An Oracle White Paper
November 2010

Best Practices for Implementing a Data Warehouse on the Oracle Exadata Database Machine

ORACLE®

Introduction

Increasingly companies are recognizing the value of an enterprise data warehouse (EDW). A true EDW provides a single 360-degree view of the business and a powerful platform for a wide spectrum of business intelligence tasks ranging from predictive analysis to near real-time strategic and tactical decision support throughout the organization. Ensuring the EDW will get the desired performance and will scale out as your data grows you need to get three fundamental things correct, the hardware configuration, the physical data model and the data loading process. By correctly designing these three corner stones you will be able to create an EDW that can seamlessly scale without constant tuning or tweaking of the system.

By using the Oracle Exadata Database Machine as your data warehouse platform you have a balanced, high performance hardware configuration. This paper focuses on the other two corner stones, data modeling and data loading, providing a set of best practices and examples for deploying a data warehouse on the Oracle Exadata Database Machine.

The paper is divided into four sections:

The first briefly describes the two fundamental **data models** used for database warehouses.

The remaining three sections explain **how to implement** these models in the most optimal manner in an Oracle database and provide detailed information on how to achieve optimal **data loading** performance.

Data Models for a Data Warehouse

A data model is an essential part of the development process for a data warehouse. It allows you to define the types of information needed in the data warehouse to answer the business questions and the logical relationships between different parts of the information. It should be simple, easily understood and have no regard for the physical database, the hardware that will be used to run the system or the tools that end users will use to access it. There are two classic models used for data warehouse, Third Normal Form and dimensional or Star (Snowflake) Schema.

Third Normal Form (3NF) is a classical relational-database modeling technique that minimizes data redundancy through normalization. A 3NF schema is a neutral schema design independent of any application, and typically has a large number of tables. It preserves a detailed record of each transaction without any data redundancy and allows for rich encoding of attributes and all relationships between data elements. Users typically require a solid understanding of the data in order to navigate the more elaborate structure reliably.

The **Star Schema** is so called because the diagram resembles a star, with points radiating from a center. The center of the star consists of one or more fact tables and the points of the star are the dimension tables.
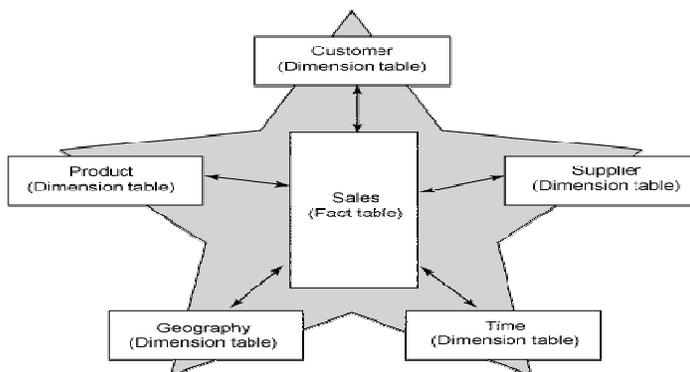

Figure 1 Star Schema - one or more fact tables surrounded by multiple dimension tables

Fact tables are the large tables that store business measurements and typically have foreign keys to the dimension tables. Dimension tables, also known as lookup or reference tables; contain the relatively static or descriptive data in the data warehouse. The Star Schema borders on a physical model, as drill paths, hierarchy and query profile are embedded in the data model itself rather than the data. This in part at least, is what makes navigation of the model so straightforward for end users. Snowflake schemas are slight variants of a simple star schema where the dimension tables are further normalized and broken down into multiple tables. The snowflake aspect only affects the dimensions and not the fact table and is therefore considered conceptually equivalent to star schemas and will not be discussed separately in this paper.

There is often much discussion regarding the 'best' modeling approach to take for any given Data Warehouse with each style, classic 3NF and dimensional having their own strengths and weaknesses. It is likely that data warehouses will need to do more to embrace the benefits of each model type rather than rely on just one - this is the approach that Oracle adopts in our Data Warehouse Reference Architecture[1]. This is also true for the majority of Oracle's customers who use a mixture of both model forms. Most important is for you to design your model according to your specific business needs.

Physical Model – Implementing the Logical Model

The starting point for the physical model is the logical model. The physical model should mirror the logical model as much as possible, although some changes in the structure of the tables and / or columns may be necessary. In addition the physical model will include staging or maintenance tables that are usually not included in the logical model. **Figure 2** below shows a blue print of the physical layers we define in our DW Reference Architecture and see in many data warehouse environments. Although your environment may not have such clearly defined layers you should have some aspects of each layer in your database to ensure it will continue to scale as it increases in size and complexity.
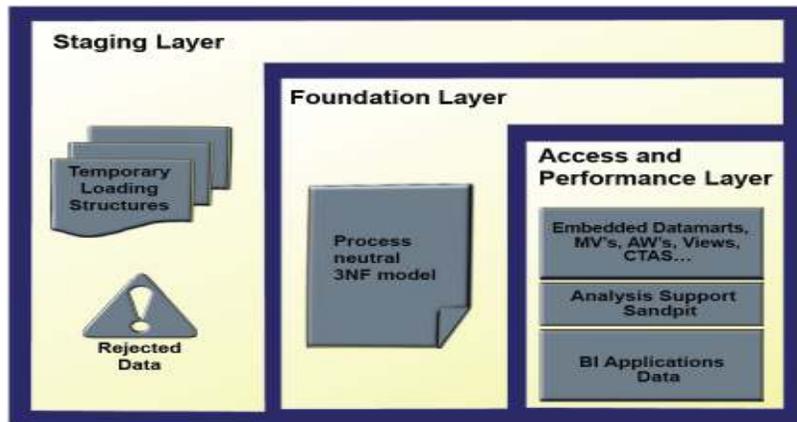

