CHAPTER 4

Designing a Data Warehouse

Organizing is what you do before you do something, so that when you do it, it is not all mixed up.

—A. A. Milne

Designing a data warehouse is one of the most important aspects of a business intelligence solution. If the data warehouse is designed correctly, all other aspects of the solution will benefit. Conversely, if it is created incorrectly, it will cause no end of problems.

In this chapter, we show techniques for designing a data warehouse, including different designs and terms used in the creation process and their proper use. Our focus is on simple practical designs that will get you building your first data warehouses quickly and easily.

When you have completed this chapter, you should be able to design data warehouses using industry standards and know the common rules that are considered "best practice" for the design process.

What Is a Data Warehouse?

At its core, a data warehouse is a collection of data designed for the easy extraction of information. It can be in any form, including a series of text files, but most often it is a relational database. Because of this, most developers think of a data warehouse simply as a reporting database. And although that is not the most highbrow definition, it is fairly accurate.

Many developers will have differing opinions on what is the best way to design a data warehouse. But there are common characteristics you can expect to see in all of them. The first common characteristic is a set of values used for reports. These are called *measures*. For example, InventoryUnits and SalesDollars can be considered measures. Another common characteristic found in data warehouses is a set of dimensions. Dimensions describe the measured data. Examples of dimensions include the dates that the InventoryUnits were documented or the zip code of the customers who bought a particular product.

We discuss more on both of these subjects in the next few pages.

What Is a Data Mart?

A data mart is also a collection of data. It, too, is designed to allow for the easy extraction of information. The information in a data mart, however, is more specific than that of a data warehouse. Typically, a data mart is created for a particular process, such as a sales event or taking inventory.

Data marts can also be designed around departments within the company. But you are typically better off defining the data mart based on a process, not a department. This is because when you define it with a process, many different departments can use the same data mart. If you define it on a department, it may be isolated from being reused, and you can end up with redundant data marts that cause more confusion rather than provide accurate information.

Data marts can be thought of as a subset of a data warehouse. Because data warehouses represent multiple processes and a data mart focuses on a single process, a data warehouse can be thought of as containing one or more data marts. Therefore, if a business has both a sales or inventory process, a sales and inventory data mart would be part of that business's data warehouse. When you build your data warehouse, you create two fact tables in one database. These two tables provide the core data for the data warehouse (Listing 4-1).

Listing 4-1. An Expression Describing a Data Mart

```
/* Data Warehouse=a set of data marts {Sales Data Mart, Inventory Data Mart} */
Select * from FactSales
Select * from FactInventory
```

This allows for easy access to the information for a particular process and a way of managing data for all of the company's processes. In summary, a data warehouse is a collection of one or more data marts, and a data mart is a collection of data around a particular process.

Competing Definitions

The definitions we use in this book are found in many other books as well, but not all. A number of competing viewpoints exist that describe what something is called in the business intelligence world. This diversity of terms began with two different industry leaders, Bill Inmon and Ralph Kimball.

In the early 1990s, Inmon published several articles on data warehousing. Later, Kimball also published articles, as well as a famous book known as *The Data Warehouse Toolkit* in the mid-to-late 1990s.

Although both agree on the principle of data warehousing providing information to the users, they differ on how to accomplish this task. Inmon believes that a data warehouse should be very comprehensive and reflect all aspects of the business before it can truly be useful. This tactic has proven to be most effective for many large companies. But it has proven to be less useful for smaller to midsize companies that do not require such complete integration from all aspects of the company in order for their businesses to operate.

Kimball's theory is that a BI solution should focus on smaller specific topics such as business processes. As soon as you implement one process in a BI solution, it is available for reporting, and work can begin on the next process to be added to the solution. This is considered a bottom-up approach.

To paraphrase, Kimball outlines a bottom-up approach by starting small and using building blocks to create something of larger scope, which can be added to over time. Inmon's theory, on the other hand, focuses on a large holistic approach building from the top down in a comprehensive manner that requires all aspects to be incorporated before the data warehouse can function as it is designed.

Many developers have adopted one philosophy or another. Sometimes they even get a bit overzealous about it, not unlike choosing an athletic team to root for in sports. Both approaches, however, have their uses and should be considered tools that, when used appropriately, provide maximum results.

Stereotypically, larger companies will benefit from the top-down approach because they often need holistic information about the many departments that make up the company. Small to midsize companies are more likely to benefit from the bottom-up approach. These companies have few departments, and communication between these departments is easier to manage.

Keep in mind that both approaches will work for companies of all sizes, so each solution must be evaluated independently, but a bottom-up approach usually is appropriate more often than not. In addition, all companies can benefit from a bottom-up approach at some point, because it requires less time and fewer resources before reporting can begin.

Starting with an OLTP Design

When you are designing a data warehouse, there is a strong chance that you will begin by reviewing an existing OLTP database. Although a data warehouse can be built on simple log files or XML data, it is much more likely that it will be built upon an OLTP database.

The standard OLTP database design has been defined for several decades. Many readers may be familiar with these patterns, but we do not expect everyone to have the same level of technical background, so let's take a brief look at these patterns before we move on.

The three basic table patterns in an OLTP database are one-to-one, one-to-many, and many-to-many relationship patterns. Figure 4-1 demonstrates these common design patterns.

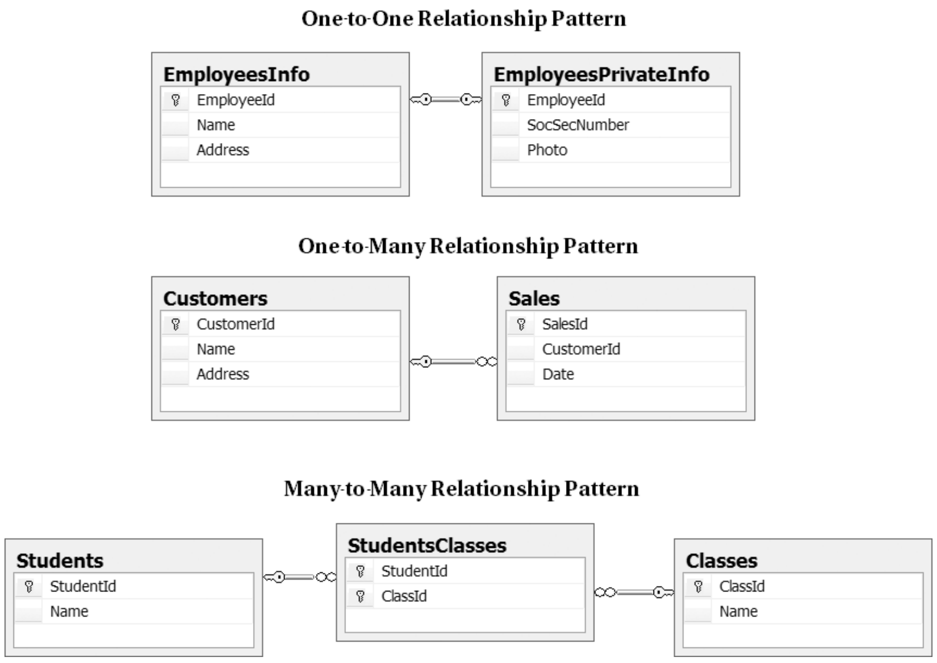


Figure 4-1. *Standard OLTP design patterns*

In a one-to-one relationship pattern, a single table is divided along its columns into two tables. Although not unheard of, this pattern is somewhat unusual. Quite often, the goal with this pattern is to separate private information from public information or possibly to partition data onto a separate hard drive for performance reasons. In Figure 4-1, two employee information tables represent a one-to-one relationship. All of the data could have been stored in one employee table, but because Social Security information is considered private, it has been separated into an additional table. Note that the EmployeeId in either table is never repeated in this type of relationship pattern.

The one-to-many relationship pattern also seen in Figure 4-1 is by far the most common pattern in OLTP databases. Our example shows a one-to-many relationship between customers and sales. These tables demonstrate that one customer can have many sales, but an individual sale is associated with only one customer.

The many-to-many relationship pattern of Figure 4-1 demonstrates the process of how one student can attend many classes and one class can hold many students. Note that a third table records the relationship data by storing a copy of the two related tables' key values.

As we show later, these design patterns are also represented in a data warehouse, but for now let's look at an example of a typical OLTP database. It enables you to contrast these OLTP designs with that of a data warehouse design.

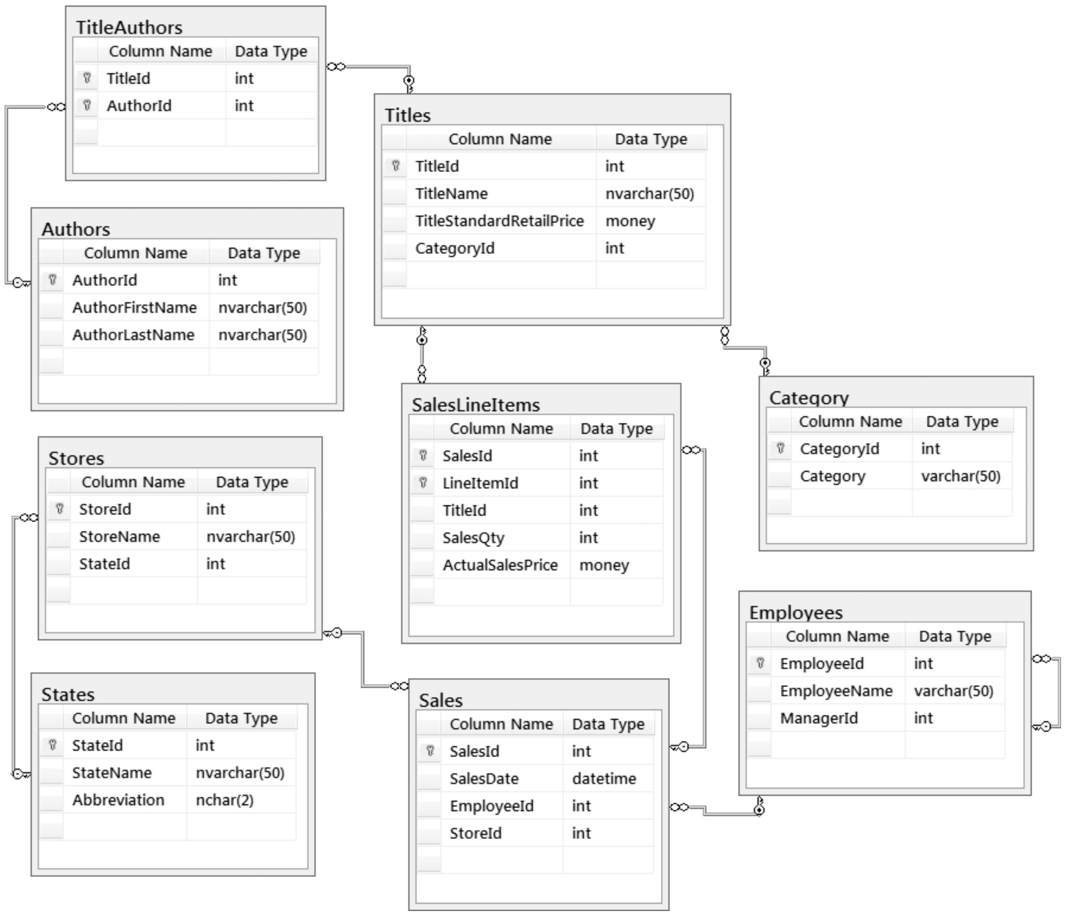
A Typical OLTP Database Design

The database we have created for our example consists of tables that record transactional data in the tables called Sales and SalesLineItems. Along with the transactional data are supporting tables. These tables describe the transactions that occur. Examples of these types of tables are the Employees, Stores, and Titles tables.

Normalized Tables

Standard OLTP databases are designed around the concept of normalization, and we have used the rules of normalization to design this example OLTP database.

In a normalized database, all columns contain a single unit of data, such as city or state, but never both city and state as a single entry. All rows contain a unique combination of values, and redundancy is eliminated wherever possible. For example, looking at Figure 4-2, you see that the Stores table has a link to the States table based on StateId.



