

DEPLOYING A DIMENSIONAL MODEL ON THE ORACLE DATABASE

Stewart Bryson, Rittman Mead

Introduction

Understanding the logical model for a data warehouse environment may seem confusing to Oracle DBAs with little or no experience in dimensional modeling. We will investigate the core physical objects in a dimensional model, and explore the differences between these objects, such as dimension tables, fact tables, and aggregate tables. We will also consider the effect different physical properties will have on these objects, such as partitioning, compression, and constraint and indexing properties. Finally, we'll explore aggregate navigation in Oracle, comparing forms of manual navigation, BI tool navigation, and finally, query rewrite, which is a form of aggregate navigation performed by the Oracle Database optimizer.

OLTP Applications versus Data Warehouses

To understand what makes a dimensional model different from third-normal form (3NF), we first need to understand what makes a BI application different from a transactional system. OLTP applications participate in “queries-per-second” processing with hundreds or perhaps even thousands of simultaneous single-record lookups and updates. Operational systems require a database model that structures—or normalizes—data across different business objects and constrains the data for effective user-entry screens. According to Ralph Kimball, “entity-relationship modeling is a discipline used to illuminate the microscopic relationships among data elements. The highest art form of entity-relationship modeling is to remove all redundancy in the data.”¹ Though the database might be awash with concurrent processing, each of the individual database operations is single-threaded. Transactional systems use the system global area (SGA) in the database because blocks returned from these queries are highly focused because they are returned using a primary key index read.

Conversely, data warehouse queries are more characteristic of “seconds-per-query” processing, because the typical query returns one or more aggregate functions (SUM, AVG, etc.) over millions or billions of rows. Because of the amount of data involved, dimensional models are designed for simplicity and speed of access de-normalizing across fewer tables and requiring fewer joins.

Oracle Next-Generation Reference Architecture

The Oracle Next-Generation Reference Architecture, depicted in Figure 1, uses three logical layers to facilitate the main functional components of the data warehouse. Each of these layers plays a role in maintaining flexibility to support structured and unstructured data sources for access by BI tools for both strategic and operational reporting. Using multiple layers in a single data warehouse allows the environment to address changing business requirements without losing reporting capabilities.

¹ Ralph Kimball, Laura Reeves, Margy Ross and Warren Thornthwaite, [The Data Warehouse Lifecycle Toolkit](#), 1st ed. (John Wiley & Sons, Inc., 1998) 141

