DSS Performance in Oracle Database 10*g*

An Oracle White Paper
September 2003

ORACLE

Executive Overview 3
Introduction 3
Test Environment 4
Performance and Feature Improvements 5
Full Table Scan 6
Large Number of Partitions 7
Optimizer CPU Cost Model 9
Auto-Statistics 12
Parallel Single Cursor 14
  Performance 14
  Memory Reduction 15
Partition Aware MV Refresh 16
Partition Outer Join 18
Native Numbers 20
  Performance 20
  Data Type and Storage 21
Conclusion 22

EXECUTIVE OVERVIEW

Two of the key requirements for data warehouses are to improve performance and to reduce manageability costs. Oracle Database 10*g* delivers on both requirements by providing significant performance enhancements automatically, with no extra deployment costs or resources to implement these enhancements. This technical white-paper details these performance features and describes the benefits they deliver.

INTRODUCTION

Data warehouses have become a cornerstone for many successful enterprises, not only supporting traditional strategic decision-making, but day-to-day operational business processes as well. With its increasing importance, the data warehouse has become a critical resource for large numbers of users, supporting a wide variety of both simple and complex processes. The data warehouse is now a mission-critical system.

With its increasing importance, the data warehouse has lost much of the flexibility and informality that one often saw years ago when data warehouses were limited to a handful of selected strategic decision makers. Data warehouses are now a mainstream part of the business operations, managed by IT with service-level agreements for availability and performance, rigorous quality-control development processes, and tight budgets.

The challenges for a data warehouse administrator are to continue to deliver improved performance and scalability, while constantly evolving the data warehouse to meet new business requirements, and at the same time reducing overall costs. While these challenges seem mutually exclusive, these are the business realities faced by most data warehouses today. These are also the challenges that Oracle Database 10*g* is designed to address.

This paper deals with only one of those technical challenges: performance.

In the area of performance, Oracle Database 10*g* delivers transparent performance gains: new features and enhancements whose benefits are realized automatically after upgrading to Oracle Database 10*g*, without explicitly implementing any new features. A data warehouse team does not have the resources to test and implement a large numbers of new database features. [Indeed, the data warehouse

team's focus for new development will be on delivering new business functionality, not implementing new database technologies to improve the current environment.] Thus, the burden for improving performance should properly fall not on the data warehouse team but on the technology itself, and this is what Oracle Database 10*g* delivers.

This paper describes several new capabilities in Oracle Database 10*g*, and characterizes the performance gains delivered by these features. With the exception of two features, each of these enhancements provides performance benefits without any specific intervention by the data warehouse administrator.

TEST ENVIRONMENT

All tests are performed on a SunFire 4800 with

- 8 x 900 MHz CPUs
- 16GB RAM
- 8 x T3BES (9 x 36GB per T3) in 72" StorEDGE cabinet
- Solaris 5.8

All the examples in this document use the Sales History schema from the Oracle Sample Schema. The Sales History table is populated with 6 years worth of sales data: Jan-1998 to Dec-2003, roughly 6GB.

The Sales History Schema consists of the following six tables:

Table	**Number of Rows**	**Average Row Length**
Sales	149,960,000	35
Times	2,191	219
Products	10,000	268
Customers	1,000,000	170
Channels	5	40
Promotions	1001	95

The Sales History table is range partitioned on time_id and each partition is populated with one month worth of sales data, total of 72 partitions.

Sales Table Columns	Distinct Values
Prod_Id	10,000
Cust_Id	1,000,000
Time_Id	2191
Channel_Id	5
Promo_Id	1001
Quantity_Sold	
Amount_Sold	

Only the primary key columns of the dimension tables are referenced in the examples; therefore, all other columns are irrelevant.

PERFORMANCE AND FEATURE IMPROVEMENTS

Oracle Database 10*g* delivers automatic performance gains, alleviating the data warehouse administrators from implementing new database features. Comparing Oracle9*i* Database with Oracle Database 10*g*, the rest of this paper describes the performance features and uses specific test cases to illustrate the improvements.