TitleAuthors

Column Name	Data Type
TitleId	int
AuthorId	int

Titles

Column Name	Data Type
TitleId	int
TitleName	nvarchar(50)
TitleStandardRetailPrice	money
CategoryId	int

Authors

Column Name	Data Type
AuthorId	int
AuthorFirstName	nvarchar(50)
AuthorLastName	nvarchar(50)

SalesLineItems

Column Name	Data Type
SalesId	int
LineItemId	int
TitleId	int
SalesQty	int
ActualSalesPrice	money

Category

Column Name	Data Type
CategoryId	int
Category	varchar(50)

Stores

Column Name	Data Type
StoreId	int
StoreName	nvarchar(50)
StateId	int

Employees

Column Name	Data Type
EmployeeId	int
EmployeeName	varchar(50)
ManagerId	int

States

Column Name	Data Type
StateId	int
StateName	nvarchar(50)
Abbreviation	nchar(2)

Sales

Column Name	Data Type
SalesId	int
SalesDate	datetime
EmployeeId	int
StoreId	int

Figure 4-2. *A typical OLTP design*

We could move the state name right into the Stores table, but because both the state name and the state abbreviation are used, we have to move both columns into the Stores table. This means that both the abbreviation and the name redundantly show for each store. Normalization moves the data into their own table and uses an integer value to link the tables together. The integer values are repeated, but they represent less redundancy than repeating both the state name and abbreviation columns.

Table Relationships

Relational databases consist of a collection of related columns. Each set of columns forms a relation otherwise known as a *table*. In Figure 4-2, you can see a relationship line between Sales and Employees. Although the line only connects the tables together, it is not difficult to guess that the relationship is between the Sales.EmployeeId and Employees.EmployeeId columns.

Relationships between the tables are a vital part of any OLTP design, and when you are trying to understand a particular database, understanding these table relationships is also vital. Before we go any further, let's review the relationships between the tables in our example database in Figure 4-2.

Many-to-Many Tables

Let's take a look at the many-to-many relationships within our example. One is the relationship between sales and titles using three tables: Titles, SalesLineItems, and Sales. This relationship dictates that one title can be on many sales and one sale can have many titles. The SalesLineItems table contains both SalesId and TitleId, making this a bridge or junction table. These bridge tables are also called *associative entities*. Whatever you choose to call them, they provide the link between tables with a many-to-many relationship, like Sales and Titles.

Another example of a many-to-many relationship is the link documented by the TitleAuthors table. This table defines that one author can write many titles and one title can be written by many authors.

One-to-Many Tables

The Stores table has a one-to-many relationship with the Sales table. The relationship declares that one store can have many sales, but each sale is associated with only one store.

A similar one-to-many relationship is found between the Sales and Employees tables. This relationship describes that one sale is associated with a single employee, but one employee can be associated with many sales.

Parent–Child One-to-Many Tables

Now, let's take a look at an additional relationship defined in the Employees table. This relationship is bound to itself in the form of employees and managers. One manager can have many employees, but one employee has only one manager, at least as defined in this database.

A Managers table could have been created instead of defining the relationship in the table itself. In doing so, however, the relationship would have a single level between a manager and an employee. As it stands now, the relationship in the Employees table can take on many levels. By that we mean an employee could report to the manager, but that manager is also an employee who could report to another manager.

The relationship of employees to managers forms a jagged (sometimes called *ragged*) hierarchy. The chain between the parent and the child may consist of one level, two levels, or many levels. The fact that the number of levels is unknown is what exemplifies the pattern of parent–child relationships, not unlike how some human children may provide a parent with grandchildren, but others may not. Those grandchildren in turn may or may not have children of their own.

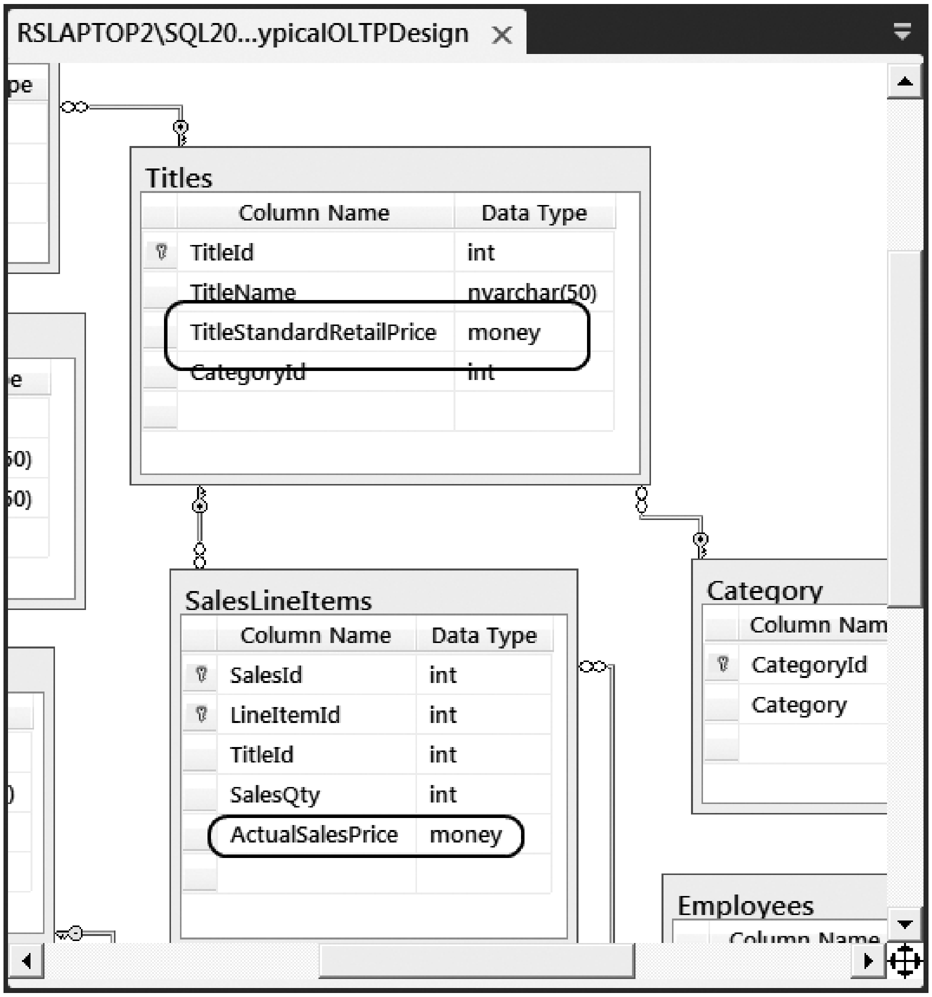
A Typical Data Warehouse Database Design

The design of the data warehouse is similar to the OLTP you just reviewed, but its focus is different. Instead of being concerned about normalization and the lack of redundancy, the focus is on report performance and simplicity. A data warehouse should provide your users with a simple, high-performance repository of report data. It should be easy to understand and consist of a minimal set of tables wherever possible.

To convert an OLTP design into an OLAP (data warehouse) design, start by identifying what reporting data is available. Using the bottom-up approach associated with Kimball's method, focus on a particular process, such as sales, and start building from there. It is important to try to make things very consistent so that additional processes can be added later, in what Kimball refers to as a *bus architecture*.

Measures

Measures are the aspects of your data warehouse that are reported on, such as if a client were to say, "I want a report on how many titles we have sold." After taking a look at the OLTP database in Figure 4-3, you know that the



Titles

Column Name	Data Type
TitleId	int
TitleName	nvarchar(50)
TitleStandardRetailPrice	money
CategoryId	int

SalesLineItems

Column Name	Data Type
SalesId	int
LineItemId	int
TitleId	int
SalesQty	int
ActualSalesPrice	money

Category

Column Nam
CategoryId
Category

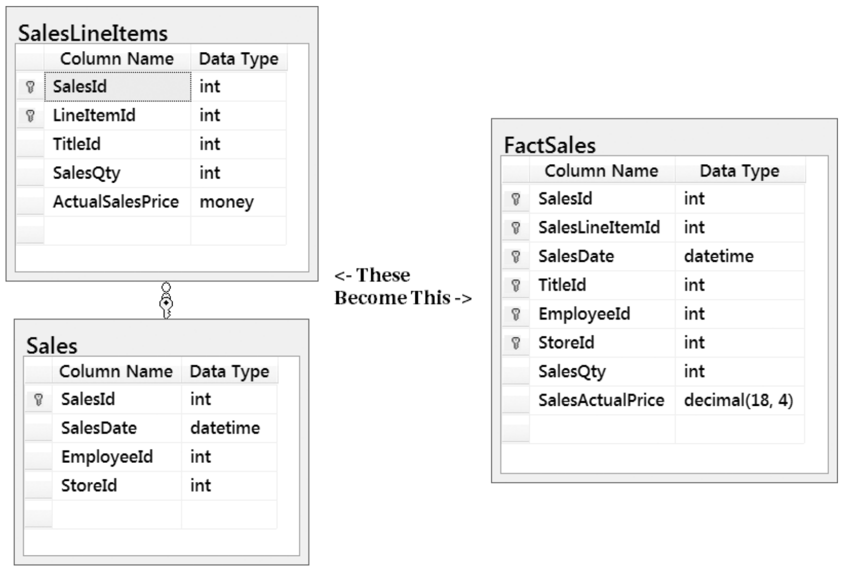
Employees

Figure 4-3. *Selecting measure candidates*

data related to the sales quantity will be required to answer this question. The *measure* provides information on a given process such as the process of selling something. The measure value, such as 15 items sold, for example, provides information about a specific event within that process.

Often when one question is asked, another question is suggested, such as, “How much was the standard retail price for that title?” Another question might be, “How much were the total sales for that title on that specific sale?” For the first question, the answer could be obtained by using the actual sales price, but for the second question, you would need to multiply the actual sales price by the sales quantity taken from specific dates to achieve a new value.

Looking at the Titles table, you see that there is a standard retail price (Figure 4-4). It would be tempting to assume that this was a measure as well. It could be used as a measure at some point in time by performing a calculation. But that might not be a mainstream calculation. Instead, it may be applicable to only a few reports.



SalesLineItems

Column Name	Data Type
SalesId	int
LineItemId	int
TitleId	int
SalesQty	int
ActualSalesPrice	money

Sales

Column Name	Data Type
SalesId	int
SalesDate	datetime
EmployeeId	int
StoreId	int

FactSales

Column Name	Data Type
SalesId	int
SalesLineItemId	int
SalesDate	datetime
TitleId	int
EmployeeId	int
StoreId	int
SalesQty	int
SalesActualPrice	decimal(18, 4)

Figure 4-4. *A typical fact table*

To understand this, take a closer look at the StandardRetailPrice column. It defines information about the title and not about an actual sale. Therefore, although the ActualSalesPrice column describes something that occurred as part of a sales event, the StandardRetailPrice column describes an additional attribute of an individual title. In essence, this means that the standard retail price is more descriptive of a title, and not something that would be moved into the measures category directly. Any items that provide additional descriptions of a process are dimensional attributes and not measures.

Granularity

Measure values typically represent the lowest level of detail tracked by a given process. For example, you could store only a grand total for the sales of a given day by categories of products, but more likely you would instead record the individual sales event throughout the day and by the individual product itself and not solely by a product’s category.

When you create a data warehouse, you can choose the level of detail you would like to report against. If you want to report only on the daily totals, you can do so by aggregating the values you stored in the OLTP data as they are turned into measures. Most of the time you will use the lowest level of detail you can obtain because it gives you the maximum amount of reporting options. Your chosen lowest level of detail defines the *granularity* of that measure.

The Fact Table

After you have defined your measures, you can begin designing your fact table. This fact table may look remarkably like the OLTP table where the measures were found, or it may combine data from multiple tables, which in turn denormalizes the data.

An example of this process would be collapsing the SalesLineItems table and the Sales table into one fact table. The SalesLineItems table defines a many-to-many relationship between sales and titles, but that relationship will still exist even if you collapse the SalesLineItems into the Sales table. When you do this, you end up with a single table with redundant values, as displayed in Figure 4-4. This denormalized design is typical in data warehouses. The SalesId will be repeated multiple times for each line item, but it is considered a small price to pay for simplification, and this pattern is representative of all fact tables.

Another common feature of the fact table is the simplification of datatypes. The actual sales price may have originally been recorded as a SQL Server money datatype, yet the fact table would represent this data as decimal (18, 4), as you see here (i.e., a total of 18 numbers with four of those numbers after the decimal point). The reason for this change is that SQL Server's money type is a custom datatype associated with Microsoft's SQL Server. Money is not an industry-standard datatype, but the decimal datatype is indeed an industry standard. The money datatype can be accurately represented by this decimal datatype; thus, it is logical to do so. Applications that use the data warehouse are more likely to work correctly using a standard datatype than a custom datatype.

