CHAPTER

1

Fundamentals of Aggregates

A decade ago, Ralph Kimball described aggregate tables as "the single most dramatic way to improve performance in a large data warehouse." Writing in *DBMS Magazine* ("Aggregate Navigation with (Almost) No Metadata," August 1996), Kimball continued:

> *Aggregates can have a very significant effect on performance, in some cases speeding queries by a factor of one hundred or even one thousand. No other means exist to harvest such spectacular gains.*

This statement rings as true today as it did ten years ago. Since then, advances in hardware and software have dramatically improved the capacity and performance of the data warehouse. Aggregates *compound* the effect of these improvements, providing performance gains that fully harness capabilities of the underlying technologies.

And the pressure to improve data warehouse performance is as strong as ever. As the baseline performance of underlying technologies has improved, warehouse developers have responded by storing and analyzing larger and more granular volumes of data. At the same time, warehouse systems have been opened to larger numbers of users, internal and external, who have come to expect instantaneous access to information.

This book empowers *you* to address these pressures. Using aggregate tables, you can achieve an extraordinary improvement in the speed of *your* data warehouse. And you can do it today, without making expensive upgrades to hardware, converting to a new database platform, or investing in exotic and proprietary technologies.

Although aggregates can have a powerful impact on data warehouse performance, they can also be misused. If not managed carefully, they can cause confusion, impose inordinate maintenance requirements, consume massive amounts of storage, and even provide inaccurate results. By following the best practices developed in this book, you can avoid these outcomes and maximize the positive impact of aggregates.

The introduction of aggregate tables to the data warehouse will touch every aspect of the data warehouse lifecycle. A set of best practices governs their selection, design, construction, and usage. They will influence data warehouse planning, project scope, maintenance requirements, and even the archive process. Before exploring each of these topics, it is necessary to establish some fundamental principles and vocabulary.

This chapter establishes the foundation on which the rest of the book builds. It introduces the *star schema*, *aggregate tables*, and the *aggregate navigator*. Guiding principles are established for the development of *invisible aggregates*, which have zero impact on production applications—other than performance, of course. Last, this chapter explores several other forms of summarization that are not invisible to applications, but may also provide useful performance benefits.

Star Schema Basics

A star schema is a set of tables in a relational database that has been designed according to the principles of *dimensional modeling*. Ralph Kimball popularized this approach to data warehouse design in the 1990s. Through his work and writings, Kimball established standard terminology and best practices that are now used around the world to design and build data warehouse systems. With coauthor Margy Ross, he provides a detailed treatment of these principles in *The Data Warehouse Toolkit, Second Edition* (Wiley, 2002).

To follow the examples throughout this book, you must understand the fundamental principles of dimensional modeling. In particular, the reader must have a basic grasp of the following concepts:

- The differences between data warehouse systems and operational systems
- How facts and dimensions support the measurement of a business process
- The tables of a star schema (fact tables and dimension tables) and their purposes

- The purpose of surrogate keys in dimension tables
- The grain of a fact table
- The additivity of facts
- How a star schema is queried
- Drilling across multiple fact tables
- Conformed dimensions and the warehouse bus
- The basic architecture of a data warehouse, including ETL software and BI software

If you are familiar with these topics, you may wish to skip to the section "Invisible Aggregates," later in this chapter.

For everyone else, this section will bring you up-to-speed. Although not a substitute for Kimball and Ross's book, this overview provides the background needed to understand the examples throughout this book. I encourage *all* readers to read *The Toolkit* for more immersion in the principles of dimensional modeling, particularly anyone involved in the design of the dimensional data warehouse.

Data warehouse designers will also benefit from reading *Data Warehouse Design Solutions*, by Chris Adamson and Mike Venerable (Wiley, 1998). This book explores the application of these principles in the service of specific business objectives and covers standard business processes in a wide variety of industries.

Operational Systems and the Data Warehouse

Data warehouse systems and operational systems have fundamentally different purposes. An operational system supports the *execution* of business process, while the data warehouse supports the *evaluation* of the process. Their distinct purposes are reflected in contrasting usage profiles, which in turn suggest that different principles will guide their design. The principles of dimensional modeling are specifically adapted to the unique requirements of the warehouse system.

Operational Systems

An operational system directly supports the *execution* of business processes. By capturing detail about significant events or transactions, it constructs a record of the activity. A sales system, for example, captures information about orders, shipments, and returns; a human resources system captures information about the hiring and promotion of employees; an accounting system captures information about the management of the financial assets and liabilities of the business. Capturing the detail surrounding these activities is often so important that the operational system becomes a part of the process.

To facilitate execution of the business process, an operational system must enable several types of database interaction, including inserts, updates, and deletes. Operational systems are often referred to as transaction systems. The focus of these interactions is almost always atomic—a specific order, a shipment, a refund. These interactions will be highly predictable in nature. For example, an order entry system must provide for the management of lists of products, customers, and salespeople; the entering of orders; the printing of order summaries, invoices, and packing lists; and the tracking of order status.

Implemented in a relational database, the optimal design for an operational schema is widely accepted to be one that is in *third normal form*. This design supports the high performance insertion, update, and deletion of atomic data in a consistent and predictable manner. This form of schema design is discussed in more detail in Chapter 8.

Because it is focused on process execution, the operational system is likely to update data as things change, and purge or archive data once its operational usefulness has ended. Once a customer has established a new address, for example, the old one is unnecessary. A year after a sales order has been fulfilled and reflected in financial reports, it is no longer necessary to maintain information about it in the order entry system.

Data Warehouse Systems

While the focus of the operational system is the *execution* of a business process, a data warehouse system supports the *evaluation* of the process. How are orders trending this month versus last? Where does this put us in comparison to our sales goals for the quarter? Is a particular marketing promotion having an impact on sales? Who are our best customers? These questions deal with the *measurement* of the overall orders process, rather than asking about individual orders.

Interaction with the data warehouse takes place exclusively through queries that retrieve data; information is not created or modified. These interactions will involve large numbers of transactions, rather than focusing on individual transactions. Specific questions asked are less predictable, and more likely to change over time. And historic data will remain important in the data warehouse system long after its operational use has passed. The differences between operational systems and data warehouse systems are highlighted in Figure 1.1.

The principles of dimensional modeling address the unique requirements of data warehouse systems. A star schema design is optimized for queries that access large volumes of data, rather than individual transactions. It supports the maintenance of historic data, even as the operational systems change or delete information. As a model of process measurements, the dimensional schema is able to address a wide variety of questions, even those that are not posed in advance of its implementation.

	Operational System	**Data Warehouse**
Also Known as	Transaction System On Line Transaction Processing (OLTP) System Source system	Analytic system Data mart
Purpose	Execution of a business process	Measurement of a business process
Primary Interaction Style	Insert, Update, Query, Delete	Query
Scope of Interaction	Individual transaction	Aggregated transactions
Query Patterns	Predictable and stable	Unpredictable and changing
Temporal Focus	Current	Current and historic
Design Principle	Third normal form (3NF)	Dimensional design (star schema)

Figure 1.1 Operational systems versus data warehouse systems.

Facts and Dimensions

A dimensional model divides the information associated with a business process into two major categories, called *facts* and *dimensions*. Facts are the measurements by which a process is evaluated. For example, the business process of taking customer orders is measured in at least three ways: quantities ordered, the dollar amount of orders, and the internal cost of the products ordered. These process measurements are listed as facts in Table 1.1.

On its own, a fact offers little value. If someone were to tell you, "Order dollars were $200,000," you would not have enough information to evaluate the process of booking orders. Over what time period was the $200,000 in orders taken? Who were the customers? Which products were sold? Without some context, the measurement is useless.

Dimensions give facts their context. They specify the parameters by which a measurement is stated. Consider the statement "January 2006 orders for packing materials from customers in the nortßheast totaled $200,000." This time, the order dollars fact is given context that makes it useful. The $200,000 represents orders taken in a specific *month* and *year* (January 2006) for all products in a *category* (packing materials) by customers in a *region* (the northeast). These dimensions give context to the order dollars fact. Additional dimensions for the orders process are listed in Table 1.1.

Table 1.1 Facts and Dimensions Associated with the Orders Process

FACTS	DIMENSIONS	
Quantity Sold	Date of Order	Sales Region
Order Dollars	Month of Order	Region Code
Cost Dollars	Year of Order	Region Vice President
	Product	Customer
	Product Description	Customer Headquarters State
	Product SKU	Customer's Billing Address
	Unit of Measure	Customer's Billing City
	Product Brand	Customer's Billing State
	Brand Code	Customer's Billing Zip Code
	Brand Manager	Customer Industry SIC Code
	Product Category	Customer Industry Name
	Category Code	Order Number
	Salesperson	Credit Flag
	Salesperson ID	Carryover Flag
	Sales Territory	Solicited Order Flag
	Territory Code	Reorder Flag
	Territory Manager	

TIP A dimensional model describes a process in terms of facts and dimensions. Facts are metrics that describe the process; dimensions give facts their context.