Figure 2 Physical layers of a Data Warehouse

Staging layer

The staging layer enables the speedy extraction, transformation and loading (ETL) of data from your operational systems into the data warehouse without impacting the business users Much of the complex data transformation and data-quality processing will occur in this layer. The tables in the staging layer are normally segregated from the query part of the data warehouse. The most basic approach for the staging layer is to have it be an identical schema to the one that exists in the source operational system(s) but with some structural changes to the tables, such as range partitioning. It is also possible that in some implementations this layer is not necessary, as all data transformation processing will be done "on the fly" as data is extracted from the source system before it is inserted directly into the Foundation Layer.

Efficient Data Loading

Whether you are loading into a staging layer or directly into the foundation layer the goal is to get the data into the warehouse in the most expedient manner. In order to achieve good performance during the load you need to begin by focusing on where the data to-being-loaded resides and how you load it into the database. For example, you should not use a serial database link or a single JDBC connection to move large volumes of data. Flat files are the most common and preferred mechanism of large volumes of data.

Staging Area

The area where flat files are stored prior to being loaded into the staging layer of a data warehouse system is commonly known as staging area. The overall speed of your load will be determined by (A) how quickly the raw data can be read from staging area and (B) how fast it can be processed and inserted into the database. It is highly recommended that you stage the raw data across as many physical disks as possible to ensure the reading it is not a bottleneck during the load.

Using the Oracle Exadata Database Machine the best place to stage the data is in an Oracle Database File System (DBFS) stored on the Exadata storage cells. DBFS creates a mountable distributed file system which can be used to access files stored in the database. It is recommended that you create the DBFS in a separate database on the Database Machine. This will allow for the DBFS to be managed and maintained separately from the Data Warehouse. The file system should also be mounted using the direct_io option to avoid thrashing the system page cache while moving the raw data files in and out of the file system. More information on setting up DBFS can be found in the Oracle® Database SecureFiles and Large Objects Developer's Guide.

Preparing the raw data files

In order to parallelize the data load Oracle needs to be able to logically break up the raw data files into chunks, known as granules. To ensure balanced parallel processing, the number of

granules is typically much higher than the number of parallel server processes. At any given point in time, a parallel server process is allocated one granule to work on; once a parallel server process completes working on its granule another granule will be allocated until all of the granules have been processed and the data is loaded.

In order to create multiple granules within a single file, Oracle needs to be able to look inside the raw data file and determine where each row of data begins and ends. This is only possible if each row has been clearly delimited by a known character such as new line or a semicolon.

If a file is not position-able and seek-able, for example the file is compressed (.zip file), then the files cannot be broken up into granules and the whole file is treated as a single granule. Only one parallel server process will be able to work on the entire file.

In order to parallelize the loading of compressed data files you need to use multiple compressed data files. The number of compressed data files used will determine the maximum parallel degree used by the load.

When loading multiple data files (compressed or uncompressed) via a single external table it is recommended that the files are similar in size and that their sizes should be a multiple of 10MB. If different size files have to be used it is recommend that the files are listed from largest to smallest.

By default, Oracle assumes that the flat file has the same character set as the database. If this is not the case you should specify the character set of the flat file in the external table definition to ensure the proper character set conversions can take place.

External Tables

Oracle offers several data loading options

- External table or SQL*Loader

- Oracle Data Pump (import & export)

- Change Data Capture and Trickle feed mechanisms (such as Oracle GoldenGate)

- Oracle Database Gateways to open systems and mainframes

- Generic Connectivity (ODBC and JDBC)

Which approach should you take? Obviously this will depend on the source and format of the data you receive. As mentioned earlier, flat files are the most common mechanism for load large volumes of data, so this paper will only focus on the loading of data from flat files.

If you are loading from files into Oracle you have two options, SQL*Loader or external tables. Oracle strongly recommends that you load using external tables rather than SQL*Loader:

- Unlike SQL*Loader, external tables allows transparent parallelization inside the database.

- You can avoid staging data and apply transformations directly on the file data using arbitrary SQL or PL/SQL constructs when accessing external tables. SQL Loader requires you to load the data as-is into the database first.

- Parallelizing loads with external tables enables a more efficient space management compared to SQL*Loader, where each individual parallel loader is an independent database sessions with its own transaction. For highly partitioned tables this could potentially lead to a lot of wasted space.

An external table is created using the standard CREATE TABLE syntax except it requires additional information about the flat files residing outside the database. The following SQL command creates an external table for the flat file 'sales_data_for_january.dat'.

```
SH@DBM1 > CREATE TABLE ext_tab_for_sales_data(
  2      Price      NUMBER(6),
  3      Quantity  NUMBER(6),
  4      Time_id    DATE)
  5   ORGANIZATION EXTERNAL(
  6   type ORACLE_LOADER
  7   default directory data_dir
  8   access parameters
  9   (   RECORDS DELIMITED BY newline
 10       BADFILE 'ulcase1.bad'
 11       LOGFILE 'ulcase1.log'
 12       FIELDS TERMINATED BY ",")
 13       LOCATION ('sales_data_for_january.dat'))
 14   parallel
 15   REJECT LIMIT UNLIMITED;
```

The most common approach when loading data from an external table is to do a CREATE TABLE AS SELECT (CTAS) statement or an INSERT AS SELECT (IAS) statement into an existing table. For example the simple SQL statement below will insert all of the rows in a flat file into partition p2 of the Sales fact
table.

```
SH@DBM1 > Insert into Sales partition(p2) Select * From ext_tab_for_sales_data
```

Parallel Direct Path Load

The key to good load performance is to use direct path loads wherever possible. A direct path load parses the input data according to the description given in the external table definition, converts the data for each input field to its corresponding Oracle data type, then builds a column array structure for the data. These column array structures are used to format Oracle data blocks and build index keys. The newly formatted database blocks are then written directly to the database, bypassing the standard SQL processing engine and the database buffer cache.