FULL TABLE SCAN

In the data warehouse environment, table scan is the foundation of many operations performed during query execution. With this in mind, the table scan performance has significantly improved in this release. Code optimization in Oracle Database 10*g* decreases CPU consumption and this translates into:

- faster table scan execution, mainly when CPU is a contention (i.e. not IO bound)
- higher concurrency, allowing for more queries to execute

In order to observe elapsed time improvement, the CPU resources available to the query was restricted by running on a single CPU. The elapsed time for a full table scan against the SALES table was measured using the following query:

```
SELECT COUNT(*) FROM sales WHERE company_id != 2;
```

.

Release	Elapsed Time (seconds)
Oracle9*i* Database	102
Oracle Database 10*g*	64

This is a huge performance improvement that will benefit all data warehouse query execution. However, the dramatic improvement (37%) seen above is mainly because the query is predominantly a full table scan and is not IO bound. For queries that perform more than just table scans, we would expect the improvements to be limited to the portion that corresponds to the table scan.

For example, the following query performs a table scan of the SALES table and applies an expensive predicate. We also execute this query using a single CPU and measure the elapsed time. The predicate evaluation takes roughly half the query execution; therefore, we will not expect the elapsed time to improve by 37%. Instead, only the table scan portion of the query will improve by that much. As a result, the overall elapsed time improvement is only 18%.

```
SELECT COUNT(*) FROM sales WHERE
(amount_sold/quantity_sold) < 1;
```

Release	Elapsed Time (seconds)
Oracle9*i* Database	208
10*g*	170

LARGE NUMBER OF PARTITIONS

Oracle's object partitioning is a very powerful feature in the data warehousing environment. For manageability, the most common partitioning method is by chronological time, such as our SALES table schema, which is range partitioned by time_id. Sales data for one specific month all reside within the same partition.

Query performance can be improved in many cases by composite partitioning, which can lead to tables with very large number of partitions. In Oracle Database 10*g*, scalability and memory usage of these objects has been dramatically improved, especially for those tables with tens of thousands of partitions.

For the purpose of this test, the SALES table was defined using range partitioning on time_id (72 partitions) and hash partitioning on prod_id (256 subpartitions per partition), this produces a composite partitioned table with 72*256=18,432 partitions:

```
CREATE TABLE SALES_COMPOSITE (
)
 PARTITION BY RANGE (time_id)
 SUBPARTITION BY HASH (prod_id)
 SUBPARTITIONS 256
```

In Oracle9*i* Database the performance of certain DDL operations such as "drop table" did not scale linearly as the number of partitions in the table increased. Performance scalability for the drop of this highly partitioned table has improved:

Release	**Elapsed Time (seconds)**
Oracle9*i* Database	754
Oracle Database 10*g*	347

Oracle Database 10*g* reduces memory consumption of partition metadata, and at the same time, allows for sharing of metadata between cursors. Unlike Oracle9*i* Database, the partition metadata is no longer copied from cursor to cursor. Instead, it is shared between multiple cursors. This improvement allows for concurrent queries to be processed without incurring the high memory cost of having duplicate copies of the partition metadata in the SGA.

Using the same composite partitioned SALES_COMPOSITE table as our example, we show the difference in cursor memory usage when executing concurrent queries against this table. Dramatic reduction of memory usage in Oracle Database 10*g* can be seen in the single query execution column. Subsequent executions using concurrent queries show the tremendous benefit of metadata sharing.

Actual benefit from this improvement will be seen on systems with high concurrency where cursor parse time is an overhead.

Shared Memory (Kbytes)	Single Query	2 Concurrent Queries	3 Concurrent Queries
Oracle9*i* Database (no metadata sharing)	607.4	1,194.8	1,715
Oracle Database 10*g* (metadata sharing)	42	54.7	63.9

OPTIMIZER CPU COST MODEL

The cost model is the optimizer's component to estimate the cost of a query. The optimizer generates different join orders and access paths in order to choose the optimal query execution plan. Cost is the criterion optimizer uses to compare plans to find the best one.