The dimensions associated with a process usually fall into groups that are readily understood within the business. The dimensions in Table 1.1 can be sorted into groups for the product (including name, SKU, category, and brand), the salesperson (including name, sales territory, and sales region), the customer (including billing information and industry classification data), and the date of the order. This leaves a group of miscellaneous dimensions, including the order number and several flags that describe various characteristics.

The Star Schema

In a dimensional model, each group of dimensions is placed in a *dimension table*; the facts are placed in a *fact table*. The result is a *star schema*, so called because it resembles a star when diagrammed with the fact table in the center. A star schema for the orders process is shown in Figure 1.2.

The dimension tables in a star schema are *wide*. They contain a large number of attributes providing rich contextual data to support a wide variety of reports and analyses. Each dimension table has a primary key, specifically assigned for the data warehouse, called a *surrogate key*. This will allow the data warehouse to track the history of changes to data elements, even if source systems do not.

Fact tables are *deep*. They contain a large number of rows, each of which is relatively compact. Foreign key columns associate each fact table row with the dimension tables. The level of detail represented by each row in a fact table must be consistent; this level of detail is referred to as *grain*.

Dimension Tables and Surrogate Keys

A dimension table contains a set of dimensional attributes and a key column. The star schema for the orders process contains dimension tables for groups of attributes describing the Product, Customer, Salesperson, Date, and Order Type. Each dimensional attribute appears as a column in one of these tables, with the exception of order_id, which is examined shortly. Each key column is a new data element, assigned during the load process and used exclusively by the warehouse.

TIP In popular usage, the word *dimension* has two meanings. It is used to describe a dimension table within a star schema, as well as the individual attributes it contains. This book distinguishes between the table and its attributes by using the terms *dimension table* for the table, and *dimension* for the attribute.

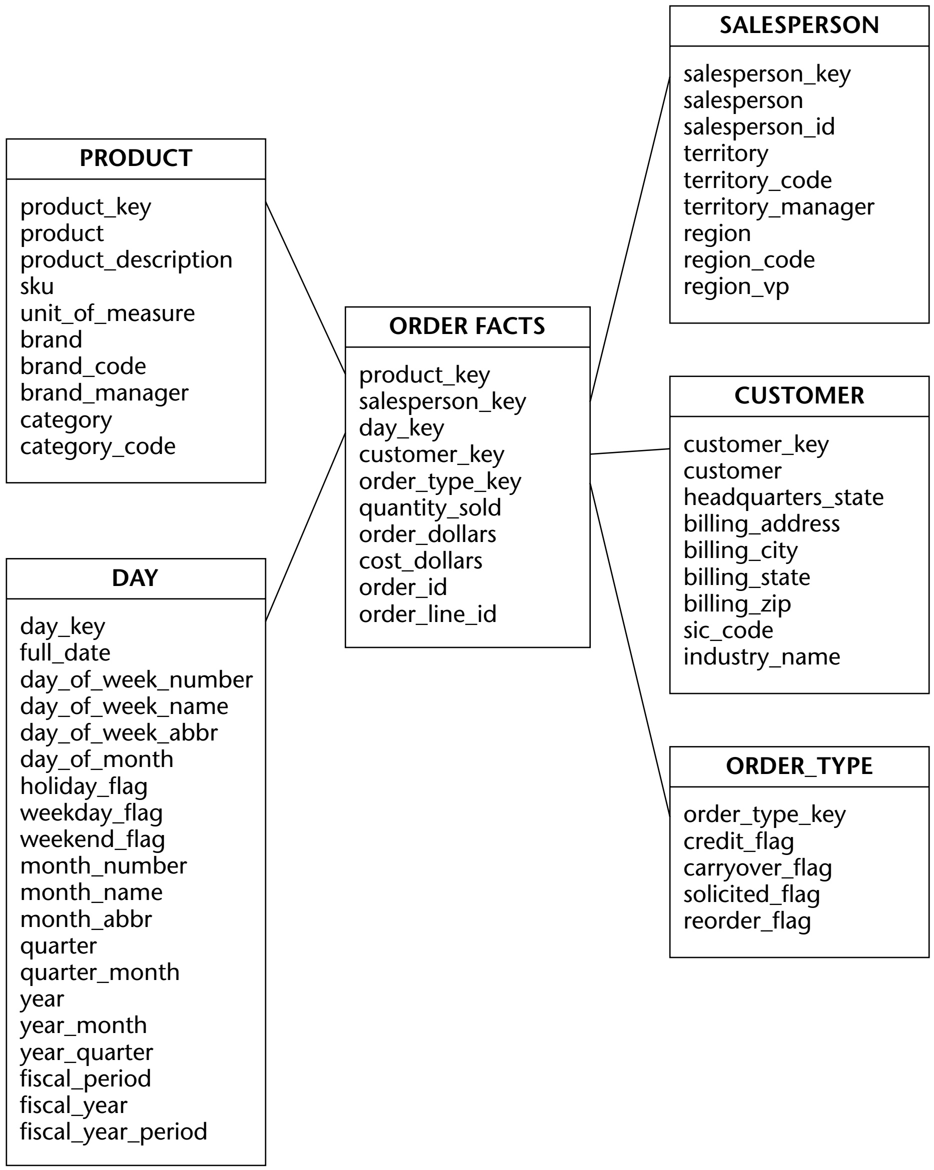


Figure 1.2 A star schema for the orders process.

Dimensions provide all context for facts. They are used to filter data for reports, drive master detail relationships, determine how facts will be aggregated, and appear with facts on reports. A rich set of descriptive dimensional attributes provides for powerful and informative reporting. Schema designers therefore focus a significant amount of time and energy identifying useful dimensional attributes. Columns whose instance values are codes, such as brand_id, may be supplemented with additional columns that decode these values into descriptive text, such as brand_name. The contents of Boolean columns or flags, such as credit_flag, are recorded in descriptive manners, such as Credit Order and Non-Credit Order. Multi-part fields or codes, such as an account code, may exist in a dimension table, along with additional columns that represent the different parts. When multiple dimensional attributes are commonly combined, such as title, first_name, and last_name, the concatenations are added to the dimension table as well.

All these efforts serve to provide a set of dimensions that will drive a rich set of reports. Dimension tables are often referred to as wide because they contain a large number of attributes, most of which are textual. Although this may consume some additional space, it makes the schema more usable. And in comparison to the fact table, the number of rows in most dimension tables is small.

TIP Dimension tables should be wide. A rich set of dimensional attributes enables useful and powerful reports.

Each dimension table has a primary key. Rather than reuse a key that exists in a transaction system, a *surrogate key* is assigned specifically for the data warehouse. Surrogate keys in this book are identifiable by the usage of the suffix *_key*, as in *product_key*, *customer_key*, and so forth. A surrogate key allows the data warehouse to track the history of changes to dimensions, even if the source systems do not. The ways in which a surrogate key supports the tracking of history will be studied in Chapter 5.

Any key that carries over from the transaction system, such as *sku* or *salesperson_id*, is referred to as a *natural key*. These columns may have analytic value, and are placed in the dimension tables as dimensional attributes. Their presence will also enable the process that loads fact tables to identify dimension records to which new fact table records will be associated. You learn more about the relationship between natural and surrogate keys, and the lookup process, in Chapter 5.

TIP A *surrogate key* is assigned to each dimension table and managed by the data warehouse load process. Key columns or unique identifiers from source systems will not participate in the primary key of the dimension table, but are included as dimensional attributes. These natural keys may have analytic value, and will also support the assignment of warehouse keys to new records being added to fact tables.

Dimension tables commonly store repeating values. For example, the Product table in Figure 1.2 contains several dimensional attributes related to a brand. Assuming there are several products within a given brand, the attribute values for these columns will be stored repeatedly, once for each product. An alternative schema design, known as the *snowflake schema*, seeks to eliminate this redundancy by further normalizing the dimensions. Figure 1.3 shows a snowflaked version of the Orders schema.

The snowflake approach saves some storage space but introduces new considerations. When the size of the dimension tables is compared to that of fact tables, the space saved by a snowflake design is negligible. In exchange for this small savings, the schema itself has become more complex. More tables must be loaded by the ETL process. Queries against the snowflake will require additional joins, potentially affecting performance. However, some data warehouse tools are optimized for a snowflake design. If such a tool is part of the data warehouse architecture, a snowflake design may be the best choice.

TIP **Avoid the snowflake design unless a component of the architecture requires it. The space saved is minimal, and complexity is added to the query and reporting processes.**

Fact Tables and Grain

A fact table contains the facts associated with a process, and foreign keys that provide dimensional context. The fact table for the order entry process appears in the center of Figure 1.2. It contains the three facts for the orders process: quantity_sold, order_dollars, and cost_dollars. In addition to the facts, it contains foreign keys that reference each of the relevant dimension tables: Customer, Product, Salesperson, Day, and order_type.

TIP **Although a fact table is not always placed in the center of a diagram, it is easily identifiable as a dependent table, carrying a set of foreign keys.**

The fact table in Figure 1.2 contains two additional attributes, order_id and order_line_id, which are neither facts nor foreign keys. These columns reference a specific order and the line number within an order respectively. They are dimensional attributes that have been placed in the fact table rather than a dimension table. Although they could have been placed in the order_type table, doing so would have dramatically increased the number of rows in the table. When a dimensional attribute is located in the fact table, it is known as a *degenerate dimension*.

