Chapter 2 discusses the following concepts:

- **Four-step process for designing dimensional models**
- **Transaction-level fact tables**
- **Additive and non-additive facts**
- **Sample dimension table attributes**
- **Causal dimensions, such as promotion**
- **Degenerate dimensions, such as the transaction ticket number**
- **Extending an existing dimension model**
- **Snowflaking dimension attributes**
- **Avoiding the "too many dimensions" trap**
- **Surrogate keys**
- **Market basket analysis**

Four-Step Dimensional Design Process

Throughout this book we will approach the design of a dimensional database by consistently considering four steps in a particular order. The meaning of these four steps will become more obvious as we proceed with the various designs, but we'll provide initial definitions at this time.

1. Select the business process to model. A process is a natural business activity performed in your organization that typically is supported by a source data-collection system. Listening to your users is the most efficient means for selecting the business process. The performance measurements that they clamor to analyze in the data warehouse result from business measurement processes. Example business processes include raw materials purchasing, orders, shipments, invoicing, inventory, and general ledger.

   It is important to remember that we're not referring to an organizational business department or function when we talk about business processes. For example, we'd build a single dimensional model to handle orders data rather than building separate models for the sales and marketing departments, which both want to access orders data. By focusing on business processes, rather than on business departments, we can deliver consistent information more economically throughout the organization. If we establish departmentally bound dimensional models, we'll inevitably duplicate data with different labels and terminology. Multiple data flows into separate dimensional models will make us vulnerable to data inconsistencies. The best way to ensure consistency is to publish the data once. A single publishing run also reduces the extract-transformation-load (ETL) development effort, as well as the ongoing data management and disk storage burden.

2. Declare the grain of the business process. Declaring the grain means specifying *exactly* what an individual fact table row represents. The grain conveys the level of detail associated with the fact table measurements. It provides the answer to the question, "How do you describe a single row in the fact table?"

   Example grain declarations include:

   - An individual line item on a customer's retail sales ticket as measured by a scanner device
   - A line item on a bill received from a doctor
   - An individual boarding pass to get on a flight
   - A daily snapshot of the inventory levels for each product in a warehouse
   - A monthly snapshot for each bank account

   Data warehouse teams often try to bypass this seemingly unnecessary step of the process. Please don't! It is extremely important that everyone on the design team is in agreement regarding the fact table granularity. It is virtually impossible to reach closure in step 3 without declaring the grain. We also should warn you that an inappropriate grain declaration will haunt a data warehouse implementation. Declaring the grain is a critical step that can't be taken lightly. Having said this, you may discover in steps 3 or 4 that the grain statement is wrong. This is okay, but then you must return to step 2, redeclare the grain correctly, and revisit steps 3 and 4 again.

3. Choose the dimensions that apply to each fact table row. Dimensions fall out of the question, "How do businesspeople describe the data that results from the business process?" We want to decorate our fact tables with a robust set of dimensions representing all possible descriptions that take on single values in the context of each measurement. If we are clear about the grain, then the dimensions typically can be identified quite easily. With the choice of each dimension, we will list all the discrete, textlike attributes that will flesh out each dimension table. Examples of common dimensions include date, product, customer, transaction type, and status.

4. Identify the numeric facts that will populate each fact table row. Facts are determined by answering the question, "What are we measuring?" Business users are keenly interested in analyzing these business process performance measures. All candidate facts in a design must be true to the grain defined in step 2. Facts that clearly belong to a different grain must be in a separate fact table. Typical facts are numeric additive figures such as quantity ordered or dollar cost amount.

Throughout this book we will keep these four steps in mind as we develop each of the case studies. We'll apply a user's understanding of the business to decide what dimensions and facts are needed in the dimensional model. Clearly, we need to consider both our business users' requirements and the realities of our source data in tandem to make decisions regarding the four steps, as illustrated in Figure 2.1. We strongly encourage you to resist the temptation to model the data by looking at source data files alone. We realize that it may be much less intimidating to dive into the file layouts and copybooks rather than interview a businessperson; however, they are no substitute for user input. Unfortunately, many organizations have attempted this path-of-least-resistance data-driven approach, but without much success.

Retail Case Study

Let's start with a brief description of the retail business that we'll use in this case study to make dimension and fact tables more understandable. We begin with this industry because it is one to which we can all relate. Imagine that we work in the headquarters of a large grocery chain. Our business has 100 grocery stores spread over a five-state area. Each of the stores has a full complement of departments, including grocery, frozen foods, dairy, meat, produce, bakery, floral, and health/beauty aids. Each store has roughly 60,000 individual products on its shelves. The individual products are called *stock keeping units* (SKUs). About 55,000 of the SKUs come from outside manufacturers and have bar codes imprinted on the product package. These bar codes are called *universal product codes* (UPCs). UPCs are at the same grain as individual SKUs. Each different package variation of a product has a separate UPC and hence is a separate SKU.

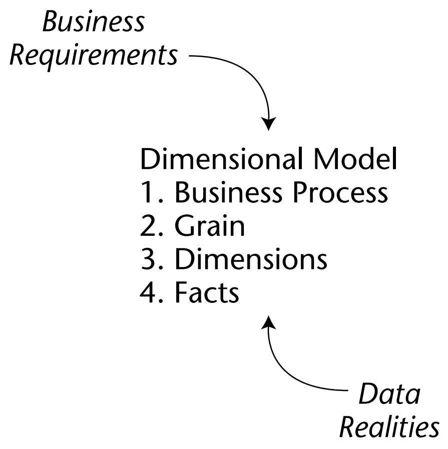

Figure 2.1 Key input to the four-step dimensional design process.

The remaining 5,000 SKUs come from departments such as meat, produce, bakery, or floral. While these products don't have nationally recognized UPCs, the grocery chain assigns SKU numbers to them. Since our grocery chain is highly automated, we stick scanner labels on many of the items in these other departments. Although the bar codes are not UPCs, they are certainly SKU numbers.

Data is collected at several interesting places in a grocery store. Some of the most useful data is collected at the cash registers as customers purchase products. Our modern grocery store scans the bar codes directly into the point-of-sale (POS) system. The POS system is at the front door of the grocery store where consumer takeaway is measured. The back door, where vendors make deliveries, is another interesting data-collection point.

At the grocery store, management is concerned with the logistics of ordering, stocking, and selling products while maximizing profit. The profit ultimately comes from charging as much as possible for each product, lowering costs for product acquisition and overhead, and at the same time attracting as many customers as possible in a highly competitive pricing environment. Some of the most significant management decisions have to do with pricing and promotions. Both store management and headquarters marketing spend a great deal of time tinkering with pricing and promotions. Promotions in a grocery store include temporary price reductions, ads in newspapers and newspaper inserts, displays in the grocery store (including end-aisle displays), and coupons. The most direct and effective way to create a surge in the volume of product sold is to lower the price dramatically. A 50-cent reduction in the price of paper towels, especially when coupled with an ad and display, can cause the sale of the paper towels to jump by a factor of 10. Unfortunately, such a big price reduction usually is not sustainable because the towels probably are being sold at a loss. As a result of these issues, the visibility of all forms of promotion is an important part of analyzing the operations of a grocery store.

Now that we have described our business case study, we'll begin to design the dimensional model.

Step 1. Select the Business Process

The first step in the design is to decide what business process(es) to model by combining an understanding of the business requirements with an understanding of the available data.

The first dimensional model built should be the one with the most impact—it should answer the most pressing business questions and be readily accessible for data extraction.

In our retail case study, management wants to better understand customer purchases as captured by the POS system. Thus the business process we're going to model is POS retail sales. This data will allow us to analyze what products are selling in which stores on what days under what promotional conditions.

Step 2. Declare the Grain

Once the business process has been identified, the data warehouse team faces a serious decision about the granularity. What level of data detail should be made available in the dimensional model? This brings us to an important design tip.

Preferably you should develop dimensional models for the most atomic information captured by a business process. Atomic data is the most detailed information collected; such data cannot be subdivided further.

Tackling data at its lowest, most atomic grain makes sense on multiple fronts. Atomic data is highly dimensional. The more detailed and atomic the fact measurement, the more things we know for sure. All those things we know for sure translate into dimensions. In this regard, atomic data is a perfect match for the dimensional approach.

Atomic data provides maximum analytic flexibility because it can be constrained and rolled up in every way possible. Detailed data in a dimensional model is poised and ready for the ad hoc attack by business users.

Of course, you can always declare higher-level grains for a business process that represent an aggregation of the most atomic data. However, as soon as we select a higher-level grain, we're limiting ourselves to fewer and/or potentially less detailed dimensions. The less granular model is immediately vulnerable to unexpected user requests to drill down into the details. Users inevitably run into an analytic wall when not given access to the atomic data. As we'll see in Chapter 16, aggregated summary data plays an important role as a performance-tuning tool, but it is not a substitute for giving users access to the lowest-level details. Unfortunately, some industry pundits have been confused on this point. They claim that dimensional models are only appropriate for summarized data and then criticize the dimensional modeling approach for its supposed need to anticipate the business question. This misunderstanding goes away when detailed, atomic data is made available in a dimensional model.

In our case study, the most granular data is an individual line item on a POS transaction. To ensure maximum dimensionality and flexibility, we will proceed with this grain. It is worth noting that this granularity declaration represents a change from the first edition of this text. Previously, we focused on POS data, but rather than representing transaction line item detail in the dimensional model, we elected to provide sales data rolled up by product and promotion in a store on a day. At the time, these daily product totals represented the state of the art for syndicated retail sales databases. It was unreasonable to expect then-current hardware and software to deal effectively with the volumes of data associated with individual POS transaction line items.