A CTAS will always use direct path load but IAS statement will not. In order to achieve direct path load with an IAS statement you must add the APPEND hint to the command.

```
[SH@DBM1 > Insert /*+ APPEND */ into Sales partition(p2) Select * From ext_tab_for_sales_data;
```

Direct path loads can also run in parallel. You can set the parallel degree for a direct path load either by adding the PARALLEL hint to the CTAS or IAS statement or by setting the PARALLEL clause on **both** the external table and the table into which the data will be loaded. Once the parallel degree has been set at CTAS will automatically do direct path load in parallel but an IAS will not. In order to enable an IAS to do direct path load in parallel you must alter the session to enable parallel DML.

```
SH@DBM1 > Alter Session Enable Parallel DML;

Session altered.

SH@DBM1 >  Insert /*+ APPEND */ into Sales partition(p2) Select * From ext_tab_for_sales_data;
```

Partition exchange loads

It is strongly recommended that the larger tables or fact tables in a data warehouse are partitioned. One of the benefits of partitioning is the ability to load data quickly and easily with minimal impact on the business users by using the exchange partition command. The exchange partition command allows you to swap the data in a non-partitioned table into a particular partition in your partitioned table. The command does not physically move data; instead it updates the data dictionary to exchange a pointer from the partition to the table and vice versa. Because there is no physical movement of data, an exchange does not generate redo and undo, making it a sub-second operation and far less likely to impact performance than any traditional data-movement approaches such as INSERT.

Let's assume we have a large table called Sales, which is range partitioned by day. At the end of each business day, data from the online sales system is loaded into the Sales table in the warehouse. The five simple steps shown in *Figure 3* ensure the daily data gets loaded into the correct partition with minimal impact to the business users of the data warehouse and optimal speed.
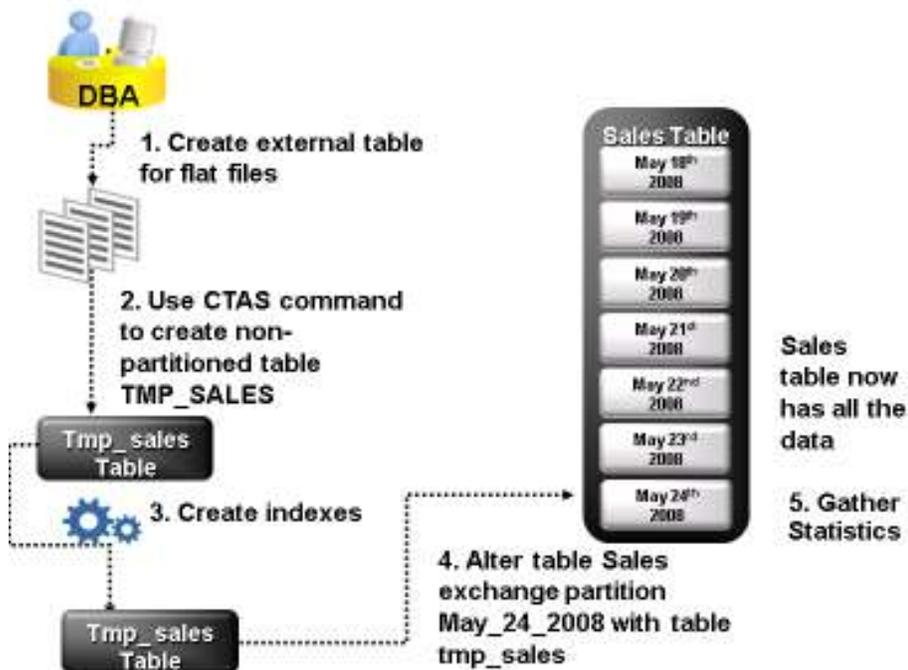
Figure 3 Partition exchange load

Partition exchange load steps

1. Create external table for the flat file data coming from the online system

2. Using a CTAS statement, create a non-partitioned table called tmp_sales that has the same column structure as Sales table

3. Build any indexes that are on the Sales table on the tmp_sales table

4. Issue the exchange partition command

```
SH@DBM1 > Alter table Sales exchange partition p2 with
  2              table tmp_sales including indexes without validation;
```

1. Gather optimizer statistics on the newly exchanged partition using incremental statistics, as described in the second paper in this series Best Practices for Workload management in a Data Warehouse on the Oracle Exadata Database Machine.

The exchange partition command in the fourth step above, swaps the definitions of the named partition and the tmp_sales table, so the data instantaneously exists in the right place in the partitioned table. Moreover, with the inclusion of the two optional extra clauses, index

definitions will be swapped and Oracle will not check whether the data actually belongs in the partition - so the exchange is very quick. The assumption being made here is that the data integrity was verified at date extraction time. If you are unsure about the data integrity then do not use the WITHOUT VALIDATION clause; the database will then check the validity of the data.

