both approaches:
  - If queries resolve down to a single partition this may be a candidate for partition level stats
  - If however the Global Table stats yield good cardinality estimates why further complicate the situation.

The Optimizer

Strategy for Gathering Column Stats

- The challenges for column statistics.
  - Data skew or uniform distribution
    - If your data values are distributed evenly you do not need histograms
    - If you have skew, histograms may help but beware of bind variable peeking issues and beware if you have more distinct values than histogram buckets.
  - Min/Max Values
    - Date and key values generated from sequences usually raise the max values for a column.
    - A predicate value outside the range of the min/max values can give unreliable cardinality estimates. Again beware of bind peeking issues.

ORACLE

The Optimizer

Strategy for Gathering Column Stats

- Data Correlation effects
  - Is there implicit association between columns rendering cardinality estimates inaccurate ?
- Functions
  - The default cardinality for functions is 5%. Is this true in reality for your where clause ?
- Bind Variable Peeking and Literals
  - Addressed pretty much in 11g but not in 10g. Have you protected yourself against these issues by removal of histograms and keeping min/max values accurate ?

ORACLE

The Optimizer

What Happens when I get a Bad Plan

- A bad plan is a function of the following
  - Bad Cardinality Estimates
    - A DBA can influence this
  - Bad Optimization Process
    - A DBA has little control over this
- The first step in the debugging process assuming a default init.ora, a good statistics strategy and hints removed would be to validate the estimated vs the actual cardinality estimates.
- This is very simply done with the GATHER_PLAN_STATISTICS hint and running the following SQL
  - `select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));`

ORACLE

The Optimizer

What Happens when I get a Bad Plan example

```
SQL> SELECT /*+ GATHER_PLAN_STATISTICS */
  2         count(DISTINCT s.claim_nbr)
  3            FROM    v_service s,
  4                    v_claim c  ,
  5                    v_prov_affiliation a
  6            WHERE   s.claim_nbr       = c.claim_nbr
  7                AND s.srv_aff_nbr     = a.aff_nbr
  8                AND s.region         IN('AT','CN','SW')
  9                AND s.srv_prov_nbr    = a.prov_nbr
 10                AND a.hat_code        = 'HTHO'
 11                AND s.bus_line        = 'GA'
 12                AND s.bus_unit        = 'GA'
 13                AND c.resolution not IN ('90','91')
 14                AND s.effective_date >= a.effective_date
 15                AND ((a.end_date BETWEEN s.effective_date AND s.end_date)
 16                 OR a.end_date  >= s.effective_date)
 17                AND s.paid_date >= add_months(trunc(sysdate,'year'),0)  -0
 18                 AND s.paid_date <= last_day(add_months((trunc(sysdate)),-1));

COUNT(DISTINCTS.CLAIM_NBR)
--------------------------
                    346432
```

ORACLE

The Optimizer

What Happens when I get a Bad Plan example

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	Reads
1	SORT GROUP BY		1	1	1	00:08:01.44	6219K	397K
* 2	FILTER		1		1728K	00:07:22.80	6219K	397K
3	NESTED LOOPS		1	1	1728K	00:07:22.80	6219K	397K
* 4	HASH JOIN		1	1	1728K	00:06:13.66	1021K	388K
5	PARTITION LIST SINGLE		1	6844	3029	00:00:00.04	537	537
* 6	INDEX RANGE SCAN	PROV_AFFILIATION_IX13	1	6844	3029	00:00:00.03	537	537
7	PARTITION LIST SINGLE		1	**5899**	**5479K**	00:06:45.52	1021K	388K
* 8	TABLE ACCESS BY LOCAL INDEX ROWID	SERVICE	1	**5899**	**5479K**	00:06:45.51	1021K	388K
* 9	INDEX SKIP SCAN	SERVICE_IX8	1	**4934**	**5479K**	00:01:11.27	34399	34399
10	PARTITION LIST SINGLE		1728K	**1**	**1728K**	00:01:05.88	5197K	9215
* 11	INDEX RANGE SCAN	CLAIM_IX7	1728K	**1**	**1728K**	00:01:01.00	5197K	9215

The Optimizer

Debugging The Problem

- In this example embedding correct cardinality hints resulted in a good execution plan.
- The bad cardinality estimates were a function of massive data correlation reducing the cardinality to 1, this resulted in poor access paths, join order and join type selection.
- If however if the cardinality estimates are good and a poor plan is still obtained a different strategy is required
  - Short Term: Fix plan with hints, outlines profiles etc.
  - Long term: Escalate to Support providing SQL statement, schema stats export, plan output results and 10053 trace.

