Rakamin Academy

HOME CREDIT

Virtual Internship Experience

Big Data

Data Warehouse Logical Design

Logical Design

Your organization has decided to build an enterprise data warehouse. You have defined the business requirements and agreed upon the scope of your business goals, and created a conceptual design. Now you need to translate your requirements into a system deliverable. To do so, you create the logical and physical design for the data warehouse. You then define:

- The specific data content
- Relationships within and between groups of data
- The system environment supporting your data warehouse
- The data transformations required
- The frequency with which data is refreshed

The logical design is more conceptual and abstract than the physical design. In the logical design, you look at the logical relationships among the objects. In the physical design, you look at the most effective way of storing and retrieving the objects as well as handling them from a transportation and backup/recovery perspective.

Orient your design toward the needs of the end users. End users typically want to perform analysis and look at aggregated data, rather than at individual transactions. However, end users might not know what they need until they see it. In addition, a well-planned design allows for growth and changes as the needs of users change and evolve.

Creating a Logical Design

A logical design is conceptual and abstract. You do not deal with the physical implementation details yet. You deal only with defining the types of information that you need.

One technique you can use to model your organization's logical information requirements is entity-relationship modeling. Entity-relationship modeling involves identifying the things of importance (entities), the properties of these things (attributes), and how they are related to one another (relationships).

The process of logical design involves arranging data into a series of logical relationships called entities and attributes. An entity represents a chunk of information. In relational databases, an entity often maps to a table. An attribute is a component of an entity that helps define the uniqueness of the entity. In relational databases, an attribute maps to a column.

To ensure that your data is consistent, you must use unique identifiers. A unique identifier is something you add to tables so that you can differentiate between the same item when it appears in different places. In a physical design, this is usually a primary key.

Entity-relationship modeling is purely logical and applies to both OLTP and data warehousing systems. It is also applicable to the various common physical schema modeling techniques found in data warehousing environments, namely normalized (3NF) schemas in Enterprise Data Warehousing environments, star or snowflake schemas in data marts, or hybrid schemas with components of both of these classical modeling techniques.

What is a Schema?

A schema is a collection of database objects, including tables, views, indexes, and synonyms. You can arrange schema objects in the schema models designed for data warehousing in a variety of ways. Most data warehouses use a dimensional model.

The model of your source data and the requirements of your users help you design the data warehouse schema. You can sometimes get the source model from your company's enterprise data model and reverse-engineer the logical data model for the data warehouse from this. The physical implementation of the logical data warehouse model may require some changes to adapt it to your system parameters—size of computer, number of users, storage capacity, type of network, and software. A key part of designing the schema is whether to use a third normal form, star, or snowflake schema, and these are discussed later.

About Third Normal Form Schemas

Third Normal Form design seeks to minimize data redundancy and avoid anomalies in data insertion, updates and deletion. 3NF design has a long heritage in online transaction processing (OLTP) systems. OLTP systems must maximize performance and accuracy when inserting, updating and deleting data. Transactions must be handled as quickly as possible or the business may be unable to handle the flow of events, perhaps losing sales or incurring other costs. Therefore, 3NF designs avoid redundant data manipulation and minimize table locks, both of which can slow inserts, updates and deletes. 3NF designs also works well to abstract the data from specific application needs. If new types of data are added to the environment, you can extend the data model with relative ease and minimal impact to existing applications. Likewise, if you have completely new types of analyses to perform in your data warehouse, a well-designed 3NF schema will be able to handle them without requiring redesigned data structures.