Providing access to the POS transaction information gives us with a very detailed look at store sales. While users probably are not interested in analyzing single items associated with a specific POS transaction, we can't predict all the ways that they'll want to cull through that data. For example, they may want to understand the difference in sales on Monday versus Sunday. Or they may want to assess whether it's worthwhile to stock so many individual sizes of certain brands, such as cereal. Or they may want to understand how many shoppers took advantage of the 50-cents-off promotion on shampoo. Or they may want to determine the impact in terms of decreased sales when a competitive diet soda product was promoted heavily. While none of these queries calls for data from one specific transaction, they are broad questions that require detailed data sliced in very precise ways. None of them could have been answered if we elected only to provide access to summarized data.

A data warehouse almost always demands data expressed at the lowest possible grain of each dimension not because queries want to see individual low-level rows, but because queries need to cut through the details in very precise ways.

Step 3. Choose the Dimensions

Once the grain of the fact table has been chosen, the date, product, and store dimensions fall out immediately. We assume that the calendar date is the date value delivered to us by the POS system. Later, we will see what to do if we also get a time of day along with the date. Within the framework of the primary dimensions, we can ask whether other dimensions can be attributed to the data, such as the promotion under which the product is sold. We express this as another design principle:

A careful grain statement determines the primary dimensionality of the fact table. It is then often possible to add more dimensions to the basic grain of the fact table, where these additional dimensions naturally take on only one value under each combination of the primary dimensions. If the additional dimension violates the grain by causing additional fact rows to be generated, then the grain statement must be revised to accommodate this dimension.

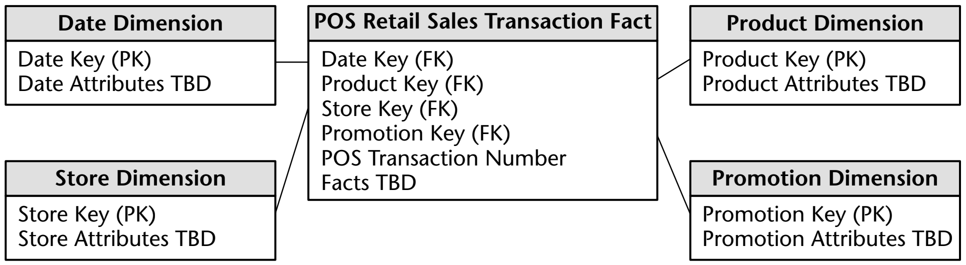

Figure 2.2 Preliminary retail sales schema.
"TBD" means "to be determined."

In our case study we've decided on the following descriptive dimensions: date, product, store, and promotion. In addition, we'll include the POS transaction ticket number as a special dimension. More will be said on this later in the chapter.

We begin to envision the preliminary schema as illustrated in Figure 2.2. Before we delve into populating the dimension tables with descriptive attributes, let's complete the final step of the process. We want to ensure that you're comfortable with the complete four-step process—we don't want you to lose sight of the forest for the trees at this stage of the game.

Step 4. Identify the Facts

The fourth and final step in the design is to make a careful determination of which facts will appear in the fact table. Again, the grain declaration helps anchor our thinking. Simply put, the facts must be true to the grain: the individual line item on the POS transaction in this case. When considering potential facts, you again may discover that adjustments need to be made to either our earlier grain assumptions or our choice of dimensions.

The facts collected by the POS system include the sales quantity (e.g., the number of cans of chicken noodle soup), per unit sales price, and the sales dollar amount. The sales dollar amount equals the sales quantity multiplied by the unit price. More sophisticated POS systems also provide a standard dollar cost for the product as delivered to the store by the vendor. Presuming that this cost fact is readily available and doesn't require a heroic activity-based costing initiative, we'll include it in the fact table. Our fact table begins to take shape in Figure 2.3.

Three of the facts, sales quantity, sales dollar amount, and cost dollar amount, are beautifully additive across all the dimensions. We can slice and dice the fact table with impunity, and every sum of these three facts is valid and correct.

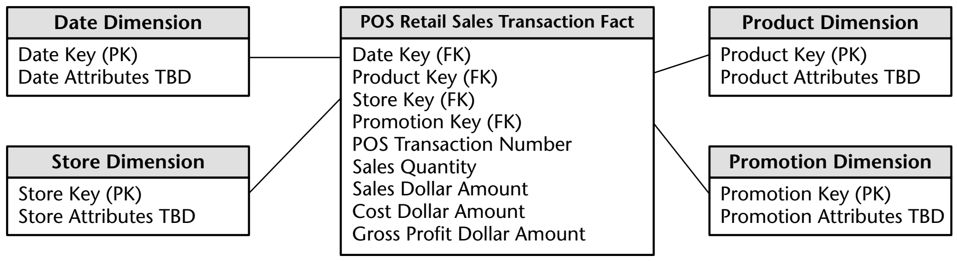

Figure 2.3 Measured facts in the retail sales schema.

We can compute the gross profit by subtracting the cost dollar amount from the sales dollar amount, or revenue. Although computed, this gross profit is also perfectly additive across all the dimensions—we can calculate the gross profit of any combination of products sold in any set of stores on any set of days. Dimensional modelers sometimes question whether a calculated fact should be stored physically in the database. We generally recommend that it be stored physically. In our case study, the gross profit calculation is straightforward, but storing it eliminates the possibility of user error. The cost of a user incorrectly representing gross profit overwhelms the minor incremental storage cost. Storing it also ensures that all users and their reporting applications refer to gross profit consistently. Since gross profit can be calculated from adjacent data within a fact table row, some would argue that we should perform the calculation in a view that is indistinguishable from the table. This is a reasonable approach if *all* users access the data via this view and *no* users with ad hoc query tools can sneak around the view to get at the physical table. Views are a reasonable way to minimize user error while saving on storage, but the DBA must allow no exceptions to data access through the view. Likewise, some organizations want to perform the calculation in the query tool. Again, this works if *all* users access the data using a common tool (which is seldom the case in our experience).

The gross margin can be calculated by dividing the gross profit by the dollar revenue. Gross margin is a nonadditive fact because it can't be summarized along any dimension. We can calculate the gross margin of any set of products, stores, or days by remembering to add the revenues and costs before dividing. This can be stated as a design principle:

Percentages and ratios, such as gross margin, are nonadditive. The numerator and denominator should be stored in the fact table. The ratio can be calculated in a data access tool for any slice of the fact table by remembering to calculate the *ratio of the sums*, not the sum of the ratios.

Unit price is also a nonadditive fact. Attempting to sum up unit price across any of the dimensions results in a meaningless, nonsensical number. In order to analyze the average selling price for a product in a series of stores or across a period of time, we must add up the sales dollars and sales quantities before dividing the total dollars by the total quantity sold. Every report writer or query tool in the data warehouse marketplace should automatically perform this function correctly, but unfortunately, some still don't handle it very gracefully.

At this early stage of the design, it is often helpful to estimate the number of rows in our largest table, the fact table. In our case study, it simply may be a matter of talking with a source system guru to understand how many POS transaction line items are generated on a periodic basis. Retail traffic fluctuates significantly from day to day, so we'll want to understand the transaction activity over a reasonable period of time. Alternatively, we could estimate the number of rows added to the fact table annually by dividing the chain's annual gross revenue by the average item selling price. Assuming that gross revenues are $4 billion per year and that the average price of an item on a customer ticket is $2.00, we calculate that there are approximately 2 billion transaction line items per year. This is a typical engineer's estimate that gets us surprisingly close to sizing a design directly from our armchairs. As designers, we always should be triangulating to determine whether our calculations are reasonable.

Dimension Table Attributes

Now that we've walked through the four-step process, let's return to the dimension tables and focus on filling them with robust attributes.

Date Dimension

We will start with the date dimension. The date dimension is the one dimension nearly guaranteed to be in every data mart because virtually every data mart is a time series. In fact, date is usually the first dimension in the underlying sort order of the database so that the successive loading of time intervals of data is placed into virgin territory on the disk.

For readers of the first edition of *The Data Warehouse Toolkit* (Wiley 1996), this dimension was referred to as the *time dimension* in that text. Rather than sticking with that more ambiguous nomenclature, we use the *date dimension* in this book to refer to daily-grained dimension tables. This helps distinguish the date and time-of-day dimensions, which we'll discuss later in this chapter.

Unlike most of our other dimensions, we can build the date dimension table in advance. We may put 5 or 10 years of rows representing days in the table so that we can cover the history we have stored, as well as several years in the future. Even 10 years' worth of days is only about 3,650 rows, which is a relatively small dimension table. For a daily date dimension table in a retail environment, we recommend the partial list of columns shown in Figure 2.4.

