1

Database Modeling Past and Present

"...a page of history is worth a volume of logic." (Oliver Wendell Holmes)

Why a theory was devised and how it is now applied, can be more significant than the theory itself.

This chapter gives you a basic grounding in database model design. To begin with, you need to understand simple concepts, such as the difference between a database model and a database. A *database model* is a blueprint for how data is stored in a database and is similar to an architectural approach for how data is stored — a pretty picture commonly known as an *entity relationship diagram* (a database on paper). A *database*, on the other hand, is the implementation or creation of a physical database on a computer. A database model is used to create a database.

In this chapter, you also examine the evolution of database modeling. As a natural progression of improvements in database modeling design, the relational database model has evolved into what it is today. Each step in the evolutionary development of database modeling has solved one or more problems.

The final step of database modeling evolution is applications and how they affect a database model design. An *application* is a computer program with a user-friendly interface. End-users use interfaces (or screens) to access data in a database. Different types of applications use a database in different ways — this can affect how a database model should be designed. Before you set off to figure out a design strategy, you must have a general idea of the kind of applications your database will serve. Different types of database models underpin different types of applications. You must understand where different types of database models apply.

It is essential to understand that a well-organized design process is paramount to success. Also, a goal to drive the design process is equally as important as the design itself. There is no sense designing or even building something unless the target goal is established first, and kept foremost in mind.

This chapter, being the first in this book, lays the groundwork by examining the most basic concepts of database modeling.

By the end of this chapter, you should understand why the relational database model evolved. You will come to accept that the relational database model has some shortcomings, but after many years it is still the most effective of available database modeling design techniques, for most application types. You will also discover that variations of the relational database model depend on the application type, such as an Internet interface, or a data warehouse reporting system.

In this chapter, you learn about the following:

- ❑ The definition of a database
- ❑ The definition of a database model
- ❑ The evolution of database modeling
- ❑ The hierarchical and network database models
- ❑ The relational database model
- ❑ The object and object-relational database models
- ❑ Database model types
- ❑ Database design objectives
- ❑ Database design methods

Grasping the Concept of a Database

A *database* is a collection of information — preferably related information and preferably organized. A database consists of the physical files you set up on a computer when installing the database software. On the other hand, a database model is more of a concept than a physical object and is used to create the tables in your database. This section examines the database, not the database model.

By definition, a database is a structured object. It can be a pile of papers, but most likely in the modern world it exists on a computer system. That structured object consists of *data* and *metadata*, with metadata being the structured part. Data in a database is the actual stored descriptive information, such as all the names and addresses of your customers. Metadata describes the structure applied by the database to the customer data. In other words, the metadata is the customer table definition. The customer table definition contains the fields for the names and addresses, the lengths of each of those fields, and datatypes. (A *datatype* restricts values in fields, such as allowing only a date, or a number). Metadata applies structure and organization to raw data.

Figure 1-1 shows a general overview of a database. A database is often represented graphically by a cylindrical disk, as shown on the left of the diagram. The database contains both metadata and raw data. The database itself is stored and executed on a database server computer.

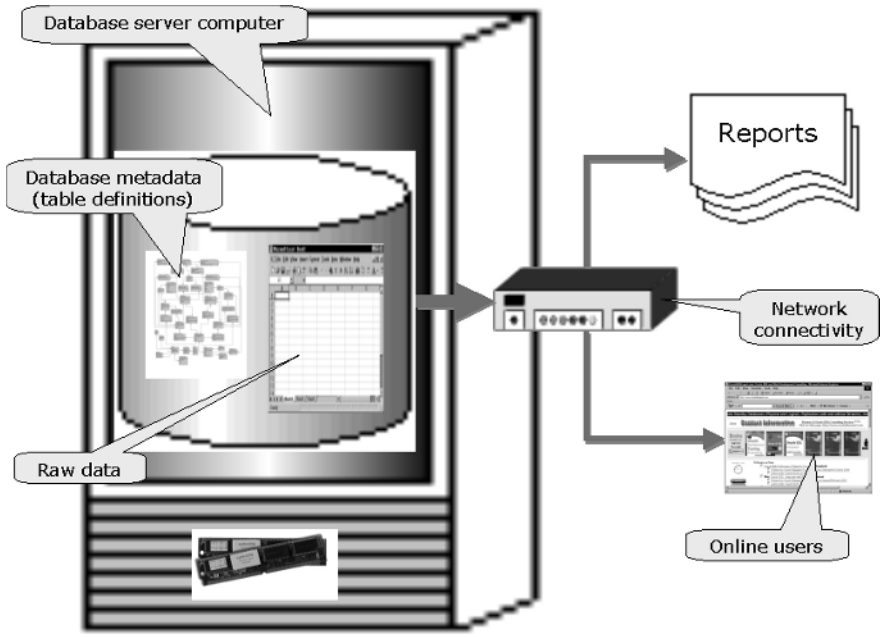

Figure 1-1: General overview of a database.

In Figure 1-1, the database server computer is connected across a network to end-users running reports, and online browser users browsing your Web site (among many other application types).