In addition to the measures, the fact table contains IDs, or keys, that connect to dimension tables. These columns are referred to as *dimensional keys*. The dimensional keys may be listed at the front of the column list or placed at the end—it does not matter. We have chosen to represent them at the beginning of the column list, but it is simply a matter of preference.

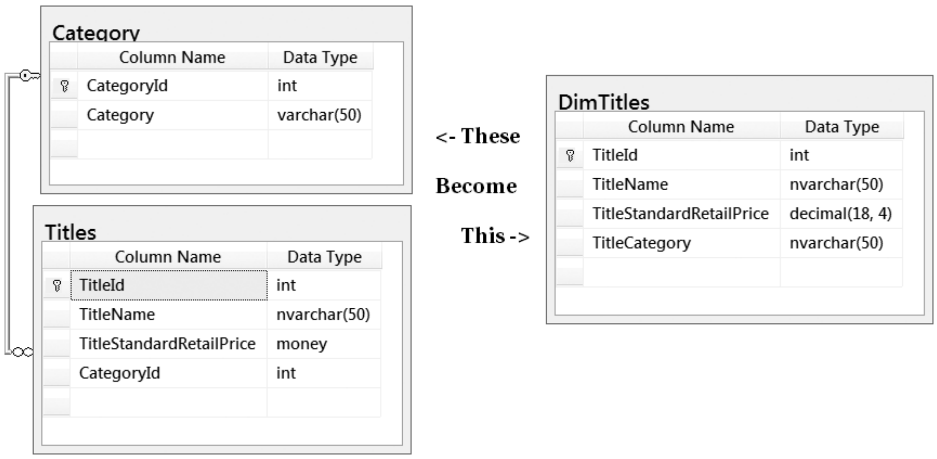
Another preference is how you name a fact table. Some developers might choose the name Sales. Others would prefer the name SalesFacts. However, we chose the name FactSales for our fact table. Placing its designation as a fact table at the beginning of the table name will help organize our tables alphabetically. Our convention does not have much significance other than that it just makes it easier to group tables together in some applications. For instance, SQL Server's Management Studio will sort tables alphabetically in its Object Explorer window.

Dimensions

Once the fact table is defined, it is time to turn your attention to describing your measures with dimensions. At a minimum, each dimension table should contain a dimensional key column and a dimensional name column. The dimensional key column will typically be something such as ProductId or CustomerId. The name column provides a human-friendly description of that particular ID. Along with the name, this ID column may need additional descriptive data to allow for better organization and a clearer understanding, for example, the name of its category.

In Figure 4-5, we have created a Titles dimension table with four dimensional attributes (TitleId, TitleName, TitleStandardRetailPrice and TitleCategory). The TitleId is the dimensional key, but all four are dimensional attributes.

In Figure 4-5, note that the money datatype is now translated into a decimal for the same reasons we described with regard to the SalesActualPrice measure. We have also collapsed the data from two tables into one table by taking the category data from the Category table and moving into the TitleCategory column into the DimTitles table. In addition, the original Category table used a varchar datatype, but the new dimensional table uses an nvarchar, or Unicode variable character datatype. Unicode has now become the standard for most modern applications, and although it takes up twice the size in bytes to store this datatype, it is more consistent with the datatypes in the other columns within the database. Simplicity is often more important to the data warehouse design than absolute efficiency, and giving all columns consistent datatypes as well as consistent sizes is, well, simpler.



Category

Column Name	Data Type
CategoryId	int
Category	varchar(50)

Titles

Column Name	Data Type
TitleId	int
TitleName	nvarchar(50)
TitleStandardRetailPrice	money
CategoryId	int

<- These

Become

This ->

DimTitles

Column Name	Data Type
TitleId	int
TitleName	nvarchar(50)
TitleStandardRetailPrice	decimal(18, 4)
TitleCategory	nvarchar(50)

Figure 4-5. *A typical dimension table*

We realize that some may not agree with this decision, so let's examine this idea. It is true that performance might be enhanced using a smaller datatype, but if it can be determined that the Titles table will have only a few hundred rows, there will be little to no measurable decrease in performance. If, however, there are millions of titles, then of course you will want to change to the smaller datatype to increase performance.

Design your tables around the philosophy that simplicity is more important than pure efficiency and performance is more important than total simplicity. That is to say, if you can make it simple and not adversely affect performance, keep it simple! If, on the other hand, simplifying things decreases your performance to a noticeable degree, ignore your simplification efforts for this occurrence. Use common sense, and evaluate your needs on a case-by-case basis. It is possible to get too caught up in defending one style over another when often there is little impact either way.

Stars and Snowflakes

At one end of a lunch table you may hear the simplicity versus performance argument, and at the other the stars versus snowflake argument. Ignoring the fact that you need to find a more exciting place to eat lunch, let's examine the difference.

Star and snowflake designs reference a pattern form between dimension tables when compared to a fact table. A better name for them, however, would have been single-table dimensions and multitable dimensions; let us explain.

In Figure 4-6, you see a star design that forms a ring of dimension tables around a centralized fact table. In the star design there is only a single circle of dimension tables around the fact table of a data mart. Whether there are three, four or a hundred dimensions in the data mart, it makes no difference; as long as there is only a single table for each of these dimensions, this pattern still forms what is known as a star design.

Dimensions containing multiple tables per dimension are snowflake designs. Snowflakes form a circle of two or more tiers of dimensional tables around the fact table of a data mart. Whether it forms a circle of three, four or hundreds of tiers, it still is a snowflake design. Figure 4-6 outlines the pattern of a snowflake design compared to a star design.

If you are thinking, "Really? Is that all there is to it?" you are not alone. Randal has had many a student enroll in his classes specifically to learn the difference between a star and a snowflake design. How anticlimactic to discover the answer is so simple!

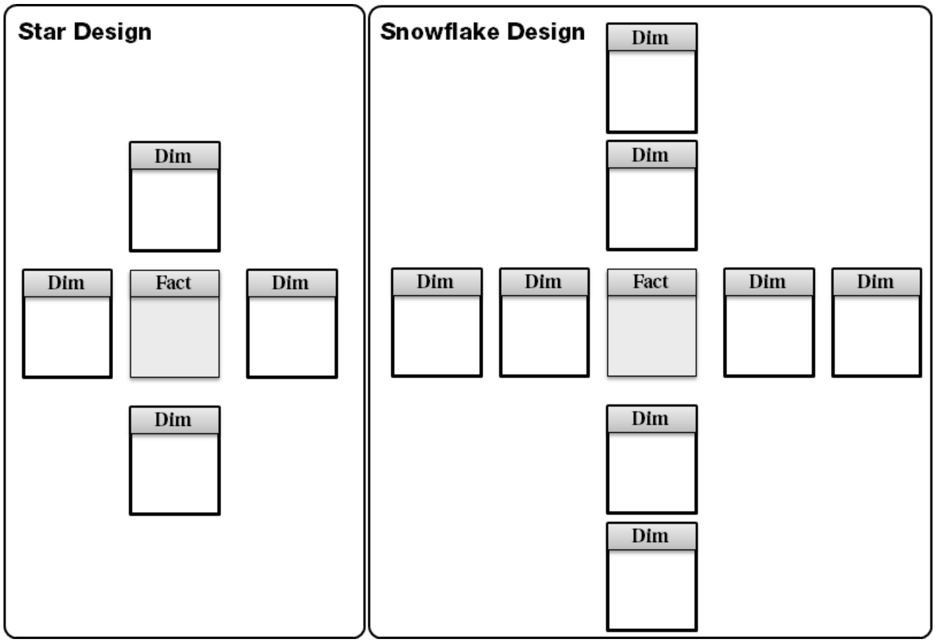


Figure 4-6. *Star and snowflake design patterns*

Keep in mind that when you are designing your tables, they may not necessarily display in a circular pattern as they do here, but it does not matter. The terms *snowflake* and *star* are simply descriptive words that portray how they are connected. Additionally, a single data warehouse can contain both design patterns.

Next, let's take a look at scenarios where one design pattern may be a better choice than the other.

Performance Considerations

In most cases, a set of tables can simply be collapsed into a single star dimension table. One argument in favor of the star design is that single tables are simpler to work with, even if they decrease storage efficiency by containing redundant values. The opposite argument is that by reducing redundancy, you increase storage efficiency; therefore, using separate tables by normalizing them is a better choice. Kimball and Inmon disagree on this subject as well, with Kimball siding with the single table star design and Inmon taking the snowflake stance.

Keep in mind that performance is relative to action, and the performance related to the action between the functionality of stars versus snowflakes is no exception. If you were to create a report based on star-designed tables, you would have fewer tables to connect to in your SQL joins. However, the redundancy of data in the tables means that there is more to transfer from the hard drive into memory before your query's results are assembled. The action of SQL Server finding and linking two tables decreases performance, but the hard drive I/O performance is lowered because of the reduction of redundancy data. From this single example, you can see that performance considerations are not as straightforward as they may appear at first.

In general, the simplified joins of a star design give the best performance for smaller tables (few columns and thousands of rows), whereas the snowflake design has better performance for larger tables (many columns and millions of rows).

■ **Tip** Remember that whether you choose to use a star or snowflake design, the data in your report is exactly the same. Even if you do not make the "best" choice, your design will still work.

In summary, it is more efficient to store data in a snowflake design, but it is more convenient to store it in a star design. Many data warehouses end up as a hybrid with some of the dimension tables designed in the star design pattern and other dimension tables designed in the snowflake design pattern. Nothing says you can't have both!

If you still can't decide which to use, follow this advice: when possible, use the star design for simplicity. If you come across a circumstance where you need to reduce redundancy, change the design to a snowflake. In the end both are simply tools that will enable you to get the job done, so choose the tool that is appropriate for the job.

Comparing Designs

Figure 4-7 compares the tables used in a star design and a snowflake design. If you design the Stores and States tables according to the star design philosophy, you would take the StateName and Abbreviation columns from the States table and collapse them into the Stores table (as shown in DimStores_StarVersion). Conversely, if you design the same two tables according to the snowflake philosophy, you leave it in two separate tables (as shown in the DimStores_SnowflakeVersion and the DimStates_SnowflakeVersion).

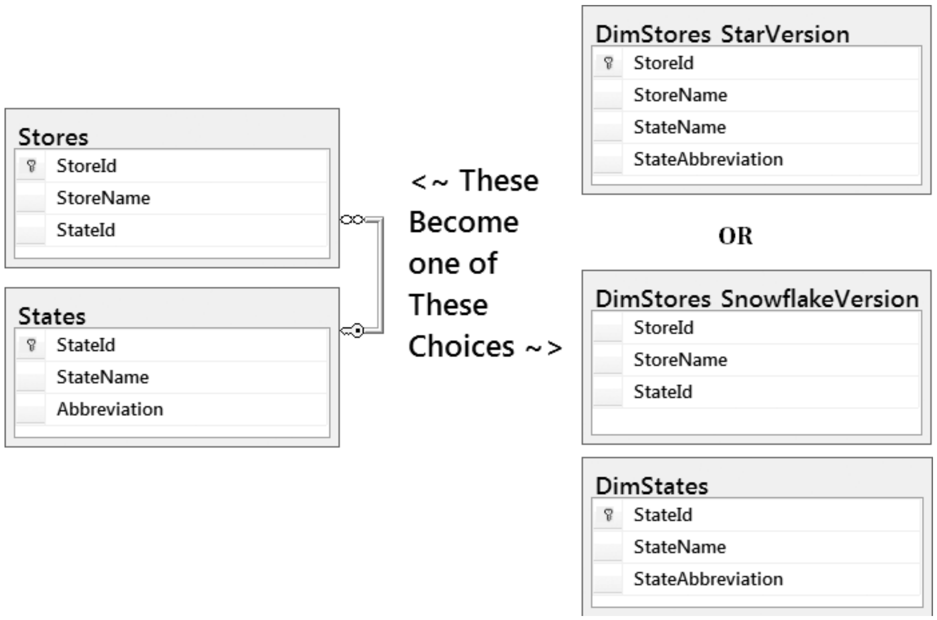


Figure 4-7. *Star versus snowflake designs*

■ **Tip** For learning purposes, we use a hybrid approach in our exercises, to provide experience with both. This means that some dimensions are designed as a snowflake and others as a star.

Notice that the Abbreviation from the States table has been changed to StateAbbreviation in the new examples. This is one example of how this change adds clarification and allows for a more workable and detailed table. Another change is that the StateId from the States table is no longer needed in the star design, as the need for that particular key is removed.