In Oracle9*i* Database, the Cost Based Optimizer uses an "IO" cost model. This cost model primarily costs everything in single block reads, and largely ignores the CPU cost or uses imprecise constants to estimate it.

In Oracle Database 10*g*, the default cost model will become "CPU+IO". In this model, the cost unit is time. In order to estimate query execution time, the optimizer estimates the number and the type of IO operations, the number of CPU cycles the database will perform during execution of the query, and uses system statistics to convert the number of CPU cycles and the number of IOs into execution time.

One of the important CPU-only operations is fetching data from the buffer cache. The cost of accessing an index or a table depends on whether the index blocks or table blocks are to be read from the buffer cache or from the disk. Oracle Database 10*g* introduces Object Level Caching Statistics to collect the following statistics for each object:

- average number of blocks in the buffer cache, and
- average cache hit ratio.

The optimizer uses these data to estimate the number of cached blocks for each index or table access. The total IO and CPU cost of a query will be the combined costs of the IO cost of reading non-cached blocks from the disk, the CPU cost of getting cached blocks from the buffer cache, and the CPU cost of processing the data.

With the new "CPU+IO" cost model, more factors are taken into account when computing the cost of a plan. As a result, better plans are selected in some cases. An example is the following join query on the SALES and PRODUCTS tables.

A unique index, PRODUCT_PK, is built on the PRODUCTS table with key = prod_id.

The SALES table has 149,960,000 rows while the PRODUCTS table only has 10,000 rows, so the small index PRODUCTS_PK built on PRODUCTS can be cached in memory.

The following query checks for missing prod_id in the PRODUCTS table:

```
SELECT * FROM sales WHERE prod_id NOT IN
(SELECT prod_id FROM products);
```

Since the query does not require accessing columns in the PRODUCTS table other than prod_id, it is sufficient to access the PRODUCTS_PK index. Using Oracle9*i*

Database's "IO" cost model, the optimizer thinks that, because the PRODUCTS_PK index is cached, the cost of the query is only the full scan of the SALES table. The optimizer will therefore choose nested loop, as shown below:

```
Id     Operation                  Name
0      SELECT STATEMENT
1       NESTED LOOPS ANTI
2        PARTITION RANGE ALL
3         TABLE ACCESS FULL    SALES
*4        INDEX UNIQUE SCAN    PRODUCTS_PK

Predicate Information (identified by operation id):
   4 - access("SALES"."PROD_ID"="PRODUCTS"."PROD_ID")
```

Using Oracle Database 10*g*'s "CPU+IO" cost model, the cost of executing the join and the cost of accessing the PRODUCTS_PK index from the buffer cache will also be considered. As a result, hash join will be chosen over nested loop join.

```
Id     Operation                  Name
0      SELECT STATEMENT
*1      HASH JOIN RIGHT ANTI
2        INDEX FAST FULL SCAN  PRODUCTS_PK
3        TABLE ACCESS FULL     SALES

Predicate Information (identified by operation id):

   1 - access("PROD_ID"="PROD_ID")
```

Release	Elapsed Time (seconds)
Oracle9*i* Database (IO cost model)	150
Oracle Database 10*g* (CPU+IO cost model)	34

This example demonstrates how a particular problem has been solved by using a more advanced approach and by taking more information into account.

Another potential benefit of the "CPU+IO" cost model is the ability to order predicates in queries. Predicate ordering is possible only when the cost of predicates is known, and this is possible only in the "CPU+IO" cost model because predicate cost has CPU as a main component.

Consider the following query:

```
SELECT COUNT(*) FROM sales
WHERE TO_NUMBER(TO_CHAR(time_id, 'YYYY')) > 1998
      AND promo_id = 98;
```

The cost of the first predicate is higher. This is because for every row, we need to transform the time_id column first with the TO_CHAR function followed by the TO_NUMBER function before we can evaluate the predicate. The transformation by the TO_CHAR and the TO_NUMBER function is expensive. The cost of the second predicate is much lower because equality condition is much cheaper. Since only 50,959 out of 143,692,928 rows satisfy the "promo_id = 98" condition, reversing the predicates means we don't have to check the next condition for those that fail the "promo_id = 98" condition.