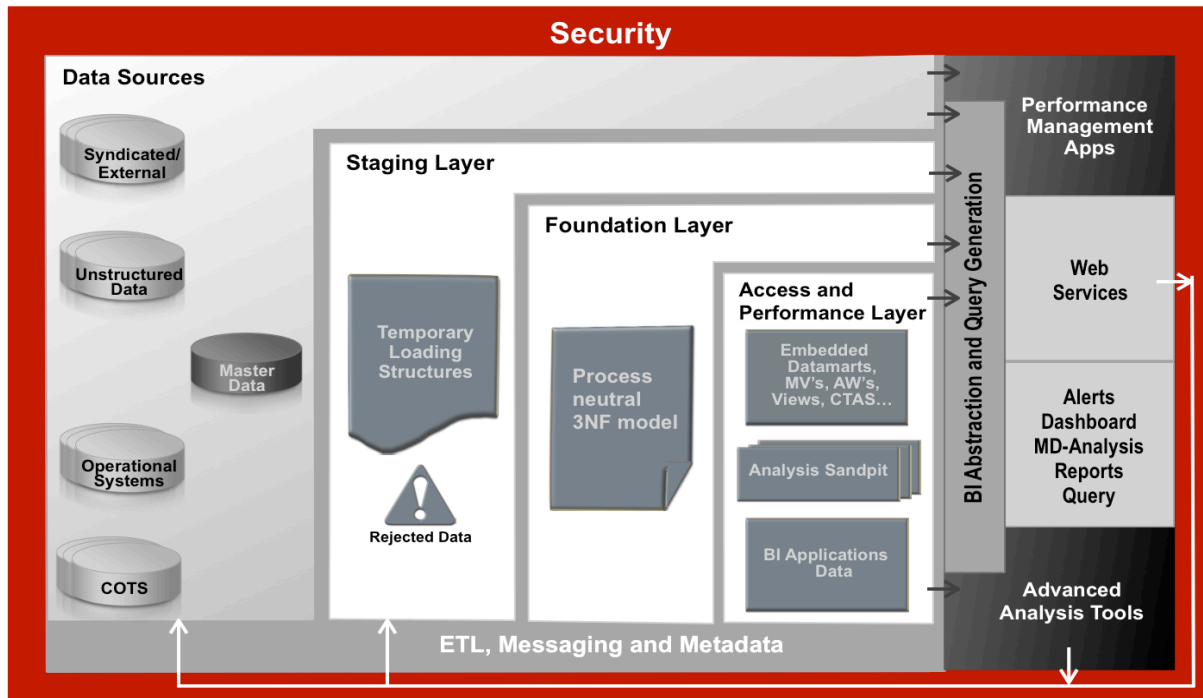


Figure 1: Oracle Next-Generation Reference Architecture

Staging Layer

The Staging layer is the “landing pad” for incoming data, and is typically made up of regular, heap-organized tables that are populated by extraction routines from a range of data sources. This includes incremental “change” tables for any method of change-data capture (CDC), including custom-CDC routines, Oracle Database CDC or GoldenGate. Also in this layer are staging tables used to assist with ETL processing, including data reject tables and external tables for querying flat-files.

Foundation Layer

While a dimensional model excels at information access, it’s cumbersome at storing data for the long-term, particularly if analysis requirements change over time. The Foundation Layer in the reference architecture acts as the information management layer, storing detail-level warehouse data in a business and query-neutral form that will persist over time. Data in this layer is typically modeled in normalized 3NF with hierarchies stored in column values rather than hard-coded in table structures. Business keys together with source system identifiers and transactional timestamps are used in this layer, as the purpose here is to maintain “lossless” data. We don’t use surrogate keys or slowly-changing dimensions in the Foundation Layer as these are aspects of dimensional modeling.

Access and Performance Layer

The Access and Performance Layer is where we find our dimensional models. Our main purpose in this layer is to optimize our model to address business requirements, and provide easy and efficient access for particular BI tools. The key benefit is that our data warehouse should now last beyond the immediate requirements that our users have for a dimensional model. While Kimball argues that we can gracefully

adapt a dimensional model over time to incorporate changes and new data, in reality this is often difficult to do.²

Dimensional Model, or “Star Schema”

The recommended modeling approach for data warehouses is to use a star schema in line with the Kimball Bus Architecture, depicted in Figure 2. The model is de-normalized with a series of attribute tables, called “dimensions”, and measure tables, called “facts.” Kimball describes the goal of dimensional modeling as “a logical design technique that seeks to present the data in a standard framework that is intuitive and allows for high-performance access.”³ Data models composed of facts and dimensions are also called “star schemas” due to their appearance on a data model diagram, with a fact table typically in the middle and all the dimension tables extending out.

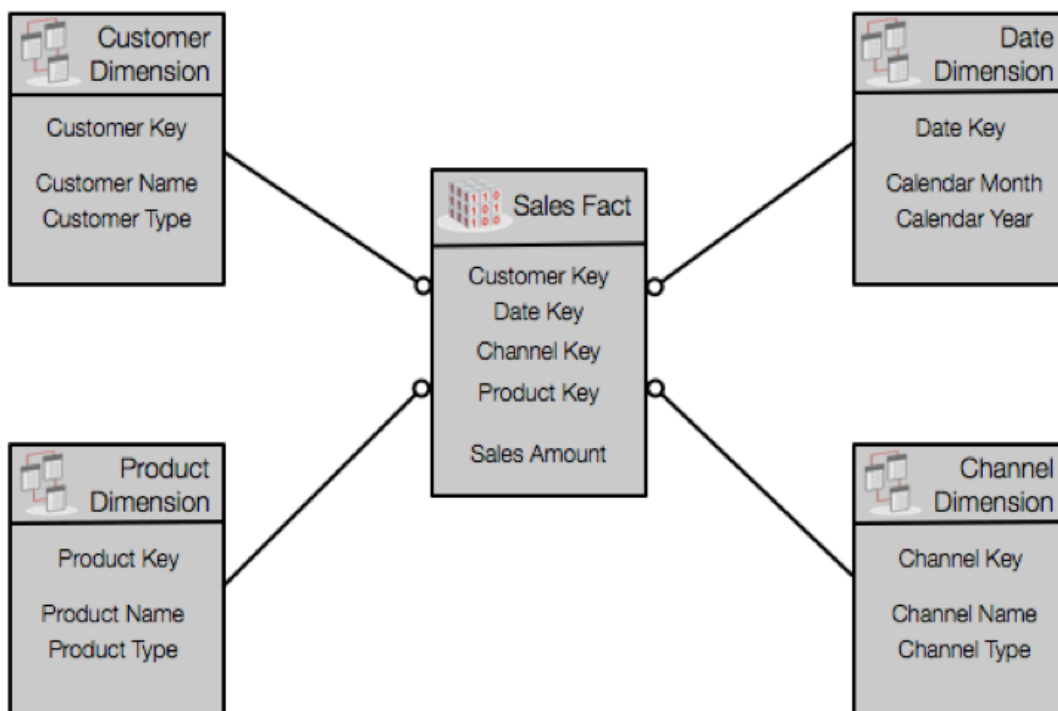


Figure 2: Dimensional Model or "Star Schema"

Dimension Tables

Kimball describes dimension tables as “entry points into the data warehouse.”⁴ They have a single “surrogate key” which is a system-generated identifier that serves as the primary key for the table. The dimension table also contains one or more columns serving as a “natural key.” The natural key uniquely defines a dimensional entity, such as a Product or a Customer, and therefore this key is typically facilitated using a primary key from a transactional system that sources the dimensional entity.