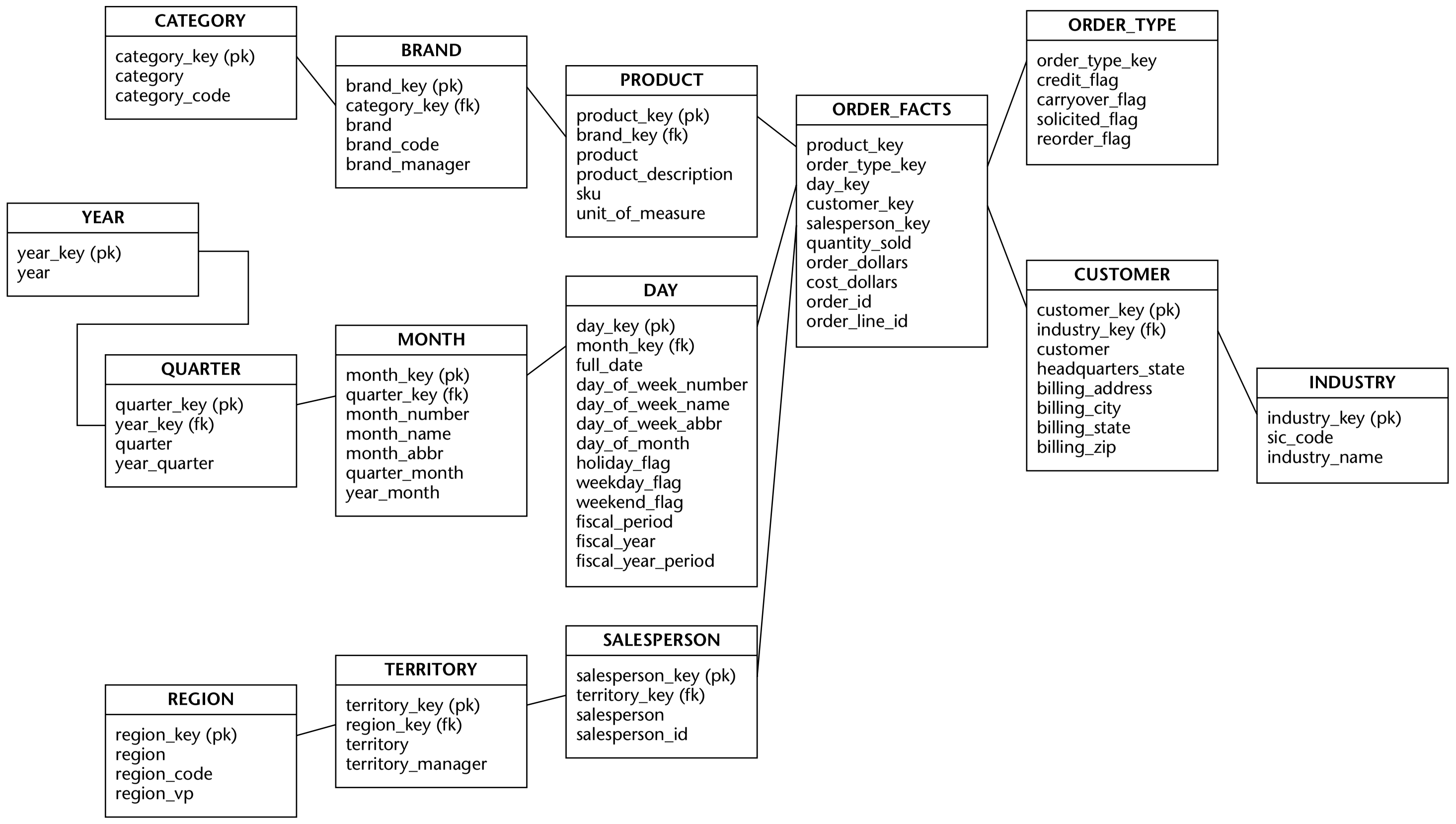


Figure 1.3 A snowflake schema for the orders process.

TIP Fact tables contain facts, foreign keys that reference dimension tables, and degenerate dimensions.

In each row of a fact table, all facts are recorded at the same level of detail. This level of detail is determined by the dimensional attributes present in the schema design. In the case of order_facts, the design requires each fact to have an associated order date, customer, salesperson, product, order_type, order_id, and order_line_id. If certain dimensional attributes do not apply to certain facts, these facts must be placed in a separate fact table. I examine designs involving multiple fact tables shortly.

The level of detail represented by a fact table row is referred to as its *grain*. Declaring the grain of a fact table is an important part of the schema design process. It ensures that there is no confusion about the meaning of a fact table row, and guarantees that all facts will be recorded at the same level of detail. A clear understanding of grain is also necessary when designing aggregate tables, as you will see in Chapter 2.

Grain may be declared in dimensional terms, or through reference to an artifact of the business process. The grain of order_facts, declared dimensionally, is "order facts by order Date, Customer, Salesperson, Product, order_id, and order_line." Because an individual line on a specific order contains all this information, the grain of order_facts can also be declared as "order facts at the order line level of detail."

The grain of a fact table row is always set at the lowest possible level of detail, based on what is available in the transaction systems where data is collected. The schema can always be used to produce less detailed summaries of the facts, as you will see in a moment. Detail not captured by the schema can never be reported on. Schema designers therefore look to capture data at an atomic level.

TIP The level of detail represented by a fact table row is referred to as its *grain*. Grain may be declared dimensionally or through reference to an artifact of the business process. The grain of a dimensional schema should capture data at the lowest possible level of detail.

The rate of growth of a fact table is greater than that of a dimension table. Although a fact table may start out small, it will quickly surpass dimensions in terms of number of rows, and become the largest table in the schema design. For example, consider the star schema for order facts. There may be several hundred products in the Product dimension table, a few hundred salespeople in the Salesperson table, one hundred thousand customers in the Customer table, sixteen possible combinations of flags in the Order_Type table, and five years worth of dates, or 1826 rows, in the Day table. Of these tables, the Product dimension table is the largest, and it may be expected to grow by about 10 percent per year. Fact table size can be estimated based on an average volume of orders. If the number of order lines per day averages 10,000, the fact table will contain 3,650,000 rows by the end of one year.

Although fact tables contain far more rows than dimension tables, each row is far more compact. Because its primary contents are facts and foreign keys, each fact table row is a very efficient consumer of storage space. Contrast this to dimension tables, which contain a large number of attributes, many of which are textual. Although the typical dimension row will require far more bytes of storage, these tables will contain far fewer rows. Over time, most fact tables become several orders of magnitude larger than the largest dimension table.

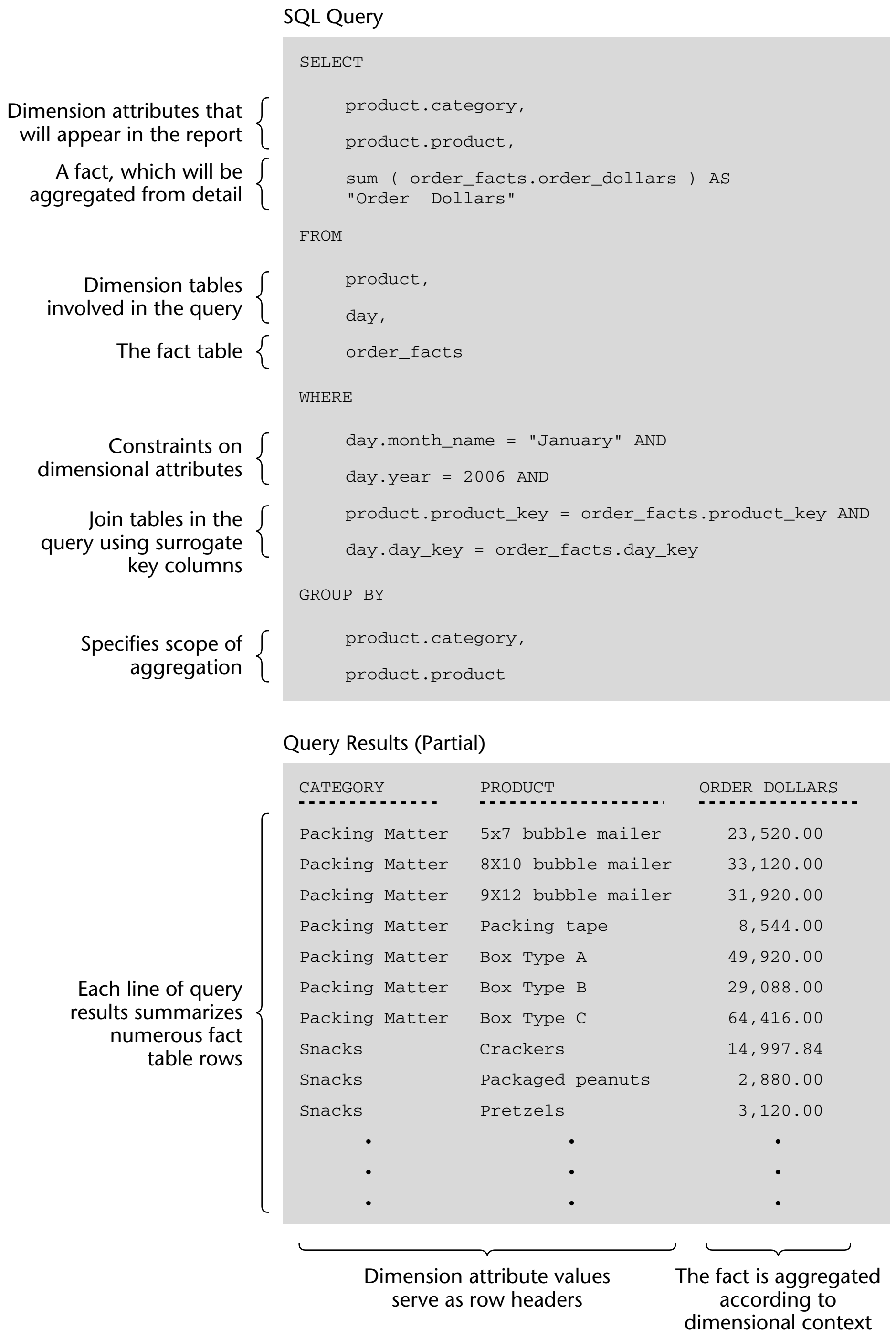
TIP Because they will accumulate a large number of rows, fact tables are often referred to as *deep*. Fact table rows are naturally compact because they contain primarily numeric data. This makes the most efficient use of physical storage.