This is verified when we execute the query using the two different cost models. Using the "CPU+IO" cost model in Oracle Database 10*g*, the predicates are reversed and the query execution is much faster.

Release	Elapsed Time (seconds)
Oracle9*i* Database (IO cost model)	70
Oracle Database 10*g* (CPU+IO cost model)	36

AUTO-STATISTICS

For the cost-based optimizer to function properly, object statistics are required to be collected using the ANALYZE command or the DBMS_STATS package. In Oracle Database 10*g*, the entire statistics collection process, including the initial and on going optimizer statistics collection, is fully automated. The fully automated statistics collection makes it easy to manage. More importantly, it would subsequently improve SQL execution performance.

In prior releases, the DBA has to collect optimizer statistics and has to keep them up-to-date as the data changes over time. Human error can cause the statistics to be missing, inaccurate (due to inadequate sample size), or stale; leading to poor SQL execution plans and poor performance. This is a non-trivial effort, especially with fairly large amount of objects and data. Fully automated statistics collection significantly reduces the chances of getting poor plans because of missing or stale statistics.

Oracle Database 10*g* automates statistics collection by analyzing objects with:

- low statistics confidence level,
- no statistics, and
- stale statistics.

Modification Monitoring mechanism is enabled by default to determine whether the statistics have become stale and to give weights for ordering so that the ones with the highest need are analyzed first. Global statistics are collected only when staleness of the tables as a whole reaches a pre-defined threshold.

Upon database creation, a GATHER_STATS_JOB is automatically generated. The Unified Scheduler in Oracle Database 10*g* schedules the GATHER_STATS_JOB according to the windows associated with MAINTENANCE_WINDOW_GROUP. By default, the MAINTENANCE_WINDOW_GROUP includes two windows: WEEKNIGHT_WINDOW and WEEKEND_WINDOW, with WEEKNIGHT_WINDOW corresponding to a window from 10PM to 6AM every night Monday through Friday, and WEEKEND_WINDOW corresponding to all day Saturday and Sunday. The scheduler schedules the GATHER_STATS_JOB for the first window that opens up. If the job did not get started during that window, the scheduler will reschedule the job for the next window.

When the schema is first created, the SALES table is loaded with sales data up to the end of year 2002, about 112 million rows. All other tables are loaded with relatively small number of rows. Because the tables are not specifically analyzed, no statistics are collected immediately after the load. Querying the table properties show that no statistics have been gathered for the tables.

```
SELECT table_name, num_rows, last_analyzed,
monitoring FROM user_tables;
```

TABLE_NAME	NUM_ROWS	LAST_ANALY	MON
SALES			YES
TIMES			YES
PRODUCTS			YES
CHANNELS			YES
PROMOTIONS			YES
CUSTOMERS			YES

Notice here that table monitoring is turned on automatically.

When the WEEKNIGHT_WINDOW opens up at 10PM, all these tables will be analyzed automatically.

```
SELECT table_name, num_rows, last_analyzed,
monitoring FROM user_tables;
```

TABLE_NAME	NUM_ROWS	LAST_ANALY	MON
SALES	112246776	2003-07-31	YES
TIMES	2191	2003-07-31	YES
PRODUCTS	10000	2003-07-31	YES
CHANNELS	5	2003-07-31	YES
PROMOTIONS	1001	2003-07-31	YES
CUSTOMERS	1000000	2003-07-31	YES

Suppose that on the second day, more data, mainly sales data for the year 2003, is loaded into the SALES table. These changes to the SALES table are captured by the Modification Monitoring mechanism. And as a result, the statistics on the SALES table will be recollected when the window opens up again at 10PM, while the statistics on the other tables remain unchanged.

```
SELECT table_name, num_rows, last_analyzed,
monitoring FROM user_tables;
```

TABLE_NAME	NUM_ROWS	LAST_ANALY	MON
SALES	149934895	2003-08-01	YES
TIMES	2191	2003-07-31	YES
PRODUCTS	10000	2003-07-31	YES
CHANNELS	5	2003-07-31	YES
PROMOTIONS	1001	2003-07-31	YES
CUSTOMERS	1000000	2003-07-31	YES

PARALLEL SINGLE CURSOR

In Oracle Database 10*g*, the parallel execution model for queries has moved from the slave SQL model to the parallel single cursor model (PSC). The PSC architecture lays the foundation for future parallel execution performance improvements by enabling new parallel operations and also yields immediate benefits in terms of performance and memory consumption.