² Kimball, *Lifecycle* 148-149

³ Kimball, *Lifecycle* 144

⁴ Kimball, *Lifecycle* 146

The purpose for a dimension table is to store attributes and descriptors for the different entities in the business. These descriptors are usually textual and perform two main functions: 1) they are used to filter the results of our queries; 2) they form the contents of our GROUP BY clauses becoming the row headers in our reports. Dimension tables also store hierarchies that exist within the entity that the table represents; there can be any number of hierarchies present in the data, and each of these hierarchies can have many different levels.

Fact Tables

A fact or measure in a dimensional model is “usually something that is not known in advance. A fact is ‘an observation in the marketplace.’”⁵ Fact tables act as containers for these observations which are predominately stored as numeric columns. Fact tables contain foreign keys to multiple dimension tables, and the logical primary key of the fact table is a multi-part key composed of the different foreign keys included in the fact. Because fact tables are event-driven and transactional, they can grow quite large, and it is fact tables that account for the majority of the data stored in a data warehouse.

Product Dimension	Sales Fact
PRODUCT_KEY	PRODUCT_KEY
product_id	CUSTOMER_KEY
product_descr	CHANNEL_KEY
product_long_descr	DATE_KEY
product_type_code	quantity
product_type_descr	amount
product_class_id	tax_amount
product_class_descr	discount_amount
product_group_id	
product_group_descr	

Figure 3: Sample Dimension and Fact Tables

Partitioning

It is difficult to properly deploy a data warehouse without partitioning. With non-partitioned fact tables, every row of new data decreases performance to some degree, and fact tables typically get thousands, if not millions, of new rows every day. Query response times for fact tables built on single-segment tables will never perform according to user expectations. With a correct partitioning strategy, queries returning a month of data should perform the same whether the table contains 10 months or 10 years of data.

Partitioning Fact Tables

Although Kimball argues that all surrogate keys should be numeric, the surrogate key to our date dimension table should be a DATE datatype instead of a NUMBER. This debate bubbles up from the two streams at work in the Oracle Data Warehousing community: the data warehousing folks, and the Oracle folks. In his earlier works, Kimball argued that a NUMBER required less space than a DATE and

⁵ Kimball, Lifecycle 165

was more attractive for that reason. He also posits that the only conceivable reason to use a DATE as a surrogate key is to bypass the date dimension and write all SQL directly against the fact tables using date functions.⁶ More recently, he makes a better argument: if our surrogate key is a DATE, then how do we handle “Not Applicable” type rows?⁷ I think that most designers who struggle with this decision point to this issue. If we use an actual DATE as our surrogate key, then what value can we use that means “no date at all”?

Oracle experts like Thomas Kyte argue that “dates belong in DATES” because the Oracle Database optimizer will work more efficiently in these situations.⁸ Kimball’s point that there is no conceivable reason to use a DATE datatype other than a desire to bypass the date dimension is flawed: there are performance reasons to do this, as Kyte demonstrates, and one of the biggest is the ability to partition the fact table using a DATE datatype.

Since every fact table will have a foreign key to the date dimension, then every fact table can be RANGE partitioned on a DATE. A single physical event, such as a sales transaction, can have several different dates associated with that event: the order date, the ship date, the receiving date, etc. As fact tables are modeled to effectively represent these events, a fact record will then need to contain multiple dates. The date dimension can act as a “role-playing” dimension: “A *role* in a data warehouse is a situation where a single dimension appears several times in the same fact table.”⁹ Choosing which date in the fact to use should be a relatively simple endeavor: use the most pervasive reporting date. End-users can usually reach consensus on what this date is in a matter of minutes.

Partitioning Dimension Tables

Dimension tables should usually exist in a single segment, but larger dimension tables — the customer or user dimension is a common example — can be as large as fact tables, and may benefit from partitioning. If there is a pervasive attribute in the dimension table that is always used to slice customer data, then LIST or RANGE partitioning on that value may prune the table in a valuable way. Otherwise, using HASH partitioning of the surrogate key — always using a power of 2 for the number of partitions — can enable a partial partition-wise join between the dimension and the fact table. This is a highly optimized join technique that allows the Oracle Database to effectively parallelize the join between the fact and the dimension table across the different hash partitions in the dimension.