Last, note that the fact table does not contain a row for every combination of dimension values. Rows are added only when there is a transaction. If a specific customer does not buy a specific product on a specific date, there is no fact table row for the associated set of keys. In most applications, the number of key combinations that actually appear in the fact table is actually relatively small. This property of the fact table is referred to as *sparsity*.

Using the Star Schema

The queries against a star schema follow a consistent pattern. One or more facts are typically requested, along with the dimensional attributes that provide the desired context. The facts are summarized as appropriate, based on the dimensions. Dimension attributes are also used to limit the scope of the query and serve as the basis for filters or constraints on the data to be fetched and aggregated.

A properly configured relational database is well equipped to respond to such a query, which is issued using Structured Query Language (SQL). Suppose that the vice president of sales has asked to see a report showing order_dollars by Category and Product during the month of January 2006. The Orders star schema from Figure 1.2 can provide this information. The SQL query in Figure 1.4 produces the required results, summarizing tens of thousands of fact table rows.


SQL Query
Dimension attributes that will appear in the report
A fact, which will be aggregated from detail
Dimension tables involved in the query
The fact table
Constraints on dimensional attributes
Join tables in the query using surrogate key columns
Specifies scope of aggregation
Query Results (Partial)
Each line of query results summarizes numerous fact table rows
Dimension attribute values serve as row headers
The fact is aggregated according to dimensional context

```
SELECT

    product.category,

    product.product,

    sum ( order_facts.order_dollars ) AS
    "Order  Dollars"

FROM

    product,

    day,

    order_facts

WHERE

    day.month_name = "January" AND

    day.year = 2006 AND

    product.product_key = order_facts.product_key AND

    day.day_key = order_facts.day_key

GROUP BY

    product.category,

    product.product
```

```
CATEGORY          PRODUCT              ORDER DOLLARS
-------------     -----------------    ---------------
Packing Matter    5x7 bubble mailer       23,520.00
Packing Matter    8X10 bubble mailer      33,120.00
Packing Matter    9X12 bubble mailer      31,920.00
Packing Matter    Packing tape             8,544.00
Packing Matter    Box Type A              49,920.00
Packing Matter    Box Type B              29,088.00
Packing Matter    Box Type C              64,416.00
Snacks            Crackers                14,997.84
Snacks            Packaged peanuts         2,880.00
Snacks            Pretzels                 3,120.00
      .                  .                    .
      .                  .                    .
      .                  .                    .
```

Figure 1.4 Querying the star schema.

The SELECT clause of the query indicates the dimensions that should appear in the query results (category and product), the fact that is requested (order dollars), and the manner in which it will be aggregated (through the SQL sum() operation). The relational database is well equipped to perform this aggregation operation. The FROM clause specifies the star schema tables that are involved in the query.

The WHERE clause constrains the query based on the values of specific dimensional attributes (month and year) and specifies the join relationships between tables in the query. Joins are among the most expensive operations the database must perform; notice that in the case of a star schema, dimension attributes are always a maximum of one join away from facts. Finally, the GROUP BY clause specifies the context to which the fact will be aggregated.

Most queries against a star schema follow the structure of this basic template. Any combination of facts and dimensions can be retrieved, subject to any filters, simply by adding and removing attributes from the various clauses of the query. More complex reports build on this same basic query structure by adding subqueries, performing set operations with the results of more than one query, or, as you will see in a moment, by merging query result sets from different fact tables.

TIP The star schema maximizes query performance by limiting the number of joins that must be performed and leveraging native RDBMS aggregation capabilities. Fact and dimension attributes can be mixed and matched as required, which enables the schema to answer a wide variety of questions— many of which were not anticipated at design time.

The example query takes advantage of the fact that order_dollars is *fully additive*. That is, individual order_dollars amounts can be added together over any of the dimensions in the schema. Order_dollar amounts can be summed over time, across products, across customers, and so forth. While most facts are fully additive, many are not. Facts that are *semi-additive* can be summed across some dimensions but not others. Other facts, such as ratios, percentages, or averages, are not additive at all. Examples of semi-additive and non-additive facts appear in Chapter 8. Also note that facts can be summarized in other ways; examples include counts, minimum values, maximum values, averages, and running totals.

Multiple Stars and Conformance

An enterprise data warehouse contains numerous star schemas. The data warehouse for a manufacturing business may include fact tables for orders, shipments, returns, inventory management, purchasing, manufacturing, sales goals, receivables, payables, and so forth. Each star schema permits detailed analysis of a specific business process.

Some business questions deal not with a single process, but with multiple processes. Answering these questions will require consulting multiple fact tables. This process is referred to as *drilling across*. Construction of drill-across reports requires each fact table be queried separately. Failure to do so can result in incorrect results.

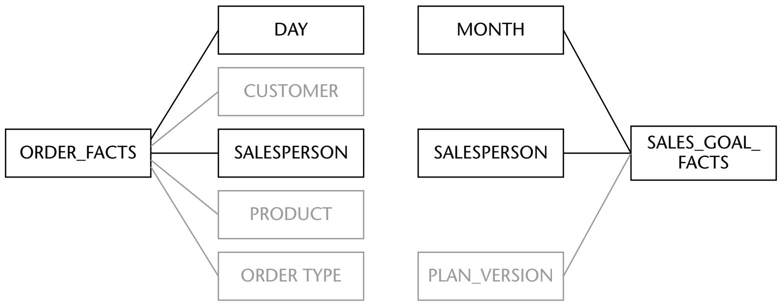
Suppose that senior management wishes to compare the performance of salespeople to sales goals on a monthly basis. The order_facts fact table, which you have already seen, tracks the actual performance of salespeople. A separate star schema records sales goals. The grain of the fact table, sales_goal_facts, is salesperson, month, and plan version. Comparison of salesperson performance to goal requires that you consult both fact tables, which appear in the top of Figure 1.5.

You cannot access both fact tables using a single query. For a given month and salesperson, there may be a single goal value in sales_goals_facts, but there may be numerous rows in order_facts. Combining both fact tables in a single query would cause the goal value for a salesperson to be repeated once for each corresponding row in order_facts. In addition, you do not want to lose track of salespeople with goals but no orders, or those with orders but no goals. Further, the month associated with an order is in the Day table, while the month associated with a sales goal is stored in the Month table. You don't want two different month values in the report.

These problems are overcome in a multiple-step process. First, each fact table is queried separately. Each query retrieves any needed facts specific to that fact table, plus dimensions that will appear in the report. The dimensions, therefore, will appear in both result sets. Next, these interim result sets are merged together. You accomplish this by performing a full outer join, based on the common dimension values. Comparisons of facts from the different fact tables can be performed once the result sets have been merged.

This process is depicted in the bottom portion of Figure 1.5. To compare order_dollars and goal_dollars by Salesperson and Month, two queries are constructed. Each is similar in form to the one described earlier in this chapter. The first query fetches Year, Month, Salesperson, and the sum of sales_dollars from the order_facts schema. The second query fetches Year, Month, Salesperson, and the sum of goal_dollars from the sales_goal_facts schema.

Because each query aggregates a fact to the same set of dimensions, the intermediate result sets now have the same grain. Each will have a maximum of one row for each combination of year, month, and salesperson. The results can now be safely joined together, without concern for double counting. This is achieved by performing a full outer join on their common dimensional attributes, which are month, year, and salesperson. The full outer join ensures that you do not lose rows from one result set that do not have a corresponding row in the other. This might occur if there is a salesperson with a goal but no orders, or vice versa. As part of this merge process, a percent of goal figure can be calculated by computing the ratio of the two facts. The result of the drill-across operation is shown in the bottom of the figure.