Each column in the date dimension table is defined by the particular day that the row represents. The day-of-week column contains the name of the day, such as Monday. This column would be used to create reports comparing the business on Mondays with Sunday business. The day number in calendar month column starts with 1 at the beginning of each month and runs to 28, 29, 30, or 31, depending on the month. This column is useful for comparing the same day each month. Similarly, we could have a month number in year (1, ... , 12). The day number in epoch is effectively a Julian day number (that is, a consecutive day number starting at the beginning of some epoch). We also could include

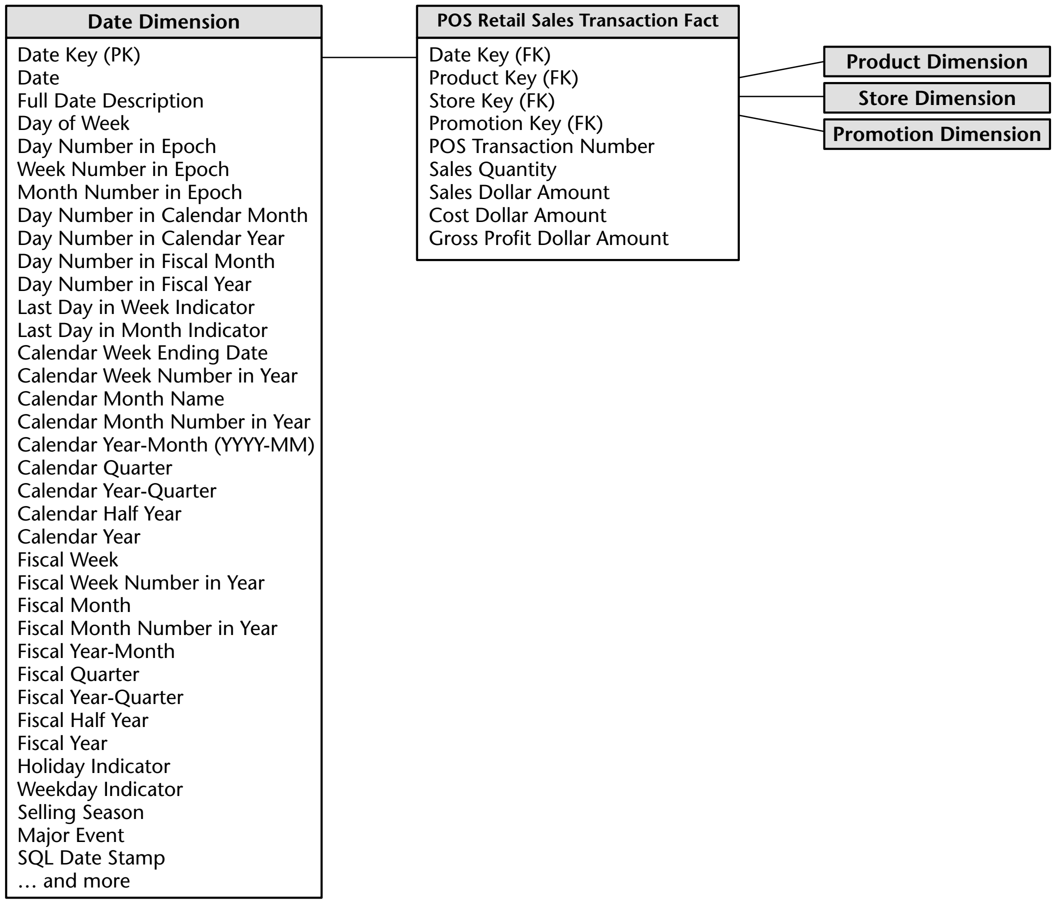

Figure 2.4 Date dimension in the retail sales schema.

absolute week and month number columns. All these integers support simple date arithmetic between days across year and month boundaries. For reporting, we would want a month name with values such as January. In addition, a year-month (YYYY-MM) column is useful as a report column header. We likely also will want a quarter number (Q1, ... , Q4), as well as a year quarter, such as 2001-Q4. We would have similar columns for the fiscal periods if they differ from calendar periods.

The holiday indicator takes on the values of Holiday or Nonholiday. Remember that the dimension table attributes serve as report labels. Simply populating the holiday indicator with a Y or an N would be far less useful. Imagine a report where we're comparing holiday sales for a given product versus non-holiday sales. Obviously, it would be helpful if the columns had meaningful values such as Holiday/Nonholiday versus a cryptic Y/N. Rather than decoding cryptic flags into understandable labels in a reporting application, we prefer that the decode be stored in the database so that a consistent value is available to all users regardless of their reporting environment.

A similar argument holds true for the weekday indicator, which would have a value of Weekday or Weekend. Saturdays and Sundays obviously would be assigned the Weekend value. Of course, multiple date table attributes can be jointly constrained, so we can easily compare weekday holidays with weekend holidays, for example.

The selling season column is set to the name of the retailing season, if any. Examples in the United States could include Christmas, Thanksgiving, Easter, Valentine's Day, Fourth of July, or None. The major event column is similar to the season column and can be used to mark special outside events such as Super Bowl Sunday or Labor Strike. Regular promotional events usually are not handled in the date table but rather are described more completely by means of the promotion dimension, especially since promotional events are not defined solely by date but usually are defined by a combination of date, product, and store.

Some designers pause at this point to ask why an explicit date dimension table is needed. They reason that if the date key in the fact table is a date-type field, then any SQL query can directly constrain on the fact table date key and use natural SQL date semantics to filter on month or year while avoiding a supposedly expensive join. This reasoning falls apart for several reasons. First of all, if our relational database can't handle an efficient join to the date dimension table, we're already in deep trouble. Most database optimizers are quite efficient at resolving dimensional queries; it is not necessary to avoid joins like the plague. Also, on the performance front, most databases don't index SQL date calculations, so queries constraining on an SQL-calculated field wouldn't take advantage of an index.

In terms of usability, the typical business user is not versed in SQL date semantics, so he or she would be unable to directly leverage inherent capabilities associated with a date data type. SQL date functions do not support filtering by attributes such as weekdays versus weekends, holidays, fiscal periods, seasons, or major events. Presuming that the business needs to slice data by these nonstandard date attributes, then an explicit date dimension table is essential. At the bottom line, calendar logic belongs in a dimension table, not in the application code. Finally, we're going to suggest that the date key is an integer rather than a date data type anyway. An SQL-based date key typically is 8 bytes, so you're wasting 4 bytes in the fact table for every date key in every row. More will be said on this later in this chapter.

Figure 2.5 illustrates several rows from a partial date dimension table.

Data warehouses always need an explicit date dimension table. There are many date attributes not supported by the SQL date function, including fiscal periods, seasons, holidays, and weekends. Rather than attempting to determine these nonstandard calendar calculations in a query, we should look them up in a date dimension table.

If we wanted to access the time of the transaction for day-part analysis (for example, activity during the evening after-work rush or third shift), we'd handle it through a separate time-of-day dimension joined to the fact table. Date and time are almost completely independent. If we combined the two dimensions, the date dimension would grow significantly; our neat date dimension with 3,650 rows to handle 10 years of data would expand to 5,256,000 rows if we tried to handle time by minute in the same table (or via an outrigger). Obviously, it is preferable to create a 3,650-row date dimension table and a separate 1,440-row time-of-day by minute dimension.

In Chapter 5 we'll discuss the handling of multiple dates in a single schema. We'll explore international date and time considerations in Chapters 11 and 14.

Date Key	Date	Full Date Description	Day of Week	Calendar Month	Calendar Year	Fiscal Year-Month	Holiday Indicator	Weekday Indicator
1	01/01/2002	January 1, 2002	Tuesday	January	2002	F2002-01	Holiday	Weekday
2	01/02/2002	January 2, 2002	Wednesday	January	2002	F2002-01	Non-Holiday	Weekday
3	01/03/2002	January 3, 2002	Thursday	January	2002	F2002-01	Non-Holiday	Weekday
4	01/04/2002	January 4, 2002	Friday	January	2002	F2002-01	Non-Holiday	Weekday
5	01/05/2002	January 5, 2002	Saturday	January	2002	F2002-01	Non-Holiday	Weekend
6	01/06/2002	January 6, 2002	Sunday	January	2002	F2002-01	Non-Holiday	Weekend
7	01/07/2002	January 7, 2002	Monday	January	2002	F2002-01	Non-Holiday	Weekday
8	01/08/2002	January 8, 2002	Tuesday	January	2002	F2002-01	Non-Holiday	Weekday

Figure 2.5 Date dimension table detail.

Product Dimension

The product dimension describes every SKU in the grocery store. While a typical store in our chain may stock 60,000 SKUs, when we account for different merchandising schemes across the chain and historical products that are no longer available, our product dimension would have at least 150,000 rows and perhaps as many as a million rows. The product dimension is almost always sourced from the operational product master file. Most retailers administer their product master files at headquarters and download a subset of the file to each store's POS system at frequent intervals. It is headquarters' responsibility to define the appropriate product master record (and unique SKU number) for each new UPC created by packaged goods manufacturers. Headquarters also defines the rules by which SKUs are assigned to such items as bakery goods, meat, and produce. We extract the product master file into our product dimension table each time the product master changes.

An important function of the product master is to hold the many descriptive attributes of each SKU. The merchandise hierarchy is an important group of attributes. Typically, individual SKUs roll up to brands. Brands roll up to categories, and categories roll up to departments. Each of these is a many-to-one relationship. This merchandise hierarchy and additional attributes are detailed for a subset of products in Figure 2.6.

For each SKU, all levels of the merchandise hierarchy are well defined. Some attributes, such as the SKU description, are unique. In this case, there are at least 150,000 different values in the SKU description column. At the other extreme, there are only perhaps 50 distinct values of the department attribute. Thus, on average, there are 3,000 repetitions of each unique value in the department attribute. This is all right! We do not need to separate these repeated values into a second normalized table to save space. Remember that dimension table space requirements pale in comparison with fact table space considerations.

Product Key	Product Description	Brand Description	Category Description	Department Description	Fat Content
1	Baked Well Light Sourdough Fresh Bread	Baked Well	Bread	Bakery	Reduced Fat
2	Fluffy Sliced Whole Wheat	Fluffy	Bread	Bakery	Regular Fat
3	Fluffy Light Sliced Whole Wheat	Fluffy	Bread	Bakery	Reduced Fat
4	Fat Free Mini Cinnamon Rolls	Light	Sweeten Bread	Bakery	Non-Fat
5	Diet Lovers Vanilla 2 Gallon	Coldpack	Frozen Desserts	Frozen Foods	Non-Fat
6	Light and Creamy Butter Pecan 1 Pint	Freshlike	Frozen Desserts	Frozen Foods	Reduced Fat
7	Chocolate Lovers 1/2 Gallon	Frigid	Frozen Desserts	Frozen Foods	Regular Fat
8	Strawberry Ice Creamy 1 Pint	Icy	Frozen Desserts	Frozen Foods	Regular Fat
9	Icy Ice Cream Sandwiches	Icy	Frozen Desserts	Frozen Foods	Regular Fat

Figure 2.6 Product dimension table detail.