When you compare the original OLTP tables (Figure 4-8) to the OLAP tables in Figure 4-9, notice that they are similar but not exactly the same. The names are changed to reflect their usage, and some of the tables are combined into a single table. The FactSales table has a composite primary key on all the dimension key columns (with one exception, AuthorId). The FactTitleAuthors contains only two primary keys, AuthorId, and TitleId, forming a bridge table just as in the OLTP design.

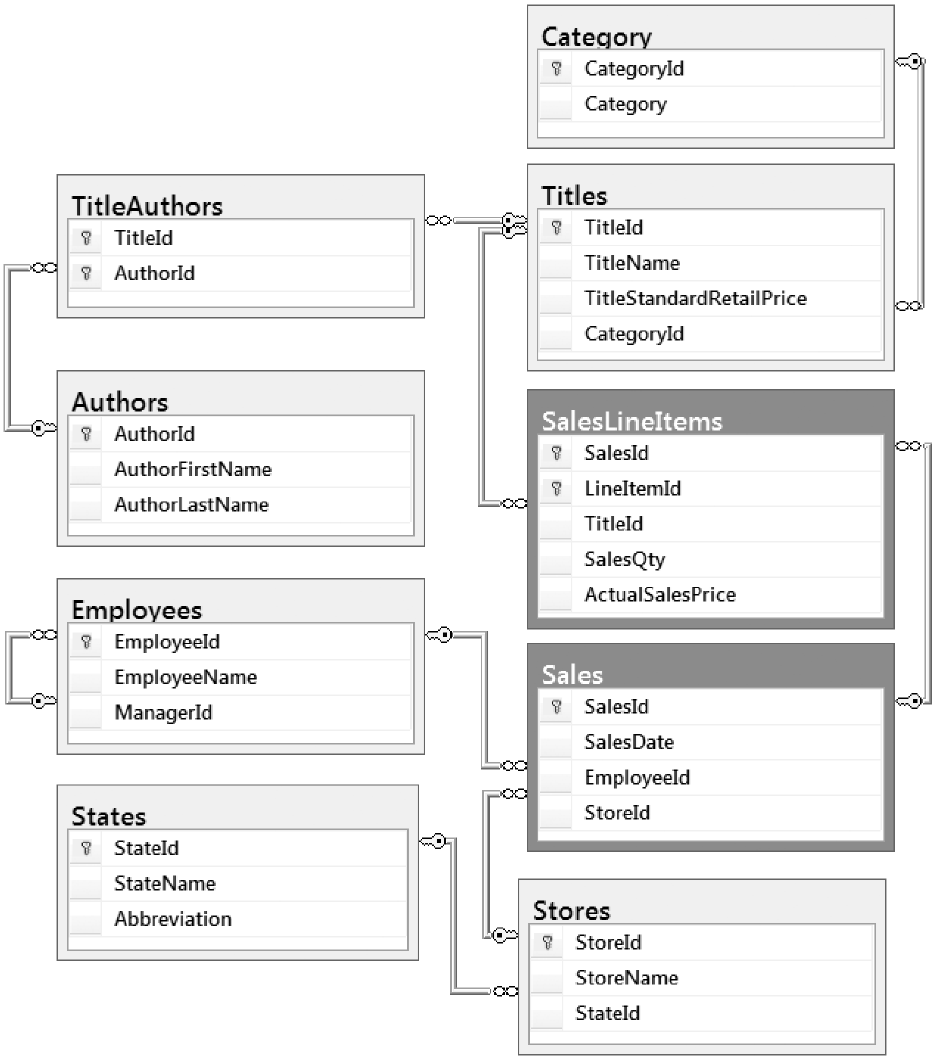


Figure 4-8. *The OLTP design*

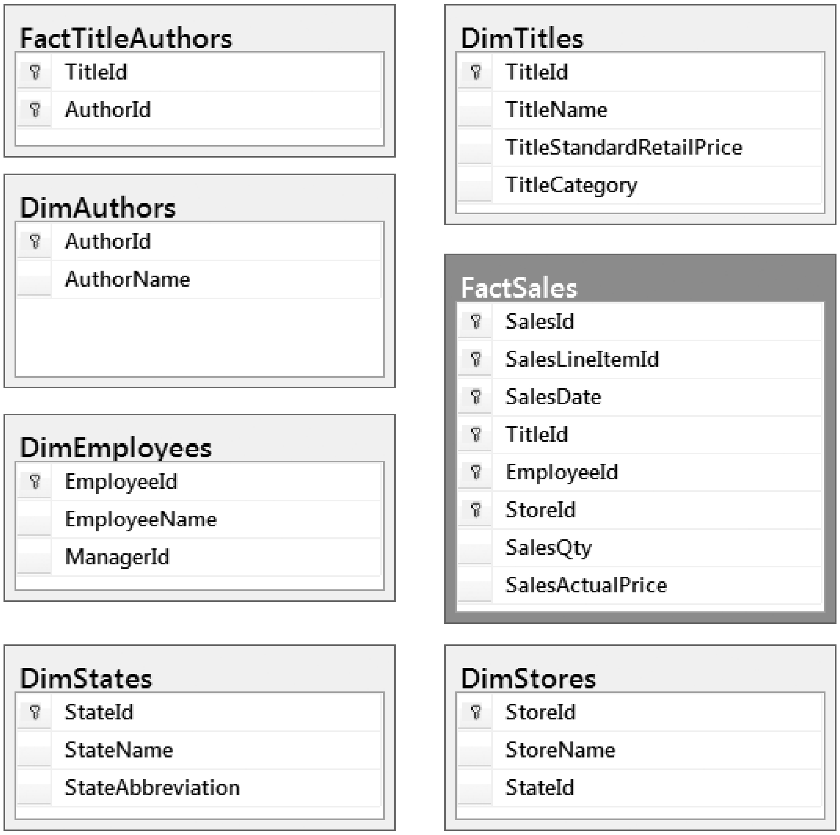


Figure 4-9. *The OLAP design*

Foreign Keys

Foreign key relations exist between the tables, but the lack of lines connecting the columns indicate that we have not placed a foreign key constraint on the tables. This can cause confusion, so let's elaborate on this feature.

A foreign key *column* is a single column where the data represents values from another table. A foreign key *constraint* stops you from putting data in a foreign key column that does not exist in the original column you are referencing. Many developers who do not work with database development on a daily basis get the two terms confused. Most database administrators would shudder at the idea of not putting foreign key constraints in an OLTP database, yet it is not nearly as common to include them in an OLAP database.

The argument against putting them in the OLAP database is that the data has already been validated in the original OLTP database, so why validate it again? The argument for placing foreign key constraints in the OLAP database is that it is considered cheap insurance. It is a type of fail-safe. If someone imports data that is somehow incorrect, the foreign key constraints will catch it as an error. We prefer using foreign key constraints in both styles of databases, but we are not doing so for this example to provide additional contrast for learning purposes.

Missing Features

Foreign key constraints are a common feature of most databases. The example in Figure 4-9 is also missing some other common features. These features include several different types of dimensions, such as a time dimension and a parent–child dimension. Although it is common for data warehouses to have some, but not all possible features, it is good to understand these other design options so that you can include them when they are required. To help with that, let's look at common dimensional patterns.

Dimensional Patterns

Dimensional patterns are different ways of creating tables to hold your dimensional data. Using the right pattern for the right set of dimensional attributes is important. For example, if you incorrectly choose a standard dimensional pattern for dealing with a many-to-many design, your analysis server cubes will come up with incorrect data. The good news is that once you have seen the patterns, they are pretty easy to recognize.

Standard Dimensions

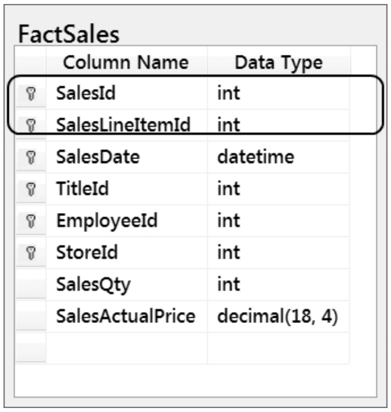
A standard dimension is a collection of one or more tables linked directly to the primary fact table (FactSales in Figure 4-9). The standard dimension is the one that you see most often, which is why it is called *standard*. Each standard dimension table should have a key column and a name column. In addition to those two columns, you can provide additional descriptive values that help further categorize the data.

In summary, in Figure 4-9, the fact table is the FactSales table. The DimTitles table is an example of a standard dimension, as are the snowflake tables, DimStores, and DimStates. These three tables represent two dimensions (Titles and Stores), and although one is in a star design and the other is in a snowflake design, they are both still considered standard dimensions.

Fact or Degenerate Dimensions

Fact dimensions (aka degenerate dimensions) have all their attributes stored in the fact table. They are most commonly referred to as *fact dimensions*, because *degenerate dimension* is not as descriptive and it sounds, well… degenerate." The names are synonymous, but for the purpose of this text, we refer to them as fact dimensions.

In a fact dimension with two dimensional attributes, both would be stored in the fact table. A classic example of this is the SalesId and the SalesLineItemId, as shown in Figure 4-10. These columns are not measures; they represent additional descriptions of the measures, and they do not link to any dimensional tables. This, by definition, makes them part of a fact dimension we call DimSales.


FactSales

Column Name	Data Type
SalesId	int
SalesLineItemId	int
SalesDate	datetime
TitleId	int
EmployeeId	int
StoreId	int
SalesQty	int
SalesActualPrice	decimal(18, 4)

Figure 4-10. *Fact dimenisons*

We could create a DimSales table and put them both in it, but there is really not much point. Leaving the SalesId and SalesLineItemId in the fact table is more straightforward for reporting, and changing this would complicate the design. Besides, this new DimSales table will effectively have a one-to-one relationship between itself and the FactSales table.

Another fact dimension column is the SalesDate. From it, we can create a time dimension. You will almost always want a time dimension for every data warehouse. Evaluating how something changes over time is one of the most common types of reporting.

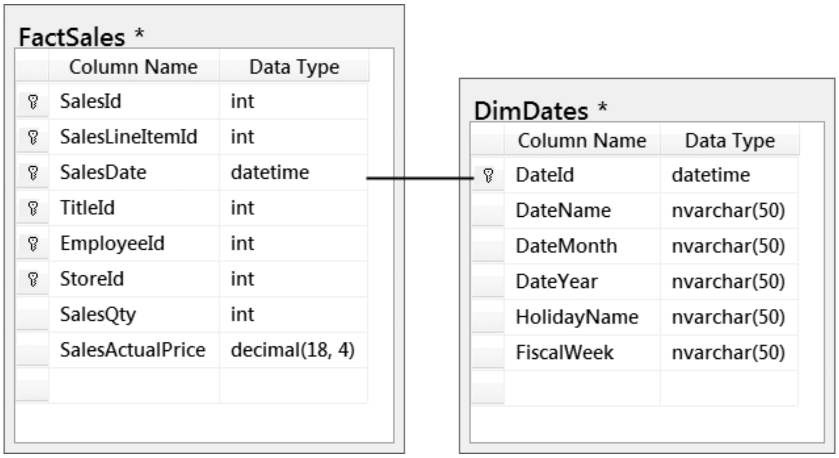
When you create time-based reports, chances are that you will want more than just a list of dates. It is likely you will want to categorize those dates into months, quarters, and years. Each of these items is an additional dimensional attribute that could be stored side by side with the date. In addition, you can extract all of these dimensional attributes from the one date by performing calculations on the data itself. Because all the dimensional attributes (either stored or calculated) are in the FactSales table, this means that, by definition, the time dimension is currently designed as a fact dimension.

Time Dimensions

Designing a time dimension as a fact dimension works but is not considered the best practice. Instead, you should create a date dimension table. A table called DimDates or DimTime is one of the most common features of every data warehouse.

At a minimum, a date table includes a date key and a date name, but it also includes other dimensional attributes such as the month, quarter, and year. These are basic date values and are easily calculated from individual dates in the fact table. But what if you want to include additional attributes such as holidays, corporate events, or fiscal weeks? These are not easy to calculate from a single date value. Creating a separate date dimension table that holds this information makes creating reports that include holidays or fiscal weeks, easy.

Consider this: Figure 4-11 shows a typical date dimension table linked to the fact table by a date ID. If you were to leave all of the additional attributes from the DimDates table in the fact table, it would dramatically increase the size of each row! Because the fact table commonly has thousands if not millions of rows, this has a big impact on the data warehouse.



