C H A P T E R 2

The Relational Database Model

In This Chapter

In this chapter, I'm going to introduce you to the development of SQL language and why it's important for relational databases. Then I'll give you a brief overview of the various structures in a relational database.

Introducing the Structured Query Language

Without the Structured Query Language (SQL), relational databases might not have become as popular as they are today. Their popularity is due mostly to how the relational database industry evolved and the standards that were an outgrowth of this evolution. Of course the business benefits of a relational database also had a great deal of impact on its acceptance.

Relational history

While the rise in popularity of relational databases appears to be tied to the personal computer revolution, the history of the relational database begins much earlier. And the origins of the relational database begin at the world's largest computer company rather than at M.I.T.S., the unknown company that designed the world's first personal computer.

The prequel to SEQUEL

In the early 1970's, one company dominated the computer business, much like Microsoft does today. That company was IBM. Today's IBM bears little resemblance to the IBM of that era. To many people at that time, computer meant IBM. IBM was a massive organization that planned things years in advance. This showed most dramatically in their research labs. They had researchers whose sole purpose was to think about alternate ways to perform computing tasks without regard to practicality. One of these researchers was Dr. E.F. Codd.

In June 1970, Dr. Codd published an article titled “A Relational Model of Data for Large Shared Data Banks” in the journal, *Communications of the Association for Computing Machinery*. This article presented a mathematical theory for storing and manipulating data using tabular data structures. This theory is the basis for the modern relational database.

SEQUEL and System/R

The idea of a relational database was appealing for many reasons and IBM authorized a research project in the mid-1970’s to create a database system based on that theory. This project became known as System/R. Along with the work on the database itself, work was also underway to develop a query language for the database. The researchers developed several different languages, including one known as SEQUEL (Structured English QUEry Language).

In the late 1970’s, IBM decided to release System/R to some of their key customers for evaluation. As part of this evaluation process, the query language was renamed SQL, although it continued to be pronounced SEQUEL. The System/R project came to a close in 1979, when IBM decided that the relational database theories developed by Codd ten years earlier were ready to be turned into a commercial product.

Ingres and Oracle

IBM was not the only company to work on relational databases during the 1970’s. Some professors at the University of California’s Berkeley computer labs also built a relational database known as Ingres. The query language for this database was known as QUEL, which is short for QUEry Language. Like SQL, this language was based on a highly structured form of the English language.

Ingres was also developed on a Unix platform, unlike System/R, which ran on an IBM mainframe operating system. At that time, Unix systems generally ran on small mini-computers such as Digital’s VAX, which were much cheaper to acquire and operate than mainframes. This made the database popular with many universities that used the database to teach the fundamental concepts of relational database implementations.

Eventually, some of the professors left the Berkeley computer labs in 1980 and formed a company known as Relational Technology, Inc. to build a commercial version of Ingres. Ingres still exists as a commercial database, though it hasn’t achieved the popularity of some other database implementations.

Another company called Relational Software, Inc. was formed in the late 1970's to create their own relational database system. Their database system, called Oracle, was first shipped in 1979, making it the first commercially available relational database system. Like IBM's System/R project, it was based on SQL, and like Ingres, it ran on a VAX mini-computer. Today, Oracle is the leading supplier of relational database systems in the world.

Back to IBM

In 1981, IBM shipped their first commercial relational database system, called SQL/Data System (SQL/DS). SQL/DS was available for the Virtual Machine (VM) operating system and was targeted primarily at computer centers that wanted to provide ad hoc query processing for relatively small databases.

IBM chose this approach for two main reasons. First, they didn't want to cannibalize sales of their highly successful hierarchical database system, Information Management System (IMS), which ran on the large Multiple Virtual Storage (MVS) based mainframes. Second, for any given workload, SQL/DS was much slower than IMS. By restricting relational database technology to a small niche, IBM was able to get practical experience with the technology without alienating its customers.

In 1983, IBM released the first relational database system to run on MVS, called Data Base/2, which is more commonly known as DB2. In the early days, DB2 gained a reputation for being a resource hog and for very slow performance. This was true when compared to the IMS database, which had many more years of tuning and optimization than DB2 had at that time. IBM's response was to suggest that DB2 should be used for ad hoc query processing, and to use IMS for high-volume transaction processing.