Constraints

A common question in data warehouse environments is whether to create constraints such as primary keys and foreign keys. The standard answer is yes — we should *create* them — but the more interesting question is whether we should *enable* them. There are situations, as we will see, where having a disabled constraints can give the database needed information.

⁶ Kimball, [Lifecycle](#) 192

⁷ Ralph Kimball, “Kimball Design Tip #51: Latest Thinking On Time Dimension Tables” (Kimball Group, 2004) 1

⁸ Thomas Kyte, (2005, February 11). “Ask Tom”. Retrieved from http://asktom.oracle.com/pls/apex/f?p=100:11:0::::P11_QUESTION_ID:4632159445946

⁹ Kimball, [Lifecycle](#) 223

In an OLTP application, constraints can form a part of the application. Instead of writing a complex series of logical tests before insertion or modification of a record, the application can instead use a constraint on the table to encapsulate and facilitate this logic. Because data warehouses source data from these systems, we don't need to re-constrain business rules that already exist in the source system. We will have new business rules that we need to constrain in the dimensional model. Unlike OLTP applications with many concurrent users, the data warehouse typically has a single ETL process responsible for loading the data, and we can build these business rules into the ETL.

So the question we have to ask is whether enabled constraints give us any benefit. Our ETL processing will take a performance hit for having to maintain these constraints during the load, or for rebuilding them after. We may get query improvements with enabled constraints, and we can also make use of constraints in our ETL. The common example is for de-duplication: we can use a constraint to guarantee we don't have duplicate records in a table.

Constraints on Dimension Tables

An enabled primary key on a dimension table gives the optimizer additional information in choosing access paths. Additionally, the index that gets created provides the possibility of an index read in the join between the dimension and the fact. In choosing to maintain the primary key constraints as enabled, we should do everything we can to decrease the cost associated with this choice.

When a primary key is created, the Oracle Database typically requires an index to use for validating the constraint. By default the database creates a new UNIQUE index for this purpose, or reuses an existing one if it already exists. If the database creates the index to support the constraint, then it considers that index as just another part of the constraint. So whenever that constraint is DISABLED, the database will drop the index as well:

```
SQL> ALTER TABLE product_dim
 2      ADD CONSTRAINT product_pk
 3      PRIMARY KEY (prod_id)
 4 /

Table altered.

Elapsed: 00:00:00.05
SQL>
SQL> SELECT index_name,
 2      uniqueness
 3      FROM all_indexes
 4      WHERE table_name='PRODUCT_DIM'
 5      AND owner = 'PHYSICAL_DW2'
 6 /

INDEX_NAME                                | UNIQUENES
-----|-----
PRODUCT_PK                                | UNIQUE

1 row selected.

Elapsed: 00:00:00.10
SQL>
SQL> ALTER TABLE product_dim
 2      DISABLE PRIMARY KEY;

Table altered.

Elapsed: 00:00:00.10
SQL>
```

```

SQL> SELECT index_name,
2         uniqueness
3       FROM all_indexes
4      WHERE table_name='PRODUCT_DIM'
5         AND owner = 'PHYSICAL_DW2'
6      /

no rows selected

Elapsed: 00:00:00.01
SQL>

```

So if an ETL process disables a constraint with a database-managed index as part of an ETL flow, the index disappears. This is disadvantageous because the database will lose all physical properties associated with it: PARALLEL degrees, STORAGE clauses, etc. To maintain the index between the DISABLE and the ENABLE, the index needs to be created *first*, so the database will consider the index to be independent of the constraint:

```

SQL> CREATE UNIQUE INDEX
2         product_pk
3       ON product_dim(prod_id)
4      /

Index created.

Elapsed: 00:00:00.05
SQL>
SQL> ALTER TABLE product_dim
2         ADD CONSTRAINT product_pk
3         PRIMARY KEY (prod_id)
4      /

Table altered.

Elapsed: 00:00:00.06
SQL>
SQL> ALTER TABLE product_dim
2         DISABLE PRIMARY KEY;

Table altered.

Elapsed: 00:00:00.01
SQL>
SQL> SELECT index_name,
2         uniqueness
3       FROM all_indexes
4      WHERE table_name='PRODUCT_DIM'
5         AND owner = 'PHYSICAL_DW2'
6      /

INDEX_NAME                                | UNIQUENES
-----
PRODUCT_PK                                | UNIQUE

1 row selected.

Elapsed: 00:00:00.01
SQL>

```

Left alone, this index will likely cause unneeded overhead in our load process, so we would want to make sure we alter the index to UNUSABLE after the constraint is disabled. Once our ETL process is complete, we would REBUILD the index and then ENABLE the constraint.