FactSales *

	Column Name	Data Type
🔑	SalesId	int
🔑	SalesLineItemId	int
🔑	SalesDate	datetime
🔑	TitleId	int
🔑	EmployeeId	int
🔑	StoreId	int
	SalesQty	int
	SalesActualPrice	decimal(18, 4)

DimDates *

	Column Name	Data Type
🔑	DateId	datetime
	DateName	nvarchar(50)
	DateMonth	nvarchar(50)
	DateYear	nvarchar(50)
	HolidayName	nvarchar(50)
	FiscalWeek	nvarchar(50)

Figure 4-11. *Date or time dimension table*

Tracking Dates and Times

It should be noted that there have been debates about whether these tables should be called DimTime instead of DimDate. Arguments for calling your table DimDate and not DimTime usually revolve around the question, "What if I may want to have a separate table for tracking hours, minutes and seconds; wouldn't that table be called DimTimes?" It sounds like a good argument, but let us examine it further.

Currently, the lowest level of detail in the DimDates table of Figure 4-11 is an individual day. Therefore, we have 365 rows for each year of dates, or at least, for three out of four years we will. Ten years of dates can be stored in a table with less than 4,000 rows, which in database terms is not that large a table.

However, what would happen if we added in hours, minutes and seconds? The size of the table would swell to more than 31 million rows! This many rows are very likely to cause problems with performance. So, what are our options?

One option would be to create a separate time dimension table that held hours, minutes, and seconds columns. You would then link this table to the fact table just as you would the date dimension table. This can work, but there is a simpler option.

Using DateTime Keys

In this second option, leave the date and time data just as it was in the OLTP design and then link the DimDate table to the fact table based in this datetime column, instead of using an integer column (Figure 4-11).

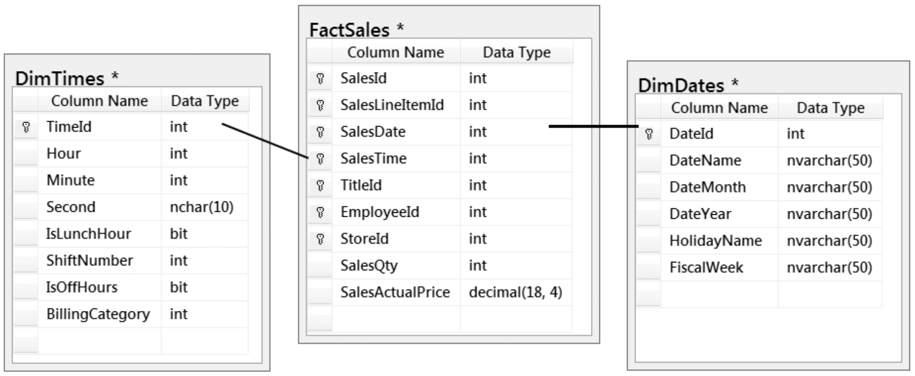
You can derive hours, minutes, and seconds quite easily from a datetime column. Because in most cases hours, minutes, and seconds do not have additional descriptors associated with them, a simple datetime column is all you need. You can then include a datetime column in your DimDate table to provide connectivity between the dimension and fact tables (Figure 4-11).

You still have additional associations for holidays. Using this design, it is easy to store time dimension values down to the milliseconds without influencing the fact table size.

Having It All

There are times where having just a datetime column might not work for you. Consider requiring additional descriptors for hours, such as lunchtime or second shift. One way of handling these descriptors is to include them in the fact table along with the datetime column. You now have dates in a dimension table, times in the fact table, and an hour description in the fact table as well. This hour description column will have considerable redundancy, but it is better than having 31 million rows in your DimDate dimension table.

Of course, if you get too many descriptors, this still will not work, and you need to consider using a separate time dimension table, as shown in Figure 4-12.



DimTimes *

Column Name	Data Type
TimeId	int
Hour	int
Minute	int
Second	nchar(10)
IsLunchHour	bit
ShiftNumber	int
IsOffHours	bit
BillingCategory	int

FactSales *

Column Name	Data Type
SalesId	int
SalesLineItemId	int
SalesDate	int
SalesTime	int
TitleId	int
EmployeeId	int
StoreId	int
SalesQty	int
SalesActualPrice	decimal(18, 4)

DimDates *

Column Name	Data Type
DateId	int
DateName	nvarchar(50)
DateMonth	nvarchar(50)
DateYear	nvarchar(50)
HolidayName	nvarchar(50)
FiscalWeek	nvarchar(50)

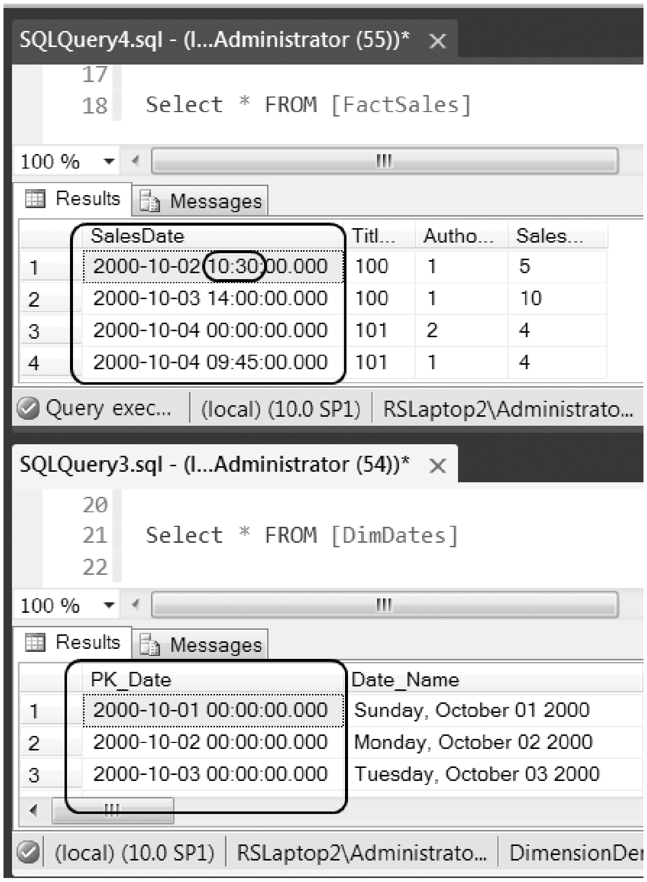
Figure 4-12. *Time and date dimension tables*

■ **Tip** Once again, we recommend keeping your design simple! Most developers find that their reports do not need details about hours or minutes. If you do not regularly need this kind of information, just create a simple date dimension table and use a datetime key to connect it to the fact table. If a rare occasion comes up where you do need more than basic information, you can generate the required results with lookup tables and programming.

Using Foreign Key Constraints

In most dimensions, you want to use an integer value to connect a dimension table to a fact table. This is a practice recommended for almost all occasions, and Kimball defends this practice strongly. However, even Kimball agrees that for the time dimension, using a datetime key instead of an integer key is more practical for tracking periods of hours, minutes, and seconds. Still, there are issues you must consider.

In SQL Server, foreign key constraints compare values to determine whether the constraint is violated. If columns do not have matching values in both tables, an error occurs. Consider the values in Figure 4-13. The values are almost the same but vary by hours and minutes. Because of this, you cannot put a SQL foreign key constraint between these tables.



SQLQuery4.sql - (l...Administrator (55))*

```
Select * FROM [FactSales]
```

	SalesDate	Titl...	Autho...	Sales...
1	2000-10-02 10:30:00.000	100	1	5
2	2000-10-03 14:00:00.000	100	1	10
3	2000-10-04 00:00:00.000	101	2	4
4	2000-10-04 09:45:00.000	101	1	4

Query exec... (local) (10.0 SP1) RSLaptop2\Administrato...

SQLQuery3.sql - (l...Administrator (54))*

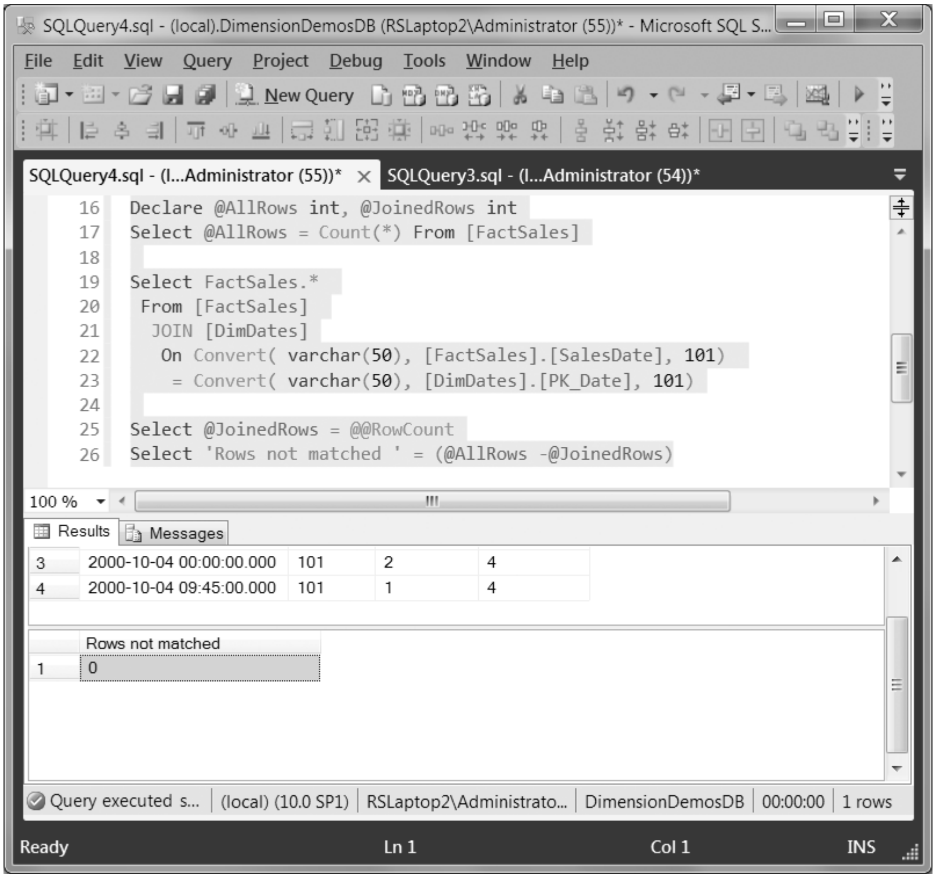
```
Select * FROM [DimDates]
```

	PK_Date	Date_Name
1	2000-10-01 00:00:00.000	Sunday, October 01 2000
2	2000-10-02 00:00:00.000	Monday, October 02 2000
3	2000-10-03 00:00:00.000	Tuesday, October 03 2000

(local) (10.0 SP1) RSLaptop2\Administrato... DimensionDe

Figure 4-13. *Time- and date-based foreign keys*

You can, however, solve this problem programmatically by using validation code that will verify the values match at the level of days. In Figure 4-14 you can see an example of this.


SQLQuery4.sql - (local).DimensionDemosDB (RSLaptop2\Administrator (55))* - Microsoft SQL S...
File Edit View Query Project Debug Tools Window Help
New Query
SQLQuery4.sql - (l...Administrator (55))*
SQLQuery3.sql - (l...Administrator (54))*
100 %
Results
Messages
Query executed s... | (local) (10.0 SP1) | RSLaptop2\Administrato... | DimensionDemosDB | 00:00:00 | 1 rows
Ready Ln 1 Col 1 INS

```
Declare @AllRows int, @JoinedRows int
Select @AllRows = Count(*) From [FactSales]

Select FactSales.*
 From [FactSales]
  JOIN [DimDates]
   On Convert( varchar(50), [FactSales].[SalesDate], 101)
    = Convert( varchar(50), [DimDates].[PK_Date], 101)

Select @JoinedRows = @@RowCount
Select 'Rows not matched ' = (@AllRows -@JoinedRows)
```

```
3   2000-10-04 00:00:00.000   101   2   4
4   2000-10-04 09:45:00.000   101   1   4

    Rows not matched
1   0
```

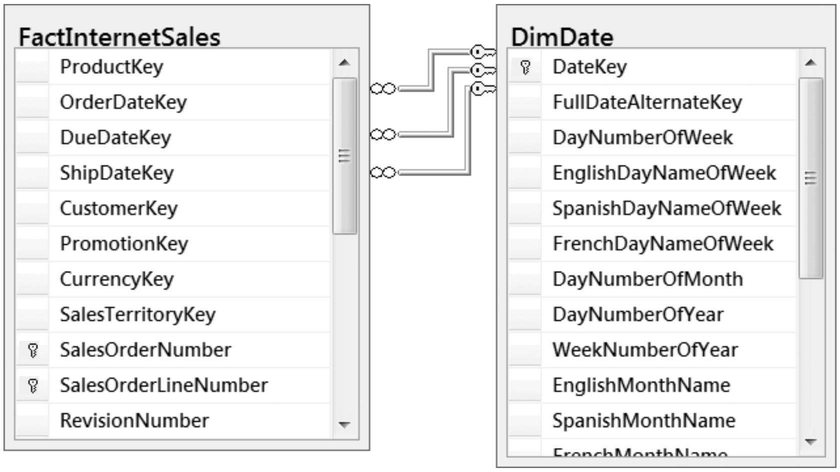
Figure 4-14. *Comparing dates between tables*

We discuss ETL programming further in Chapter 6, but for now let's move on to an example of how a date dimension table can be used in multiple ways.