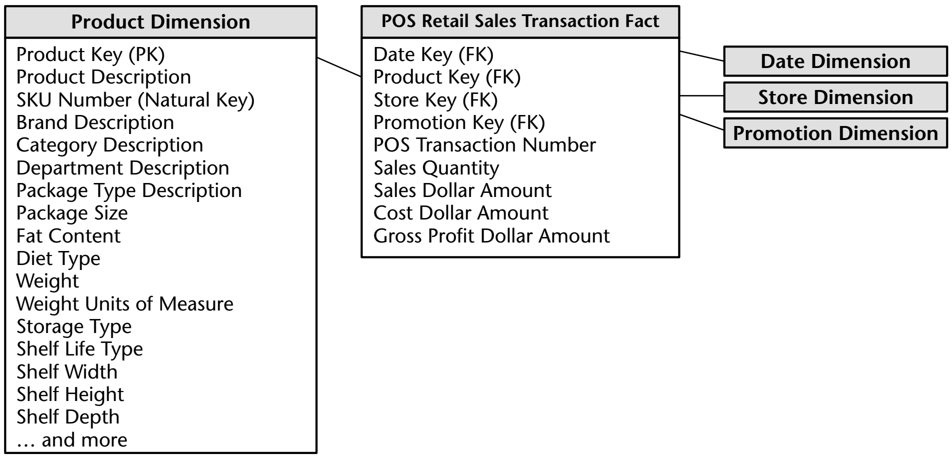

Figure 2.7 Product dimension in the retail sales schema.

Many of the attributes in the product dimension table are not part of the merchandise hierarchy. The package-type attribute, for example, might have values such as Bottle, Bag, Box, or Other. Any SKU in any department could have one of these values. It makes perfect sense to combine a constraint on this attribute with a constraint on a merchandise hierarchy attribute. For example, we could look at all the SKUs in the Cereal category packaged in Bags. To put this another way, we can browse among dimension attributes whether or not they belong to the merchandise hierarchy, and we can drill up and drill down using attributes whether or not they belong to the merchandise hierarchy. We can even have more than one explicit hierarchy in our product dimension table.

A recommended partial product dimension for a retail grocery data mart would look similar to Figure 2.7.

A reasonable product dimension table would have 50 or more descriptive attributes. Each attribute is a rich source for constraining and constructing row headers. Viewed in this manner, we see that drilling down is nothing more than asking for a row header that provides more information. Let's say we have a simple report where we've summarized the sales dollar amount and quantity by department.

Department Description	Sales Dollar Amount	Sales Quantity
Bakery	$12,331	5,088
Frozen Foods	$31,776	15,565

If we want to drill down, we can drag virtually any other attribute, such as brand, from the product dimension into the report next to department, and we automatically drill down to this next level of detail. A typical drill down within the merchandise hierarchy would look like this:

Department Description	Brand Description	Sales Dollar Amount	Sales Quantity
Bakery	Baked Well	$3,009	1,138
Bakery	Fluffy	$3,024	1,476
Bakery	Light	$6,298	2,474
Frozen Foods	Coldpack	$5,321	2,640
Frozen Foods	Freshlike	$10,476	5,234
Frozen Foods	Frigid	$7,328	3,092
Frozen Foods	Icy	$2,184	1,437
Frozen Foods	QuickFreeze	$6,467	3,162

Or we could drill down by the fat-content attribute, even though it isn't in the merchandise hierarchy roll-up.

Department Description	Fat Content	Sales Dollar Amount	Sales Quantity
Bakery	Non-Fat	$6,298	2,474
Bakery	Reduced Fat	$5,027	2,086
Bakery	Regular Fat	$1,006	528
Frozen Foods	Non-Fat	$5,321	2,640
Frozen Foods	Reduced Fat	$10,476	5,234
Frozen Foods	Regular Fat	$15,979	7,691

We have belabored the examples of drilling down in order to make a point, which we will express as a design principle.

Drilling down in a data mart is nothing more than adding row headers from the dimension tables. Drilling up is removing row headers. We can drill down or up on attributes from more than one explicit hierarchy and with attributes that are part of no hierarchy.

The product dimension is one of the two or three primary dimensions in nearly every data mart. Great care should be taken to fill this dimension with as many descriptive attributes as possible. A robust and complete set of dimension attributes translates into user capabilities for robust and complete analysis. We'll further explore the product dimension in Chapter 4, where we'll also discuss the handling of product attribute changes.

Store Dimension

The store dimension describes every store in our grocery chain. Unlike the product master file that is almost guaranteed to be available in every large grocery business, there may not be a comprehensive store master file. The product master needs to be downloaded to each store every time there's a new or changed product. However, the individual POS systems do not require a store master. Information technology (IT) staffs frequently must assemble the necessary components of the store dimension from multiple operational sources at headquarters.

The store dimension is the primary geographic dimension in our case study. Each store can be thought of as a location. Because of this, we can roll stores up to any geographic attribute, such as ZIP code, county, and state in the United States. Stores usually also roll up to store districts and regions. These two different hierarchies are both easily represented in the store dimension because both the geographic and store regional hierarchies are well defined for a single store row.

It is not uncommon to represent multiple hierarchies in a dimension table. Ideally, the attribute names and values should be unique across the multiple hierarchies.

A recommended store dimension table for the grocery business is shown in Figure 2.8.

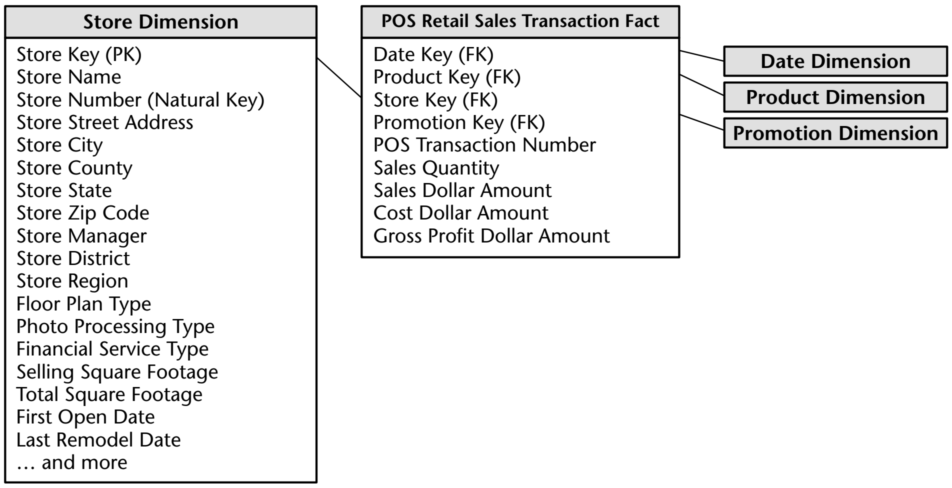

Figure 2.8 Store dimension in the retail sales schema.

The floor plan type, photo processing type, and finance services type are all short text descriptors that describe the particular store. These should not be one-character codes but rather should be 10- to 20-character standardized descriptors that make sense when viewed in a pull-down list or used as a report row header.

The column describing selling square footage is numeric and theoretically additive across stores. One might be tempted to place it in the fact table. However, it is clearly a constant attribute of a store and is used as a report constraint or row header more often than it is used as an additive element in a summation. For these reasons, we are confident that selling square footage belongs in the store dimension table.

The first open date and last remodel date typically are join keys to copies of the date dimension table. These date dimension copies are declared in SQL by the VIEW construct and are semantically distinct from the primary date dimension. The VIEW declaration would look like

```
CREATE VIEW FIRST_OPEN_DATE (FIRST_OPEN_DAY_NUMBER, FIRST_OPEN_MONTH ...)
    AS SELECT DAY_NUMBER, MONTH, ...
    FROM DATE
```

Now the system acts as if there is another physical copy of the date dimension table called FIRST_OPEN_DATE. Constraints on this new date table have nothing to do with constraints on the primary date dimension table. The first open date view is a permissible outrigger to the store dimension. Notice that we have carefully relabeled all the columns in the view so that they cannot be confused with columns from the primary date dimension. We will further discuss outriggers in Chapter 6.

Promotion Dimension

The promotion dimension is potentially the most interesting dimension in our schema. The promotion dimension describes the promotion conditions under which a product was sold. Promotion conditions include temporary price reductions, end-aisle displays, newspaper ads, and coupons. This dimension is often called a *causal* dimension (as opposed to a casual dimension) because it describes factors thought to cause a change in product sales.

Managers at both headquarters and the stores are interested in determining whether a promotion is effective or not. Promotions are judged on one or more of the following factors:

- Whether the products under promotion experienced a gain in sales during the promotional period. This is called the *lift*. The lift can only be measured if the store can agree on what the baseline sales of the promoted products would have been without the promotion. Baseline values can be estimated from prior sales history and, in some cases, with the help of sophisticated mathematical models.

- Whether the products under promotion showed a drop in sales just prior to or after the promotion, canceling the gain in sales during the promotion (time shifting). In other words, did we transfer sales from regularly priced products to temporarily reduced-priced products?
- Whether the products under promotion showed a gain in sales but other products nearby on the shelf showed a corresponding sales decrease (cannibalization).
- Whether all the products in the promoted category of products experienced a net overall gain in sales taking into account the time periods before, during, and after the promotion (market growth).
- Whether the promotion was profitable. Usually the profit of a promotion is taken to be the incremental gain in profit of the promoted category over the baseline sales taking into account time shifting and cannibalization, as well as the costs of the promotion, including temporary price reductions, ads, displays, and coupons.

The causal conditions potentially affecting a sale are not necessarily tracked directly by the POS system. The transaction system keeps track of price reductions and markdowns. The presence of coupons also typically is captured with the transaction because the customer either presents coupons at the time of sale or does not. Ads and in-store display conditions may need to be linked from other sources.