Understanding a Database Model

There are numerous, precise explanations as to what exactly a *database model* or *data model* is. A database model can be loosely used to describe an organized and ordered set of information stored on a computer. This ordered set of data is often structured using a data modeling solution in such a way as to make the retrieval of and changes to that data more efficient. Depending on the type of applications using the database, the database structure can be modified to allow for efficient changes to that data. It is appropriate to discover how different database modeling techniques have developed over the past 50 years to accommodate efficiency, in terms of both data retrieval and data changes. Before examining database modeling and its evolution, a brief look at applications is important.

What Is an Application?

In computer jargon, an *application* is a piece of software that runs on a computer and performs a task. That task can be interactive and use a graphical user interface (GUI), and can execute reports requiring the click of a button and subsequent retrieval from a printer. Or it can be completely transparent to end-users. *Transparency* in computer jargon means that end-users see just the pretty boxes on their screens and not the inner workings of the database, such as the tables. From the perspective of database modeling, different application types can somewhat (if not completely) determine the requirements for the design of a database model.

An *online transaction processing* (OLTP) database is usually a specialized, highly *concurrent* (shareable) architecture requiring rapid access to very small amounts of data. OLTP applications are often well served by rigidly structured OLTP transactional database models. A *transactional database model* is designed to process lots of small pieces of information for lots of different people, all at the same time.

On the other side of the coin, a *data warehouse* application that requires frequent updates and extensive reporting must have large amounts of properly sorted data, low concurrency, and relatively low response times. A data warehouse database modeling solution is often best served by implementing a denormalized duplication of an OLTP source database.

Figure 1-2 shows the same image as in Figure 1-1, except that in Figure 1-2, the reporting and online browser applications are made more prominent. The most important point to remember is that database modeling requirements are generally determined by application needs. It's all about the applications. End-users use your applications. If you have no end-users, you have no business.

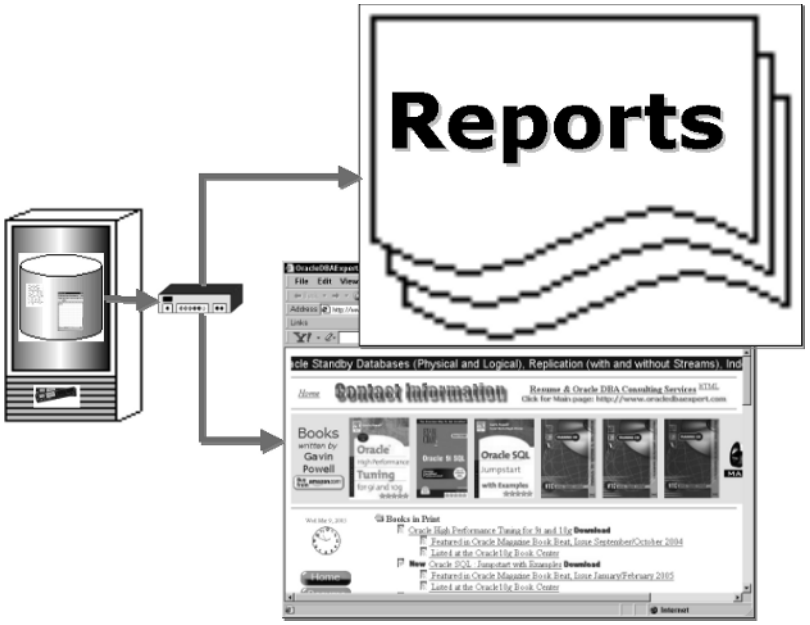

Figure 1-2: Graphic image of an application.

The Evolution of Database Modeling

The various data models that came before the relational database model (such as the hierarchical database model and the network database model) were partial solutions to the never-ending problem of how to store data and how to do it efficiently. The relational database model is currently the best solution for both storage and retrieval of data. Examining the relational database model from its roots can help you understand critical problems the relational database model is used to solve; therefore, it is essential that you understand how the different data models evolved into the relational database model as it is today.

The evolution of database modeling occurred when each database model improved upon the previous one. The initial solution was virtually no database model at all: the *file system* (also known as *flat files*). The file system is the operating system. You can examine files in the file system of the operating system by running a `dir` command in DOS, an `ls` command in UNIX, or searching through the Windows Explorer in Microsoft Windows. The problem that using a file system presents is no database structure at all.

Figure 1-3 shows that evolutionary process over time from around the late 1940s through and beyond the turn of the millennium, 50 years later. It is very unlikely that network and hierarchical databases are still in use.