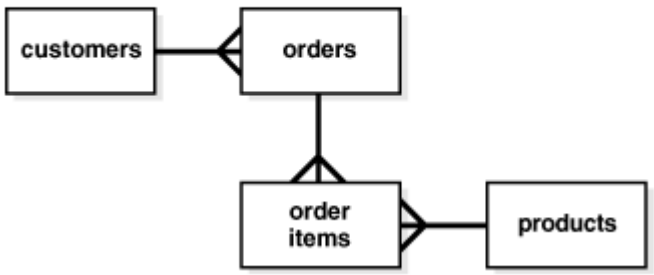
3NF designs have great flexibility, but it comes at a cost. 3NF databases use very many tables and this requires complex queries with many joins. For full scale enterprise models built in 3NF form, over one thousand tables are commonly encountered in the schema. With the kinds of

queries involved in data warehousing, which will often need access to many rows from many tables, this design imposes understanding and performance penalties. It can be complex for query builders, whether they are humans or business intelligence tools and applications, to choose and join the tables needed for a given piece of data when there are very large numbers of tables available. Even when the tables are readily chosen by the query generator, the 3NF schema often requires that a large number of tables be used in a single query. More tables in a query mean more potential data access paths, which makes the database query optimizer's job harder. The end result can be slow query performance.

The issue of slow query performance in a 3NF system is not necessarily limited to the core queries used to create reports and analyses. It can also show up in the simpler task of users browsing subsets of data to understand the contents. Similarly, the complexity of a 3NF schema may impact generating the pick-lists of data used to constrain queries and reports. Although these may seem relatively minor issues, speedy response time for such processes makes a big impact on user satisfaction.

Figure 2-1 presents a tiny fragment of a 3NF Schema. Note how order information is broken into order and order items to avoid redundant data storage. The "crow's feet" markings on the relationship between tables indicate one-to-many relationships among the entities. Thus, one order may have multiple order items, a single customer may have many orders, and a single product may be found in many order items. Although this diagram shows a very small case, you can see that minimizing data redundancy can lead to many tables in the schema.

Figure 2-1 Fragment of a Third Normal Form Schema



About Normalization

Normalization is a data design process that has a high level goal of keeping each fact in just one place to avoid data redundancy and insert, update, and delete anomalies. There are multiple levels of normalization, and this section describes the first three of them. Considering how fundamental the term third normal form (3NF) term is, it only makes sense to see how 3NF is reached.

Consider a situation where you are tracking sales. The core entity you track is sales orders, where each sales order contains details about each item purchased (referred to as a line item): its name, price, quantity, and so on. The order also holds the name and address of the customer and more. Some orders have many different line items, and some orders have just one.

In first normal form (1NF), there are no repeating groups of data and no duplicate rows. Every intersection of a row and column (a field) contains just one value, and there are no groups of columns that contain the same facts. To avoid duplicate rows, there is a primary key. For sales orders, in first normal form, multiple line items of each sales order in a single field of the table are not displayed. Also, there will not be multiple columns showing line items.

Then comes second normal form (2NF), where the design is in first normal form and every non-key column is dependent on the complete primary key. Thus, the line items are broken out into a table of sales order line items where each row represents one line item of one order. You can look at the line item table and see that the names of the items sold are not dependent on the primary key of the line items table: the sales item is its own entity. Therefore, you move the sales item to its own table showing the item name. Prices charged for each item can vary by order (for instance, due to discounts) so these remain in the line items table. In the case of sales order, the name and address of the customer is not dependent on the primary key of the sales order: customer is its own entity. Thus, you move the customer name and address columns out into their own table of customer information.

Next is third normal form, where the goal is to ensure that there are no dependencies on non-key attributes. So the goal is to take columns that do not directly relate to the subject of the row (the primary key), and put them in their own table. So details about customers, such as customer name or customer city, should be put in a separate table, and then a customer foreign key added into the orders table.

Another example of how a 2NF table differs from a 3NF table would be a table of the winners of tennis tournaments that contained columns of tournament, year, winner, and winner's date of birth. In this case, the winner's date of birth is vulnerable to inconsistencies, as the same person could be shown with different dates of birth in different records. The way to avoid this potential problem is to break the table into one for tournament winners, and another for the player dates of birth.