1. Query the Orders Star

```
SELECT
     month,
     year,
     salesperson,
     sum (order_dollars)
FROM
     . . .
```

2. Query the Sales Goals Star

```
SELECT
     month,
     year,
     salesperson,
     sum (goal_dollars)
FROM
     . . .
```

3. Merge Result Sets, Compute Ratio

```
Full outer join on common
dimension attributes:

     month,
     year,
     salesperson

Compute ratio:
     sales dollars / goal dollars
```

```
                                                 Goal       Sales     Pct
 Year       Month          Salesperson          Dollars    Dollars    Goal
------     --------     ----------------       --------   --------   ------
 2006      January      Bachman, Dee            $75,000    $73,577     98%
 2006      January      Chen, Ming                    -    $17,247       -
 2006      January      Davids, Richard        $110,000   $131,301    119%
 2006      January      Lopez, Arturo           $75,000    $77,204    103%
 2006      January      McGinty, Riley          $75,000          -      0%
 2006      January      Mitra, James           $120,000   $120,012    100%
 2006      January      Parker, Stephen         $90,000    $88,845     99%
 2006      January      Rodriguez, Alicia       $75,000    $98,407    131%
 2006      January      Smith, Susan           $110,000   $102,081     93%
 2006      January      Thompson, Tad           $99,000    $84,412     94%
 2006      January      Tolbert, Joseph        $110,000   $122,222    111%
 2006      January      Wilson, Rhonda          $90,000    $91,684    102%
```

Figure 1.5 Constructing a drill-across report.

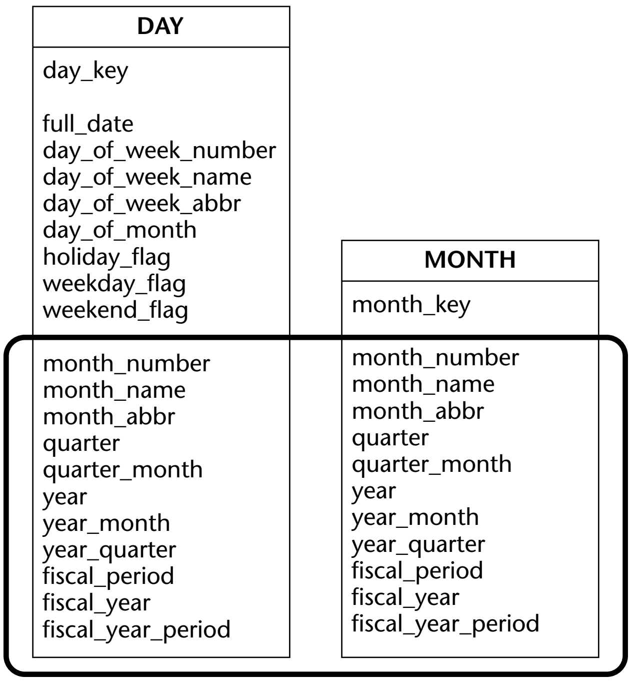
TIP Drill-across reports combine data from multiple fact tables by querying each star separately and then merging the result sets by performing a full outer join on the common dimension values.

The way the drill-across operation is carried out varies based on the tools available in the data warehouse environment and the preferences of the report developer. One option is to fetch multiple result sets from the database and then join them on a client machine or application server. Another technique stores interim result sets in temporary tables and then issues a new query that joins them. Extended SQL syntax can also be leveraged, permitting construction of a single statement that performs a full outer join on the results of multiple `SELECT` statements. Some query and reporting software products are star schema aware, and will automatically perform drill-across operations when required, using one or more of these techniques.

Successful drill-across reports cannot be constructed without consistent representation of dimensional attributes and values. The comparison of goal dollars to order dollars would not have been possible if each schema represented salesperson differently. To drill across the two fact tables, the salesperson dimension attributes and values must be identical. This is guaranteed if each star uses the same physical Salesperson table. This may not be possible if the two stars reside in different databases, perhaps even running RDBMS products from different vendors. But if the Salesperson table is the same in both databases, it is still possible to drill across. The warehouse team must ensure that both tables have the same columns, the attribute values are identical, and the same combinations of attribute values exist in both tables. When these conditions are met, the tables are identical in form and content, and are said to *conform*.

Dimensions are also said to conform if one is a perfect subset of the other. The Month table, for example, contains a subset of the attributes of the Day table. This is illustrated in Figure 1.6. The values of the common attributes must be recorded identically, and each table must have exactly the same set of distinct values for the common attributes. When these conditions are met, Month is referred to as a *conformed rollup* of the Day dimension.

Without dimensional conformance, it is not possible to pose business questions that involve multiple fact tables. You would not be able to compare performance to goals, orders to shipments, sales to inventory, contracts to payments, and so forth. Instead, individual stars become known as *stovepipes*. Each works on its own but cannot integrate with the others. And incompatible representations of the same business concept, such as product or customer, may foster distrust in the individual stars as well.



Conformed Dimensions:

- Attributes of one table are a subset of the other's.

  The only exception is the key column.

- Attribute values are recorded identically.

  "January" does not conform with "JANUARY" or "Jan"

- Each table contains the same distinct combinations of values for the common attributes.

  There is no month/year combination present in one table but not in the other.

Figure 1.6 Month is a conformed rollup of Day.

To maximize the value and success of a dimensional data warehouse, it is therefore critical to ensure dimensional conformance. A common set of dimensions is planned and cross-referenced with the various business processes that will be represented by fact tables. This common dimensional framework is referred to as the *data warehouse bus*. As you will see in Chapter 7, planning dimensional conformance in advance allows the data warehouse to be implemented one subject area at a time, while avoiding the potential of stovepipes. These subject areas are called *data marts*. Each data mart provides direct value when implemented, and will integrate with others as they are brought on-line.

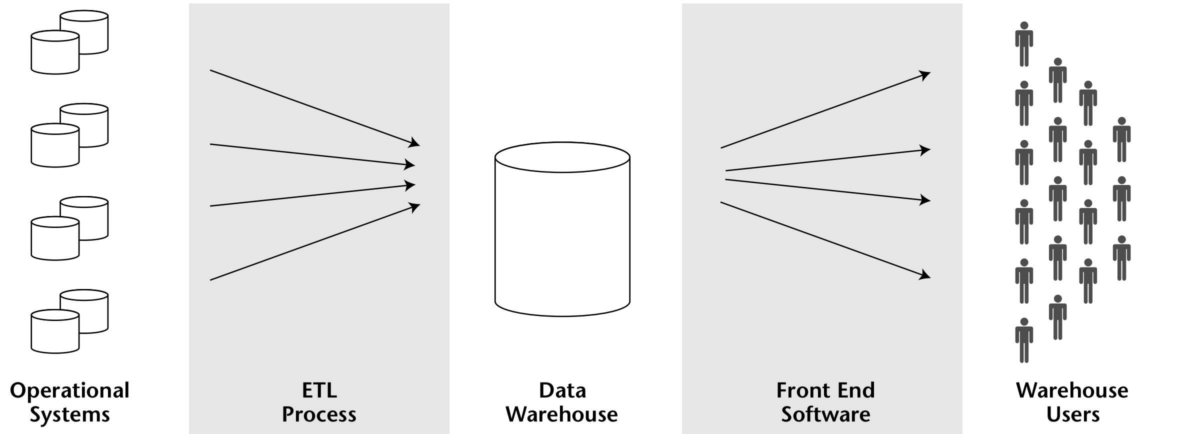
TIP Conformed dimensions are required to compare data from multiple fact tables. The set of conformed dimensions for an enterprise data warehouse is referred to as the *warehouse bus*. Planned in advance, the warehouse bus avoids incompatibilities between subject areas.

Data Warehouse Architecture

Before exploring the use of aggregate tables to augment star schema performance, it is necessary to introduce the basic technical architecture of a data warehouse. Two major components of the data warehouse have already been discussed: the operational systems and the data warehouse. In addition to these databases, every data warehouse requires two additional components: software programs that move data from the operational systems to the data warehouse, and software that is used to develop queries and reports. These major components are illustrated in Figure 1.7.

The architecture of every data warehouse includes each of these fundamental components. Each component may comprise one or more products or physical servers. Whether custom-built or implemented using commercial off-the-shelf products, each of these components is a necessary piece of infrastructure.

- **Operational systems:** An operational system is an application that supports the execution of a business process, recording business activity and serving as the system of record. Operational systems may be packaged or custom-built applications. Their databases may reside on a variety of platforms, including relational database systems, mainframe-based systems, or proprietary data stores. For some data, such as budgeting information, the system of record may be as simple as a user spreadsheet.
- **Dimensional data warehouse:** The dimensional data warehouse is a database that supports the measurement of enterprise business processes. It stores a copy of operational data that has been organized for analytic purposes, according to the principles of dimensional modeling. Information is organized around a set of conformed dimensions, supporting enterprise-wide cross-process analysis. A subject area within the data warehouse is referred to as a *data mart*. The dimensional data warehouse is usually implemented on a relational database management system (RDBMS.)