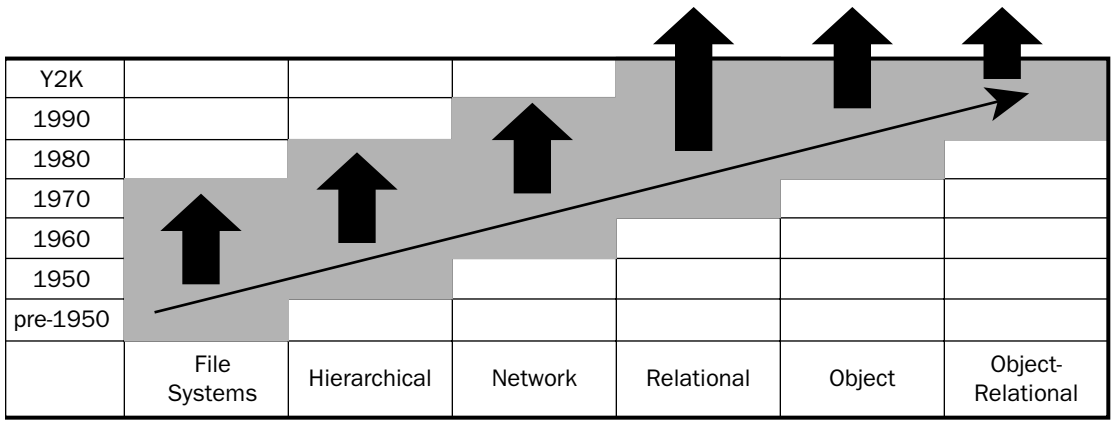

Figure 1-3: The evolution of database modeling techniques.

File Systems

Using a file system database model implies that no modeling techniques are applied and that the database is stored in *flat files* in a file system, utilizing the structure of the operating system alone. The term "flat file" is a way of describing a simple text file, containing no structure whatsoever — data is simply dumped in a file.

> *By definition, a comma-delimited file (CSV file) contains structure because it contains commas. By definition, a comma-delimited file is a flat file. However, flat file databases in the past tended to use huge strings, with no commas and no new lines. Data items were found based on a position in the file. In this respect, a comma-delimited CSV file used with Excel is not a flat file.*

Any searching through flat files for data has to be explicitly programmed. The advantage of the various database models is that they provide some of this programming for you. For a file system database, data can be stored in individual files or multiple files. Similar to searching through flat files, any relationships and validation between different flat files would have to be programmed and likely be of limited capability.

Hierarchical Database Model

The hierarchical database model is an inverted tree-like structure. The tables of this model take on a child-parent relationship. Each *child table* has a single *parent table*, and each parent table can have multiple child tables. Child tables are completely dependent on parent tables; therefore, a child table can exist only if its parent table does. It follows that any entries in child tables can only exist where corresponding parent entries exist in parent tables. The result of this structure is that the hierarchical database model supports *one-to-many* relationships.

Figure 1-4 shows an example hierarchical database model. Every task is part of a project, which is part of a manager, which is part of a division, which is part of a company. So, for example, there is a one-to-many relationship between companies and departments because there are many departments in every company. The disadvantages of the hierarchical database model are that any access must originate at the root node, in the case of Figure 1-4, the Company. You cannot search for an employee without first finding the company, the department, the employee's manager, and finally the employee.

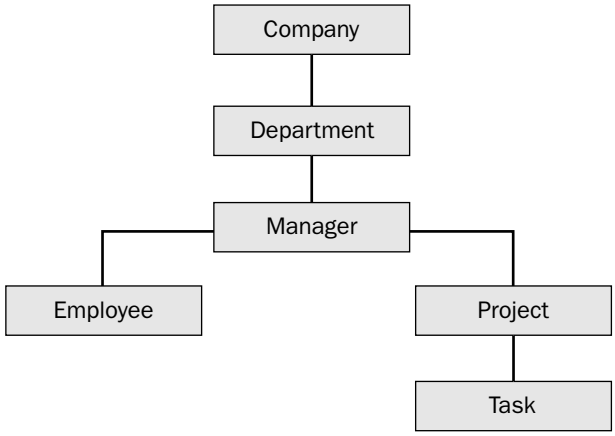

Figure 1-4: The hierarchical database model.

Network Database Model

The network database model is essentially a refinement of the hierarchical database model. The network model allows child tables to have more than one parent, thus creating a networked-like table structure. Multiple parent tables for each child allows for *many-to-many* relationships, in addition to one-to-many relationships. In an example network database model shown in Figure 1-5, there is a many-to-many relationship between employees and tasks. In other words, an employee can be assigned many tasks, and a task can be assigned to many different employees. Thus, many employees have many tasks, and visa versa.

Figure 1-5 shows how the managers can be part of both departments and companies. In other words, the network model in Figure 1-5 is taking into account that not only does each department within a company have a manager, but also that each company has an overall manager (in real life, a Chief Executive Officer, or CEO). Figure 1-5 also shows the addition of table types where employees can be defined as being of different types (such as full-time, part-time, or contract employees). Most importantly to note from Figure 1-5 is the new Assignment table allowing for the assignment of tasks to employees. The creation of the

Assignment table is a direct result of the addition of the multiple-parent capability between the hierarchical and network models. As already stated, the relationship between the employee and task tables is a many-to-many relationship, where each employee can be assigned multiple tasks and each task can be assigned to multiple employees. The Assignment table resolves the dilemma of the many-to-many relationship by allowing a unique definition for the combination of employee and task. Without that unique definition, finding a single assignment would be impossible.