ORACLE

Sessions and Cursors

A 15 year Old Problem

- Poor session and cursor management from within application code has been documented for over 15 years now at Oracle Open World.
- The issue comes down to the following issues
  - System Performance
  - System Availability
- Having too many connections will render your statistics meaningless. You may see excessive latch, buffer contention with high process counts. In many cases the best way to remove the contention is to cut the connection count.

ORACLE

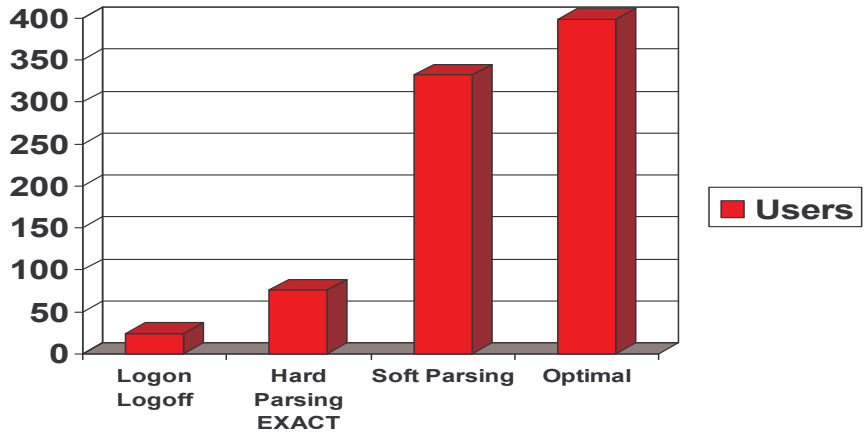
Sessions and Cursors
Performance

- The core performance issues
  - Running with the optimal number of connections
    - Try and target no more than 10 per Core in a true three tier application server architecture
    - This can be fine tuned upon amount of physical I/O taking place
  - Running with minimal logon/off activity
    - Eliminate continuous logon/off to protect the system from connection storms
  - Running with minimal parsing
    - Use bind variables to share SQL for OLTP applications
    - Reduce soft parsing if possible also

ORACLE

Session and Cursors

Performance (A classic Slide !)



ORACLE

Sessions and Cursors

Availability Because of Bad Programming

- Session Leakage
  - Sessions being allocated and not being dropped when finished with. A classic programming problem traditionally seen with memory. Java programmers forgot all about this one.
  - Symptoms
    - Increase in database connections
    - Increasing sys time in the Operating system
    - Reduced memory on the system
    - Sessions and processes init.ora parameter always being increased
- Cursor leakage
  - Similar to session leakage caused by programmer failing to close or reuse a cursor correctly
  - Symptoms
    - High parse rate(hard and soft)
    - Shared pool memory being exhausted
    - open_cursors set really high !

ORACLE

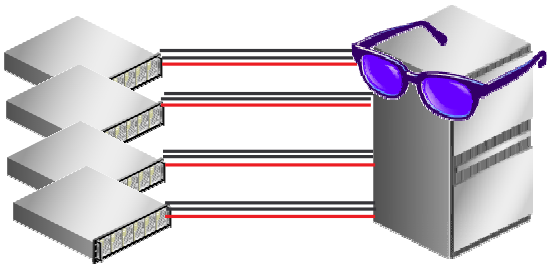
Sessions and Cursors

Availability Problems Caused by Poor Middleware Design/Configuration

- Connection Storms
  - Caused by application servers that allow a pool of connections to the database to increase when the pool is exhausted.
  - A connection storm may take the number of connections from hundreds to thousands in a matter of seconds
  - The process creation and logon activity may mask the real problem of why the connection pool was exhausted and make debugging a longer process.
  - A connection storm may render the database server unstable, panicked, hung, etc.