In Oracle9*i* Database, a parallel plan is built by the Query Coordinator (QC). This plan has a SQL cursor corresponding to each DFO (Data Flow Object — i.e a subplan of the parallel plan which is scheduled to be run on a slave set). So, running a parallel query involves parsing, unparsing, and executing a parallel cursor on the QC side and then parsing and executing on a slave set, one cursor per DFO. In Oracle Database 10*g*, we build a single cursor that contains all the information needed for parallel execution and it is used for the entire parallel execution process.

The benefits of single cursor approach are -

- performance gains due to the parallelization of operations that used to run in serial:
  - explicit correlated subqueries in the WHERE/HAVING clause (which are not unnested, i.e. transformed, into joins) are now parallelized
  - queries with implicit or explicit correlated subqueries in the SELECT list are parallelized
- reduction in shared memory usage due to global parallel plan

Performance

Here is an example of a correlated subquery in the HAVING clause which now runs in parallel.

Select products and average price for products whose average sale price is at least 10% greater than the product's minimum sale price.

```
SELECT
      s.prod_id,
      AVG(s.amount_sold/s.quantity_sold)
      AVG_selling_price
FROM sales s
GROUP BY s.prod_id
HAVING EXISTS ( SELECT p.prod_id FROM products p
                WHERE s.prod_id = p.prod_id  AND
                AVG(s.amount_sold/s.quantity_sold)
> p.prod_min_price*1.10)
ORDER BY 1;
```

In Oracle9*i* Database, the SALES table is scanned in serial and the correlated subquery is evaluated on the QC for each row with a serial lookup for the

PRODUCTS table. In Oracle Database 10*g*, the SALES table is scanned in parallel and the correlated subquery is evaluated on each slave for a given row of sales that is scanned by that slave.

Release	Elapsed Time (seconds)
Oracle9*i* Database	660
Oracle Database 10*g*	90

Memory Reduction

In the Oracle9*i* Database execution model, each DFO is associated to a particular SQL cursor and slaves only execute cursors associated to its slave set. Therefore, there is one top-level parallel cursor and at least N distinct DFO cursors, where the parallel plan has been broken into N DFO's.

In Oracle Database 10*g*, a global parallel plan is built for a parallel cursor and all slaves on the same instance are usually able to share that cursor. In the PSC model, only one parallel cursor is compiled. Since only one cursor is compiled and shared by all the slaves on the same instance regardless of their slave set, the saving in shared memory is proportional to the number of DFO's. This implies that the savings will scale up as parallel queries get more complex and consequently have a larger number of DFO's.

Here is an example query with two DFO's:

```
SELECT p.prod_subcategory,
      SUM(s.quantity_sold) "Total Quantity"
FROM sales s, products p
WHERE s.prod_id = p.prod_id
      AND  s.time_id
      BETWEEN TO_DATE('01-jan-1998', 'dd-mon-
      yyyy')
      AND TO_DATE('31-dec-2003', 'dd-mon-yyyy')
GROUP BY p.prod_subcategory
ORDER BY SUM(s.quantity_sold);
```

In Oracle9*i* Database, this query would have used 5 cursors: one for the serial plan, two for each of the DFO's to access SALES and PRODUCTS table, and one each for group by and order by DFO's of the parallel plan. In Oracle Database 10*g*, we have one cursor for the entire query.

Release	Shared Memory Consumption (KBytes)
Oracle9*i* Database	60.3
Oracle Database 10*g*	19.3

PARTITION AWARE MV REFRESH

Oracle9*i* Database introduced Partition Change Tracking (PCT) materialized view refresh, which recognized and used partition operations to generate efficient refresh expressions.