Another way to maintain the existence of the index is to create a DEFERRABLE constraint instead of a standard constraint. With a DEFERRABLE constraint, the database will always maintain the index separate from the constraint regardless of whether the index existed before the creation of the constraint or not. Additionally, DEFERRABLE constraints use non-unique indexes for validation.

```
SQL> ALTER TABLE product_dim
 2      ADD CONSTRAINT product_pk
 3      PRIMARY KEY (prod_id)
 4      DEFERRABLE
 5 /

Table altered.

Elapsed: 00:00:00.10
SQL>
SQL> ALTER TABLE product_dim
 2      DISABLE PRIMARY KEY;

Table altered.

Elapsed: 00:00:00.01
SQL>
SQL> SELECT index_name,
 2      uniqueness
 3      FROM all_indexes
 4      WHERE table_name='PRODUCT_DIM'
 5      AND owner = 'PHYSICAL_DW2'
 6 /

INDEX_NAME                                | UNIQUENES
-----
PRODUCT_PK                                | NONUNIQUE

1 row selected.

Elapsed: 00:00:00.00
SQL>
```

DEFERRABLE constraints have the added benefit of allowing for deferred constraint checking. Instead of validating constraints on a record-by-record basis at the time each row is inserted or updated, a deferred constraint is instead checked at commit time, when the entire transaction is completed. This can have a positive affect on batch processes involving thousands or millions or rows.

Constraints on Fact Tables

Does it make sense to create primary keys on fact table? Dimension tables actually join on their primary keys, so the underlying index is a possible access path for data retrieval. This is not true for fact tables. The multi-column index that would be created to support the primary key is worthless. This index would occupy and waste a lot of space while providing zero benefit for data retrieval.

The multi-part primary key might be valuable in validating our ETL process, however. In these cases, we can create the primary key DISABLED and VALIDATED. In this scenario, the Oracle Database checks to make sure the constraint is logically valid, but does not create and maintain an index to do so:

```
SQL> ALTER TABLE sales_fact
 2      ADD CONSTRAINT sales_fact_pk
 3      PRIMARY KEY ( prod_id,
 4                  cust_id,
 5                  time_id,
 6                  channel_id,
 7                  promo_id
```

```

8          )
9      DISABLE VALIDATE
10 /

```

Table altered.

Elapsed: 00:00:08.32

SQL>

```

SQL> SELECT index_name,
2      uniqueness
3      FROM all_indexes
4      WHERE table_name='SALES_FACT'
5      AND owner = 'PHYSICAL_DW2'
6
7 /

```

no rows selected

Elapsed: 00:00:00.00

SQL>

While the constraint is in a DISABLE VALIDATE state, no DML is allowed against the table. This allows us to maintain the validated nature of the constraint even though no index exists:

```

SQL> UPDATE sales_fact
2      SET prod_id = -13
3      WHERE prod_id = 13
4 /
UPDATE sales_fact
*
ERROR at line 1:
ORA-25128: No insert/update/delete on table with constraint (PHYSICAL_DW2.SALES_FACT_PK) disabled and
validated

Elapsed: 00:00:00.01
SQL>

```

In order to enable DML operations against the table again, the constraint will need to be DISABLED completely:

```

SQL> ALTER TABLE sales_fact
2      DISABLE PRIMARY KEY
3 /

```

Table altered.

Elapsed: 00:00:00.07

SQL>

```

SQL> UPDATE sales_fact
2      SET prod_id = -13
3      WHERE prod_id = 13
4 /

```

6002 rows updated.

Elapsed: 00:00:00.58

SQL>

```

SQL> ALTER TABLE sales_fact
2      MODIFY PRIMARY key
3      DISABLE VALIDATE
4 /

```

Table altered.

Elapsed: 00:00:08.56

SQL>

Additionally, partition-exchange loading can occur while the constraint is in the DISABLE VALIDATE mode.

Bitmap Indexes

Bitmap indexes are more appropriate for data warehouse environments than B-tree indexes. One of the myths about bitmap indexes is that they are only applicable for low-cardinality columns, such as Y/N attributes. However, when considering large data sets, cardinality is a relative term, and with millions or billions of rows in a table, 1000 or even 10,000 distinct values is a very low cardinality. Bitmaps also enable the optimization strategy called Star Transformation, discussed below.