Sessions and Cursors

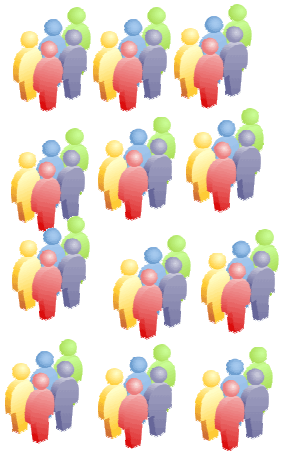
The Anatomy of a Connection Storm

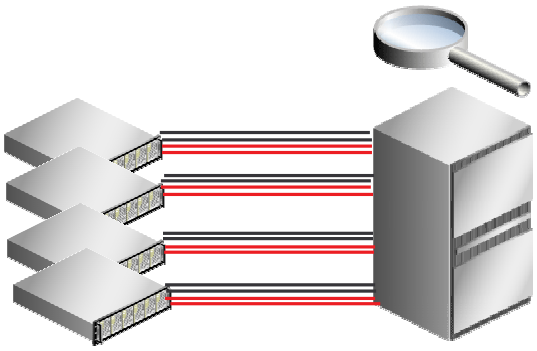


ORACLE

Sessions and Cursors

The Anatomy of a Connection Storm

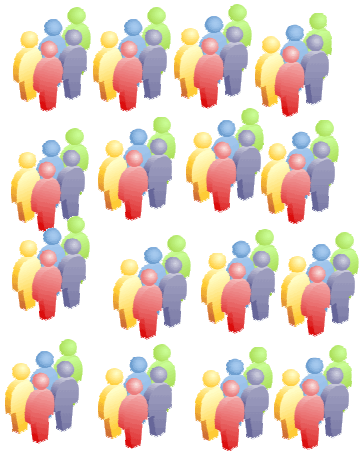


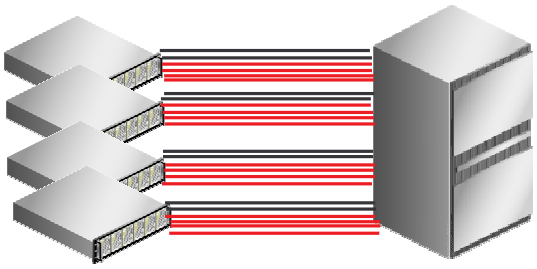


ORACLE

Sessions and Cursors

The Anatomy of a Connection Storm

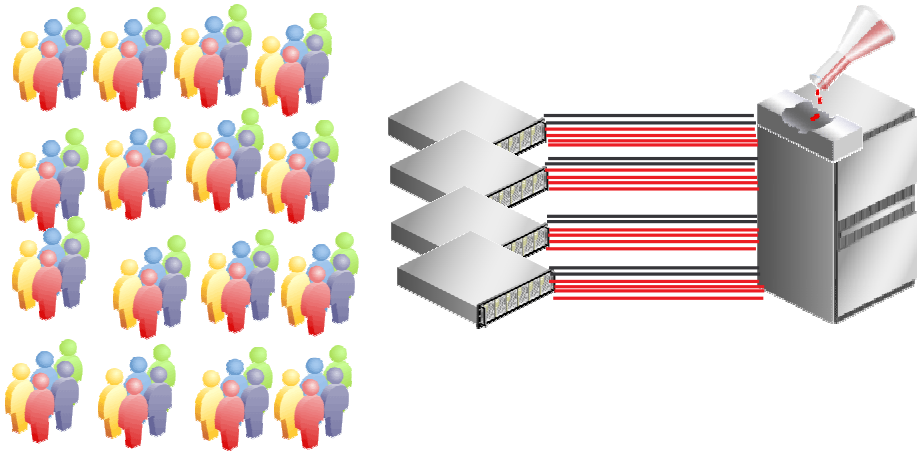




ORACLE

Sessions and Cursors

The Anatomy of a Connection Storm



ORACLE

Sessions and Cursors

Are you Vunerable to a Connection Storm ?

- Does your AWR, statspack report show a logon/off rate > 1 per second over a period of time ?
- Have you set processes to a high value as a band aid ?
- Have you even thought about it ?

ORACLE

Questions

Please no debugging on stage !

ORACLE

For More Information

search.oracle.com

or

oracle.com

ORACLE

Real World Performance Group Sessions

Session ID: S291301
Session Title: Current Trends in Database Performance
Track: TECHNOLOGY; Business Intelligence ; Database; Enterprise Management; Linux and Open Source Solutions
Room: 102
Date: 2007-11-14
Start Time: 09:45

Session ID: S291306
Session Title: Roundtable: The Real-World Performance Group
Track: TECHNOLOGY; Business Intelligence ; Database; Enterprise Management; Linux and Open Source Solutions
Room: 104
Date: 2007-11-14
Start Time: 11:15

Session ID: S294625
Session Title: Performance 2.0
Track: TECHNOLOGY; Business Intelligence ; Database; Enterprise Management; Linux and Open Source Solutions
Room: Yerba Buena Theatre
Date: 2007-11-14
Start Time: 15:00

Session ID: S291304
Session Title: When to Use the Appropriate Database Technology
Track: TECHNOLOGY; Business Intelligence ; Database; Enterprise Management; Linux and Open Source Solutions
Room: 102
Date: 2007-11-15
Start Time: 08:30

ORACLE


ORACLE®