In Oracle Database 10*g*, PCT refresh is enhanced to use dimensions and query rewrite. In addition to partition operations, functional dependencies defined in dimensions and foreign key relationships are also used to generate much faster refresh statements.

Data from Jan-1998 to Oct-2003, about 142 million rows, is loaded into the SALES table. The data is stored in a monthly fashion with partitioning by range on time_id. Several materialized views that pre-compute the data into monthly, quarterly, and yearly aggregates are created on the table. The SQL clauses used to create the views are:

```
CREATE MATERIALIZED VIEW MV_SALES_MONTHLY
PARTITION BY RANGE (calendar_month_id)
( /* one partition per month */
)
AS SELECT
      p.prod_id, c.cust_id, t.calendar_month_id,
      ch.channel_id, pr.promo_id,
      sum(fact.quantity_sold)quantity_sold,
      sum(fact.amount_sold)amount_sold
FROM
      product p, customer c, times t, channels ch,
      promotions pr, sales fact
WHERE
      fact.prod_id = p.prod_id and fact.cust_id =
      c.cust_id and fact.time_id = t.time_id and
      fact.channel_id = ch.channel_id and
      fact.promo_id = pr.promo_id
GROUP BY
      p.prod_id, c.cust_id, ch.channel_id,
      pr.promo_id, t.calendar_month_id;

CREATE MATERIALIZED VIEW MV_SALES_QUARTERLY
PARTITION BY RANGE (calendar_quarter_id)
( /* one partition per quarter */
)
AS SELECT
      p.prod_id, c.cust_id, t.calendar_quarter_id,
      ch.channel_id, pr.promo_id,
      sum(fact.quantity_sold)quantity_sold,
      sum(fact.amount_sold)amount_sold
FROM
      product p, customer c, times t, channels ch,
      promotions pr, sales fact
WHERE
      fact.prod_id = p.prod_id and fact.cust_id =
      c.cust_id and fact.time_id = t.time_id and
      fact.channel_id = ch.channel_id and
      fact.promo_id = pr.promo_id
GROUP BY
      p.prod_id, c.cust_id, ch.channel_id,
      pr.promo_id, t.calendar_quarter_id;
```

```
CREATE MATERIALIZED VIEW MV_SALES_ANNUALLY
PARTITION BY RANGE (calendar_year_id)
( /* one partition per year */
)
AS SELECT
      p.prod_id, c.cust_id, t.calendar_year_id,
      ch.channel_id, pr.promo_id,
      sum(fact.quantity_sold)quantity_sold,
      sum(fact.amount_sold)amount_sold
FROM
      product p, customer c, times t, channels ch,
      promotions pr, sales fact
WHERE
      fact.prod_id = p.prod_id and fact.cust_id =
      c.cust_id and fact.time_id = t.time_id and
      fact.channel_id = ch.channel_id and
      fact.promo_id = pr.promo_id
GROUP BY
      p.prod_id, c.cust_id, ch.channel_id,
      pr.promo_id, t.calendar_year_id
;
```

The following steps are performed to measure refresh performance:

- add two new partitions to the SALES table, Nov-2003 and Dec-2003
- load two months of new data into the SALES table, about 7.6 million rows
- refresh the materialized views using METHOD=>'?', requesting Oracle to find the most optimal refresh expressions:

```
dbms_mview.refresh_dependent(failures, 'SALES',
method =>'?', refresh_after_errors => TRUE,
atomic_refresh => FALSE);
```

Using Oracle Database 10*g*, we are able to provide fast refresh by

- computing only the two new months in MV_SALES_MONTHLY
- truncating and re-computing only the affected partition in MV_SALES_QUARTERLY(Q4-2003) and MV_SALES_ANNUALLY (2003)
- using query rewrite to recompute the quarterly data (Q4-2003) from the materialized view MV_SALES_MONTHLY, and the annual data (2003) from the materialized view MV_SALES_QUARTERLY

Using Oracle9*i* Database, we are unable to provide any form of incremental refresh due to the change in partitioning scheme of the SALES table.