Design Concepts for 3NF Schemas

The following section discusses some basic concepts when modeling for a data warehousing environment using a 3NF schema approach. The intent is not to discuss the theoretical foundation for 3NF modeling (or even higher levels of normalization), but to highlight some key components relevant for data warehousing.

Some key 3NF schema design concepts that are relevant to data warehousing are as follows:

- Identifying Candidate Primary Keys
- Foreign Key Relationships and Referential Integrity Constraints
- Denormalization

Identifying Candidate Primary Keys

A primary key is an attribute that uniquely identifies a specific record in a table. Primary keys can be identified through single or multiple columns. It is normally preferred to achieve unique identification through as little columns as possible - ideally one or two - and to either use a column that is most likely not going to be updated or even changed in bulk. If your data model does not lead to a simple unique identification through its attributes, you would require too many attributes to uniquely identify a single records, or the data is prone to changes, the usage of a surrogate key is highly recommended.

Specifically, 3NF schemas rely on proper and simple unique identification since queries tend to have many table joins and all columns necessary to uniquely identify a record are needed as join condition to avoid row duplication through the join.

Foreign Key Relationships and Referential Integrity Constraints

3NF schemas in data warehousing environments often resemble the data model of its OLTP source systems, in which the logical consistency between data entities is expressed and enforced through primary key - foreign key relationships, also known as parent-child relationship. A foreign key resolves a 1-to-many relationship in relational system and ensures logical consistency: for example, you cannot have an order line item without an order header, or an employee working for a non-existent department.

While such referential are always enforced in OLTP system, data warehousing systems often implement them as declarative, non-enforced conditions, relying on the ETL process to ensure data consistency. Whenever possible, foreign keys and referential integrity constraints should be defined as non-enforced conditions, since it enables better query optimization and cardinality estimates.

Denormalization

Proper normalized modelling tends to decompose logical entities - such as a customer. a product, or an order - into many physical tables, making even the retrieval of perceived simple information requiring to join many tables. While this is not a problem from a query processing perspective, it can put some unnecessary burden on both the application developer (for writing code) as well as the database (for joining information that is always used together). It is not

uncommon to see some sensible level of denormalization in 3NF data warehousing models, in a logical form as views or in a physical form through slightly denormalized tables.

Care has to be taken with the physical denormalization to preserve the subject-neutral shape and therefore the flexibility of the physical implementation of the 3NF schema.

About Facts and Dimensions in Star Schemas

Star schemas divide data into facts and dimensions. Facts are the measurements of some event such as a sale and are typically numbers. Dimensions are the categories you use to identify facts, such as date, location, and product.

The name "star schema" comes from the fact that the diagrams of the schemas typically show a central fact table with lines joining it to the dimension tables, so the graphic impression is similar to a star. Figure 2-2 is a simple example with sales as the fact table and products, times, customers, and channels as the dimension table.

About Facts and Dimensions in Star Schemas

Star schemas divide data into facts and dimensions. Facts are the measurements of some event such as a sale and are typically numbers. Dimensions are the categories you use to identify facts, such as date, location, and product.

The name "star schema" comes from the fact that the diagrams of the schemas typically show a central fact table with lines joining it to the dimension tables, so the graphic impression is similar to a star. Figure 2-2 is a simple example with sales as the fact table and products, times, customers, and channels as the dimension table.

About Fact Tables in Data Warehouses

Fact tables have measurement data. They have many rows but typically not many columns. Fact tables for a large enterprise can easily hold billions of rows. For many star schemas, the fact table will represent well over 90 percent of the total storage space. A fact table has a composite key made up of the primary keys of the dimension tables of the schema.

A fact table contains either detail-level facts or facts that have been aggregated. Fact tables that contain aggregated facts are often called summary tables. A fact table usually contains facts with the same level of aggregation. Though most facts are additive, they can also be semi-additive or non-additive. Additive facts can be aggregated by simple arithmetical addition. A common example of this is sales. Non-additive facts cannot be added at all. An example of this is

averages. Semi-additive facts can be aggregated along some of the dimensions and not along others. An example of this is inventory levels stored in physical warehouses, where you may be able to add across a dimension of warehouse sites, but you cannot aggregate across time.