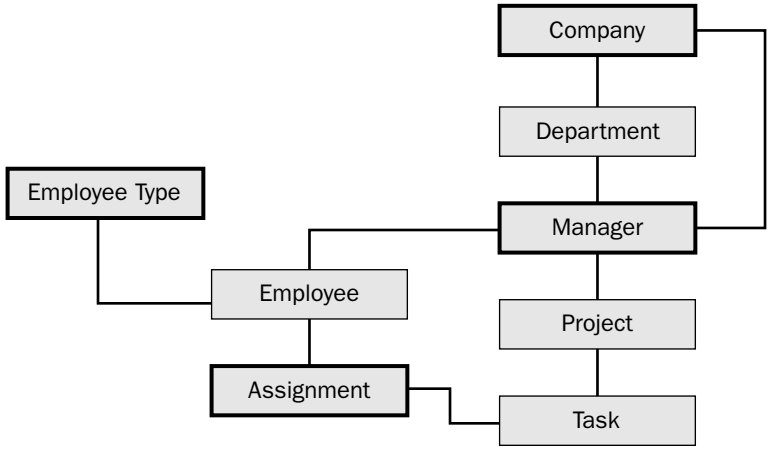

Figure 1-5: The network database model.

Relational Database Model

The relational database model improves on the restriction of a hierarchical structure, not completely abandoning the hierarchy of data, as shown in Figure 1-6. Any table can be accessed directly without having to access all parent objects. The trick is to know what to look for — if you want to find the address of a specific employee, you have to know which employee to look for, or you can simply examine all employees. You don't have to search the entire hierarchy, from the company downward, to find a single employee.

Another benefit of the relational database model is that any tables can be linked together, regardless of their hierarchical position. Obviously, there should be a sensible link between the two tables, but you are not restricted by a strict hierarchical structure; therefore, a table can be linked to both any number of parent tables and any number of child tables.

Figure 1-7 shows a small example section of the relational database model shown in Figure 1-6. The tables shown are the Project and Task tables. The `PROJECT_ID` field on the Project table uniquely identifies each project in the Project table. The relationship between the Project and Task tables is a one-to-many relationship using the `PROJECT_ID` field, duplicated from the Project table to the Task table. As can be seen in Figure 1-7, the first three entries in the Task table are all part of the *Software sales data mart* project.

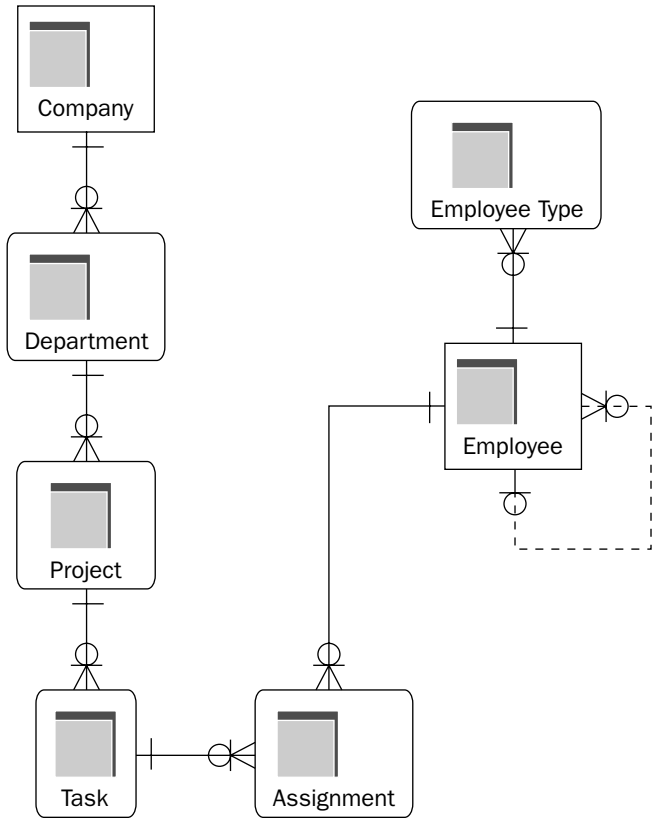

Figure 1-6: The relational database model.

Project

PROJECT_ID	DEPARTMENT_ID	PROJECT	COMPLETION	BUDGET
1	1	Software sales data mart	4-Apr-05	35,000
2	1	Software development costing application	24-Apr-05	50,000
3	2	Easy Street construction project	15-Dec-08	25,000,000
4	1	Company data warehouse	31-Dec-06	250,000

Task

TASK_ID	PROJECT_ID	TASK
1	1	Acquire data from outside vendors
2	1	Build transformation code
3	1	Test all ETL process
4	2	Assess vendor costing applications
5	3	Hire an architect
6	3	Hire an engineer
7	3	Buy lots of bricks
8	3	Buy lots of concrete
9	3	Find someone to do this because we don't know how

Figure 1-7: The relational database model — a picture of the data.

Relational Database Management System

A *relational database management system* (RDBMS) is a term used to describe an entire suite of programs for both managing a relational database and communicating with that relational database engine. Sometimes Software Development Kit (SDK) front-end tools and complete management kits are included with relational database packages. Microsoft Access is an example of this. Both the relational database and front-end development tools, for building input screens, are all packaged within the same piece of software. In other words, an RDBMS is both the database engine and any other tools that come with it. RDBMS is just another name for a relational database product. It's no big deal.