The various possible causal conditions are highly correlated. A temporary price reduction usually is associated with an ad and perhaps an end-aisle display. Coupons often are associated with ads. For this reason, it makes sense to create one row in the promotion dimension for each combination of promotion conditions that occurs. Over the course of a year, there may be 1,000 ads, 5,000 temporary price reductions, and 1,000 end-aisle displays, but there may only be 10,000 combinations of these three conditions affecting any particular product. For example, in a given promotion, most of the stores would run all three promotion mechanisms simultaneously, but a few of the stores would not be able to deploy the end-aisle displays. In this case, two separate promotion condition rows would be needed, one for the normal price reduction plus ad plus display and one for the price reduction plus ad only. A recommended promotion dimension table is shown in Figure 2.9.

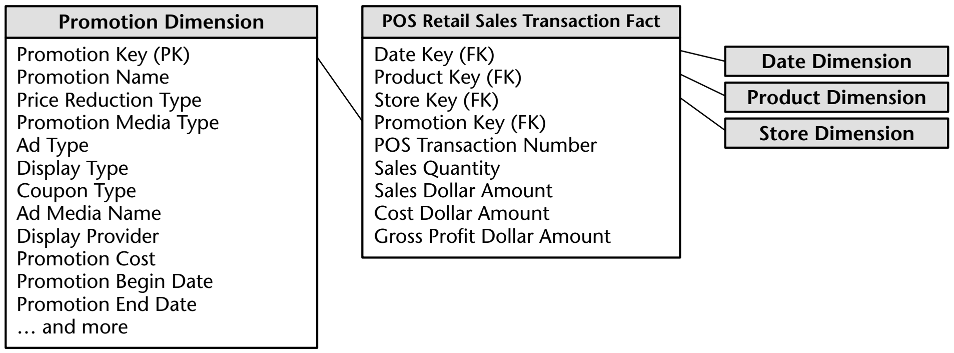

Figure 2.9 Promotion dimension in the retail sales schema.

From a purely logical point of view, we could record very similar information about the promotions by separating the four major causal mechanisms (price reductions, ads, displays, and coupons) into four separate dimensions rather than combining them into one dimension. Ultimately, this choice is the designer's prerogative. The tradeoffs in favor of keeping the four dimensions together include the following:

- Since the four causal mechanisms are highly correlated, the combined single dimension is not much larger than any one of the separated dimensions would be.
- The combined single dimension can be browsed efficiently to see how the various price reductions, ads, displays, and coupons are used together. However, this browsing only shows the possible combinations. Browsing in the dimension table does not reveal which stores or products were affected by the promotion. This information is found in the fact table.

The tradeoffs in favor of separating the four causal mechanisms into distinct dimension tables include the following:

- The separated dimensions may be more understandable to the business community if users think of these mechanisms separately. This would be revealed during the business requirement interviews.
- Administration of the separate dimensions may be more straightforward than administering a combined dimension.

Keep in mind that there is no difference in the information content in the data warehouse between these two choices.

Typically, many sales transaction line items involve products that are not being promoted. We will need to include a row in the promotion dimension, with its own unique key, to identify "No Promotion in Effect" and avoid a null promotion key in the fact table. Referential integrity is violated if we put a null in a fact table column declared as a foreign key to a dimension table. In addition to the referential integrity alarms, null keys are the source of great confusion to our users because they can't join on null keys.

You must avoid null keys in the fact table. A proper design includes a row in the corresponding dimension table to identify that the dimension is not applicable to the measurement.

Promotion Coverage Factless Fact Table

Regardless of the handling of the promotion dimension, there is one important question that cannot be answered by our retail sales schema: What products were on promotion but did not sell? The sales fact table only records the SKUs actually sold. There are no fact table rows with zero facts for SKUs that didn't sell because doing so would enlarge the fact table enormously. In the relational world, a second promotion coverage or event fact table is needed to help answer the question concerning what didn't happen. The promotion coverage fact table keys would be date, product, store, and promotion in our case study. This obviously looks similar to the sales fact table we just designed; however, the grain would be significantly different. In the case of the promotion coverage fact table, we'd load one row in the fact table for each product on promotion in a store each day (or week, since many retail promotions are a week in duration) regardless of whether the product sold or not. The coverage fact table allows us to see the relationship between the keys as defined by a promotion, independent of other events, such as actual product sales. We refer to it as a *factless fact table* because it has no measurement metrics; it merely captures the relationship between the involved keys. To determine what products where on promotion but didn't sell requires a two-step process. First, we'd query the promotion coverage table to determine the universe of products that were on promotion on a given day. We'd then determine what products sold from the POS sales fact table. The answer to our original question is the set difference between these two lists of products. Stay tuned to Chapter 12 for more complete coverage of factless fact tables; we'll illustrate the promotion coverage table and provide the set difference SQL. If you're working with data in a multidimensional online analytical processing (OLAP) cube environment, it is often easier to answer the question regarding what didn't sell because the cube typically contains explicit cells for nonbehavior.

Degenerate Transaction Number Dimension

The retail sales fact table contains the POS transaction number on every line item row. In a traditional parent-child database, the POS transaction number would be the key to the transaction header record, containing all the information valid for the transaction as a whole, such as the transaction date and store identifier. However, in our dimensional model, we have already extracted this interesting header information into other dimensions. The POS transaction number is still useful because it serves as the grouping key for pulling together all the products purchased in a single transaction.

Although the POS transaction number looks like a dimension key in the fact table, we have stripped off all the descriptive items that might otherwise fall in a POS transaction dimension. Since the resulting dimension is empty, we refer to the POS transaction number as a *degenerate dimension* (identified by the DD notation in Figure 2.10). The natural operational ticket number, such as the POS transaction number, sits by itself in the fact table without joining to a dimension table. Degenerate dimensions are very common when the grain of a fact table represents a single transaction or transaction line item because the degenerate dimension represents the unique identifier of the parent. Order numbers, invoice numbers, and bill-of-lading numbers almost always appear as degenerate dimensions in a dimensional model.

Degenerate dimensions often play an integral role in the fact table's primary key. In our case study, the primary key of the retail sales fact table consists of the degenerate POS transaction number and product key (assuming that the POS system rolls up all sales for a given product within a POS shopping cart into a single line item). Often, the primary key of a fact table is a subset of the table's foreign keys. We typically do not need every foreign key in the fact table to guarantee the uniqueness of a fact table row.

Operational control numbers such as order numbers, invoice numbers, and bill-of-lading numbers usually give rise to empty dimensions and are represented as degenerate dimensions (that is, dimension keys without corresponding dimension tables) in fact tables where the grain of the table is the document itself or a line item in the document.

If, for some reason, one or more attributes are legitimately left over after all the other dimensions have been created and seem to belong to this header entity, we would simply create a normal dimension record with a normal join. However, we would no longer have a degenerate dimension.

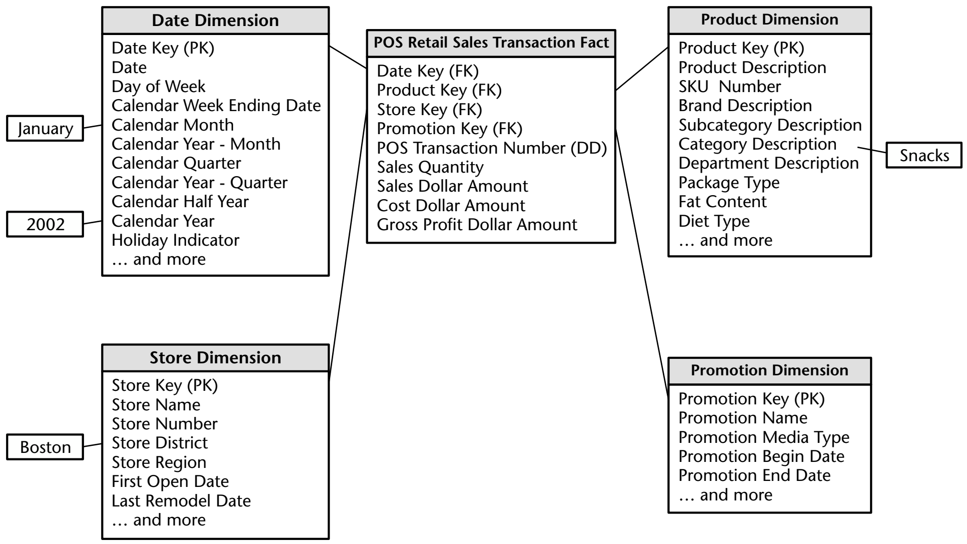

Figure 2.10 Querying the retail sales schema.

Retail Schema in Action

With our retail POS schema designed, let's illustrate how it would be put to use in a query environment. A business user might be interested in better understanding weekly sales dollar volume by promotion for the snacks category during January 2002 for stores in the Boston district. As illustrated in Figure 2.10, we would place query constraints on month and year in the date dimension, district in the store dimension, and category in the product dimension.

If the query tool summed the sales dollar amount grouped by week-ending date and promotion, the query results would look similar to those below. You can plainly see the relationship between the dimensional model and the associated query. High-quality dimension attributes are crucial because they are the source of query constraints and result set labels.

Calendar Week Ending Date	Promotion Name	Sales Dollar Amount
January 6, 2002	No Promotion	22,647
January 13, 2002	No Promotion	4,851
January 20, 2002	Super Bowl Promotion	7,248
January 27, 2002	Super Bowl Promotion	13,798

If you are using a data access tool with more functionality, the results may appear as a cross-tabular report. Such reports are more appealing to business users than the columnar data resulting from an SQL statement.