Operational Systems

Purpose

- Business process execution
- Authoritative system of record

Profile

- Packaged or custom-built applications
- DBMS Systems:
  - Relational
  - Hierarchical
  - Network
  - Proprietary
- Spreadsheets or desktop databases

Architecture

- Server or mainframe based

ETL Process

Purpose

- Extract source data
- Transform for star schema
- Load into warehouse
- Process automation

Profile

- Packaged tools and/or custom coded routines
- Additional utilities as needed

Architecture

- Server based or host resident
- May be metadata driven, supported by an RDBMS

Data Warehouse

Purpose

- Business process measurement

Profile

- Relational Database Management System
- Star Schema Design

Architecture

- Server based
- Centralized or distributed

Front End Software

Purpose

- Query data warehouse
- Present information to users

Profile

- Packaged software, custom front ends, or combination
- Products may include:
  - Business Intelligence
  - Enterprise Reporting
  - Ad Hoc Query Tools
  - Data Mining Tools

Architecture

- Server based, desktop based, or combination
- Additional services may include authorization and authentication, automation and distribution, portal-based access

Warehouse Users

Profile

- Consumers of warehouse data
- Internal and external
- Operational and strategic focus

Figure 1.7 Data warehouse architecture.

- **Extract Transform Load (ETL) software:** ETL software is used to move data into data warehouse tables. This process involves fetching data from source systems (*extract*), reorganizing it as required by the star schema design (*transform*), and inserting it into warehouse tables (*load*). ETL may be accomplished using specialized, packaged software, or by writing custom code. The ETL process may rely on a number of additional utilities and databases for staging data, cleansing it, automating the process, and so forth. A detailed overview of the ETL process is provided in Chapter 5.
- **Front-end software:** Any tool that consumes information from the data warehouse, typically by issuing a SQL query to the data warehouse and presenting results in a number of different formats. Most architectures incorporate more than one front-end product. Common front-end tools include business intelligence (BI) software, enterprise reporting software, ad hoc query tools, data mining tools, and basic SQL execution tools. These services may be provided by commercial off-the-shelf software packages or custom developed. Front-end software often provides additional services, such as user- and group-based security administration, automation of report execution and distribution, and portal-based access to available information products.

Having developed a basic understanding of the dimensional model and the data warehouse architecture, you are now ready to begin studying aggregate tables.

Invisible Aggregates

Aggregate tables improve data warehouse performance by reducing the number of rows the RDBMS must access when responding to a query. At the simplest level, this is accomplished by partially summarizing the data in a base fact table and storing the result in a new fact table. Some new terminology will be necessary to differentiate aggregate tables from those in the original schema.

If the design of an aggregate schema is carefully managed, a query can be rewritten to leverage it through simple substitution of aggregate table names for base table names. Rather than expecting users or application developers to perform this substitution, an aggregate navigator is deployed. This component of the data warehouse architecture intercepts all queries and rewrites them to leverage aggregates, allowing users and applications to issue SQL written for the original schema.

For all of this to come off smoothly, it is important that the aggregates be designed and built according to some basic principles. These principles will shape the best practices detailed throughout this book.

Improving Performance

As you have seen, the best practices of dimensional design dictate that fact table grain is set at the lowest possible level of detail. This ensures that it will be possible to present the facts in any dimensional context desired. But most queries do not call for presentation of these individual atomic measurements. Instead, some group of these measurements will be aggregated. Although the atomic measurements do not appear in the final query results, the RDBMS must access them in order to compute their aggregates.

Consider again the query from Figure 1.4, which requests order_dollars by category and product for the month of January 2006. Each row of the final result set summarizes a large number of fact table rows. That's because the grain of the order_facts table, shown in Figure 1.2, is an individual order line. With over 10,000 order lines generated per day, the sample query requires the RDBMS to access over 300,000 rows of fact table data.

Most of the time the RDBMS spends executing the query will be spent reading these 300,000 rows of data. More specifically, the time will be spent waiting for the storage hardware to provide the data, as described in Chapter 2. Of course, the RDBMS has other tasks to perform. For example, it must first identify which rows of data it needs. And after the data has been returned, it needs to perform the necessary joins and aggregation. But in comparison to the time spent reading data, these tasks will be completed relatively quickly.

Aggregate tables seek to improve query performance by reducing the amount of data that must be accessed. By pre-aggregating the data in the fact table, they reduce the amount of work the RDBMS must perform to respond to a query. Put simply, query performance is increased when the number of rows that must be accessed is decreased.

Suppose a summarization of the order facts across all customers and orders is precomputed and stored in a new table called order_facts_aggregate. Shown in Figure 1.8, this table contains the same facts as the original fact table, but at a coarser grain. Gone are the relationships to the customer and order_type tables. Also omitted are the degenerate dimensions order_id and order_line_id.

All the data necessary to answer the query is present in this summary schema—it contains order_dollars, category, product, month, and year. But in this summarized version of order_facts, there are approximately 1,000 rows per day, compared to 10,000 order lines per day in the original fact table. Using this table, the RDBMS would have to access one-tenth the number of rows. Reading data from disk is one of the most time-consuming tasks the RDBMS performs while executing a query, as described in Chapter 2. By reducing the amount of data read by a factor of ten, response time is improved dramatically.

TIP An aggregate table improves response time by reducing the number of rows that must be accessed in responding to a query.

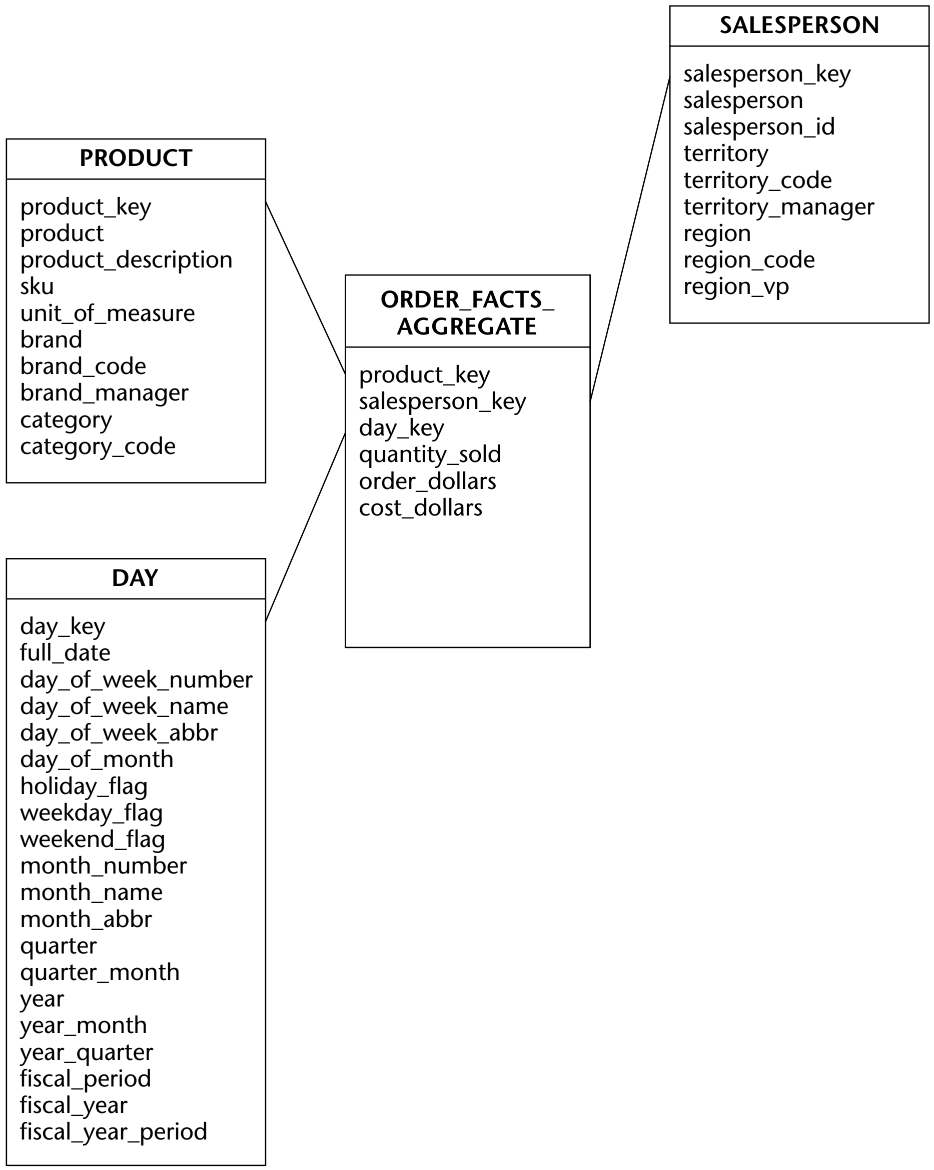


Figure 1.8 An aggregate schema.

Notice that the design of the aggregate schema in Figure 1.8 does not attempt to reuse base schema tables to store the aggregated data. Instead, a new table was created to store aggregated facts. The use of separate tables as a best practice for the storage of aggregated data is established in Chapter 3, after fully exploring the alternatives.