Release	**Elapsed Time (seconds)**
Oracle9*i* Database (complete refresh)	2820
Oracle Database 10*g* (PCT refresh)	1500

PARTITION OUTER JOIN

OLAP users are used to seeing data densified along the time dimension. The lack of this feature means that OLAP densification queries are extremely awkward, using a combination of DISTINT, CROSS JOIN, and OUTER JOIN operations.

Oracle Database 10*g* introduces an extension to the ANSI join operator. Using "PARTITION OUTER JOIN", one can now express densification along the dimensions of interest. The syntax allows OUTER JOINS to be performed within a partition of a table.

Suppose we wish to compare the quarter-over-quarter sales of products. Not all products have results for all quarters, e.g. seasonal items. Gaps (or sparsity) in the time series make this comparison difficult. If we can fill in the gaps, or densify, the data along the time dimension, we can compare the quarterly results by referring from one row to another at a fixed distance.

We pre-aggregate 2 years of sales data into quarterly results, MV_SALES_8QTR, and we compare the differences in SQL expressions.

With ANSI SQL in Oracle9*i* Database, the statement would look like:

```
SELECT fact.prod_id, fact.cust_id,
      fact.channel_id, fact.promo_id,
      V1.calendar_quarter_id, fact.quantity_sold
FROM mv_sales_8qtr fact RIGHT OUTER JOIN
SELECT prod_id, cust_id, channel_id, promo_id,
      calendar_quarter_id
FROM
      (SELECT DISTINCT sales.prod_id,
            sales.cust_id, sales.channel_id,
            sales.promo_id) CROSS JOIN times_8qtr
      t
      )V1 ON
(fact.prod_id = V1.prod_id AND fact.cust_id =
V1.cust_id AND fact.channel_id = V1.channel_id AND
fact.promo_id = V1.promo_id AND
fact.calendar_quarter_id =
V1.calendar_quarter_id);
```

With the proposed ANSI extension in Oracle Database 10*g*, the following much simpler expression can be written:

```
SELECT fact.prod_id, fact.cust_id,
      fact.calendar_quarter_id, fact.channel_id,
      fact.promo_id, fact.quantity_sold
FROM mv_sales_8qtr fact
PARTITION BY
      (fact.prod_id, fact.cust_id,
      fact.channel_id, fact.promo_id)
RIGHT OUTER JOIN times_8qtr t ON
(fact.calendar_month_id = t.calendar_month_id);
```

A few rows from the above expression show the densification results:

Prod	Cust	QTR	Channel	Promo	Sold
40	34750	1998-Q1	8	286	
40	34750	1998-Q2	8	286	
40	34750	1998-Q3	8	286	
40	34750	1998-Q4	8	286	
40	34750	1999-Q1	8	286	
40	34750	1999-Q2	8	286	
40	34750	1999-Q3	8	286	14
40	34750	1999-Q4	8	286	

Densified results can then be processed using analytic functions such as LEAD and LAG.

Execution of the Oracle Database 10*g* expression is also much faster.

Release	Elapsed Time (seconds)
Oracle9*i* Database (ANSI)	505
Oracle Database 10*g* (ANSI extension)	152

NATIVE NUMBERS

Oracle Database 10*g* introduces two new data types for storing numbers which implement most of the IEEE 754 standard for Binary Floating Point Arithmetic - binary-float and binary-double. The benefits include better performance for arithmetic on numeric columns, potential for reduced storage for numeric columns, enhanced functionality, and ease of use.

The performance benefit of floating point data types increases as the precision used by the values increases. The cost of arithmetic operations using Oracle number data type scales linearly with the number of bytes used to store the values. The cost of arithmetic operations for floating point data type is relatively constant. Complex math functions are usually much faster for floating point data types than for Oracle number data type. Among the basic math operators, division and multiplication are most likely to be much faster for floating point data types.