Calendar Week Ending Date	Super Bowl Promotion Sales Dollar Amount	No Promotion Sales Dollar Amount
January 6, 2002	0	22,647
January 13, 2002	0	4,851
January 20, 2002	7,248	0
January 27, 2002	13,793	0

Retail Schema Extensibility

Now that we've completed our first dimensional model, let's turn our attention to extending the design. Assume that our retailer decides to implement a frequent shopper program. Now, rather than knowing that an unidentified shopper had 26 items in his or her shopping cart, we're able to see exactly what a specific shopper, say, Julie Kimball, purchases on a weekly basis. Just imagine the interest of business users in analyzing shopping patterns by a multitude of geographic, demographic, behavioral, and other differentiating shopper characteristics.

The handling of this new frequent shopper information is relatively straightforward. We'd create a frequent shopper dimension table and add another foreign key in the fact table. Since we can't ask shoppers to bring in all their old cash register receipts to tag our historical sales transactions with their new frequent shopper number, we'd substitute a shopper key corresponding to a "Prior to Frequent Shopper Program" description on our historical fact table rows. Likewise, not everyone who shops at the grocery store will have a frequent shopper card, so we'd also want to include a "Frequent Shopper Not Identified" row in our shopper dimension. As we discussed earlier with the promotion dimension, we must avoid null keys in the fact table.

As we embellished our original design with a frequent shopper dimension, we also could add dimensions for the time of day and clerk associated with the transaction, as illustrated in Figure 2.11. Any descriptive attribute that has a single value in the presence of the fact table measurements is a good candidate to be added to an existing dimension or be its own dimension. The decision regarding whether a dimension can be attached to a fact table should be a binary yes/no based on the declared grain. If you are in doubt, it's time to revisit step 2 of the design process.

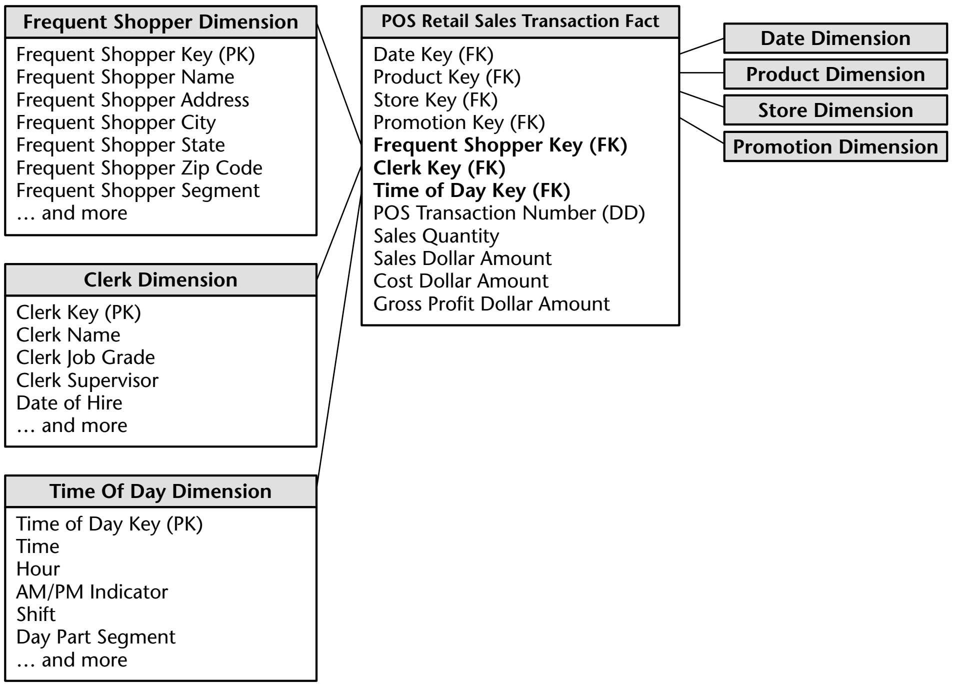

Figure 2.11 Embellished retail sales schema.

Our original schema gracefully extends to accommodate these new dimensions largely because we chose to model the POS transaction data at its most granular level. The addition of dimensions that apply at that granularity did not alter the existing dimension keys or facts; all preexisting applications continue to run without unraveling or changing. If we had declared originally that the grain would be daily retail sales (transactions summarized by day, store, product, and promotion) rather than at transaction line detail, we would not have been able to easily incorporate the frequent-shopper, time-of-day, or clerk dimensions. Premature summarization or aggregation inherently limits our ability to add supplemental dimensions because the additional dimensions often don't apply at the higher grain.

Obviously, there are some changes that can never be handled gracefully. If a data source ceases to be available and there is no compatible substitute, then the data warehouse applications depending on this source will stop working. However, the predictable symmetry of dimensional models allow them to absorb some rather significant changes in source data and/or modeling assumptions without invalidating existing applications. We'll describe several of these unexpected modification categories, starting with the simplest:

New dimension attributes. If we discover new textual descriptors of a product, for example, we add these attributes to the dimension as new columns. All existing applications will be oblivious to the new attributes and continue to function. If the new attributes are available only after a specific point in time, then "Not Available" or its equivalent should be populated in the old dimension records.

New dimensions. As we just illustrated in Figure 2.11, we can add a dimension to an existing fact table by adding a new foreign key field and populating it correctly with values of the primary key from the new dimension.

New measured facts. If new measured facts become available, we can add them gracefully to the fact table. The simplest case is when the new facts are available in the same measurement event and at the same grain as the existing facts. In this case, the fact table is altered to add the new columns, and the values are populated into the table. If the ALTER TABLE statement is not viable, then a second fact table must be defined with the additional columns and the rows copied from the first. If the new facts are only available from a point in time forward, then null values need to be placed in the older fact rows. A more complex situation arises when new measured facts occur naturally at a different grain. If the new facts cannot be allocated or assigned to the original grain of the fact table, it is very likely that the new facts belong in their own fact table. It is almost always a mistake to mix grains in the same fact table.

Dimension becoming more granular. Sometimes it is desirable to increase the granularity of a dimension. In most cases, the original dimension attributes can be included in the new, more granular dimension because they roll up perfectly in a many-to-one relationship. The more granular dimension often implies a more granular fact table. There may be no alternative but to drop the fact table and rebuild it. However, all the existing applications would be unaffected.

Addition of a completely new data source involving existing dimensions as well as unexpected new dimensions. Almost always, a new source of data has its own granularity and dimensionality, so we create a new fact table. We should avoid force-fitting new measurements into an existing fact table of consistent measurements. The existing applications will still work because the existing fact and dimension tables are untouched.

Resisting Comfort Zone Urges

With our first dimensional design behind us, let's directly confront several of the natural urges that tempt modelers coming from a more normalized background. We're consciously breaking some traditional modeling rules because we're focused on delivering business value through ease of use and performance, not on transaction processing efficiencies.

Dimension Normalization (Snowflaking)

The flattened, denormalized dimension tables with repeating textual values may make a normalization modeler uncomfortable. Let's revisit our case study product dimension table. The 150,000 products roll up into 50 distinct departments. Rather than redundantly storing the 20-byte department description in the product dimension table, modelers with a normalized upbringing want to store a 2-byte department code and then create a new department dimension for the department decodes. In fact, they would feel more comfortable if all the descriptors in our original design were normalized into separate dimension tables. They argue that this design saves space because we're only storing cryptic codes in our 150,000-row dimension table, not lengthy descriptors.

In addition, some modelers contend that the normalized design for the dimension tables is easier to maintain. If a department description changes, they'd only need to update the one occurrence rather than the 3,000 repetitions in our original product dimension. Maintenance often is addressed by normalization disciplines, but remember that all this happens back in the staging area, long before the data is loaded into a presentation area's dimensional schema.

Dimension table normalization typically is referred to as *snowflaking*. Redundant attributes are removed from the flat, denormalized dimension table and placed in normalized secondary dimension tables. Figure 2.12 illustrates the partial snowflaking of our original schema. If the schema were fully snowflaked, it would appear as a full third-normal-form entity-relationship diagram. The contrast between Figure 2.12 and the earlier design in Figure 2.10 is startling. While the fact tables in both figures are identical, the plethora of dimension tables (even in our simplistic representation) is overwhelming.

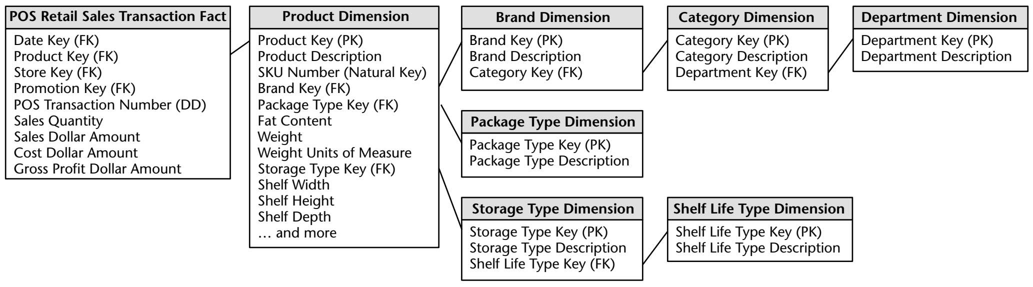

Figure 2.12 Partially snowflaked product dimension.

While snowflaking is a legal extension of the dimensional model, in general, we encourage you to resist the urge to snowflake given our two primary design drivers, ease of use and performance.