The History of the Relational Database Model

The relational database was invented by an IBM researcher named Dr. E. F. Codd, who published a number of papers over a period of time. Other people have enhanced Dr. Codd's original research, bringing the relational database model to where it is today.

Essentially, the relational database model began as a way of getting groups of data from a larger data set. This could be done by removing duplication from the data using a process called *normalization*. Normalization is composed of a number of steps called *normal forms*. The result was a general data access language ultimately called the Structured Query Language (SQL) that allowed for queries against organized data structures. All the new terms listed in this paragraph (including normalization, normal forms, and SQL) are explained in later chapters.

Much of what happened after Dr Codd's initial theoretical papers was vendor development and involved a number of major players. Figure 1-8 shows a number of distinct branches of development. These branches were DB2 from IBM, Oracle Database from Oracle Corporation, and a multitude of relational databases stemming from Ingres (which was initially conceived by two scientists at Berkeley). The more minor relational database engines such as dBase, MS-Access, and Paradox tended to cater to single-user, small-scale environments, and often included free front-end application development kits.

The development path of the different relational database vendors proceeded as follows. Development from one database to another usually resided in different companies, and was characterized by movement of personnel rather than of database source code. In other words, the people invented the different databases (not the companies), and people moved between different companies. Additionally, numerous object databases have been developed. Object databases generally have very distinct applications. Some object databases have their roots in relational technology, once again in terms of the movement of personnel skills.

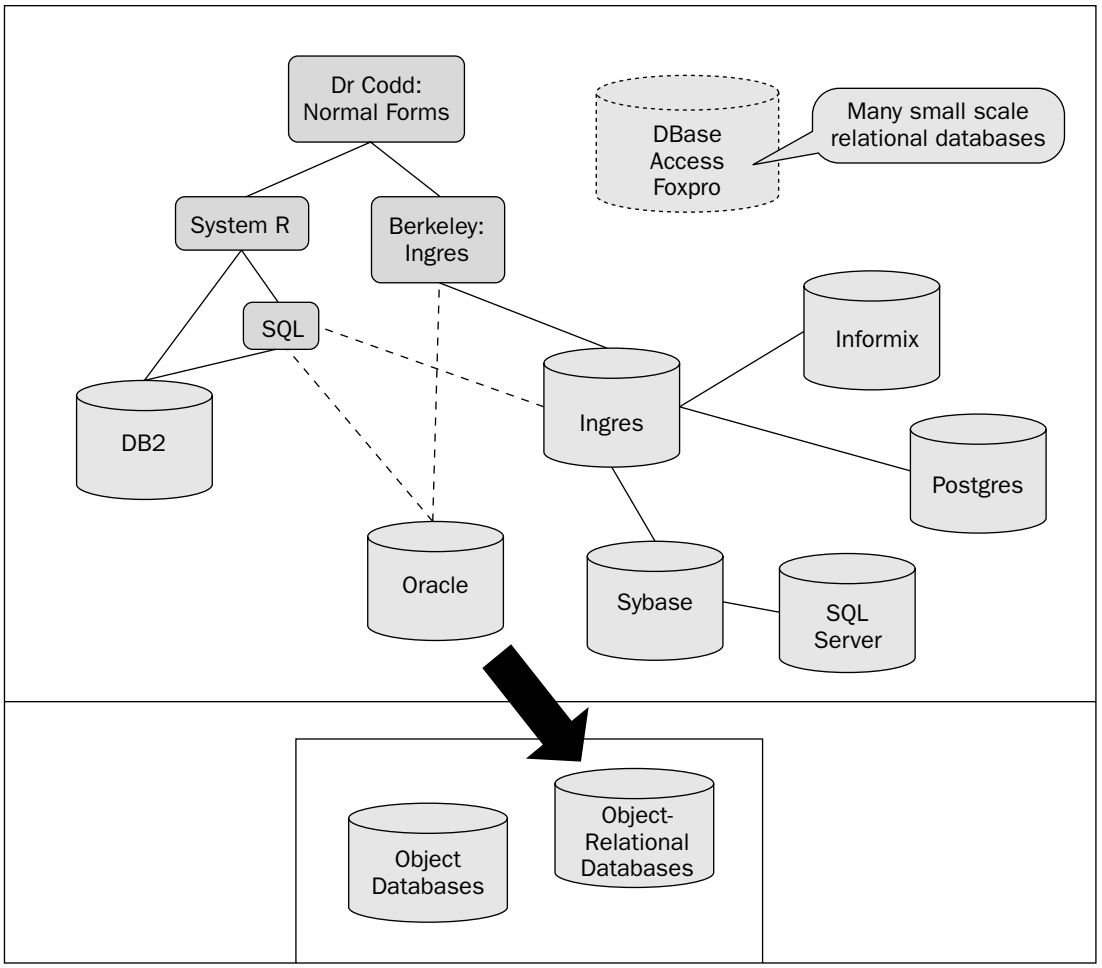

Figure 1-8: The history of the relational database model.

Object Database Model

An object database model provides a three-dimensional structure to data where any item in a database can be retrieved from any point very rapidly. Whereas the relational database model lends itself to retrieval of groups of records in two dimensions, the object database model is efficient for finding unique items. Consequently, the object database model performs poorly when retrieving more than a single item, at which the relational database model is proficient.