Role-Playing Dimensions

Role-playing dimension tables are used repeatedly for slightly different purposes or, well, roles. So, the term *role-playing* comes from the fact that the same table plays many roles. The classic example of a role-playing dimension is a time dimension that links to a fact table multiple times. Figure 4-15 shows an example that can be found in Microsoft's `AdventureWorks2008DW` demonstration database.

This may seem to be too simplistic an explanation, but this is really all there is to it. It is possible to have role-playing dimensions that do not involve dates. An example might be geographic regions. But dates are by far the most common example.



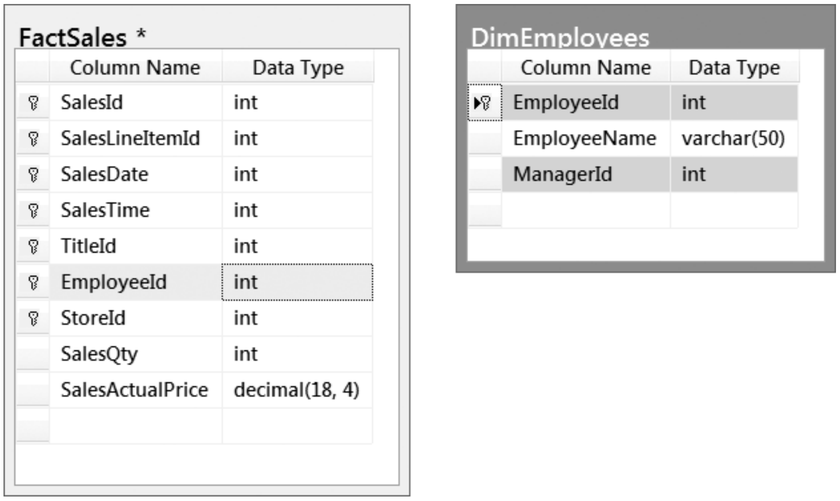
FactInternetSales
- ProductKey
- OrderDateKey
- DueDateKey
- ShipDateKey
- CustomerKey
- PromotionKey
- CurrencyKey
- SalesTerritoryKey
- SalesOrderNumber
- SalesOrderLineNumber
- RevisionNumber

DimDate
- DateKey
- FullDateAlternateKey
- DayNumberOfWeek
- EnglishDayNameOfWeek
- SpanishDayNameOfWeek
- FrenchDayNameOfWeek
- DayNumberOfMonth
- DayNumberOfYear
- WeekNumberOfYear
- EnglishMonthName
- SpanishMonthName
- FrenchMonthName

Figure 4-15. *A role-playing dimension*

Parent–Child Dimensions

Parent–child dimension tables look like their OLTP counterparts. The classic example of a parent–child dimension is an employee table where the relationship between employees and managers is mapped by associating a manager ID (parent) to an employee ID (child). To create a parent–child dimension table in the data warehouse, all you need to do is use the same design as you would in an OLTP environment, much as the one you see in Figure 4-16.



FactSales *

Column Name	Data Type
SalesId	int
SalesLineItemId	int
SalesDate	int
SalesTime	int
TitleId	int
EmployeeId	int
StoreId	int
SalesQty	int
SalesActualPrice	decimal(18, 4)

DimEmployees

Column Name	Data Type
EmployeeId	int
EmployeeName	varchar(50)
ManagerId	int

Figure 4-16. *A parent–child dimension*

Junk Dimensions

You may discover that you end up with a number of dimension tables that hold nothing more than the key column, name column and just a few rows. In a situation where there are a large number of these small tables, your data warehouse can become quite cluttered. One way to simplify things is to create a junk dimension. Although the name sounds a bit silly, a junk dimension can be quite useful.

With a junk dimension, you can bind a bunch of small dimensional tables into a single table. By taking the combination of possible values from these smaller tables and storing the combined values in a single table, you create a junk dimension table.

For example, if the values of one dimension table were "go to lunch" and "go home" and another dimension table had the values "yes" and "no," then the possible combinations would be as follows:

- Go to Lunch, Yes
- Go to Lunch, No
- Go Home, Yes
- Go Home, No

If you build a junk dimension table to hold these values, you will need an ID to link to the fact table and, in this case, two other columns to hold a combination of values. Figure 4-17 shows an example of taking two

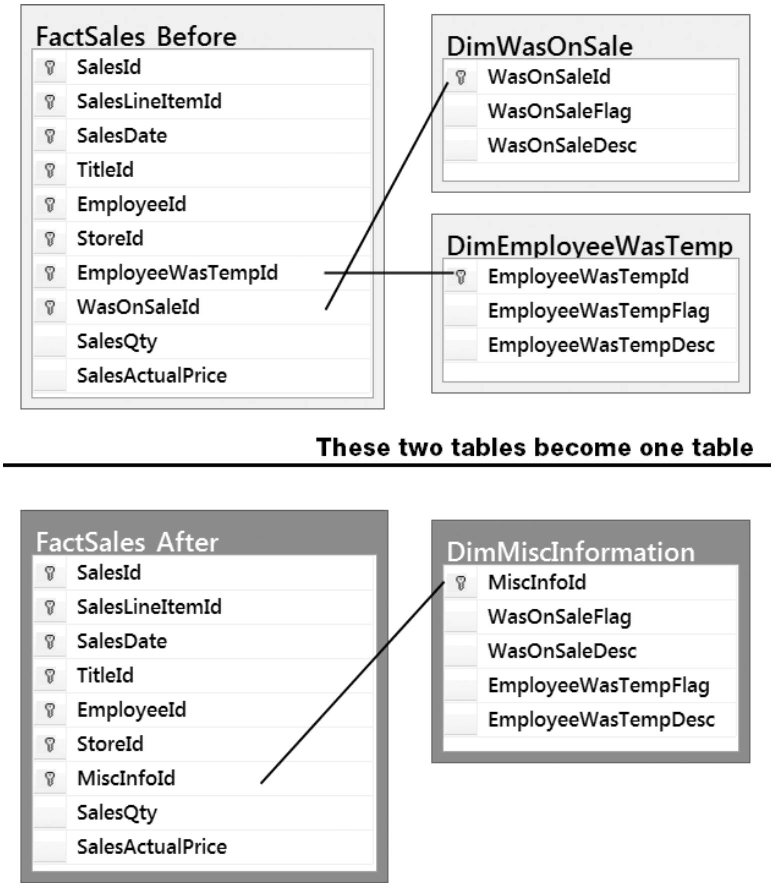


Figure 4-17. *Converting to a junk dimension*

dimensional tables and combining them into a single table. The table DimWasOnSale is combined with the table DimEmployeeWasTemp to form one junk dimension table, DimMiscInformation.

Another option could have been to move all columns in the fact table and create a fact dimension; however, that would add quite a lot of extra data to the fact table. With the current design, you end up with only a single integer linking to a particular combination of values.

Be careful not to go overboard on combining too many dimensions into a large junk dimension. We have seen this happen, and it does not work out well. It is very much like having a miscellaneous folder on your hard drive where you always have trouble finding what you are looking for because it is filled up with such a huge collection of junk.

Many-to-Many Dimensions

A many-to-many dimension is another common component in data warehouses. In these dimensions, you have a set of two tables connected by a bridge table. The bridge table defines the many-to-many relationship.

In Figure 4-18, our FactTitleAuthors bridge table links the DimAuthors and DimTitles together. In order to process the author's information, however, you have to go through the DimTitles table and its associative bridge table.

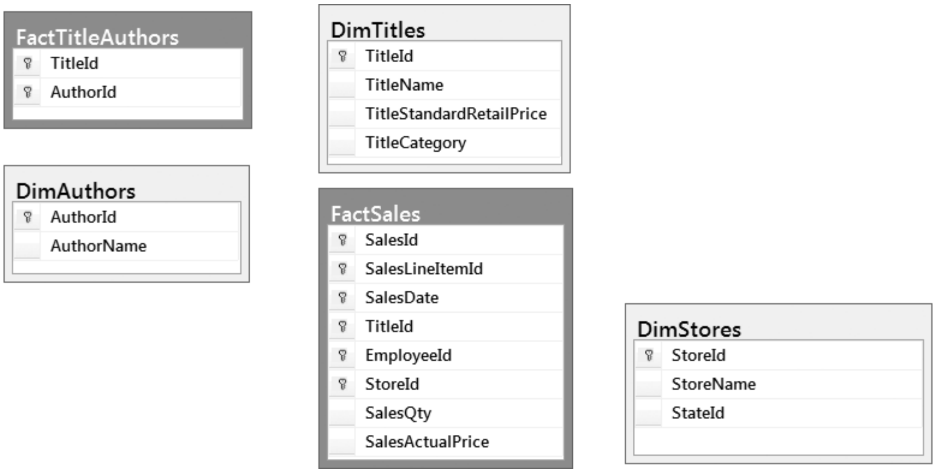


Figure 4-18. *A many-to-many dimension*

DimAuthors is not a standard dimension design because it does not link directly to the primary fact table, FactSales, and is instead linked to the bridge table, FactTitleAuthors.

Fact vs. Bridge Tables

Bridge tables are fact tables, just not the primary fact table of a data mart. A bridge table's purpose is to provide a connection between dimension tables, not store measures, and it is often referred to as a *factless fact table*.

On closer inspection, notice that all fact tables represent a many-to-many relationship between dimension tables; it is inherent in their design. Therefore, in our design (Figure 4-18), one title can sell in many stores, and one store can sell many titles. In this example, the FactSales table bridges DimTitles and DimStores and is the primary table of the data mart.

Still, this does not make both DimTitles and DimStores many-to-many dimensions; instead, they are just two regular dimensions connected to the primary fact table, FactSales. DimAuthors, on the other hand, does not connect directly to the primary fact table. Instead, it connects indirectly through DimTitles. This indirect connection is the hallmark of a many-to-many dimension.

Changing the Connection

It may seem confusing that the Authors dimension does not connect directly to the fact table with the measures, and you may be tempted to fix this by creating a direct connection. But if you try to do so, you will lose the correct many-to-many relationship between DimTitles and DimAuthors.

When you attach the DimAuthors table to the FactSales table, your reports will have issues. Initially this may seem like it works, but once you create an SSAS cube in the data warehouse, you will quickly spot the incorrect values in your cubes.

Wait a second! Didn't we say that all fact tables map a many-to-many relationship? What gives?