The Base Schema and the Aggregate Schema

The star schema in Figure 1.8 has already been referred to as an *aggregate schema*. It provides a partially aggregated summarization of data that is already stored in the data warehouse. The original schema containing the granular transactions is referred to as the *base schema*. Together, the base and aggregate stars form a *schema family*.

Similar terminology describes the individual tables in each schema. The fact table in the aggregate schema is referred to as an *aggregate fact table*. It summarizes information from the *base fact table*.

Notice that the dimension tables in the aggregate schema from Figure 1.8 are identical to those in the base schema. This aggregate schema has summarized order_facts by completely omitting the salesperson and order_type dimension tables.

For some aggregate schemas, partial summarization across a dimension may be useful. Instead of completely omitting a dimension table, it is partially summarized. For example, a monthly summary of order_facts by customer and salesperson would further reduce the number of rows in the aggregate fact table, and still be able to respond to the sample query.

In order to construct this aggregate schema, a Month dimension table is needed. The base schema includes only a Day dimension, so a new Month table must be built. This *aggregate dimension table* will be based on the Day dimension table in the base schema. As you will see in Chapter 2, its design is subject to the same rules of dimensional conformance discussed earlier in this chapter. In fact, a conformed Month table was already built for the sales goals schema, as shown in Figures 1.5 and 1.6. This same Month table can be used in an aggregate of the orders schema, as shown in Figure 1.9.

Although the aggregate schema from Figure 1.9 will go still further in improving the performance of our sample query, its overall usefulness is more limited than the daily aggregate from Figure 1.8. For example, it cannot respond to a query that requests order_dollars by day and product. Because this query requires specific days, it can be answered only by the aggregate schema in Figure 1.8, or by the base schema.

This characteristic can be generalized as follows: The more highly summarized an aggregate table is, the fewer queries it will be able to accelerate. This means that choosing aggregates involves making careful tradeoffs between the performance gain offered and the number of queries that will benefit. Chapter 2 explores how these factors are balanced when choosing aggregate tables.

With a large number of users issuing a diverse set of queries, it is to be expected that no single aggregate schema will improve the performance of every query. Notice, however, that improving the response time of a few resource-intensive queries can improve the overall throughput of the DBMS dramatically. Still, it is reasonable to expect that a *set* of aggregate tables will be deployed in support of a given star, rather than just one.

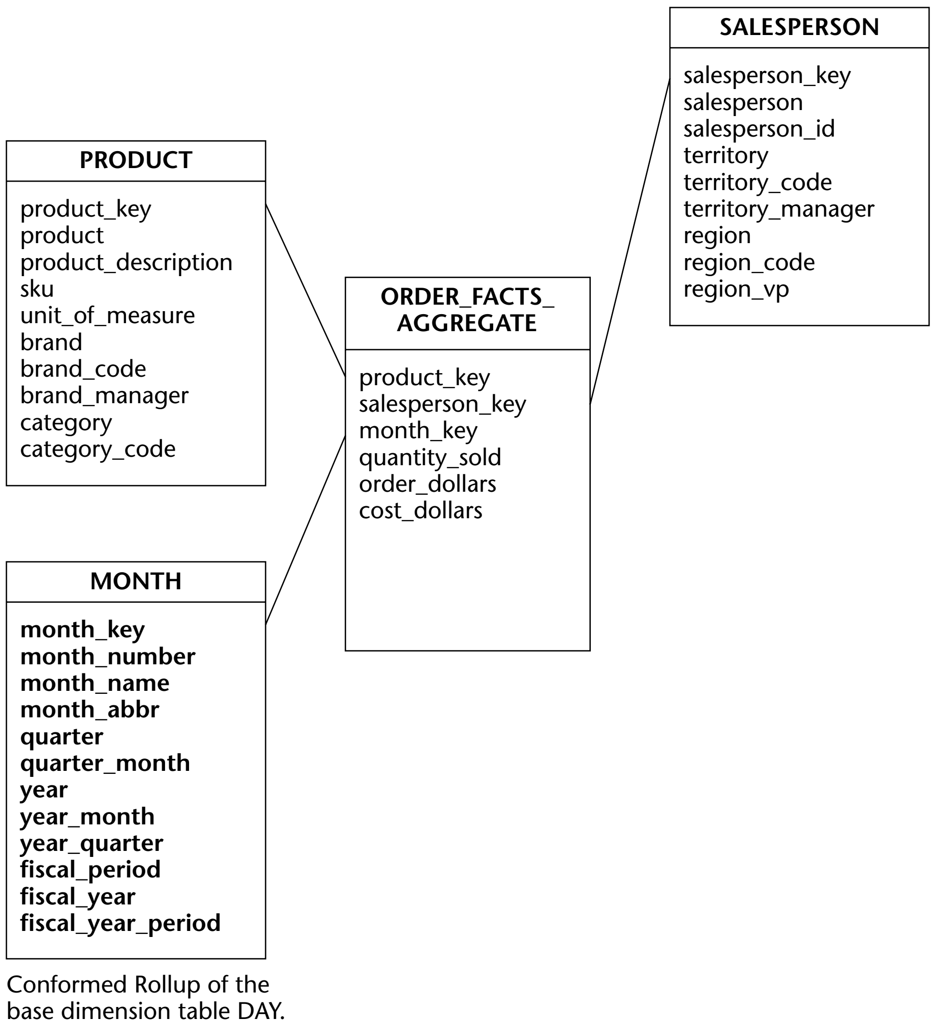


Figure 1.9 Month as an aggregate dimension table.

The Aggregate Navigator

To receive the performance benefit offered by an aggregate schema, a query must be written to use the aggregate. Rewriting the sample query was a simple matter of substituting the name of the aggregate fact table for that of the base fact table. This may be easy for technical personnel, but can prove confusing for an end user. The complexity grows as additional aggregate tables are added. Technical and business users alike may err in assessing which table will provide the best performance. And, if more aggregate tables are added or old ones removed, existing queries must be rewritten.

The key to avoiding these pitfalls lies in implementation of an *aggregate navigator*. A component of the data warehouse infrastructure, the aggregate navigator assumes the task of rewriting user queries to utilize aggregate tables. Users and developers need only be concerned with the base schema. At runtime, the aggregate navigator redirects the query to the most efficient aggregate schema. This process is depicted in Figure 1.10.

A number of commercial products offer aggregate navigation capabilities. Ideally, the aggregate navigator provides this service to all queries, regardless of the front-end application being used, and all back-end databases, regardless of physical location or technology. It maintains all information needed to rewrite queries, and keeps track of the status of each aggregate. This allows its services to be adjusted as aggregates are added and removed from the database, and permits aggregates to be taken off-line when being rebuilt. Detailed requirements for the aggregate navigator are presented in Chapter 4, which also describes how the aggregation strategy is altered if there is no aggregate navigator.

Principles of Aggregation

This chapter has proposed using a set of invisible aggregate tables to improve data warehouse performance. A family of aggregate star schemas will partially summarize information in a base star schema. These dimensional aggregates will be invisible to end-user applications, which will continue to issue SQL targeted at the base schema. An aggregate navigator will intercept these queries and rewrite them to leverage the aggregate tables.

For all this to come off smoothly, two key principles must be followed in the design and construction of the aggregate schemas. Subsequent chapters of this book enumerate best practices surrounding the selection, design, usage, and implementation of aggregates.

Providing the Same Results

The data warehouse must always provide accurate and consistent results. If it does not, business users will lose faith in its capability to provide accurate measurement of business processes. Use of the data warehouse will drop. Worse, those who do use it may base business decisions on inaccurate data.

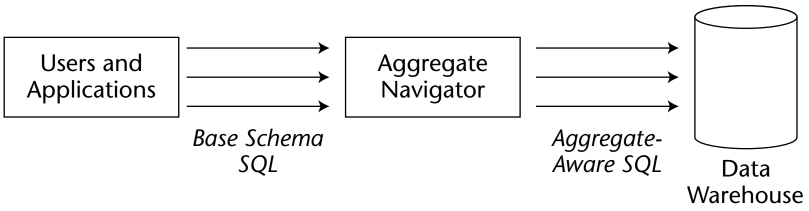


Figure 1.10 The aggregate navigator.

The importance of accuracy leads to the first guiding principle for aggregate tables. Assuming the base schema has been designed to publish operational data accurately and consistently, any aggregates must do the same. More specifically, the following can be stated:

An aggregate schema must always provide exactly the same results as the base schema.

If a rewritten query returns different results, it is returning wrong results. The data warehouse will bear the consequences. Every aggregate table must be an accurate and comprehensive summarization of its corresponding base table. This may seem obvious, but it is important to state it explicitly.

The effects of this principle are encountered throughout this book. Design options that violate this principle will be ruled out in Chapters 3 and 8. If the base schema and aggregate schema are not to be loaded simultaneously, this principle will require aggregates to be taken off-line during the load process, as discussed in Chapter 5. Design options for summaries that handle the absence of data differently than the base schema can produce results that are accurate, but different. In Chapter 9 these derived schemas are not awarded the status of invisibility and serve as base schema tables instead.