In terms of adding rows to data in a fact table, there are three main approaches:

- Transaction-based
  Shows a row for the finest level detail in a transaction. A row is entered only if a transaction has occurred for a given combination of dimension values. This is the most common type of fact table.
- Periodic Snapshot
  Shows data as of the end of a regular time interval, such as daily or weekly. If a row for the snapshot exists in a prior period, a row is entered for it in the new period even if no activity related to it has occurred in the latest interval. This type of fact table is useful in complex business processes where it is difficult to compute snapshot values from individual transaction rows.
- Accumulating Snapshot
  Shows one row for each occurrence of a short-lived process. The rows contain multiple dates tracking major milestones of a short-lived process. Unlike the other two types of fact tables, rows in an accumulating snapshot are updated multiple times as the tracked process moves forward.

About Dimension Tables in Data Warehouses

- Dimension tables provide category data to give context to the fact data. For instance, a star schema for sales data will have dimension tables for product, date, sales location, promotion and more. Dimension tables act as lookup or reference tables because their information lets you choose the values used to constrain your queries. The values in many dimension tables may change infrequently. As an example, a dimension of geographies showing cities may be fairly static. But when dimension values do change, it is vital to update them fast and reliably. Of course, there are situations where data warehouse dimension values change frequently. The customer dimension for an enterprise will certainly be subject to a frequent stream of updates and deletions.
- A key aspect of dimension tables is the hierarchy information they provide. Dimension data typically has rows for the lowest level of detail plus rows for aggregated dimension values. These natural rollups or aggregations within a dimension table are called hierarchies and add great value for analyses. For instance, if you want to calculate the

share of sales that a specific product represents within its specific product category, it is far easier and more reliable to have a predefined hierarchy for product aggregation than to specify all the elements of the product category in each query. Because hierarchy information is so valuable, it is common to find multiple hierarchies reflected in a dimension table.

- Dimension tables are usually textual and descriptive, and you will use their values as the row headers, column headers and page headers of the reports generated by your queries. While dimension tables have far fewer rows than fact tables, they can be quite wide, with dozens of columns. A location dimension table might have columns indicating every level of its rollup hierarchy, and may show multiple hierarchies reflected in the table. The location dimension table could have columns for its geographic rollup, such as street address, postal code, city, state/province, and country. The same table could include a rollup hierarchy set up for the sales organization, with columns for sales district, sales territory, sales region, and characteristics.

Design Concepts in Star Schemas

Here we touch on some of the key terms used in star schemas. This is by no means a full set, but is intended to highlight some of the areas worth your consideration.

Data Grain

One of the most important tasks when designing your model is to consider the level of detail it will provide, referred to as the grain of the data. Consider a sales schema: will the grain be very fine, storing every single item purchased by each customer? Or will it be a coarse grain, storing only the daily totals of sales for each product at each store? In modern data warehousing there is a strong emphasis on providing the finest grain data possible, because this allows for maximum analytic power. Dimensional modeling experts generally recommend that each fact table store just one grain level. Presenting fact data in single-grain tables supports more reliable querying and table maintenance, because there is no ambiguity about the scope of any row in a fact table.

Working with Multiple Star Schemas

Because the star schema design approach is intended to chunk data into distinct processes, you need reliable and performant ways to traverse the schemas when queries span multiple schemas. One term for this ability is a data warehouse bus architecture. A data warehouse bus architecture can be achieved with conformed dimensions and conformed facts.

Conformed Dimensions

Conformed dimensions means that dimensions are designed identically across the various star schemas. Conformed dimensions use the same values, column names and data types consistently across multiple stars. The conformed dimensions do not have to contain the same number of rows in each schema's copy of the dimension table, as long as the rows in the shorter tables are a true subset of the larger tables.