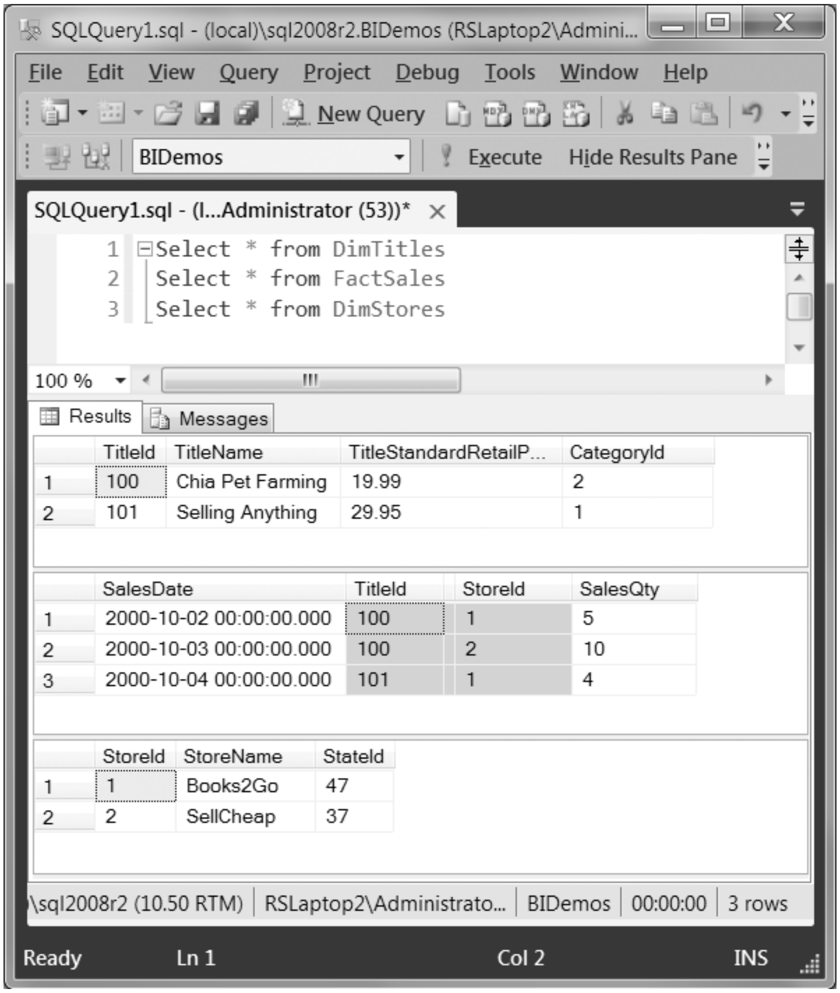
The problem is that there are two types of many-to-many relationships here, at least from the perspective of the primary fact table. We call them direct and indirect many-to-many relationships. Direct many-to-many relationships can be connected directly to the primary fact table, whereas indirect many-to-many relationships must be connected with a bridge table.

The distinction is based on measure granularity in the primary fact table, but to really understand what this means, we need to examine the two designs in more detail.

Direct Many-to-Many Relationships

In the direct design, one row in a dimension table is associated with another row in a different table, but it could be a different row for different events recorded in the primary fact table. A measured value in the fact table is associated with only one row of data from each of the dimension tables.

Let's look at an example. In Figure 4-19, we show the contents of two dimension tables, DimTitles and DimStores, and one fact table called FactSales. The relationship between DimTitles and DimStores is a many-to-many one, because sometimes a title is sold by one store, but on a different sales event, it sells in another store. This means that from the perspective of the sales event, there will be only one store ID for each title ID sold. After all, it makes sense that one part of the book will never be sold in one store and the other part of the book sold in another. So, although one title can sell in many stores and one store can sell many titles, there will never come a time where a single title has more than one store ID for a particular sales event. Therefore, from the position of a FactSales table, a row in the DimTitles table will map to one and only one row in the DimStores table for a given measure (such as the sales quantity), and each row in the fact table will have one title ID and one store ID (Figure 4-19).


SQLQuery1.sql - (local)\sql2008r2.BIDemos (RSLaptop2\Admini...
File Edit View Query Project Debug Tools Window Help
New Query
BIDemos
Execute
Hide Results Pane
SQLQuery1.sql - (I...Administrator (53))*
100 %
Results
Messages
\sql2008r2 (10.50 RTM) | RSLaptop2\Administrato... | BIDemos | 00:00:00 | 3 rows
Ready Ln 1 Col 2 INS

```
Select * from DimTitles
Select * from FactSales
Select * from DimStores
```

	TitleId	TitleName	TitleStandardRetailP...	CategoryId
1	100	Chia Pet Farming	19.99	2
2	101	Selling Anything	29.95	1

	SalesDate	TitleId	StoreId	SalesQty
1	2000-10-02 00:00:00.000	100	1	5
2	2000-10-03 00:00:00.000	100	2	10
3	2000-10-04 00:00:00.000	101	1	4

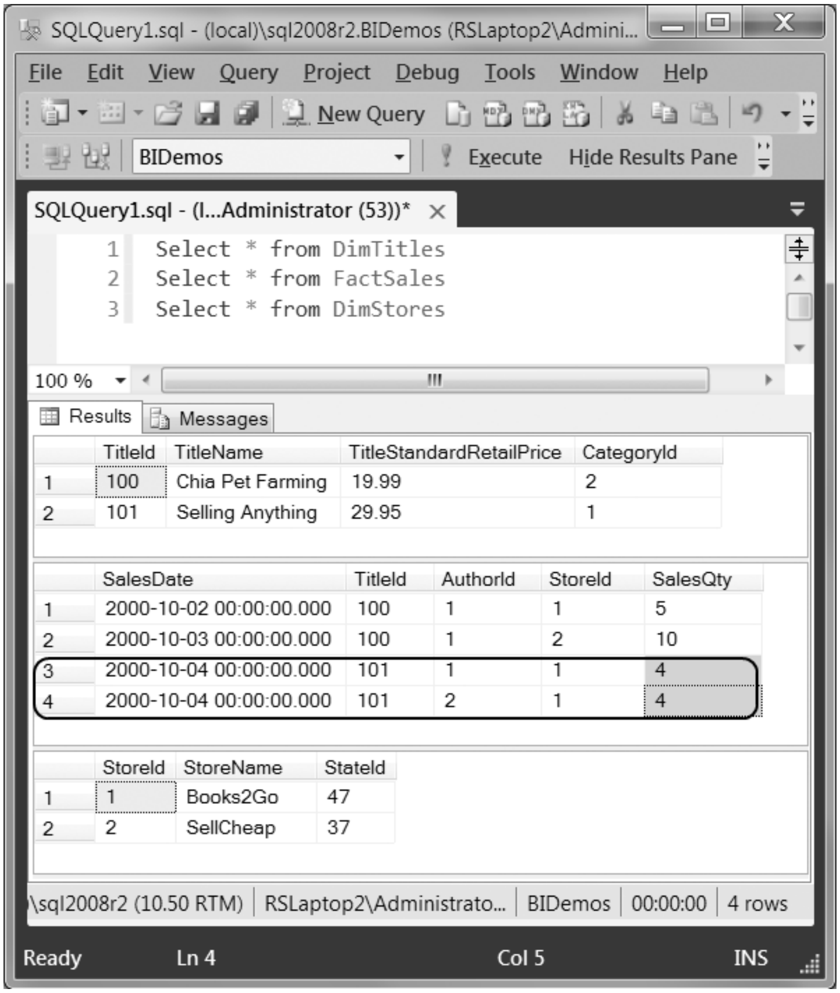
	StoreId	StoreName	StateId
1	1	Books2Go	47
2	2	SellCheap	37

Figure 4-19. *A direct many-to-many dimension*

Indirect Many-to-Many Relationships

In an indirect many-to-many relationship, one row is associated with multiple rows of another table. A measured value in the primary fact table can be associated with one or more rows of data from each of the dimension tables. For example, one title may have only one author, whereas another has many authors. If a particular book has two authors, it will always have two authors regardless of the sales event in the fact table with which we are concerned. Therefore, from the position of a fact table's measures (such as sales quantity), sometimes a quantity will be associated with one author and other times with many authors!

If we try to track this in a fact table, we will need two rows for each title sold, as shown in Figure 4-20. Note, however, that this can lead to a problem known as *double counting*.



SQLQuery1.sql - (local)\sql2008r2.BIDemos (RSLaptop2\Admini...

```
Select * from DimTitles
Select * from FactSales
Select * from DimStores
```

	TitleId	TitleName	TitleStandardRetailPrice	CategoryId
1	100	Chia Pet Farming	19.99	2
2	101	Selling Anything	29.95	1

	SalesDate	TitleId	AuthorId	StoreId	SalesQty
1	2000-10-02 00:00:00.000	100	1	1	5
2	2000-10-03 00:00:00.000	100	1	2	10
3	2000-10-04 00:00:00.000	101	1	1	4
4	2000-10-04 00:00:00.000	101	2	1	4

	StoreId	StoreName	StateId
1	1	Books2Go	47
2	2	SellCheap	37

\sql2008r2 (10.50 RTM) | RSLaptop2\Administrato... | BIDemos | 00:00:00 | 4 rows

Ready Ln 4 Col 5 INS

Figure 4-20. *Double counting with a many-to-many dimension*

On the surface, the fact that there are two rows for this one sales event does not seem too bad. When you make a report showing sales by authors, the data is even correct! There is a problem, however, if you make a report showing sales by titles alone. If you place the AuthorId in the primary fact table, you have to list the sales event multiple times, and title sales will aggregate once for each author. In software such as SSAS that performs aggregations of the sales quantity for a given day, title, and store for you, it will double-count the sale. Oops!

With a report like this, humans can figure out pretty quickly what the issue is. But computers? Well, computers are fast but not smart. You have to configure them around this issue because software, such as SSAS, for example, expects the many-to-many dimensions to use a bridge table.

When you move the AuthorId back out of the primary fact table and access it only indirectly through the bridge table, this problem is solved, and the sales event for that title will still be associated with all the authors who wrote that book.

The Takeaway

OK, no doubt we have made your head hurt with this one! The thing to remember is that many-to-many relationships connect to a fact table either directly or indirectly.

- In the *direct* many-to-many dimension, place both dimension keys from the two dimension tables in the central fact table.
- In the *indirect* many-to-many dimension, you need to have a separate, bridging fact table that is just between the two dimension tables.
- If you design them the wrong way, you will get incorrect values in your reports, so always check your values! If it's wrong, try the other way!

Conformed Dimensions

Conformed dimensions are not really dimensions in and of themselves; it is simply a phrase to describe dimensions that can be used from multiple data marts within a data warehouse. Basically, conformed dimension tables are utilized by many fact tables with each fact table based in a different subject.

For example, if we have a sales table and an inventory table, both of them can use the date dimension table. Since that is the case, the date dimension is a conformed dimension.

This is pretty straightforward, but it is important to be sure the granularity is appropriate. By that we mean, if the inventory is done monthly and the sales are tracked daily, you need a means of linking them based on their individual grain. A simple way of accomplishing this is by including both a DateID and a MonthID in the date table. You often see date dimension tables designed with two columns such as a MonthName column with a MonthID column as well. This can be further built upon by adding a YearID, aWeekID or whatever other increments that may be useful.

Other dimensions can be conformed as well. For example, consider the Titles dimension; in some databases, it is likely that inventory counts and sales will both be associated with titles and this, by definition, means that the titles dimension is also a conformed dimension.

If we continue with this example, the Authors dimension will not be a conformed dimension because it is unlikely that you will be taking inventory on authors.

Adding Surrogate Keys

It is a common practice to add artificial key columns. The idea is that instead of having values that naturally occur in the OLTP environment, you have artificial integer values that make up your data warehouse dimensional keys.

An example of this would be to add a new column to the DimAuthors table as you see in Figure 4-21. The new column, called AuthorKey, will connect to the fact table (not shown here) by replacing the AuthorId column with the new AuthorKey column. The artificial AuthorKey column is referred to as the *surrogate key*, whereas the original AuthorId column is known as the *natural key*.

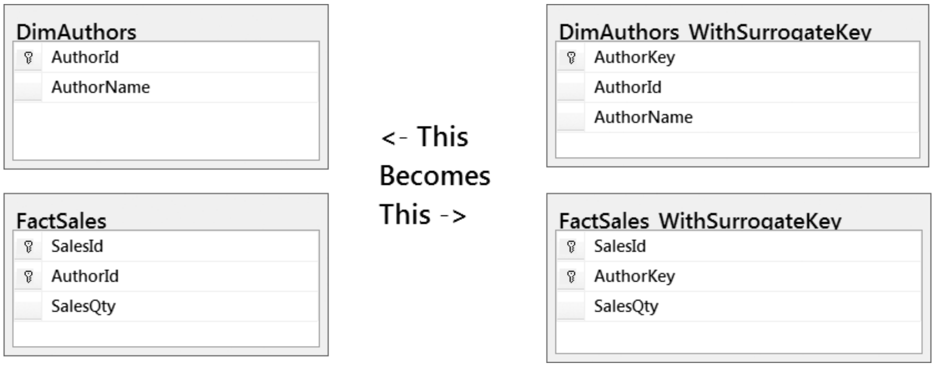


Figure 4-21. *Adding a surrogate key*

It is considered best practice to do this on all tables with the possible exception of the time dimension table, as mentioned previously. The existence of surrogate keys helps when you are merging data into a data warehouse from many different OTLP databases as well as when you are tracking any changes to dimensional values, such as an author changing his or her name.

Slowly Changing Dimensions

Most of the time, dimensional values do not change over time; for example, it is unlikely that the names of the months will change any time soon. Some data does change, however, such as people names. We use the term *slowly changing dimension* (SCD) to describe such data.