Bitmap Index Maintenance

Bitmap indexes may cause issues during DML statements. Though the performance of DML statements on tables with bitmap indexes has increased through the various versions of the database from 9i through 11g, it is still recommended to alter bitmap indexes to UNUSABLE before DML operations, and to REBUILD them once the DML operation is complete. Because all LOCAL indexes, bitmaps included, can be marked UNUSABLE at the partition level, this is beneficial for loading event-based fact tables because a daily load would usually affect only a few partitions during an incremental load:

```
SQL> alter table sales_fact
 2      modify partition sales_q4_2003
 3      unusable local indexes
 4 /
```

Table altered.

Elapsed: 00:00:00.05

```
SQL>
SQL> INSERT INTO sales_fact
 2      ( prod_id,
 3      cust_id,
 4      time_id,
 5      channel_id,
 6      promo_id,
 7      quantity_sold,
 8      amount_sold )
 9      VALUES
10      ( 14,
11      3001,
12      '11/05/2002 12:00:00 AM',
13      3,
14      999,
15      1,
16      1259.99 )
17 /
```

1 row created.

Elapsed: 00:00:00.00

```
SQL>
SQL> alter table sales_fact
 2      modify partition sales_q4_2002
 3      unusable local indexes
 4 /
```

Table altered.

Elapsed: 00:00:00.05

```
SQL>
```

Star Transformation

Bitmap indexes should be included on all surrogate key columns on the fact tables, with foreign keys created and either enabled or added with the rely clause. Either enabled or disabled primary keys should also be created on the surrogate keys for all dimension tables. This will enable the optimizer to use star transformation, a powerful optimization technique which makes queries against star schemas perform much faster and require fewer resources on from the database.

Summary Management

Summary management is a powerful technique to improve query response time in data warehouses. Instead of aggregating millions of rows of data every time a user runs a report, storing pre-calculated summary data allows these values to be calculated only once. But in building these aggregate structures, we have to decide how we will *navigate* to them. Aggregate navigation describes the process of finding and querying usable aggregates in lieu of the base fact tables.

Manual Navigation

Manual navigation is the most simplistic form of aggregate navigation. Aggregate tables are delivered as part of the logical model. The end user decides which physical table contains the appropriate level of aggregation and uses that table in his or her report. The data warehouse team would publish a catalog to the end users so they can understand the different levels of aggregation for each fact table. The aggregate tables are loaded using ETL processes resembling the ones loading the base fact tables.

Manual navigation also requires different versions of dimension tables to correspond to each level of aggregation. If sales data is aggregated at the month level in a table called SALES_MONTH_AGG, then the existing date dimension table that exists at the grain of an individual day would not be applicable for this aggregate. We would require an additional date dimension table that contains a single row for each month in the data warehouse, and only the attributes at the month level and higher would be included. Kimball refers to these higher-level dimension tables as “shrunk dimensions.”¹⁰ Shrunk dimension tables would be published to end users in the catalog.

Tool Navigation

Tool navigation is just an advanced form of manual navigation. The same physical structures have to be delivered: many different versions of the same fact table at different levels of aggregation, and the shrunk dimension tables that go with them.

Sophisticated BI tools such as Oracle Business Intelligence Enterprise Edition (OBIEE) are able to hide all the different versions of a fact table — the base fact and all the different aggregates derived from it — in a single “logical fact table” that is aware of the different levels of aggregation. OBIEE does the same thing with dimension tables: it exposes a single “logical dimension table” that subsumes the base dimension table and all the shrunk dimensions that are created to go with it. As reports and dashboards reference these logical facts and dimension tables instead of the actual ones, new aggregates can be added where they didn’t exist before without affecting the presentation layer.

OBIEE is an interesting option to consider for aggregate navigation when implementing a summary management strategy. OBIEE has an optimizer of sorts called the BI Server which considers all the

¹⁰ Kimball, [Lifecycle](#) 557

different ways the underlying physical structures can be joined to present efficient queries to the database. But the most efficient method of summary management is to let the Oracle Database optimizer handle aggregate navigation with a feature called query rewrite.

Query Rewrite

The BI Server can only hide the complexity of numerous facts and dimension tables from the presentation tools within the OBIEE framework. Query rewrite can provide aggregate navigation to *any* tool that connects to the database and issue SQL queries. Query rewrite is implemented using materialized views, a common, easy and familiar feature in the database. Oracle also provides the DBMS_ADVISOR package which can analyze query patterns and recommend generic aggregates.

The Oracle Database also provides functionality for incrementally refreshing aggregate tables that are based on partitioned tables, which all base fact tables should be. The feature is called partition-change tracking (PCT) and it uses the metadata in the Oracle Database to determine which partitions in the base fact table have been modified with new or updated data. Because the database understands which partition aggregate rows points back to, a materialized view refresh is able to only modify the aggregate calculations with underlying data that has changed.