Data Compression

Another key decision that you need to make during the load phase is whether or not to compress your data. Using table compression obviously reduces disk and memory usage, often resulting in better scale-up performance for read-only operations. Table compression can also speed up query execution by minimizing the number of round trips required to retrieve data from the disks. Compressing data however imposes a performance penalty on the load speed of the data. The overall performance gain typically outweighs the cost of compression.

Oracle offers three types of compression on the Oracle Exadata Database Machine; basic compression, OLTP compression (component of the Advanced Compression option), and Exadata Hybrid Columnar Compression (EHCC).

With standard compression Oracle compresses data by eliminating duplicate values in a database block. Standard compression only works for direct path operations (CTAS or IAS, as discussed earlier). If the data is modified using any kind of conventional DML operation (for example updates), the data within that database block will be uncompressed to make the modifications and will be written back to disk uncompressed.

With OLTP compression, just like standard compression, Oracle compresses data by eliminating duplicate values in a database block. But unlike standard compression OLTP compression allows data to remain compressed during all types of data manipulation operations, including conventional DML such as INSERT and UPDATE. More information on the OLTP table compression features can be found in the Oracle® Database Administrator's Guide 11g.

Exadata Hybrid Columnar Compression (EHCC) achieves its compression using a different compression technique. A logical construct called the compression unit is used to store a set of Exadata Hybrid Columnar-compressed rows. When data is loaded, a set of rows is pivoted into a columnar representation and compressed. After the column data for a set of rows has been compressed, it is fit into the compression unit. If conventional DML is issued against a table with EHCC, the necessary data is uncompressed in order to do the modification and then written back to disk using a block-level compression algorithm. If your data set is frequently modified using conventional DML EHCC is not recommended, instead the use of OLTP compression is recommended.

EHCC provides different levels of compression, focusing on query performance or compression ratio respectively. With EHCC optimized for query, less compression algorithms are applied to the data to achieve good compression with little to no performance impact. Compression for

archive on the other hand tries to optimize the compression on disk, irrespective of its potential impact on the query performance. See the Oracle Exadata documentation for further details.

If you decide to use compression, consider sorting your data before loading it to achieve the best possible compression rate. The easiest way to sort incoming data is to load it using an ORDER BY clause on either your CTAS or IAS statement. You should ORDER BY a NOT NULL column (ideally non numeric) that has a large number of distinct values (1,000 to 10,000).

Foundation layer - Third Normal Form

From staging, the data will transition into the foundation or integration layer via another set of ETL processes. Data begins to take shape and it is not uncommon to have some end-user application access data from this layer especially if they are time sensitive, as data will become available here before it is transformed into the dimension / performance layer. Traditionally this layer is implemented in the Third Normal Form (3NF).

Optimizing 3NF

Optimizing a 3NF schema in Oracle requires the three Ps – Power, Partitioning and Parallel Execution. Power means that the hardware configuration must be balanced, which is the case on the Oracle Exadata Database Machine. The larger tables should be partitioned using composite partitioning (range-hash or list-hash). There are three reasons for this:

1. Easier manageability of terabytes of data

2. Faster accessibility to the necessary data

3. Efficient and performant table joins

Finally Parallel Execution enables a database task to be parallelized or divided into smaller units of work, thus allowing multiple processes to work concurrently. By using parallelism, a terabyte of data can be scanned and processed in minutes or less, not hours or days. Parallel execution will be discussed in more detail in the system management section below.

Partitioning

Partitioning allows a table, index or index-organized table to be subdivided into smaller pieces. Each piece of the database object is called a partition. Each partition has its own name, and may optionally have its own storage characteristics. From the perspective of a database administrator, a partitioned object has multiple pieces that can be managed either collectively or individually.

This gives the administrator considerable flexibility in managing partitioned object. However, from the perspective of the application, a partitioned table is identical to a non-partitioned table; no modifications are necessary when accessing a partitioned table using SQL DML commands. Partitioning can provide tremendous benefits to a wide variety of applications by improving manageability, availability, and performance.

Partitioning for manageability

Range partitioning will help improve the manageability and availability of large volumes of data. Consider the case where two year's worth of sales data or 100 terabytes (TB) is stored in a table. At the end of each day a new batch of data needs to be too loaded into the table and the oldest days worth of data needs to be removed. If the Sales table is ranged partitioned by day the new data can be loaded using a partition exchange load as described above. This is a sub-second operation and should have little or no impact to end user queries. In order to remove the oldest day of data simply issue the following command:

```
SH@DBM1 > Alter table Sales drop partition Sales_Q4_2009;
```