Conformed Facts

If the fact columns in multiple fact tables have exactly the same meaning, then they are considered conformed facts. Such facts can be used together reliably in calculations even though they are from different tables. Conformed facts should have the same column names to indicate their conformed status. Facts that are not conformed should always have different names to highlight their different meanings.

Surrogate Keys

Surrogate or artificial keys, usually sequential integers, are recommended for dimension tables. By using surrogate keys, the data is insulated from operational changes. Also, compact integer keys may allow for better performance than large and complex alphanumeric keys.

Degenerate Dimensions

Degenerate dimensions are dimension columns in fact tables that do not join to a dimension table. They are typically items such as order numbers and invoice numbers. You will see them when the grain of a fact table is at the level of an order line-item or a single transaction.

Junk Dimensions

Junk dimensions are abstract dimension tables used to hold text lookup values for flags and codes in fact tables. These dimensions are referred to as junk, not because they have low value, but because they hold an assortment of columns for convenience, analogous to the idea of a "junk drawer" in your home. The number of distinct values (cardinality) of each column in a junk dimension table is typically small.

Embedded Hierarchy

Classic dimensional modeling with star schemas advocates that each table contain data at a single grain. However, there are situations where designers choose to have multiple grains in a table, and these commonly represent a rollup hierarchy. A single sales fact table, for instance, might contain both transaction-level data, then a day-level rollup by product, then a month-level

rollup by product. In such cases, the fact table will need to contain a level column indicating the hierarchy level applying to each row, and queries against the table will need to include a level predicate.

Factless Fact Tables

Factless fact tables do not contain measures such as sales price or quantity sold. Instead, the rows of a factless fact table are used to show events not represented by other fact tables. Another use for factless tables is as a "coverage table" which holds all the possible events that could have occurred in a given situation, such as all the products that were part of a sales promotion and might have been sold at the promotional price.

Slowly Changing Dimensions

One of the certainties of data warehousing is that the way data is categorized will change. Product names and category names will change. Characteristics of a store will change. The areas included in sales territories will change. The timing and extent of these changes will not always be predictable. How can these slowly changing dimensions be handled? Star schemas treat these in three main ways:

- Type 1
  The dimension values that change are simply overwritten, with no history kept. This creates a problem for time-based analyses. Also, it invalidates any existing aggregates that depended on the old value of the dimension.
- Type 2
  When a dimension value changes, a new dimension row showing the new value and having a new surrogate key is created. You may choose to include date columns in our dimension showing when the new row is valid and when it is expired. No changes need be made to the fact table.
- Type 3
  When a dimension value is changed, the prior value is stored in a different column of the same row. This enables easy query generation if you want to compare results using the current and prior value of the column.

In practice, Type 2 is the most common treatment for slowly changing dimensions.

About Snowflake Schemas

The snowflake schema is a more complex data warehouse model than a star schema, and is a type of star schema. It is called a snowflake schema because the diagram of the schema resembles a snowflake.

Snowflake schemas normalize dimensions to eliminate redundancy. That is, the dimension data has been grouped into multiple tables instead of one large table. For example, a product dimension table in a star schema might be normalized into a `products` table, a `product_category` table, and a `product_manufacturer` table in a snowflake schema. While this saves space, it increases the number of dimension tables and requires more foreign key joins. The result is more complex queries and reduced query performance. Figure 2-3 presents a graphical representation of a snowflake schema.

About the Oracle In-Memory Column Store

Traditional analytics has certain limitations or requirements that need to be managed to obtain good performance for analytic queries. You need to know user access patterns and then customize your data structures to provide optimal performance for these access patterns. Existing indexes, materialized views, and OLAP cubes need to be tuned. Certain data marts and reporting databases have complex ETL and thus need specialized tuning. Additionally, you need to strike a balance between performing analytics on stale data and slowing down OLTP operations on the production databases.