When changes like these occur, you may want to track them for reporting purposes. For example, if the reports show sales connected with an old name, the author of those sales may not be given proper credit. It could be that whoever is reviewing the report knows only the new name of that author and has no way of knowing the previous name.

Slowly changing dimensions are certainly not a new concept. There have been several designs created over the years to help handle these types of situations. These designs are categorized into named types. The most common of these named types being used today are types I, II, and III.

Type I

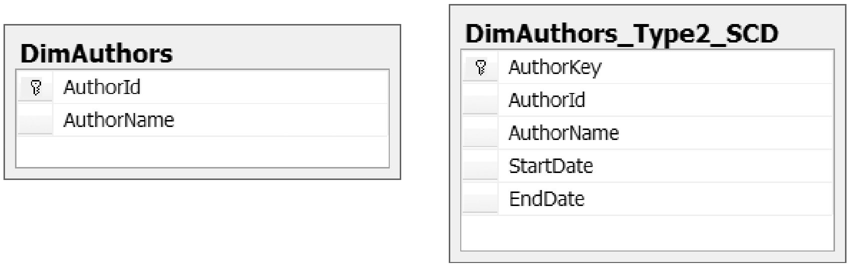
The SCD type I, oddly enough, does not track changes at all. This is the normal state of a dimension table in which no additional columns or rows have been added for the purpose of tracking changes. If an author changes his or her name, simply record the new name and (using your best New York accent), forget about it. All reports now show the new name, and old ones will not be corrected. With type I you are not tracking historical changes, and perhaps that is preferred. The important thing here is that you made a conscious decision instead of letting it be the default behavior of your dimensions.

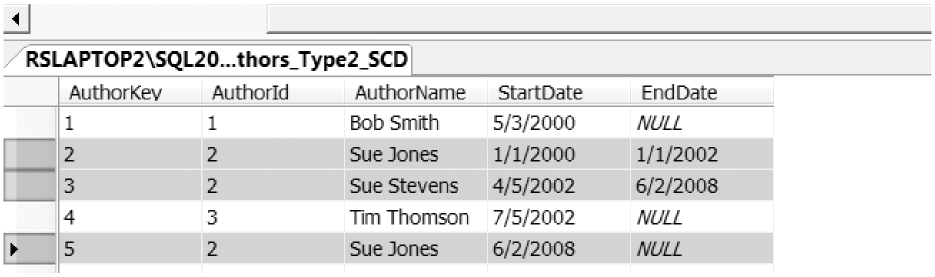
Type II

SCD type II is the complete opposite of type I. Type II tracks all changes rather than no changes at all. To do this, add additional columns and rows to the table specifically for the purpose of tracking the changes.

In Figure 4-22, the Authors table has been modified to include SCD columns. The modifications in this case included adding a column to indicate when a particular author's name was recorded using the start date column and when it ended using the end day column. In this way, every time an author changes his or her name, it can be recorded when the name change occurred and the length of time that it was applicable.

As shown in Figure 4-22, type II dimensions are distinguished by tracking the changes using additional rows. Each change adds a new row to the dimension table.




RSLAPTOP2\SQL20...thors_Type2_SCD

AuthorKey	AuthorId	AuthorName	StartDate	EndDate
1	1	Bob Smith	5/3/2000	NULL
2	2	Sue Jones	1/1/2000	1/1/2002
3	2	Sue Stevens	4/5/2002	6/2/2008
4	3	Tim Thomson	7/5/2002	NULL
5	2	Sue Jones	6/2/2008	NULL

Figure 4-22. *Adding type II slow changing dimension columns*

Type III

In the SCD type III, the process is simplified by tracking only the current and previous values. If an additional change occurs, the current value becomes a previous value, and the value before will be overwritten. Although not as popular as type II, it is simple to implement and may be appropriate on occasion. In Figure 4-23, the column AuthorPreviousName tracks the previous name, and the ChangeDate column tracks the date the change occurred.

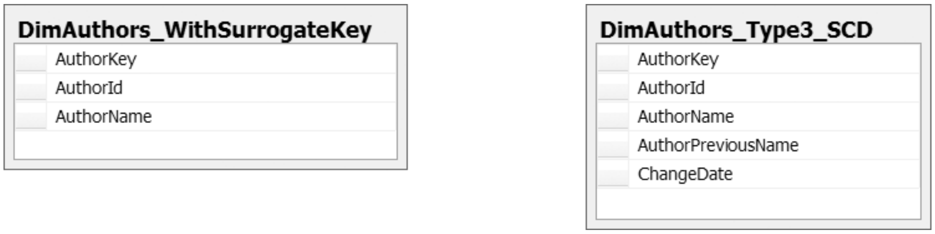


Figure 4-23. *A type III slow-changing dimension*

The distinguishing characteristic of a type III SCD is that changes are tracked by adding columns, not rows. So if you wanted to track the first, second, and third changes, you simply add columns for each version. You might even give each column a name such as AuthorNameVersion2, AuthorNameVersion3, and so on. The number of changes you track are determined by how many columns you add. Typically, though, it is most common in type III to track only one change, which would be the previously used version (AuthorPreviousName in this example).

Now that you have learned about the common dimensional design patterns, it is time to use your knowledge by performing the following exercise.

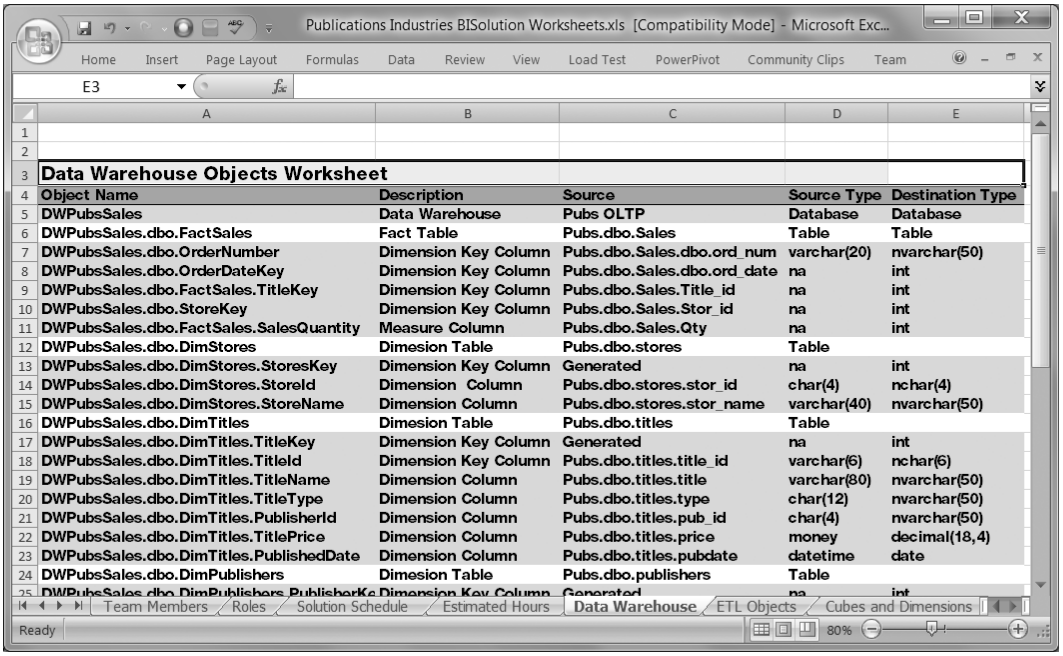
EXERCISE 4-1. REVIEWING THE DWPUBS DATA WAREHOUSE

In this exercise, you review a data warehouse design for the Pubs database using the design created by the authors.

Once again, there is not much to do in this exercise, but we did not see the point in having you fill in the spreadsheet line by line when reviewing and decided it would be just as enlightening and less tedious if we did it for you. However, in the "Learn by Doing" exercise of this chapter, you indeed get this type of practice.

Here are the steps to follow:

1. Open the `PubsBISolutionworksheet.xslx` file found in the downloadable book files.
2. Navigate to the data warehouse worksheet and compare the tables listed to the designs presented in this chapter.
3. Review the columns and datatypes defined in this worksheet comparing the original source datatypes to the destination datatypes. Figure 4-24 is an example of what this worksheet looks like, but the details will be more visible in the worksheet itself. (You create these tables in the next chapter, and details are provided there as well.)



Data Warehouse Objects Worksheet

Object Name	Description	Source	Source Type	Destination Type
DWPubsSales	Data Warehouse	Pubs OLTP	Database	Database
DWPubsSales.dbo.FactSales	Fact Table	Pubs.dbo.Sales	Table	Table
DWPubsSales.dbo.OrderNumber	Dimension Key Column	Pubs.dbo.Sales.dbo.ord_num	varchar(20)	nvarchar(50)
DWPubsSales.dbo.OrderDateKey	Dimension Key Column	Pubs.dbo.Sales.dbo.ord_date	na	int
DWPubsSales.dbo.FactSales.TitleKey	Dimension Key Column	Pubs.dbo.Sales.Title_id	na	int
DWPubsSales.dbo.StoreKey	Dimension Key Column	Pubs.dbo.Sales.Stor_id	na	int
DWPubsSales.dbo.FactSales.SalesQuantity	Measure Column	Pubs.dbo.Sales.Qty	na	int
DWPubsSales.dbo.DimStores	Dimesion Table	Pubs.dbo.stores	Table	
DWPubsSales.dbo.DimStores.StoresKey	Dimension Key Column	Generated	na	int
DWPubsSales.dbo.DimStores.StoreId	Dimension Column	Pubs.dbo.stores.stor_id	char(4)	nchar(4)
DWPubsSales.dbo.DimStores.StoreName	Dimension Column	Pubs.dbo.stores.stor_name	varchar(40)	nvarchar(50)
DWPubsSales.dbo.DimTitles	Dimesion Table	Pubs.dbo.titles	Table	
DWPubsSales.dbo.DimTitles.TitleKey	Dimension Key Column	Generated	na	int
DWPubsSales.dbo.DimTitles.TitleId	Dimension Key Column	Pubs.dbo.titles.title_id	varchar(6)	nchar(6)
DWPubsSales.dbo.DimTitles.TitleName	Dimension Column	Pubs.dbo.titles.title	varchar(80)	nvarchar(50)
DWPubsSales.dbo.DimTitles.TitleType	Dimension Column	Pubs.dbo.titles.type	char(12)	nvarchar(50)
DWPubsSales.dbo.DimTitles.PublisherId	Dimension Column	Pubs.dbo.titles.pub_id	char(4)	nvarchar(50)
DWPubsSales.dbo.DimTitles.TitlePrice	Dimension Column	Pubs.dbo.titles.price	money	decimal(18,4)
DWPubsSales.dbo.DimTitles.PublishedDate	Dimension Column	Pubs.dbo.titles.pubdate	datetime	date
DWPubsSales.dbo.DimPublishers	Dimesion Table	Pubs.dbo.publishers	Table	
DWPubsSales.dbo.DimPublishers.PublisherKe	Dimension Key Column	Generated	na	int

Figure 4-24. The PubsBISolution data warehouse worksheet

In this exercise, you reviewed the data warehouse design created by the authors. Now it's time to move on to creating the database using SQL Server 2012. As we do so, we revisit this spreadsheet in more detail (Figure 4-24).

Moving On

In this chapter, you learned techniques for designing a data warehouse. You also reviewed an example of an OLAP database created by the authors and compared it to its OLTP counterpart. Finally, you reviewed a design for a data warehouse built upon the Pubs OLTP database. Now it's time to implement that design by building a data warehouse in the next chapter.

LEARN BY DOING

In this "Learn by Doing" exercise, you perform the processes defined in this chapter using the Northwind database. We have included an outline of the steps you performed in this chapter and an example of how the authors handled them in two Word documents. These documents are found in the folder `C:\_BISolutionsBookFiles\_LearnByDoing\Chapter04Files`. Please see the `ReadMe.doc` file for detailed instructions.

What's Next?

We just gave you quite a bit of information on designing a data warehouse, but there is always more to tell. In this chapter, we focused only on core concepts that you can anticipate seeing on a regular basis. Because of this, you may be interested in researching more on the subject.

For more information, articles and videos on this subject can be found at `www.NorthwestTech.org/ProBISolutons`. Design tips posted by the Kimball Group can be found on its website at `www.kimballgroup.com/html/designtips.html`.

We also recommend the following books: *The Data Warehouse Toolkit* and *The Data Warehouse ETL Toolkit*, both by Ralph Kimball (Wiley).