Partitioning for easier data access

Range partitioning will also help ensure only the necessary data to answer a query will be scan. Let's assume that the business users predominately accesses the sales data on a weekly basis, e.g. total sales per week then range partitioning this table by day will ensure that the data is accessed in the most efficient manner, as only 7 partition needs to be scanned to answer the business users query instead of the entire table. The ability to avoid scanning irrelevant partitions is known as *partition pruning.*
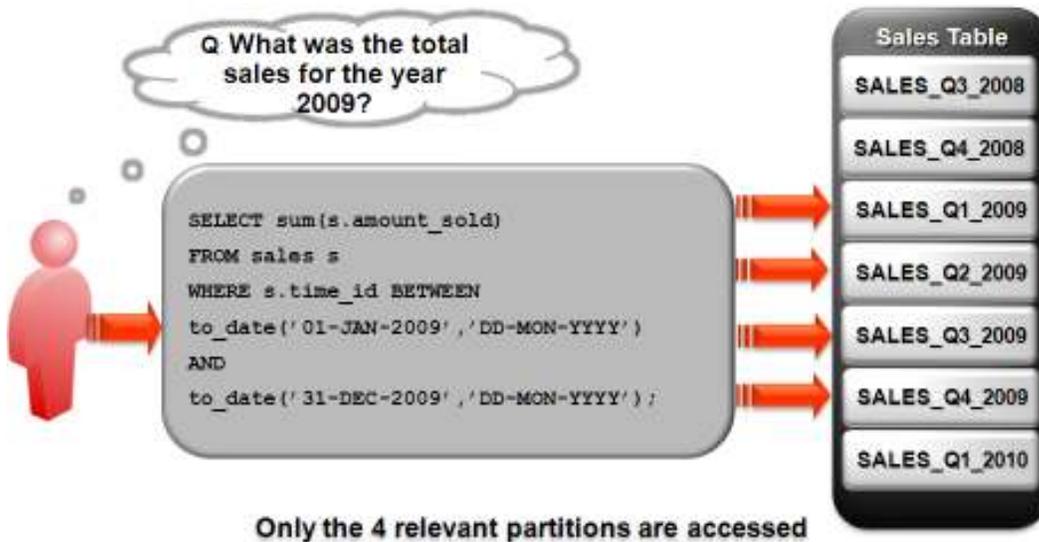

Figure 4 Partition pruning: only the relevant partition is accessed

Partitioning for join performance

Sub-partitioning by hash is used predominately for performance reasons. Oracle uses a linear hashing algorithm to create sub-partitions. In order to ensure that the data gets evenly distributed

among the hash partitions it is highly recommended that the number of hash partitions is a power of 2 (for example, 2, 4, 8, etc). Each hash partition should be at least 16MB in size. Any small and they will not have efficient scan rates with parallel query.

One of the main performance benefits of hash partitioning is partition-wise joins. Partition-wise joins reduce query response time by minimizing the amount of data exchanged among parallel execution servers when joins execute in parallel. This significantly reduces response time and improves both CPU and memory resource usage. In a clustered data warehouse, this significantly reduces response times by limiting the data traffic over the interconnect (IPC), which is the key to achieving good scalability for massive join operations. Partition-wise joins can be full or partial, depending on the partitioning scheme of the tables to be joined.

A full partition-wise join divides a join between two large tables into multiple smaller joins. Each smaller join, performs a joins on a pair of partitions, one for each of the tables being joined. For the optimizer to choose the full partition-wise join method, both tables must be equi-partitioned on their join keys. That is, they have to be partitioned on the same column with the same partitioning method. Parallel execution of a full partition-wise join is similar to its serial execution, except that instead of joining one partition pair at a time, multiple partition pairs are joined in parallel by multiple parallel query servers. The number of partitions joined in parallel is determined by the Degree of Parallelism (DOP).
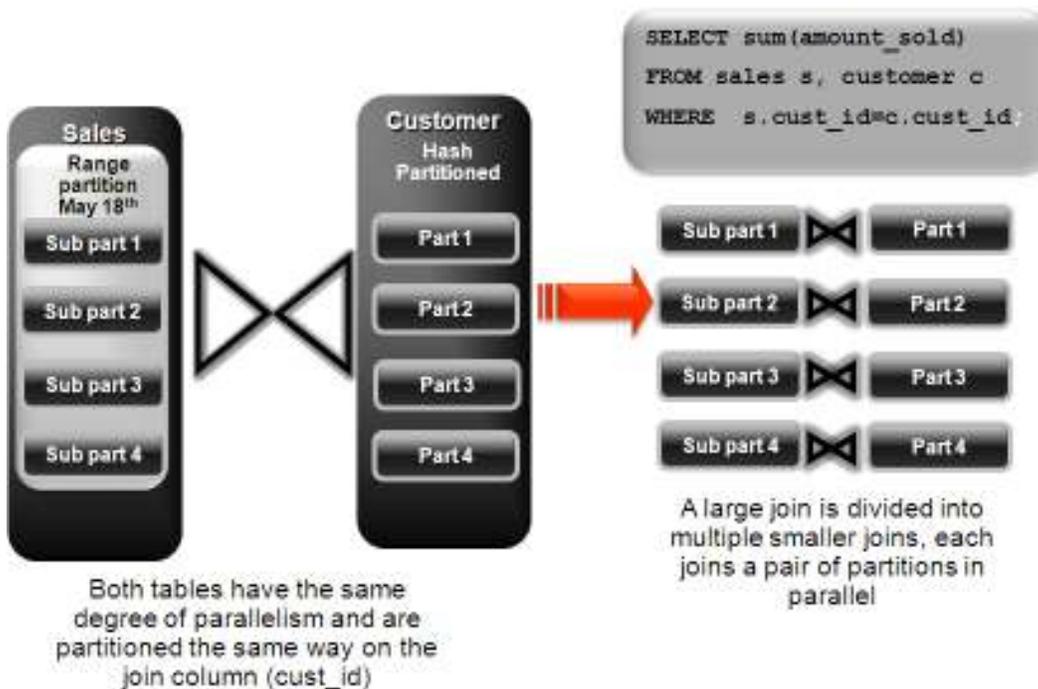

Figure 5 Full Partition-Wise Join

Figure 5 illustrates the parallel execution of a full partition-wise join between two tables, Sales and Customers. Both tables have the same degree of parallelism and the same number of partitions. They are range partitioned on a date field and sub partitioned by hash on the cust_id field. As illustrated in the picture, each partition pair is read from the database and joined directly. There is no data redistribution necessary, thus minimizing IPC communication, especially across nodes. *Figure 6* shows the execution plan you would see for this join.