IBM also labeled DB2 as a strategic product. In IBM speak, this meant that a customer that was concerned about the future should be running DB2 today. While this helped DB2 sales, it didn't help DB2 performance. However, as time went on, two things helped to change DB2 performance dramatically. First, DB2 underwent several major version changes, with each change bringing a significant improvement in performance. At the same time, the price of hardware dropped immensely, which meant that customers could buy a lot more computing power for the same amount of money. And nothing solves a performance problem as well as more hardware.

Tip

A lot of memory is good and too much is just right: When dealing with a relational database, nothing improves performance as much as adding memory. Adding more RAM to a computer allows the database to reduce the amount of I/O needed to process a query. Adding RAM for a disk cache will also work wonders.

ANSI Standard SQL

In 1982, the American National Standards Institute (ANSI) began a process to determine a standard for a relational database query language. Over the next four years they reviewed several different languages, including SQL and QUEL; however, with IBM's weight behind DB2, SQL was eventually selected as the standard. Even though SQL was selected as the standard, there were some major differences between the actual syntax used by DB2 and the final syntax adopted by ANSI in 1986.

The final syntax adopted in 1986 had a lot of open holes, which prompted ANSI to revise the standard for a second time in 1989 and again in 1992. The standards have become known as SQL-86, SQL-89, and SQL-92. Because the standards allowed various levels of compliance, many database vendors could claim that their database supported the ANSI SQL standard.

Caution

A non-standard standard: The ANSI standard permits vendors to add their own proprietary extensions to their SQL implementations. In practice, this means that each vendor's implementation is significantly different from each other. While the really simple statements are compatible between vendors, many complex statements may not be compatible between database vendors. If you are trying to use the one-program fits all database systems approach to application design, you need to verify that the statements you use are appropriate for all database systems.

Ingres finally bowed to the pressure and in 1986, added support for SQL alongside QUEL. Since that time, nearly all database vendors have added support for SQL to their databases. Even some database vendors with non-relational database systems now support the Structured Query Language.

Other relational database vendors followed Oracle's lead and jumped on the SQL bandwagon, such as Sybase, Informix, Digital, and Microsoft. For the most part, these vendors ignored the high-end mainframes and concentrated their efforts on mini-computers and/or personal computer networked servers, which turned out to be the right decision since the demand for mainframe databases is declining as the demand for mainframes remains stagnant.

Note

Mainframes and Unix servers: Having worked with computers of many different sizes over the years, I've learned that there isn't as much difference between a mainframe and some of the largest Unix servers on the market as most people believe. Much of the technology used in the high-end Unix servers was first developed for the mainframe, while the innovative design approaches used to build affordable Unix servers were incorporated into mainframes. This blending of technology and design means that mainframes will continue to be around for a long time as an alternative to the high-end Unix servers.

Business benefits of a relational database

Obviously, SQL wouldn't have achieved its success unless it met people's expectations. SQL succeeded because of several key factors, such as the underlying relational database technology, the existence of an independent ANSI standard, and the fact that the language itself is very robust and that most of the database systems using SQL are based on modern client/server architectures.

Relational database technology

Designing a non-relational database can be a difficult task and using one can be even worse. In most non-relational database systems, creating a database is a difficult task due to the fact that the hierarchies must be completely described or all of the possible relationships must be thought out before the database is created. A relational database allows you to make changes easily and on the fly. You can add and remove database structures while the database is running.

The relational database model also permits people to use relationships that haven't been predefined. This lets an organization change their databases gradually over time, which reduces the impact and scope of any single change.

Pros and cons of the ANSI standard

Many people believe that the ANSI standard guarantees that programs written to use SQL can be moved from one database platform to another with minimal problems, in much the same way that people can move an ANSI C program from one machine to another. This gives many users the feeling that if they need to, they can switch database vendors. In reality, unless you make a special effort to design your application to the ANSI standard, most people will find moving from one database system to another more trouble than it's worth.

However, there are many advantages to having an independent standard than simply portability. One of the largest problems today is finding knowledgeable people to develop applications. Often people are hired with some of the skills, but need to be trained in others. While the details of the SQL syntax may vary from one database vendor to another, the basic statements are the same. This makes it possible to train someone in a new database system by focusing only on how their SQL is different from the standard rather than starting with a whole new language. This means that people can move from one database system to another with much less effort than would have been possible without an independent standard.

Powerful query language

Prior to SQL, most database systems used one method to access a database interactively and another method for programmatic access. This forced programmers to learn multiple languages to access their database. Also, the interactive query language wasn't powerful enough to answer certain types of questions. In addition, there was typically a third language that was used to define the structure of the database. This made it much harder than it really needed to be for many application programmers. SQL solves this problem by providing a single query language. This same language can be used interactively, programmatically and to define the database structures.