The object database model does resolve some of the more obscure complexities of the relational database model, such as removal of the need for types and many-to-many relationship replacement tables. Figure 1-9 shows an example object database model structure equivalent of the relational database model structure shown in Figure 1-6. The assignment of tasks to employees is catered for using a collection inclusion in the manager, employee, and employee specialization classes. Also note that the different types of employees are catered for by using specializations of the employee class.

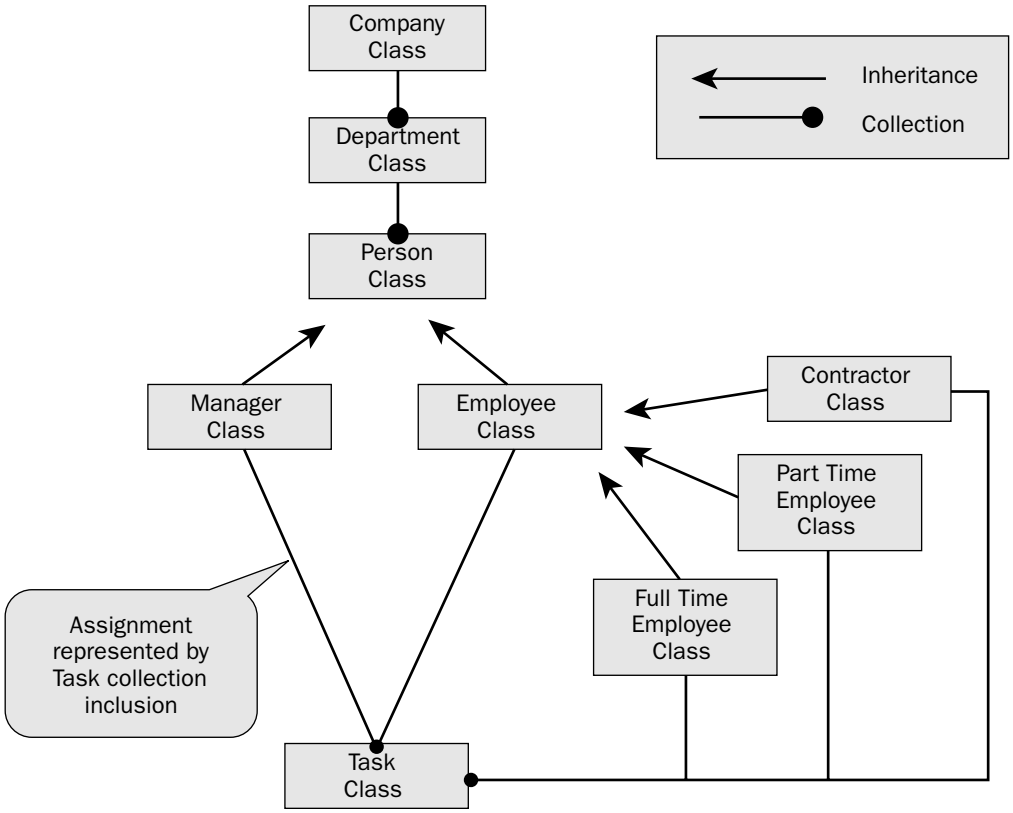

Figure 1-9: The object database model.

Another benefit of the object database model is its inherent ability to manage and cater for extremely complex applications and database models. This is because of a basic tenet of object methodology whereby highly complex elements can be broken down into their most basic parts, allowing explicit access to, as well as execution against and within those basic parts. In other words, if you can figure out how all the little pieces work individually, it makes the big picture (complex by itself) a combination of a number of smaller, much simpler constituent pieces.

A discussion of the object database model in a book covering the relational database model is important because many modern applications are written using object methodology based SDKs such as Java. One of the biggest sticking points between object programmed applications and relational databases is the performance of the mapping process between the two structural types: object and relational. Object and relational structure is completely different. It is, therefore, essential to have some understanding of object database modeling techniques to allow development of efficient use of relational databases by object-built applications.

Object-Relational Database Model

The object database model is somewhat spherical in nature, allowing access to unique elements anywhere within a database structure, with extremely high performance. The object database model performs extremely poorly when retrieving more than a single data item. The relational database model, on the other hand, contains records of data in tables across two dimensions. The relational database model is best suited for retrieval of groups of data, but can also be used to access unique data items fairly efficiently. The object-relational database model was created in answer to conflicting capabilities of relational and object database models.

Essentially, object database modeling capabilities are included in relational databases, but not the other way around. Many relational databases now allow binary object storage and limited object method coding capabilities, with varying degrees of success. The biggest issue with storage of binary objects in a relational database is that potentially large objects are stored in what is actually a small-scale structural element as a single field-record entry in a table. This is not always strictly the case because some relational databases allow storage of binary objects in separate disk files outside the table's two-dimensional record structures.

The evolution of database modeling began with what was effectively no database model whatsoever with file system databases, evolving on to hierarchies to structure, networks to allow for special relationships, onto the relational database model allowing for unique individual element access anywhere in the database. The object database model has a specific niche function at handling high-speed application of small data items within large highly complex data sets. The object-relational model attempts to include the most readily accountable aspects of the object database model into the structure of the relational database model, with varying (and sometimes dubious) degrees of success.