Partition Hash All above the join &
single PQ set indicate partition-wise join

ID	Operation	Name	Pstart	Pstop	TQ	PQ Distrib
0	SELECT STATEMENT					
1	PX COORDINATOR					
2	PX SEND QC (RANDOM)	:TQ10001			Q1,01	QC (RAND)
3	SORT GROUP BY				Q1,01	
4	PX RECEIVE				Q1,01	
5	PX SEND HASH	:TQ10000			Q1,00	HASH
6	SORT GROUP BY				Q1,00	
7	PX PARTITION HASH ALL		1	128	Q1,00	
8	HASH JOIN				Q1,00	
9	TABLE ACCESS FULL	Customers	1	128	Q1,00	
10	TABLE ACCESS FULL	Sales	1	128	Q1,00	

Figure 6 Execution plan for Full Partition-Wise Join

To ensure that you get optimal performance when executing a partition-wise join in parallel, the number of partitions in each of the tables should be larger than the degree of parallelism used for the join. If there are more partitions than parallel servers, each parallel server will be given one pair of partitions to join, when the parallel server completes that join, it will requests another pair of partitions to join. This process repeats until all pairs have been processed. This method enables the load to be balanced dynamically (for example, 128 partitions with a degree of parallelism of 32).

What happens if only one of the tables you are joining is partitioned? In this case the optimizer could pick a partial partition-wise join. Unlike full partition-wise joins, partial partition-wise joins can be applied if only one table is partitioned on the join key. Hence, partial partition-wise joins are more common than full partition-wise joins. To execute a partial partition-wise join, Oracle dynamically repartitions the other table based on the partitioning strategy of the partitioned table. Once the other table is repartitioned, the execution is similar to a full partition-wise join. The redistribution operation involves exchanging rows between parallel execution servers. This operation leads to interconnect traffic in RAC environments, since data needs to be repartitioned across node boundaries.
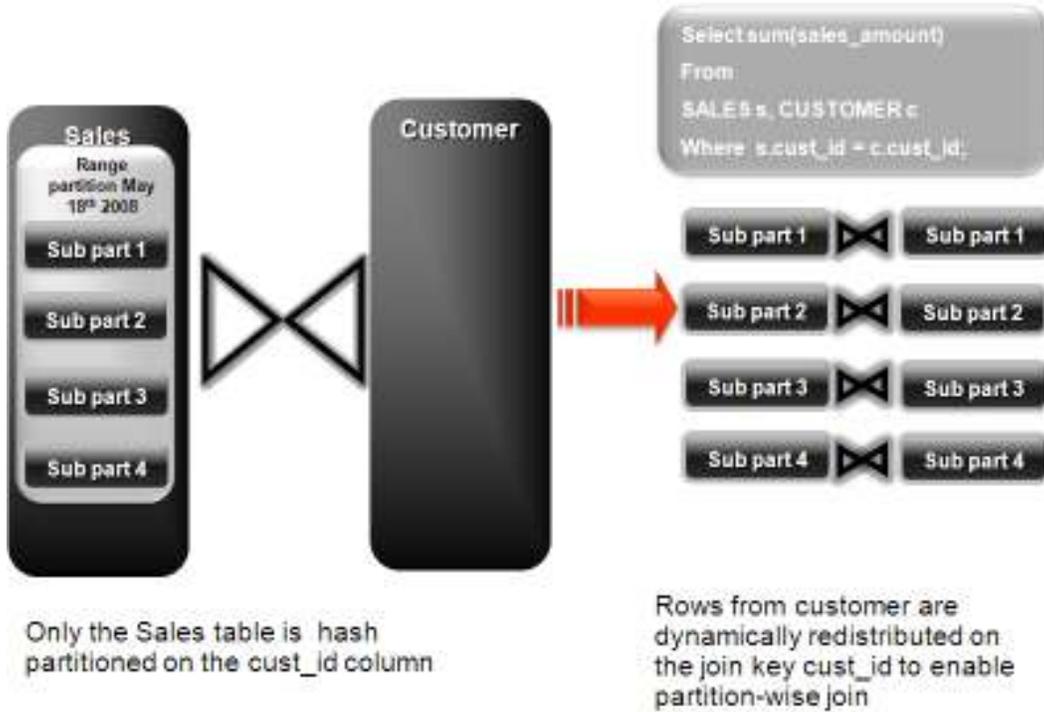
Figure 7 Partial Partition-Wise Join

Figure 7 illustrates a partial partition-wise join. It uses the same example as in ***Figure 5***, except that the customer table is not partitioned. Before the join operation is executed, the rows from the customers table are dynamically redistributed on the join key.