- The multitude of snowflaked tables makes for a much more complex presentation. Users inevitably will struggle with the complexity. Remember that simplicity is one of the primary objectives of a denormalized dimensional model.
- Likewise, database optimizers will struggle with the complexity of the snowflaked schema. Numerous tables and joins usually translate into slower query performance. The complexities of the resulting join specifications increase the chances that the optimizer will get sidetracked and choose a poor strategy.
- The minor disk space savings associated with snowflaked dimension tables are insignificant. If we replaced the 20-byte department description in our 150,000-row product dimension table with a 2-byte code, we'd save a whopping 2.7 MB (150,000 x 18 bytes), but we may have a 10-GB fact table! Dimension tables are almost always geometrically smaller than fact table. Efforts to normalize most dimension tables in order to save disk space are a waste of time.
- Snowflaking slows down the users' ability to browse within a dimension. Browsing often involves constraining one or more dimension attributes and looking at the distinct values of another attribute in the presence of these constraints. Browsing allows users to understand the relationship between dimension attribute values.

  Obviously, a snowflaked product dimension table would respond well if we just wanted a list of the category descriptions. However, if we wanted to see all the brands within a category, we'd need to traverse the brand and category dimensions. If we then wanted to also list the package types for each brand in a category, we'd be traversing even more tables. The SQL needed to perform these seemingly simple queries is quite complex, and we haven't even touched the other dimensions or fact table.
- Finally, snowflaking defeats the use of bitmap indexes. Bitmap indexes are very useful when indexing low-cardinality fields, such as the category and department columns in our product dimension tables. They greatly speed the performance of a query or constraint on the single column in question. Snowflaking inevitably would interfere with your ability to leverage this performance-tuning technique.

The dimension tables should remain as flat tables physically. Normalized, snowflaked dimension tables penalize cross-attribute browsing and prohibit the use of bit-mapped indexes. Disk space savings gained by normalizing the dimension tables typically are less than 1 percent of the total disk space needed for the overall schema. We knowingly sacrifice this dimension table space in the spirit of performance and ease-of-use advantages.

There are times when snowflaking is permissible, such as our earlier example with the date outrigger on the store dimension, where a clump of correlated attributes is used repeatedly in various independent roles. We just urge you to be conservative with snowflaked designs and use them only when they are obviously called for.

Too Many Dimensions

The fact table in a dimensional schema is naturally highly normalized and compact. There is no way to further normalize the extremely complex many-to-many relationships among the keys in the fact table because the dimensions are not correlated with each other. Every store is open every day. Sooner or later, almost every product is sold on promotion in most or all of our stores.

Interestingly, while uncomfortable with denormalized dimension tables, some modelers are tempted to denormalize the fact table. Rather than having a single product foreign key on the fact table, they include foreign keys for the frequently analyzed elements on the product hierarchy, such as brand, subcategory, category, and department. Likewise, the date key suddenly turns into a series of keys joining to separate week, month, quarter, and year dimension tables. Before you know it, our compact fact table has turned into an unruly monster that joins to literally dozens of dimension tables. We affectionately refer to these designs as *centipedes* because the fact tables appear to have nearly 100 legs, as shown in Figure 2.13. Clearly, the centipede design has stepped into the too-many-dimensions trap.

Remember, even with its tight format, the fact table is the behemoth in a dimensional design. Designing a fact table with too many dimensions leads to significantly increased fact table disk space requirements. While we're willing to use extra space for dimension tables, fact table space consumption concerns us because it is our largest table by orders of magnitude. There is no way to index the enormous multipart key effectively in our centipede example. The numerous joins are an issue for both usability and query performance.

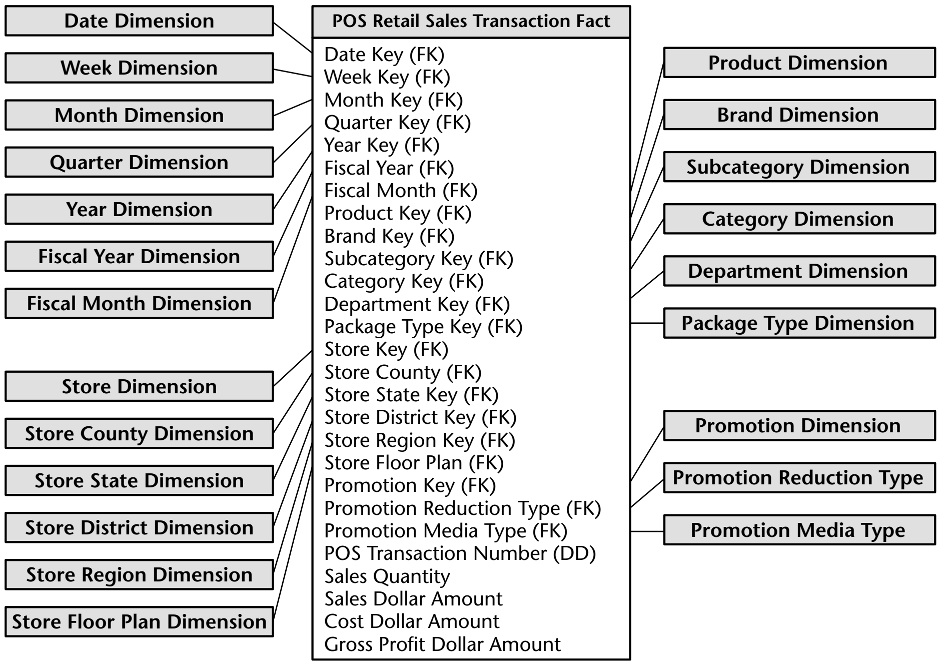

Figure 2.13 Centipede fact table with too many dimensions.

Most business processes can be represented with less than 15 dimensions in the fact table. If our design has 25 or more dimensions, we should look for ways to combine correlated dimensions into a single dimension. Perfectly correlated attributes, such as the levels of a hierarchy, as well as attributes with a reasonable statistical correlation, should be part of the same dimension. You have made a good decision to combine dimensions when the resulting new single dimension is noticeably smaller than the Cartesian product of the separate dimensions.

A very large number of dimensions typically is a sign that several dimensions are not completely independent and should be combined into a single dimension. It is a dimensional modeling mistake to represent elements of a hierarchy as separate dimensions in the fact table.

Surrogate Keys

We strongly encourage the use of surrogate keys in dimensional models rather than relying on operational production codes. Surrogate keys go by many other aliases: *meaningless keys*, *integer keys*, *nonnatural keys*, *artificial keys*, *synthetic keys*, and so on. Simply put, surrogate keys are integers that are assigned sequentially as needed to populate a dimension. For example, the first product record is assigned a product surrogate key with the value of 1, the next product record is assigned product key 2, and so forth. The surrogate keys merely serve to join the dimension tables to the fact table.

Modelers sometimes are reluctant to give up their natural keys because they want to navigate the fact table based on the operational code while avoiding a join to the dimension table. Remember, however, that dimension tables are our entry points to the facts. If the fifth through ninth characters in the operational code identify the manufacturer, then the manufacturer's name should be included as a dimension table attribute. In general, we want to avoid embedding intelligence in the data warehouse keys because any assumptions that we make eventually may be invalidated. Likewise, queries and data access applications should not have any built-in dependency on the keys because the logic also would be vulnerable to invalidation.

Every join between dimension and fact tables in the data warehouse should be based on meaningless integer surrogate keys. You should avoid using the natural operational production codes. None of the data warehouse keys should be smart, where you can tell something about the row just by looking at the key.

Initially, it may be faster to implement a dimensional model using operational codes, but surrogate keys definitely will pay off in the long run. We sometimes think of them as being similar to a flu shot for the data warehouse—like an immunization, there's a small amount of pain to initiate and administer surrogate keys, but the long-run benefits are substantial.

One of the primary benefits of surrogate keys is that they buffer the data warehouse environment from operational changes. Surrogate keys allow the warehouse team to maintain control of the environment rather than being whipsawed by operational rules for generating, updating, deleting, recycling, and reusing production codes. In many organizations, historical operational codes (for example, inactive account numbers or obsolete product codes) get reassigned after a period of dormancy. If account numbers get recycled following 12 months of inactivity, the operational systems don't miss a beat because their business rules prohibit data from hanging around for that long. The data warehouse, on the other hand, will retain data for years. Surrogate keys provide the warehouse with a mechanism to differentiate these two separate instances of the same operational account number. If we rely solely on operational codes, we also are vulnerable to key overlap problems in the case of an acquisition or consolidation of data. Surrogate keys allow the data warehouse team to integrate data from multiple operational source systems, even if they lack consistent source keys.

There are also performance advantages associated with the use of surrogate keys. The surrogate key is as small an integer as possible while ensuring that it will accommodate the future cardinality or maximum number of rows in the dimension comfortably. Often the operational code is a bulky alphanumeric character string. The smaller surrogate key translates into smaller fact tables, smaller fact table indices, and more fact table rows per block input-output operation. Typically, a 4-byte integer is sufficient to handle most dimension situations. A 4-byte integer is a single integer, not four decimal digits. It has 32 bits and therefore can handle approximately 2 billion positive values (2^{32-1}) or 4 billion total positive and negative values (-2^{32-1} to $+2^{32-1}$). As we said, this is more than enough for just about any dimension. Remember, if you have a large fact table with 1 billion rows of data, every byte in each fact table row translates into another gigabyte of storage.

As we mentioned earlier, surrogate keys are used to record dimension conditions that may not have an operational code, such as the "No Promotion in Effect" condition. By taking control of the warehouse's keys, we can assign a surrogate key to identify this condition despite the lack of operational coding.

Similarly, you may find that your dimensional models have dates that are yet to be determined. There is no SQL date value for "Date to be Determined" or "Date Not Applicable." This is another reason we advocate using surrogate keys for your date keys rather than SQL date data types (as if our prior rationale wasn't convincing enough).

The date dimension is the one dimension where surrogate keys should be assigned in a meaningful, sequential order. In other words, January 1 of the first year would be assigned surrogate key value 1, January 2 would be assigned surrogate key 2, February 1 would be assigned surrogate key 32, and so on. We don't want to embed extensive calendar intelligence in these keys (for example, YYYY-MM-DD) because doing so may encourage people to bypass the date lookup dimension table. And, of course, in using this smart format, we would again have no way to represent "Hasn't happened yet" and other common date situations. We just want our fact table rows to be in sequential order. Treating the surrogate date key as a date sequence number will allow the fact table to be physically partitioned on the basis of the date key. Partitioning a large fact table on the basis of date is highly effective because it allows old data to be removed gracefully and new data to be loaded and indexed without disturbing the rest of the fact table.