Examining the Types of Databases

At this stage, we need to branch into both the database and application arenas because the choice of database modeling strategy is affected by application requirements. After all, the reason you build a database is to service some need. That need is influenced by one or more applications. Applications should present user-friendly interfaces to end-users. End-users should not be expected to know anything at all about database modeling. The objective is to provide something useful to a banker, an insurance sales executive, or anyone else most likely not in the computer industry, and probably not even in a technical field. You need to take into account the function of what a database achieves, rather than the complicated logic that goes into designing the specific database.

Databases functionally fall into three general categories:

- Transactional
- Decision support system (DSS)
- Hybrid

Transactional Databases

A *transactional database* is a database based on small changes to the database (that is, small transactions). The database is transaction-driven. In other words, the primary function of the database is to add new data, change existing data, delete existing data, all done in usually very small chunks, such as individual records.

The following are some examples of transactional databases:

- *Client-Server database* — A client-server environment was common in the pre-Internet days where a transactional database serviced users within a single company. The number of users could range from as little as one to thousands, depending on the size of the company. The critical factor was actually a mixture of both individual record change activity and modestly sized reports. Client-server database models typically catered for low concurrency and low throughput at the same time because the number of users was always manageable.
- *OLTP database* — OLTP databases cause problems with concurrency. The number of users that can be reached over the Internet is an unimaginable order of magnitude larger than that of any in-house company client-server database. Thus, the concurrency requirements for OLTP database models explode well beyond the scope of previous experience with client-server databases. The difference in scale can only be described as follows:
  - *Client-server database* — A client-server database inside a company services, for example, 1,000 users. A company of 1,000 people is unlikely to be corporate and global, so all users are in the same country, and even likely to be in the same city, perhaps even in and around the same office. Therefore, the client-server database services 1,000 people, 8 hours per day, 5 days a week, perhaps 52 weeks a year. The standard U.S. work year is estimated at a maximum of about 2,000 hours per year. That's a maximum of 2,000 hours per year, per person. Also, consider how many users will access the database at exactly the same millisecond. The answer is probably 1! You get the picture.
  - *OLTP database* — An OLTP database, on the other hand, can have millions of potential users, 24 hours per day, 365 days per year. An OLTP database must be permanently online and concurrently available to even in excess of 1,000 users every millisecond. Imagine if half a million people are watching a home shopping network on television and a Web site appears offering something for free that everyone wants. How many people hit the Web site at once and make demands on the OLTP database behind that Web site? The quantities of users are potentially staggering. This is what an OLTP database has to cater to — enormously high levels of concurrent database access.

Decision Support Databases

Decision support systems are commonly known as DSS databases, and they do just that — they support decisions, generally more management-level and even executive-level decision-type of objectives. Following are some DSS examples:

- *Data warehouse database* — A *data warehouse* database can use the same data modeling approach as a transactional database model. However, data warehouses often contain many years of historical data to provide effective forecasting capabilities. The result is that data warehouses can become excessively large, perhaps even millions of times larger than their counterpart OLTP source

databases. The OLTP database is the source database because the OLTP database is the database where all the transactional information in the data warehouse originates. In other words, as data becomes not current in an OLTP database, it is moved to a data warehouse database. Note the use of the word "moved," implying that the data is copied to the data warehouse and deleted from the OLTP database. Data warehouses need specialized relational database modeling techniques.

- *Data mart* — A *data mart* is essentially a small subset of a larger data warehouse. Data marts are typically extracted as small sections of data warehouses, or created as small section data chunks during the process of creating a much larger data warehouse database. There is no reason why a data mart should use a different database modeling technique than that of its parent data warehouse.
- *Reporting database* — A *reporting database* is often a data warehouse type database, but containing only active (and not historical or archived) data. A simple reporting database is of small size compared to a data warehouse database, and likely to be much more manageable and flexible.

Data warehouse databases are typically inflexible because they can get so incredibly large.

Hybrid Databases

A *hybrid database* is simply a mixture containing both OLTP type concurrency requirements and data warehouse type throughput requirements. In less-demanding environments (or in companies running smaller operations), a smaller hybrid database is often a more cost-effective option, simply because there is one rather than two databases — fewer machines, fewer software licenses, fewer people, the list goes on.

This section has described what a database does. The function of the database can determine the way in which the database model is built. The following section goes back to the database model design process, but approaching it from a conceptual perspective.

Understanding Database Model Design

Do you really need to design stuff? When designing a computer system or a database model, you might wonder why you need to design it. And exactly what is *design*? Design is to writing software like what architecture is to civil engineering. Architects learn all the arty stuff such as where the bathrooms go and how many bathrooms there are, and whether or not there are bathrooms. If the architecture were left to the civil engineers, they might forget the bathrooms or leave the occupants of the completed structure with Portaloos or outhouses.