Client/server architecture

Another benefit of using database systems based on the SQL language is that most database systems have been developed using a client/server architecture. This makes it a natural fit for modern personal computer networks. This also makes it easy to scale the size of the computer running the database server, which in turn allows an organization to spend less money on computing resources.

Parts of a Relational Database

In order to understand how to access a relational database, you need to understand the basic parts of a relational database and how they fit together.

Tip

Separation of data: I often do my development directly on a Windows 2000/NT Server system, with a local database server. This way, I can test applications I'm working on against a test database without impacting my production database server.

Tables and rows of data

A database holds your data in a series of one or more *tables*. Each table is similar to a spreadsheet, with the data organized into a series of rows and columns. Each row corresponds to a record in a file and represents a unique instance of a piece of data. A column corresponds to a field in a file and represents a single attribute of a piece of data.

Consider a table that contains information about a customer. Each row in the table contains information about a single customer, while each column in the table contains a specific attribute about that customer, such as their name or street address.

Columns and data types

The data in a column is often called *atomic*, which implies that the data in the column can't be subdivided. For instance, consider a field called Name, which consists of three subparts: Name.First, Name.Middle, and Name.Last. The Name field is not atomic since it can be broken into smaller pieces. On the other hand, a field that contains an unstructured name value would be atomic since there is no structure to the field.

A *repeating group* is a field that contains more than one instance of a data value. Thus any field that is an array or a collection is a repeating group. Since a repeating group is also not atomic, it can't be represented by a single column.

Associated with each column is a *data type* that identifies the type of data you plan to store in the column. In general there are four main data types:

✦ **Numbers** — Integers, fixed decimal point and floating decimal point numbers fall into this category.

✦ **Character strings** — *Fixed-length* character strings always use the same amount of storage in your table, while *variable-length* character strings store only the characters in the string, plus some information that allows the database to track the length of the string.

✦ **Date/time values** — This data type contains a date value, a time value, or a combination date and time value. Each database server will determine the exact meaning of these values and how they are stored.

✦ **Binary values** — This is basically an unformatted, store-it-yourself field. You can store items such as images, sound clips, and even application programs using a binary data type.

Note

All binary data types are not the same: The database server you are using will determine the characteristics for a binary data type.

In addition to storing a data value in a column, a special flag known as **Null** is also available. This flag indicates whether the column contains a valid value. If the column has a valid value, this flag will be set to false and the column will be known as **Not Null**. However if you don't assign a value to this column, then the flag will be set to true and the column will be known as **Null**.

Note

Nulls aren't empty: Don't confuse an empty string with **Null**. An empty string represents a value, while **Null** means that no value is associated with the column.

Indexes and keys

In the theoretical relational database model there isn't any organization to how the rows are stored in the table. However, from a practical viewpoint, you need a mechanism that allows you to quickly select a row or set of rows. This mechanism is known as an *index*.

An index is a special database structure that maintains a set of column values, sorted in a way that minimizes the search time. Typically, an index is implemented as an inverted list, similar to that used in the indexed database model. This allows the database server to quickly search through the list of values in the index to select the desired rows from the table. Indexes are separate from the table and can be added and removed on the fly.

A column is said to depend on another field when there is only one possible value of the second field for a specific value of the first column. If this sounds confusing, consider the following: you have a table with a name column and a social security number column. Since the social security number is unique for each individual and each individual has only one name, the name field is dependent on the social security number. Note that the reverse isn't true, since it is possible to have more than one social security number for a given name. In other words, there may be more than one person with the name of John Smith in this country, and each will have their own unique social security number.

The list of columns included in the index is known as the *key*. Every table should have a key whose value is unique for each row in the table. This is known as the *primary key*. For example, a table that contains information about customers might have a column called CustomerId as the primary key. A key that contains more than one column is also known as a *composite key*.

Any other keys in the table are referred to as *secondary keys*. Unlike the primary key, secondary keys need not be unique. They merely help you locate a common subset of rows in a table.

When a group of columns in one table matches the primary key definition in another table, the group of columns is called a *foreign key*. Foreign keys are useful when you want to establish a relationship between two tables. For example, in the customer table, one of the columns might be zip code. You could add a ZIP code table to your database, which contains a list of ZIP codes each with their corresponding city name. Thus, for any given ZIP code, you could determine the name of the city where the customer lives. This way you could automatically fill in the city name when a customer enters their ZIP code.