Finally, surrogate keys are needed to support one of the primary techniques for handling changes to dimension table attributes. This is actually one of the most important reasons to use surrogate keys. We'll devote a whole section in Chapter 4 to using surrogate keys for slowly changing dimensions.

Of course, some effort is required to assign and administer surrogate keys, but it's not nearly as intimidating as many people imagine. We'll need to establish and maintain a cross-reference table in the staging area that will be used to substitute the appropriate surrogate key on each fact and dimension table row. In Chapter 16 we lay out a flow diagram for administering and processing surrogate keys in our dimensional schemas.

Before we leave the topic of keys, we want to discourage the use of concatenated or compound keys for dimension tables. We can't create a truly surrogate key simply by gluing together several natural keys or by combining the natural key with a time stamp. Also, we want to avoid multiple parallel joins between the dimension and fact tables, sometimes referred to as *double-barreled joins*, because they have an adverse impact on performance.

While we don't typically assign surrogate keys to degenerate dimensions, you should evaluate each situation to determine if one is required. A surrogate key is necessary if the transaction control numbers are not unique across locations or get reused. For example, our retailer's POS system may not assign unique transaction numbers across stores. The system may wrap back to zero and reuse previous control numbers once its maximum has been reached. Also, your transaction control number may be a bulky 24-byte alphanumeric column. In such cases, it would be advantageous to use a surrogate key. Technically, control number dimensions modeled in this way are no longer degenerate.

For the moment, let's assume that the first version of the retail sales schema represents both the logical and physical design of our database. In other words, the relational database contains only five actual tables: retail sales fact table and date, product, store, and promotion dimension tables. Each of the dimension tables has a primary key, and the fact table has a composite key made up of these four foreign keys, in addition to the degenerate transaction number. Perhaps the most striking aspect of the design at this point is the simplicity of the fact table. If the four foreign keys are tightly administered consecutive integers, we could reserve as little as 14 bytes for all four keys (4 bytes each for date, product, and promotion and 2 bytes for store). The transaction number might require an additional 8 bytes. If the four facts in the fact table were each 4-byte integers, we would need to reserve only another 16 bytes. This would make our fact table row only 38 bytes wide. Even if we had a billion rows, the fact table would occupy only about 38 GB of primary data space. Such a streamlined fact table row is a very typical result in a dimensional design.

Our embellished retail sales schema, illustrated in Figure 2.11, has three additional dimensions. If we allocate 4 bytes each for shopper and clerk and 2 bytes for the time of day (to the nearest minute), then our fact table width grows to only 48 bytes. Our billion-row fact table occupies just 48 GB.

Market Basket Analysis

The retail sales schema tells us in exquisite detail what was purchased at each store and under what conditions. However, the schema doesn't allow us to very easily analyze which products were sold in the same market basket together. This notion of analyzing the combination of products that sell together is known by data miners as *affinity grouping* but more popularly is called *market basket analysis*. Market basket analysis gives the retailer insights about how to merchandise various combinations of items. If frozen pasta dinners sell well with cola products, then these two products perhaps should be located near each other or marketed with complementary pricing. The concept of market basket analysis can be extended easily to other situations. In the manufacturing environment, it is useful to see what products are ordered together because we may want to offer product bundles with package pricing.

The retail sales fact table cannot be used easily to perform market basket analyses because SQL was never designed to constrain and group across line item fact rows. Data mining tools and some OLAP products can assist with market basket analysis, but in the absence of these tools, we'll describe a more direct approach below. Be forewarned that this is a rather advanced technique; if you are not doing market basket analysis today, simply skim this section to get a general sense of the techniques involved.

In Figure 2.14 we illustrate a market basket fact table that was derived from retail sales transactions. The market basket fact table is a periodic snapshot representing the pairs of products purchased together during a specified time period. The facts include the total number of baskets (customer tickets) that included products A and B, the total number of product A dollars and units in this subset of purchases, and the total number of product B dollars and units purchased. The basket count is a semiadditive fact. For example, if a customer ticket contains line items for pasta, soft drinks, and peanut butter in the market basket fact table, this single order is counted once on the pasta-soft drinks fact row, once on the row for the pasta-peanut butter combination, and so on. Obviously, care must be taken to avoid summarizing purchase counts for more than one product.

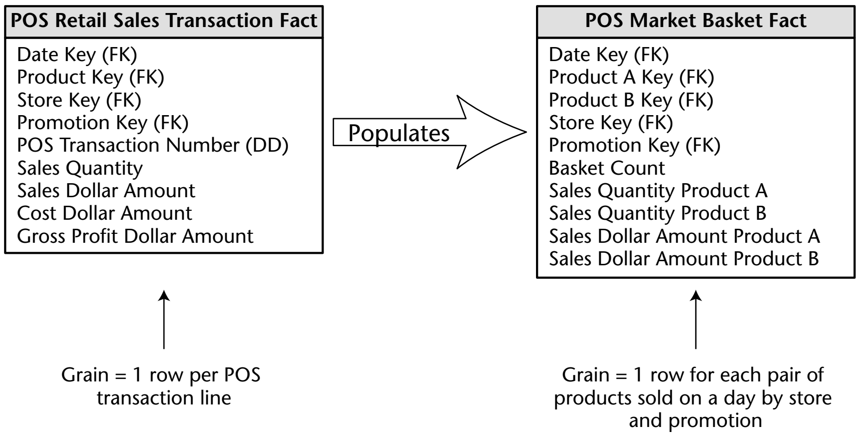

Figure 2.14 Market basket fact table populated from purchase transactions.

You will notice that there are two generalized product keys (product keys A and B) in the market basket fact table. Here we have built a single product dimension table that contains entries at multiple levels of the hierarchy, such as individual products, brands, and categories. This specialized variant of our normal product dimension table contains a small number of rather generic attributes. The surrogate keys for the various levels of the product hierarchy have been assigned so that they don't overlap.

Conceptually, the idea of recording market basket correlations is simple, but the sheer number of product combinations makes the analysis challenging. If we have *N* products in our product portfolio and we attempt to build a table with every possible pair of product keys encountered in product orders, we will approach N^2 product combinations [actually N x $(N - 1)$ for the mathematicians among you]. In other words, if we have 10,000 products in our portfolio, there would be nearly 100,000,000 pairwise combinations. The number of possible combinations quickly approaches absurdity when we're dealing with a large number of products. If a retail store sells 100,000 SKUs, there are 10 billion possible SKU combinations.

The key to realistic market basket analysis is to remember that the primary goal is to understand the *meaningful* combinations of products sold together. Thinking about our market basket fact table, we would first be interested in rows with high basket counts. Since these product combinations are observed frequently, they warrant further investigation. Second, we would look for situations where the dollars or units for products A and B were in reasonable balance. If the dollars or units are far out of balance, all we've done is find high-selling products coupled with insignificant secondary products, which wouldn't be very helpful in making major merchandising or promotion decisions.

In order to avoid the combinatorial explosion of product pairs in the market basket fact table, we rely on a progressive pruning algorithm. We begin at the top of the product hierarchy, which we'll assume is category. We first enumerate all the category-to-category market basket combinations. If there are 25 categories, this first step generates 625 market basket rows. We then prune this list for further analysis by selecting only the rows that have a reasonably high order count and where the dollars and units for products A and B (which are categories at this point) are reasonably balanced. Experimentation will tell you what the basket count threshold and balance range should be.

We then push down to the next level of detail, which we'll assume is brand. Starting with the pruned set of combinations from the last step, we drill down on product A by enumerating all combinations of brand (product A) by category (product B). Similarly, we drill down one level of the hierarchy for product B by looking at all combinations of brand (product A) by brand (product B). Again, we prune the lists to those with the highest basket count frequencies and dollar or unit balance and then drill down to the next level in the hierarchy.

As we descend the hierarchy, we produce rows with smaller and smaller basket counts. Eventually, we find no basket counts greater than the reasonable threshold for relevance. It is permissible to stop at any time once we've satisfied the analyst's curiosity. One of the advantages of this top-down approach is that the rows found at each point are those with the highest relevance and impact. Progressively pruning the list provides more focus to already relevant results. One can imagine automating this process, searching the product hierarchy downward, ignoring the low basket counts, and always striving for balanced dollars and units with the high basket counts. The process could halt when the number of product pairs reached some desired threshold or when the total activity expressed in basket count, dollars, or units reached some lower limit.

A variation on this approach could start with a specific category, brand, or even a product. Again, the idea would be to combine this specific product first with all the categories and then to work down the hierarchy. Another twist would be to look at the mix of products purchased by a given customer during a given time period, regardless of whether they were in the same basket. In any case, much of the hard work associated with market basket analysis has been off-loaded to the staging area's ETL processes in order to simplify the ultimate query and presentation aspects of the analysis.

Summary

In this chapter we got our first exposure to designing a dimensional model. Regardless of industry, we strongly encourage the four-step process for tackling dimensional model designs. Remember that it is especially important that we clearly state the grain associated with our dimensional schema. Loading the fact table with atomic data provides the greatest flexibility because we can summarize that data "every which way." As soon as the fact table is restricted to more aggregated information, we'll run into walls when the summarization assumptions prove to be invalid. Also remember that it is vitally important to populate our dimension tables with verbose, robust descriptive attributes.

In the next chapter we'll remain within the retail industry to discuss techniques for tackling a second business process within the organization, ensuring that we're leveraging our earlier efforts while avoiding stovepipes.