The Oracle In-Memory Column Store (IM column store) within the Oracle Database provides improved performance for both ad-hoc queries and analytics on live data. The live transactional database is used to provide instant answers to queries, thus enabling you to seamlessly use the same database for OLTP transactions and data warehouse analytics.

The IM column store is an optional area in the SGA that stores copies of tables, table partitions, and individual columns in a compressed columnar format that is optimized for rapid scans. Columnar format lends itself to easily to vector processing thus making aggregations, joins, and certain types of data retrieval faster than the traditional on-disk formats. The columnar format exists only in memory and does not replace the on-disk or buffer cache format. Instead, it supplements the buffer cache and provides an additional, transaction-consistent, copy of the table that is independent of the disk format.

Benefits of Using the Oracle In-Memory Column Store

The IM column store enables the Oracle Database to perform scans, joins, and aggregates much faster than when it uses the on-disk format exclusively. Business applications, ad-hoc analytic queries, and data warehouse workloads benefit most. Pure OLTP databases that perform short transactions using index lookups benefit less.

The IM column store seamlessly integrates with the Oracle Database. All existing database features, including High Availability features, are supported with no application changes required. Therefore, by configuring the IM column store, you can instantly improve the performance of existing analytic workloads and ad-hoc queries.
The Oracle Optimizer is aware of the IM column store making it possible for the Oracle Database to seamlessly send analytic queries to the IM column store while OLTP queries and DML are sent to the row store.
The advantages offered by the IM column store for data warehousing environments are:

- Faster scanning of large number of rows and applying filters that use operators such as =,<,>, and IN.
- Faster querying of a subset of columns in a table, for example, selecting 5 of 100 columns. See "Faster Performance for Analytic Queries".
- Enhanced performance for joins by converting predicates on small dimension tables to filters on a large fact table. See "Enhanced Join Performance Using Vector Joins".
- Efficient aggregation by using VECTOR GROUP BY transformation and vector array processing. See "Enhanced Aggregation Using VECTOR GROUP BY Transformations".
- Reduced storage space and significantly less processing overhead because fewer indexes, materialized views, and OLAP cubes are required when IM column store is used.

Faster Performance for Analytic Queries
Storing data in memory using columnar format provides fast throughput for analyzing large amounts of data. This is useful for ad-hoc queries with unanticipated access patterns. Columnar format uses fixed-width columns for most numeric and short string data types. This enables very fast vector processing that answers queries faster. Only the columns necessary for the specific data analysis task are scanned instead of entire rows of data. Data can be analyzed in real-time, thus enabling you to explore different possibilities and perform iteration. Using the IM column store requires fewer OLAP cubes to be created to obtain query results.
For example, you need to find the number of sales in the state of California this year. This data is stored in the SALES table. When this table is stored in the IM column store, the database needs to just scan the State column and count the number of occurrences of state California.
Enhanced Join Performance Using Vector Joins
IM column store takes advantage of vector joins. Vector joins speed up joins by converting predicates on small dimension tables to filters on large fact tables. This is useful when performing a join of multiple dimensions with one large fact table. The dimension keys on fact tables have lots of repeat values. The scan performance and repeat value optimization speeds up joins.
Enhanced Aggregation Using VECTOR GROUP BY Transformations
An important aspect of analytics is to determine patterns and trends by aggregating data. Aggregations and complex SQL queries run faster when data is stored in the IM column store.
VECTOR GROUP BY transformations enable efficient in-memory array-based aggregation. During a fact table scan, aggregate values are accumulated into in-memory arrays and efficient algorithms are used perform aggregation. Performing joins based on the primary

key and foreign key relationships are optimized for both star schemas and snowflake schemas.