You can choose to make a relationship — a required relationship — in which case the database server will prevent you from adding a row to the table unless the value in the foreign key columns is found as the primary key in the corresponding table. Thus if you choose to enforce the above relationship, you can only insert a row into the customer's table if the customer's ZIP code exists in the ZIP code table.

Another advantage of indexes is that you can use an index to ensure that a value in the table is unique. While the primary key of the table must be unique, in some cases you may want the secondary key to be unique as well.

Theory vs. Reality

One of the advantages of a relational database is the use of set theory to describe operations against its data. It allows a high degree of separation between how the data is viewed by the database user and how the data is actually stored internally. However, just because something makes sense in theory doesn't necessarily mean it's a good idea, because the theory doesn't take into consideration the physical limitations of your database server.

Without indexes, every query would have to read every row in a table. On small tables this wouldn't be a big handicap, since it doesn't take very long to read all of the rows. However, on large tables this can be a major problem.

Views

One of the most important concepts in a relational database is known as a *view*. A view is simply a "virtual table" created from one or more base tables in your database. Views come in two flavors: updateable and non-updateable. As their names imply, updateable views can be updated and are usually created from a single table. Non-updateable views are generally created from two or more tables or contain columns whose values are calculated in some fashion.

Normalization

Normalization is a way to classify your database structure. From a theoretical viewpoint, the more normalized you database is, the better. These are the four basic levels of normalization that are commonly found in normal database designs:

- **Unnormalized:** No rules are imposed on the database structure.
- **First normal form:** Each field must be atomic. Repeating groups and composite fields are not permitted.
- **Second normal form:** Every non-key field must depend on the entire primary key. A field must not depend on only part of a composite primary key. The database must also be in first normal form.
- **Third normal form:** A non-key field can't depend on another non-key field. The database must also be in the second normal form.

Relational database theory also describes a fourth normal form, a fifth normal form, and a Boyce/Codd normal form. These last forms often result in a database that has too many tables to offer good performance.

You can't build an unnormalized relational database since repeating groups and composite fields aren't permitted. (Of course there are ways around even these restrictions in most database servers.) The most important thing to understand is that as you move up the normalization ladder, the database's large tables become broken down into more and more small tables. This is done in the name of reducing data duplication.

For example, in a truly normalized database, you wouldn't store city and state information with someone's address, since you can get this information from the person's ZIP code. You would add another table to your database that uses the ZIP code as a primary key and have the corresponding city and state information for that ZIP code. However, to get someone's address, you now must access two different tables, which is far more expensive than accessing a single table with all of the customer's address information.

Thoughts on Relational Databases

Relational databases have a much longer history than most people would believe. However, I believe this history played a very important part in shaping the relational databases you use today.

Unlike other types of databases that were either developed by a committee or by a single vendor who were able to unilaterally impose their standards on their customers, relational databases were developed through intense competition. This competition has existed for nearly twenty years and shows no sign of letting up.

When selecting a database, rely less on the features of a particular relational database, and more on the database vendor's track record. Relational databases are usually updated every couple of years and sometimes more often than that. When one particularly innovative feature shows up in one database system, the other vendors will be quick to copy and improve on it and you'll most likely see it released in the next version of their database software.

Many customers will make a decision on which database to buy based on which database vendor has the best features at the time of their evaluation. I've seen organizations use this approach, only to have problems down the road when the vendor decided to freeze their product at a particular version and not enhance it any more. This left the organization heavily dependent on a dead-end product and forced them to undergo a major conversion effort to a similar product from another vendor. This cost a lot of time and effort that could have been used to develop new applications.

Carefully consider these issues when picking your database product. Remember that the investment you make in your database is not just for a year or two but for the next ten to twenty years. There are many vendors in today's market that have a long track record and have proven themselves over time. Choosing one of these vendors will save you time and money in the long run.

Summary

In this chapter you learned that:

- The evolution of the relational database resulted in many database vendors offering products that appear to be compatible because they are based on the SQL database language, but in reality they are highly incompatible.
- There are many business benefits to using relational database systems in your applications, including having a single language for data definition, data manipulation and query processing, and a client server architecture for efficient sharing of data.
- The major parts of a relational database include tables, columns, rows, indexes, and views.
- Normalizing a database is not necessarily a good thing.

✦ ✦ ✦