Access layer - Star Schema

The access layer represents data, which is in a form that most users and applications can understand. It is in this layer you are most likely to see a star schema.
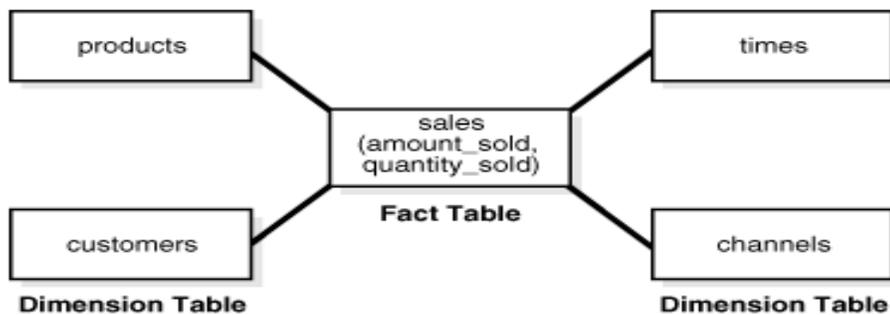
Figure 8 Star Schema - one or more fact tables surrounded by multiple dimension tables

A typical query in the access layer will be a join between the fact table and some number of dimension tables and is often referred to as a star query. In a star query each dimension table will be joined to the fact table using a primary key to foreign key join. Normally the dimension tables don't join to each other. A business question that could be asked against the star schema in **Figure 8** would be "What was the total number of umbrellas sold in Boston during the month of May 2008?" The resulting SQL query for this question is shown in **Figure 9**.
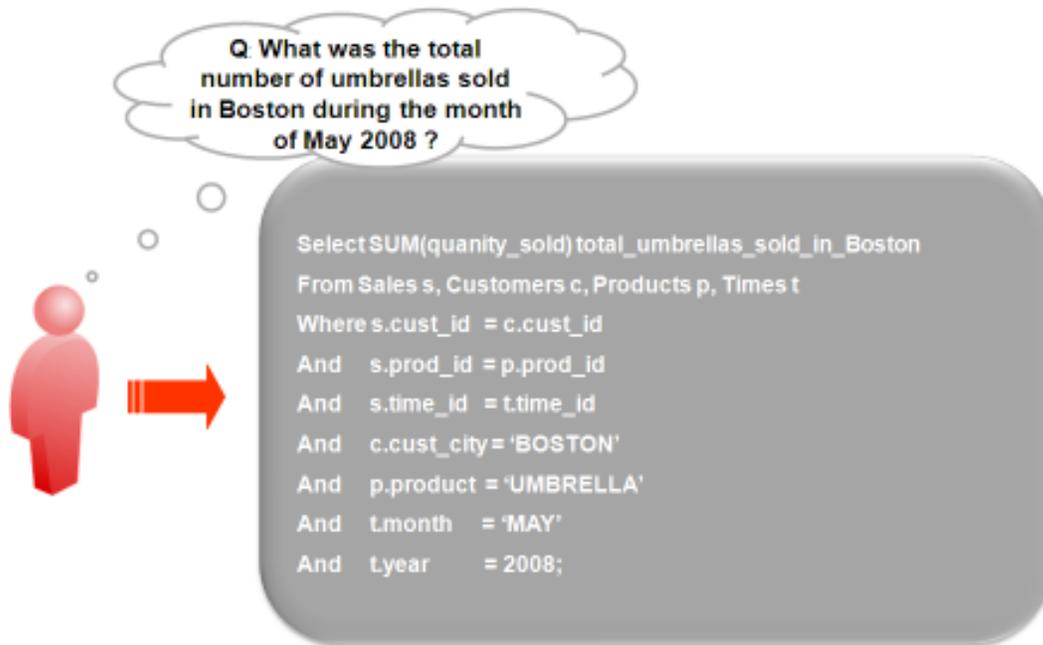
Figure 9 Typical Star Query with all where clause predicates on the dimension tables

As you can see all of the where clause predicates are on the dimension tables and the fact table (Sales) is joined to each of the dimensions using their foreign key, primary key relationship. So, how do you go about optimizing for this style of query?

Optimizing Star Queries

Tuning a star query is very straight forward. The two most important criteria are;

- Create a bitmap index on each of the foreign key columns in the fact table or tables

- Set the initialization parameter STAR_TRANSFORMATION_ENABLED to TRUE. This will enable the optimizer feature for star queries which is off by default for backward compatibility.

If your environment meets these two criteria your star queries should use a powerful optimization technique that will rewrite or transform you SQL called star transformation. Star transformation executes the query in two phases, the first phase retrieves the necessary rows from the fact table (row set) while the second phase joins this row set to the dimension tables.

The rows from the fact table are retrieved by using bitmap joins between the bitmap indexes on all of the foreign key columns. The end user never needs to know any of the details of STAR_TRANSFORMATION, as the optimizer will automatically choose STAR_TRANSFORMATION when its appropriate.