We start by creating a standard materialized view, but we make sure we add FOR QUERY REWRITE in the DDL:

```
SQL> CREATE MATERIALIZED VIEW product_sales_mv
 2      BUILD IMMEDIATE
 3      ENABLE QUERY REWRITE
 4      AS SELECT prod_name,
 5                prod_category,
 6                calendar_quarter_desc,
 7                SUM(amount_sold) AS dollar_sales,
 8                COUNT(*) AS cnt,
 9                COUNT(amount_sold) AS cnt_amt
10     FROM sales_fact
11     JOIN product_dim
12     USING (prod_id)
13     JOIN time_dim
14     USING (time_id)
15     GROUP BY prod_name,
16                prod_category,
17                calendar_quarter_desc
18 /
```

Materialized view created.

Elapsed: 00:00:07.55

SQL>

Now, when we execute a query that accesses the base fact table SALES_FACT, we can see that the optimizer steers us clear of that segment and instead queries the materialized view:

```
SQL> SELECT prod_name,
 2          prod_category,
 3          calendar_quarter_desc,
 4          SUM(amount_sold) AS dollar_sales
 5     FROM sales_fact
 6     JOIN product_dim
 7     USING (prod_id)
 8     JOIN time_dim
 9     USING (time_id)
10     GROUP BY prod_name,
11                prod_category,
12                calendar_quarter_desc
```

```
13 /
```

```
1063 rows selected.
```

```
Elapsed: 00:00:00.13
```

```
Execution Plan
```

```
-----
```

```
Plan hash value: 4155884679
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1063	80788	6 (0)	00:00:01
1	MAT_VIEW REWRITE ACCESS FULL	PRODUCT_SALES_MV	1063	80788	6 (0)	00:00:01

```
Note
```

```
-----
```

```
- dynamic sampling used for this statement (level=2)
```

```
SQL>
```

This example may cause us to believe that we need a materialized view for each specific query that we want to execute, but this is not the case. The database is able to do generic query rewrite — meaning that the query doesn't have to match the materialized exactly — using complex heuristics and metadata stored in the data dictionary. In the following example, we see that the optimizer can even do a GROUP BY on top of an aggregate table to aggregate the data further:

```
SQL> SELECT prod_name,
2         calendar_quarter_desc,
3         SUM(amount_sold) AS dollar_sales
4 FROM sales_fact
5 JOIN product_dim
6 USING (prod_id)
7 JOIN time_dim
8 USING (time_id)
9 GROUP BY prod_name,
10        calendar_quarter_desc
11 /
```

```
1063 rows selected.
```

```
Elapsed: 00:00:00.06
```

```
Execution Plan
```

```
-----
```

```
Plan hash value: 109899774
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1063	52087	7 (15)	00:00:01
1	HASH GROUP BY		1063	52087	7 (15)	00:00:01
2	MAT_VIEW REWRITE ACCESS FULL	PRODUCT_SALES_MV	1063	52087	6 (0)	00:00:01

```
Note
```

```
-----
```

```
- dynamic sampling used for this statement (level=2)
```

```
SQL>
```

Notice that the dimension tables are joined in the materialized view statement and actually form the elements in the GROUP BY. Though this provides the best performance for query rewrite, it introduces a major limitation in refreshing materialized views using PCT. PCT requires that all tables referenced in

the materialized view statement are partitioned, and PCT will only be effective if the partitioning strategy segments the data in intelligent ways, such as RANGE partitioning on a DATE attribute. In the typical data warehouse, it's only our fact tables that are partitioned in this way. PCT will still work if the non-partitioned tables haven't been modified, which is the case with static dimension tables such as our date dimension, or junk dimension tables designed to be used with specific fact tables. The majority of our dimension tables are loaded with fresh data every day. Can we still get the reporting effectiveness of query rewrite with the intelligent refresh provided with PCT?

We can have the best of both worlds by using a feature in query rewrite called dimension join-back. In this scenario, we create aggregates that only reference fact tables. The optimizer will still be able to use query rewrite against the fact table, but instead of storing the required dimensional attributes necessary, the database creates a shrunken dimension table in memory and joins it to the materialized view. Dimension join-back provides the right mix between query performance and refresh performance, and allows us to create very generic aggregate tables that will be used in the majority of queries.

We start by recreating our materialized view, but this time, only referencing the fact table in the DDL:

```
SQL> CREATE MATERIALIZED VIEW product_sales_mv
 2      BUILD IMMEDIATE
 3      ENABLE QUERY REWRITE
 4      AS SELECT prod_id,
 5                time_id,
 6                SUM(amount_sold) AS dollar_sales,
 7                COUNT(*) AS cnt,
 8                COUNT(amount_sold) AS cnt_amt
 9      FROM sales_fact
10     GROUP BY prod_id,
11            time_id
12 /
```

Materialized view created.