Using the Oracle In-Memory Column Store

You can store data using columnar format in the IM column store for existing databases or for new database that you plan to create. IM column store is simple to configure and does not impact existing applications. Depending on the requirement, you can configure one or more tablespaces, tables, materialized views, or partitions to be stored in memory.

Using Vector Joins to Enhance Join Performance

Joins are an integral part of data warehousing workloads. IM column store enhances the performance of joins when the tables being joined are stored in memory. Simple joins that use bloom filters and complex joins between multiple tables benefit by using the IM column store. In a data warehousing environment, the most frequently-used joins are ones in which one or more dimension tables are joined with a fact table.
The following types of joins run faster when the tables being joined are stored in the IM column store:

- Joins that are amenable to using bloom filters
- Joins of multiple small dimension tables with one fact table
- Joins between two tables that have a PK-FK relationship

The IM column store runs queries that contain joins more efficiently and quickly by using vector joins. Vector joins allow the Oracle Database to take advantage of the fast scanning and vector processing capability of the IM column store. A vector join transforms a join between a dimension and fact table to filter that can be applied as part of the scan of the fact table. This join conversion is performed with the use of bloom filters, which enhance hash join performance in the Oracle Database. Although bloom filters are independent of IM column store, they can be applied very efficiently to data stored in memory through SIMD vector processing.
Consider the following query that performs a join of the CUSTOMERS dimension table with the SALES fact table:

```
SELECT c.customer_id, s.quantity_sold, s.amount_sold
FROM CUSTOMERS c, SALES s
WHERE c.customer_id = s.customer_id AND c.country_id = 'FR';
```

When both these tables are stored in the IM column store, SIMD vector processing is used to quickly scan the data and apply filters. Figure 2-4 displays a graphical representation of the how vector joins are used to implement the query. The predicate on the CUSTOMERS table, c.country_id='FR' is converted into a filter on the SALES fact table. The filter is country_id='FR'. Because the SALES table is stored in memory using columnar format, just one column needs to be scanned to determine the result of this query.

Automatic Big Table Caching to Improve the Performance of In-Memory Parallel Queries

Automatic big table caching enhances the in-memory query capabilities of Oracle Database. When a table does not fit in memory, the database decides which buffers to cache based on access patterns. This provides efficient caching for large tables, even if they do not fully fit in the buffer cache.
An optional section of the buffer cache, called the big table cache, is used to store data for table scans. The big table cache is integrated with the buffer cache and uses a temperature-based, object-level replacement algorithm to manage the big table cache contents. This is different from the access-based, block level LRU algorithm used by the buffer cache.

Typical data warehousing workloads scan multiple tables. Performance may be impacted if the combined size of these tables is greater than the combined size of the buffer cache. With automatic big table caching, the scanned tables are stored in the big table cache instead of the buffer cache. The temperature-based, object-level replacement algorithm used by the big table cache can provide enhanced performance for data warehousing workloads by:

- Selectively caching the "hot" objects
  Each time an object is accessed, Oracle Database increments the temperature of that object. An object in the big table cache can be replaced only by another object whose temperature is higher than its own temperature.
- Avoiding thrashing
  Partial objects are cached when objects cannot be fully cached.

In Oracle Real Application Clusters (Oracle RAC) environments, automatic big table caching is supported only for parallel queries. In single instance environments, this functionality is supported for both serial and parallel queries.
To use automatic big table caching, you must enable the big table cache. To use automatic big table caching for serial queries, you must set the DB_BIG_TABLE_CACHE_PERCENT_TARGET initialization parameter to a nonzero value. To use automatic big table caching for parallel queries, you must set PARALLEL_DEGREE_POLICY to AUTO or ADAPTIVE and DB_BIG_TABLE_CACHE_PERCENT_TARGET to a nonzero value.