But how exactly will STAR_TRANSFORMATION effect or rewrite our star-query in **Figure 9**. As mentioned above, the query will be processed in two phases. In the first phase Oracle will transform or rewrite our query so that each of the joins to the fact table is rewritten as sub-queries. You can see exactly how the rewritten query looks in **Figure 10.** By rewriting the query in this fashion we can now leverage the strengths of bitmap indexes. Bitmap indexes provide set-based processing within the database, allowing us to use very fact methods for set operations such as AND, OR, MINUS and COUNT. So, we will use the bitmap index on time_id to identify the set of rows in the fact table corresponding to sales in May 2008. In the bitmap the set of rows will actually be represented as a string of 1's and 0's. A similar bitmap is retrieved for the fact table rows corresponding to the sale of umbrellas and another is accessed for sales made in Boston. At this point we have three bitmaps, each representing a set of rows in the fact table that satisfy an individual dimension constraint. The three bitmaps are then combined using a bitmap AND operation and this newly created final bitmap is used to extract the rows from the fact table needed to evaluate the query.
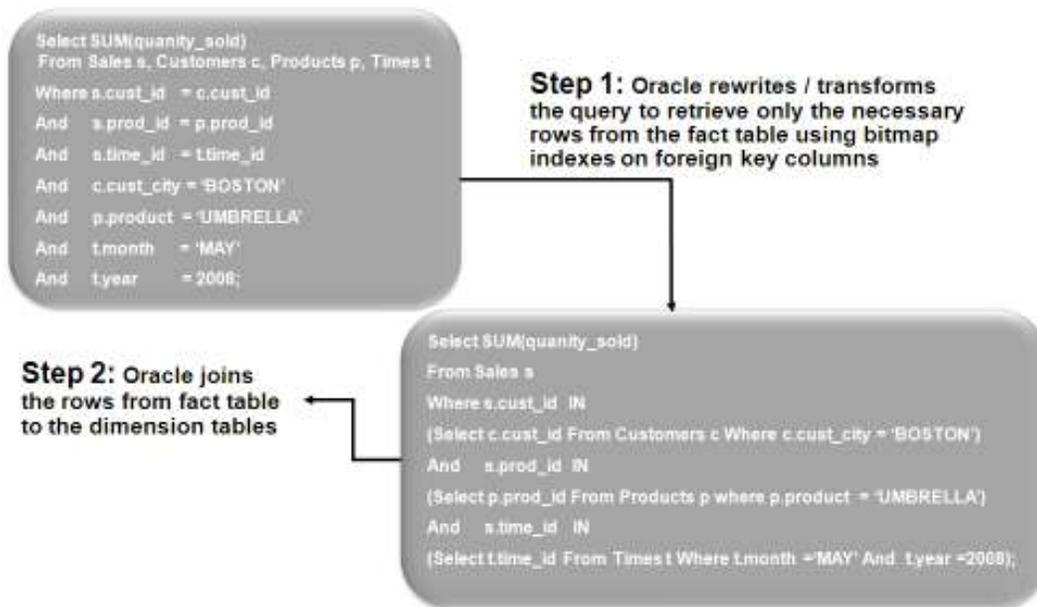

Figure 10 Star Transformation is a two-phase process

The second phase is to join the rows from the fact table to the dimesion tables. The join back to the dimension tables are normally done using a hash join but the Oracle Optimizer will select the most efficient join method depending on the size of the dimension tables.

Figure 11 shows the typical execution plan for a star query where STAR_TRANSFORMATION has kicked in. The execution plan may not look exactly how you imagined it. You may have noticed that we do not join back to the customer table after the rows have been sucessfully retrieved from the Sales table. If you look closely at the select list we don't actually select anything from the Customers table so the optimizer knows not to bother joining back to that dimension table. You may also notice that for some queries even if STAR_TRANFORMATION does kick in it may not use all of the bitmap indexes on the fact table. The optimizer decides how many of the bitmap indexes are required to retrieve the necessary rows from the fact table. If an additional bitmap indexes will not improve the selectivity the optimizer will not use it. The only time you will see the dimension table that corresponds to the excludued bitmap in the execution plan will be during the second phase or the join back phase.

ID	Operation	Name	Rows	Pstart	Pstop	
0	SELECT STATEMENT		1			
1	SORT GROUP BY NO SORT		1			Phase 2
2	HASH JOIN		3			
3	TABLE ACCESS FULL	PRODUCTS	2			
4	HASH JOIN		1			
5	TABLE ACCESS FULL	TIMES	1			
6	PARTITION RANGE SUBQUERY		44144	1	16	
7	TABLE ACCESS BY LOCAL INDEX ROWID	SALES	44144	1	16	
8	BITMAP CONVERSION TO ROWIDS					Phase 1
9	BITMAP AND					
10	BITMAP MERGE					
11	BITMAP KEY ITERATION					
12	BUFFER SORT					
13	TABLE ACCESS FULL	TIMES	1			
14	BITMAP INDEX RANGE SCAN	SALES_TIME_BIX		1	16	
15	BITMAP MERGE					
16	BITMAP KEY ITERATION					
17	BUFFER SORT					
18	TABLE ACCESS FULL	CUSTOMERS	1			
19	BITMAP INDEX RANGE SCAN	SALES_CUST_BIX		1	16	
20	BITMAP MERGE					
21	BITMAP KEY ITERATION					
22	BUFFER SORT					
23	TABLE ACCESS FULL	PRODUCTS	2			
24	BITMAP INDEX RANGE SCAN	SALES_PROD_BIX		1	16	

Figure 11 Typical Star Query execution plan

Summary

In order to guarantee you will get the optimal performance from your data warehouse and to ensure it will scale out as the data increases you need to get the following three fundamental things right:

- The hardware configuration. It must be balanced and must achieve the necessary IO throughput required to meet the systems peak load

- The data model. If it is a 3NF it should always achieve partition-wise joins or if it's a Star Schema it should use star transformation

- The data loading process. It should be as fast as possible and have zero impact on the business user

By selecting a Oracle Exadata Database Machine you can ensure the hardware configuration will perform. Following the best practices for deploying a data warehouse outlined in this paper will allow you to seamlessly scale out your EDW without having to constantly tune or tweak the system.

ORACLE®

Oracle is committed to developing practices and products that help protect the environment