Elapsed: 00:00:03.93
SQL>

Now, when we query the SALES_FACT table, the optimizer is intelligent enough to realize that a join between the required dimension tables and PRODUCT_SALES_MV will be more efficient than joining them to SALES_FACT:

```
SQL> SELECT prod_name,
 2          prod_category,
 3          calendar_quarter_desc,
 4          SUM(amount_sold) AS dollar_sales
 5 FROM sales_fact
 6 JOIN product_dim
 7   USING (prod_id)
 8 JOIN time_dim
 9   USING (time_id)
10 GROUP BY prod_name,
11          prod_category,
12          calendar_quarter_desc
13 /
```

1063 rows selected.

Elapsed: 00:00:00.45

Execution Plan

Plan hash value: 2094493660

```
-----
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		252	20412	70 (6)	00:00:01
1	HASH GROUP BY		252	20412	70 (6)	00:00:01
* 2	HASH JOIN		252	20412	69 (5)	00:00:01
3	VIEW	VW_GBC_9	252	16380	51 (6)	00:00:01
4	HASH GROUP BY		252	25452	51 (6)	00:00:01
* 5	HASH JOIN		36954	3644K	49 (3)	00:00:01
6	TABLE ACCESS FULL	PRODUCT_DIM	72	3384	3 (0)	00:00:01
7	MAT_VIEW REWRITE ACCESS FULL	PRODUCT_SALES_MV	36954	1948K	45 (0)	00:00:01
8	TABLE ACCESS FULL	TIME_DIM	1826	29216	18 (0)	00:00:01

```
-----
```

Predicate Information (identified by operation id):

- ```

```
- 2 - access("ITEM\_1"="TIME\_DIM"."TIME\_ID")
  - 5 - access("PRODUCT\_SALES\_MV"."PROD\_ID"="PRODUCT\_DIM"."PROD\_ID")

Note

- ```
-----
```
- dynamic sampling used for this statement (level=2)

SQL>

It's even capable of rolling up to higher levels of granularity as we saw above, but still using the dimension join-back feature:

```
SQL> SELECT prod_name,
2         calendar_quarter_desc,
3         SUM(amount_sold) AS dollar_sales
4 FROM sales_fact
5 JOIN product_dim
6 USING (prod_id)
7 JOIN time_dim
8 USING (time_id)
9 GROUP BY prod_name,
10        calendar_quarter_desc
11 /
```

1063 rows selected.

Elapsed: 00:00:00.16

Execution Plan

```
-----
```

Plan hash value: 2043978574

```
-----
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1005	81405	69 (5)	00:00:01
1	HASH GROUP BY		1005	81405	69 (5)	00:00:01
* 2	HASH JOIN		36954	2923K	68 (3)	00:00:01
3	TABLE ACCESS FULL	TIME_DIM	1826	29216	18 (0)	00:00:01
* 4	HASH JOIN		36954	2345K	49 (3)	00:00:01
5	TABLE ACCESS FULL	PRODUCT_DIM	72	2160	3 (0)	00:00:01
6	MAT_VIEW REWRITE ACCESS FULL	PRODUCT_SALES_MV	36954	1263K	45 (0)	00:00:01

```
-----
```

Predicate Information (identified by operation id):

- ```

```
- 2 - access("PRODUCT\_SALES\_MV"."TIME\_ID"="TIME\_DIM"."TIME\_ID")
  - 4 - access("PRODUCT\_SALES\_MV"."PROD\_ID"="PRODUCT\_DIM"."PROD\_ID")

Note

- ```
-----
```
- dynamic sampling used for this statement (level=2)

Conclusion

Dimensional models makes the data model and hence the data in the warehouse easy to understand, which has a twofold benefit. First, business users are able to intuitively query and hence analyze data in the warehouse. The objects in the warehouse are defined in business terms, and the model is designed to clearly and simply represent business entities. Users do not have to understand the intricacies of 3NF to interrogate the data warehouse. Second, reporting tools are easy to integrate with the warehouse, as the underlying schema is effectively an industry standard. The dimensional model provides a common structure that the majority of the tools in the industry can plug-in to.

The Oracle Database also understands dimensional models, and many of the high-end data warehousing features are designed to work with star schemas. Star transformation requires a series of constraints on the dimension and fact tables so the optimizer can be sure it is querying a star schema. Query rewrite relies on the dimensional model, including the hierarchies constructed in dimension tables, to effectively navigate between different levels of aggregation. Understanding the nuances of dimensional modeling, and how it differs from standard ERD modeling, will make these high-end data warehousing features accessible to more traditional DBA's.