The Same Facts and Dimension Attributes as the Base Schema

The successful deployment of aggregate tables depends largely on their invisibility, particularly when more than one set of aggregates is available. The aggregate navigator will be relied upon to quickly and accurately rewrite queries through the substitution of table names and join columns in SQL queries. This suggests a second guiding principle:

The attributes of each aggregate table must be a subset of those from a base schema table. The only exception to this rule is the surrogate key for an aggregate dimension table.

The introduction of a new attribute to an aggregate table will either destroy its invisibility by requiring the aggregate be addressed by application SQL directly, or complicate the query rewrite process by requiring complex transformations of SQL syntax. Examples of potential new attributes, and their impact on schema usage, are explored in Chapter 3.

The capabilities of specific technologies may permit this principle to be relaxed slightly. For example, database features such as materialized views or materialized query tables can safely perform certain calculations on base schema attributes, or combine attributes from multiple tables into a single summary table. These possibilities are explored in Chapter 4.

Other Types of Summarization

The aggregate tables discussed thus far provide the same results as the base schema, and are leveraged by substituting the names of aggregate tables in SQL queries. Throughout this book, the term *aggregate* will be reserved for tables that exhibit these characteristics, even if an aggregate navigator is not deployed.

However, there are other ways to summarize the information in a star schema. Although not meeting the requirements for invisibility, these summary tables may provide value to the data warehouse in other ways.

Pre-Joined Aggregates

Like the dimensional aggregates you have seen so far, a pre-joined aggregate summarizes a fact across a set of dimension values. But unlike the aggregate star schemas from Figures 1.8 and 1.9, the pre-joined aggregate places the results in a single table. By doing so, the pre-joined aggregate eliminates the need for the RDBMS to perform a join operation at query time.

An example of a pre-joined aggregate appears in Figure 1.11. This table collects some dimensional attributes from Product, Day, and Salesperson, placing them in a single table with the facts from order_facts.

Like the aggregate schemas from the previous section, this pre-joined aggregate will improve the performance of our sample query by reducing the number of rows that must be consulted. While it also eliminates join processing requirements, this effect is less dramatic. All the major relational databases in use today have native support for star-join operations, optimizing the process by which fact table data is accessed and combined with dimensional data.

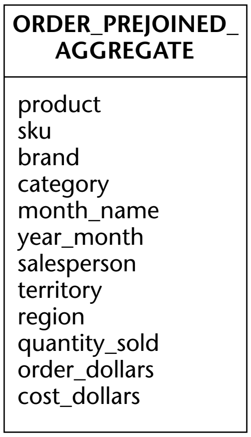


Figure 1.11 A pre-joined aggregate.

The pre-joined aggregate also has one drawback: It requires dramatically more storage space. Earlier, you saw that fact tables, although deep, consisted of very compact rows. Dimension tables contained wider rows, but were relatively small. The pre-joined aggregate is likely to be *wide and deep*. It is therefore likely to consume a very large amount of space, unless it is highly summarized. The more highly summarized an aggregate, the fewer queries it will accelerate.

TIP A pre-joined aggregate table combines aggregated facts and dimension attributes in a single table. While it can improve query performance, the pre-joined aggregate tends to consume excessive amounts of storage space.

Nonetheless, pre-joined aggregates are often implemented in support of specific queries or reports. And if the second principle of aggregation is relaxed slightly, the aggregate navigator may be leveraged to rewrite base-level SQL to access pre-joined aggregates as well. The aggregate navigator will have to replace all table names with the single pre-joined aggregate, and eliminate joins from the `WHERE` clause.

The processes of choosing and designing pre-joined aggregates are similar to those for standard dimensional aggregates, as you learn in Chapters 2 and 3. In Chapter 4, you see that database features such as materialized views work nicely to maintain pre-joined aggregates and rewrite queries. And in Chapter 5, you see that the pre-joined aggregate is also easy to build manually because it does not require any surrogate keys.

Derived Tables

Another group of summarization techniques seeks to improve performance by altering the structure of the tables summarized or changing the scope of their content. These tables are not meant to be invisible. Instead, they are provided as base schema objects. Users or report developers must explicitly choose to use these tables; the aggregate navigator does not come into play. There are three major types of derived tables: the merged fact table, the pivoted fact table, and the sliced fact table.

The *merged fact table* combines facts from more than one fact table at a common grain. This technique is often used to construct powerful data marts that draw data from multiple business areas. The merged fact table eliminates the need to perform drill-across operations, but introduces subtle differences in the way facts are recorded. Some facts will be stored with a value of zero, where the base fact table recorded nothing.

The *pivoted fact table* transforms a set of metrics in a single row into multiple rows with a single metric, or vice versa. Pivoted fact tables greatly simplify reporting by configuring the storage of facts to match the desired presentation format. Performing a similar transformation within SQL or a reporting tool can be slow and cumbersome. Like the merged fact table, the pivoted fact table may be forced to store facts with a value of zero, even when the base fact table contains nothing.

The *sliced fact table* does nothing to transform the structure of the original schema, but does change its content. A sliced fact table contains a subset of the records of the base fact table, usually in coordination with a specific dimension attribute. This technique can be used to relocate subsets of data closer to the individuals that need it. Conversely, it can also be used to consolidate regional data stores that are identical in structure. Because the slice can be derived from the whole, or vice versa, which table is derived is a function of the business environment.

In all three cases, the derived fact tables are not expected to serve as invisible stand-ins for the base schema. The merged and pivoted fact tables significantly alter schema structure, while the sliced fact table alters its content. These tables should be considered part of the base schema, and accessed by users or report developers explicitly. Derived tables are explored in further detail in Chapter 9.

Tables with New Facts

In a paradoxical twist, the final type of summary table actually contains attributes not present in the original schema. Such summaries occur when a fact in the original schema does not exhibit the characteristic of additivity, as described previously. While these facts can be summarized, the semantics and usage of the result differ from those of the original table. Like the derived schemas, these summarizations are not expected to remain invisible to end users. Instead, they are called upon directly by application SQL.

In all the examples in this chapter, facts have been aggregated by summing their values. For some facts, this is not appropriate. Semi-additive facts may not be added together across a particular dimension; non-additive facts are never added together. In these situations, you may choose to aggregate by means other than summation.

An account balance, for example, is semi-additive. On a given day, you can add together the balances of your various bank accounts to compute a total balance. Unfortunately, you cannot add together a series of daily balances to compute a monthly total. But it may be useful to compute a monthly average of these values. The average daily balance can be used to determine the interest allied at month's end, for example. But an average daily balance means something different than the balance dollars fact of the original schema.

When you aggregate by means other than summation, you create new facts. These facts have different meanings than the original fact, and their usage is governed by different rules. Tables with new facts, therefore, are not expected to remain invisible. Like the derived tables, they will be made available as part of the base schema. Users and applications will access them explicitly. Tables with new facts will be encountered in Chapter 8, which explores the impact of advanced dimensional design techniques on aggregates.

Summary

This chapter has laid the foundation for the chapters to come, reviewing the basics of star schema design, introducing the aggregate table and aggregate navigator, defining some standard vocabulary, and establishing some guiding principles for invisible aggregates.

- While operational systems focus on process *execution*, data warehouse systems focus on process *evaluation*. These contrasting purposes lead to distinct operational profiles, which in turn suggest different principles to guide schema design.
- The principles of *dimensional modeling* govern the development of warehouse systems. Process evaluation is enabled by identifying the *facts* that measure a business process and the *dimensions* that give them context. These attributes are grouped into tables that form a star schema design.
- *Dimension tables* contain sets of dimensional attributes. They drive access to the facts, constrain queries, and serve as row headers on reports. The use of a *surrogate key* permits the dimension table to track history, regardless of how changes are handled in operational systems.
- Facts are placed in *fact tables*, along with foreign key references to the appropriate dimension tables. The *grain* of a fact table identifies the level of detail represented by each row. It is set at the lowest level possible, as determined by available data.
- Although the specific questions asked by end users are unpredictable and change over time, queries follow a *standard pattern*. Questions that cross subject areas can be answered through a process called *drilling across*, provided the warehouse had been designed around a set of conformed dimensions referred to as the warehouse bus.
- Aggregate tables improve the response time of a star schema query by reducing the number of rows the database must read. Ideally, the aggregate tables are *invisible* to end uses and applications, which issue queries to the base schema. An *aggregate navigator* is deployed, rewriting these queries to leverage aggregates as appropriate.

- To facilitate this invisibility, two basic principles guide aggregate schema design. Aggregates must always provide the same results as the base schema, and the attributes of each aggregate table must be a subset of those of a base table.
- Not all forms of summarization meet the requirements of invisible aggregates. Other forms of summarization include *pre-joined aggregates*, *derived tables*, and tables with *new facts*. Although not serviced by the aggregate navigator, these summaries can serve as useful additions to the base schema of the data warehouse, accessed explicitly through application SQL.

With these fundamentals out of the way, you are ready to turn to the first and most perplexing task: choosing which aggregates to design and build.