About In-Memory Aggregation

In-memory aggregation uses the VECTOR GROUP BY operation to enhance the performance of queries that aggregate data and join one or more relatively small tables to a larger table, as often occurs in a star query. VECTOR GROUP BY will be chosen by the SQL optimizer based on cost estimates. This will occur more often when the query selects from in-memory columnar tables and the tables include unique or numeric join keys (regardless of whether the uniqueness is forced by a primary key, unique constraint or schema design). VECTOR GROUP BY aggregation will only by chosen for GROUP BY. It will not be chosen for GROUP BY ROLLUP, GROUPING SETS or CUBE.

VECTOR GROUP BY Aggregation and the Oracle In-Memory Column Store

Although using the IM column store is not a requirement for using VECTOR GROUP BY aggregation, it is strongly recommended that you use both features together. Storing tables in memory using columnar format provides the foundation storage that VECTOR GROUP BY aggregation leverages to provide transactionally consistent results immediately after a schema is updated without the need to wait until the data marts are populated.

When to Use VECTOR GROUP BY Aggregation

Not all queries and scenarios benefit from the use of VECTOR GROUP BY aggregation. The following sections provide guidelines about the situations in which using this aggregation can be beneficial.

Situations Where VECTOR GROUP BY Aggregation Is Useful

VECTOR GROUP BY aggregation provides benefits in the following scenarios:

- The schema contains "mostly" unique keys or numeric keys for the columns that are used to join the fact and dimensions. The uniqueness can be enforced using a primary key, unique constraint or by schema design.
- The fact table is at least 10 times larger than the dimensions.
- The IM column store is used to store the dimensions and fact table in memory.

Situations Where VECTOR GROUP BY Aggregation Is Not Advantageous

Using VECTOR GROUP BY aggregation does not provide substantial performance benefits in the following scenarios:

- Joins are performed between two very large tables
  By default, the VECTOR GROUP BY transformation is used only if the fact table is at least 10 times larger than the dimensions.
- Dimensions contain more than 2 billion rows
  The VECTOR GROUP BY transformation is not used if a dimension contains more than 2 billion rows.
- The system does not have sufficient memory resources
  Most systems that use the IM column store will be able to benefit from using the VECTOR GROUP BY transformation.

When Is VECTOR GROUP BY Aggregation Used to Process Analytic Queries?

VECTOR GROUP BY aggregation is integrated with the Oracle Optimizer and no new SQL or initialization parameters are required to enable the use of this transformation. It also does not need additional indexes, foreign keys, or dimensions.

By default, Oracle Database decides whether or not to use VECTOR GROUP BY aggregation for a query based on the cost, relative to other execution plans that are determined for this query. However, you can direct the database to use VECTOR GROUP BY aggregation for a query by using query block hints or table hints.

VECTOR GROUP BY aggregation can be used to process a query that uses a fact view that is derived from multiple fact tables.

Oracle Database uses VECTOR GROUP BY aggregation to perform data aggregation when the following conditions are met:

- The queries or subqueries aggregate data from a fact table and join the fact table to one or more dimensions.
  Multiple fact tables joined to the same dimensions are also supported assuming that these fact tables are connected only through joins to the dimension. In this case, VECTOR GROUP BY aggregates fact table separately and then joins the results on the grouping keys.
- The dimensions and fact table are connected to each other only through join columns.
  Specifically, the query must not have any other predicates that refer to columns across multiple dimensions or from both a dimension and the fact table. If a query performs a join between two or more tables and then joins the result to the fact, then VECTOR GROUP BY aggregation treats the multiple dimensions as a single dimension.

The best performance for VECTOR GROUP BY aggregation is obtained when the tables being joined are stored in the IM column store.

VECTOR GROUP BY aggregation does not support the following:

- Semi- and anti-joins across multiple dimensions or between a dimension and the fact table.
- Equi-joins across multiple dimensions.
- Aggregations performed using DISTINCT.
- Bloom filters
  VECTOR GROUP BY aggregation and bloom filters are mutually exclusive.
  If bloom filters are used to perform joins for a query, then VECTOR GROUP BY aggregation is not applicable to the processing of this query.