Binary float offers precision for up to 6-9 decimal digits and binary-double offers precision up to 15-17 decimal digits. A floating-point number data type is represented in a binary system (one that uses base 2), as in the IEEE-754 standard, while Oracle number data type uses the decimal system (one that uses base 10). The base affects many properties of the format, including how a numeric value is rounded. Although the binary float data types have certain performance and storage advantages, due to the mathematical properties of this data type, binary float cannot be safely used in all applications. Binary floating-point is, therefore, suitable for applications that require numerically intensive work or mathematical analysis where performance is the main concern. It is not suitable for financial and commercial applications that cannot tolerate loss of precision due to truncation and approximations caused due to rounding. Unsuitable applications would include Banking, Order Entry and Retail Sales accounting. Suitable applications would include those where absolute precision is not required such as statistical analysis and many data mining applications.

Performance

The SALES table can be used to evaluate the number of items that are purchased per promotion. Because individual products have a wide range of quantity sold, the geometric mean is used to offset the extreme values and is an appropriate measure of the central tendency.

This can be implemented in two different ways:

- change in schema - define the column as a binary-float data type
- change the query SQL - use the TO_BINARY_FLOAT function to avoid changes to the schema or where the use of Oracle number data type is required (e.g. if the column we are performing computations on requires decimal accuracy or precision beyond that offered by the floating-point native data type)

In this example, quantity_sold can be an Oracle number or a binary-float data type.

```
SELECT promotion_name, EXP (geo_mean_temp/count)
AS geometric_mean FROM
    (SELECT p.promo_name AS promotion_name,
 SUM   (LN   (quantity_sold))   AS   geo_mean_temp,
COUNT(*) AS count
 FROM
 sales_native s,
 promotions p
 WHERE
 time_id BETWEEN TO_DATE ('01-jan-1998', 'dd-mon-
yyyy')
 AND   TO_DATE   ('31-dec-1998','dd-mon-yyyy')   AND
quantity_sold > 0
 AND p.promo_id = s.promo_id GROUP BY promo_name)
ORDER BY geometric_mean DESC;
```

To benefit from the native computation in Oracle Database 10*g*, it is also possible to modify the query to use the TO_BINARY_FLOAT conversion function when referencing the quantity_sold column as an Oracle number:

```
SUM (LN (TO_BINARY_FLOAT(quantity_sold))) AS
```

Release	Elapsed Time (seconds)
Oracle9*i* Database (Oracle numbers)	326
Oracle Database 10*g* (SQL change)	23
Oracle Database 10*g* (schema change)	12

Data Type and Storage

Oracle numbers are variable-width, requiring from 3 to 22 bytes. One byte is needed for the length, one for the sign and exponent, and the other 1 to 20 bytes for the mantissa. One byte is needed for each base 100 digit. Also, Oracle number data type uses an additional byte for negative values.

Values with trailing zeros use lesser storage as an Oracle number due to the decimal representation. In contrast, native number data types are fixed-width, the width is determined by the data type. Binary-float data type requires 5 bytes, one byte for length, 1 sign bit, 8 exponent bits and 23 significant bits, and the binary double data type requires 9,1,11 and 52 bits respectively.

An Oracle number data type will use 6 and 10 bytes on disk for the maximum amount of precision offered by binary float and binary double. In contrast, binary float and binary double use 5 and 9 bytes. Oracle numbers are variable width and native numbers are fixed width. Therefore, there are scenarios where Oracle numbers occupy less bytes on disk as compared to native numbers.

CONCLUSION

Data warehouses today need improved performance as well as reduced manageability cost. To address these requirements, Oracle Database 10*g* delivers new features and enhancements whose performance benefits are available automatically, without needing large deployment resources from the data warehouse team. By upgrading to Oracle Database 10*g*, data warehouses can automatically benefit from the performance improvements while focusing on the delivery of new business functionalities.

DSS Performance in Oracle Database 10*g*
September 2003
Author: Susy Fan
Contributing Authors: Linan Jiang, George Lumpkin, Saroj Sancheti

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com

Oracle is a registered trademark of Oracle Corporation. Various product and service names referenced herein may be trademarks of Oracle Corporation. All other product and service names mentioned may be trademarks of their respective owners.

Copyright © 2001 Oracle Corporation
All rights reserved.