Civil engineers ensure that it all stands up without falling down on our heads. Architects make it habitable. So, where does that lead us with software, database modeling, and having to design the database model? Essentially, the design process involves putting your ideas on paper before actually constructing your object, and perhaps experimenting with moving parts and pieces around a bit just to see what they look like. Civil engineers are not in the habit of erecting millions of tons of precast concrete slabs into the forms of bridges and skyscrapers and then moving bits around (such as whole corners and sections of structures) just to see what the changes look like. You see my point. You must design it and build it on paper first. You could use something like a computer-aided design (CAD) package to sort out the *seeing what it looks like* stage. In terms of the database model, you must design it before you build it and then start filling it with data and hooking it up to applications.

Database design is so important because all applications written against that database model design are completely dependent on the structure of that underlying database. If the database model must be altered at a later stage, everything constructed based on the database model probably must be changed and perhaps even completely rewritten. That's all the applications — and I mean all of them! That can get very expensive and time consuming. Design the database model in the same way that you would design an application — using tools, flowcharts, pretty pictures, Entity Relationship Diagrams (ERDs), and anything else that might help to ensure that what you intend to build is not only what you need, but also will actually work, and preferably work without ever breaking.

Of course, liability issues place far more stringent requirements on the process of design for architects and civil engineers when building concrete structures than that compared with computer systems. Just imagine how much it costs to build a skyscraper! Skyscrapers can take 10 years to build. The cost in wages alone is probably in the hundreds of millions. A computer system, however, and database model that ultimately turns into a complete dud as a result of poor planning and design can cost a company more money than it is prepared to spend and perhaps more than a company is even able to lose.

Design is the process of ensuring that it all works without actually building it. Design is a little like testing something on paper before spending thousands of hours building it in possibly the wrong way.

Design is needed to ensure that it works before spending humungous amounts of money finding out that it doesn't. The idea is to fix as many teething problems and errors in the design. Fixing the design is much easier than fixing a finished product. A design on paper costs a fraction of what building and implementing the real thing would cost. Waste a small amount of money in planning, rather than lose more than can be afforded when it's too late to fix it.

Defining the Objectives

Defining objectives is probably the single most important task done in planning any project, be it a skyscraper or a database model. You could, of course, just start anywhere and dive right into the project with your eyes shut. But that is not planning. The more you plan what you are going to do, the more likely the final result will fit your requirements.

Aside from planning, you must know what to plan in the first place. *Defining the objectives* is the basic step of defining how you are going to get from A to B.

So, now that you know you have to plan your steps, you also have to know what the steps are that you are planning for (be those steps the final result or smaller steps in between). There are, of course, a number of points to guide the establishment of design objectives for a proper relational database model design:

- *Aim for a well-structured database model* — A well-structured database model is simple, easy to read, and easy to comprehend. If your company has a database model made up of 50 pieces of A4-sized paper taped to an entire wall, and links between tables taking 20 minutes to trace, you have a problem. That problem is poor structure. If you are interviewed as a contractor to sort out a problem like this, you might be faced with a Herculean task.
- *Data integrity* — Integrity is a set of rules in a database model, ensuring that data is not lost within the database, and that data is only destroyed when it should be.

- *Support both planned queries and ad-hoc or unplanned queries* — The fewer ad-hoc queries, the better, of course, and in some circumstances (such as very high-concurrency OLTP databases), ad-hoc queries might have to be banned altogether, or perhaps shifted to a more appropriate data warehouse platform.

An ad-hoc query is a query that is submitted to the database by a non-programmer such as a sales executive. People who are not programmers are not expected to know how to build the most elegant solution to a query and will often write queries quite to the contrary.

- Ad-hoc queries can cause serious performance issues. Customer-facing applications that require millisecond response times (which depend solely on a high-performance OLTP database) do not get along well with ad-hoc queries. Don't risk losing your customers and wind up with no business to speak of. Do not allow anyone to do anything ad-hoc in an application-controlled OLTP database.
- *Support the objectives of the business* — Highly normalized table structures do not necessarily represent business structures directly. Denormalized, data warehouse, fact-dimensional structures tend to look a lot more like a business operationally. The latter is acceptable because a data warehouse is much more likely to be subjected to ad-hoc queries by management, business planning, and executive staff. Subjecting a customer-facing OLTP database to ad-hoc activity could be disastrous for operational effectiveness of the business. In other words, don't normalize a database model simply because the rules of normalization state this is the accepted practice. Different types of databases, and even different types of applications, are often better served with less application of normalization.
- *Provide adequate performance for any required change activity* — Be it single record changes in an OLTP database or high-speed batch operations in a data warehouse (or both), this is important.
- *Each table in a database model should preferably represent a single subject or topic* — Don't over-design a database model. Don't create too many tables. OLTP databases can thrive on more detail and more tables, but not always. Data warehouses can fall apart when data is divided up into too many tables.
- *Future growth must always be a serious consideration* — Some databases can grow at astronomical rates. Where data warehouse growth is potentially predictable from one load to the next, sometimes OLTP database growth can surprise you with sudden interest in an Internet site because of advertising, or just blind luck. When a sudden jump in online user interest increases load on an OLTP database astronomically, however, a database model that is not designed for potential astronomical growth could lose all newly acquired customers just as quickly as their interest was gained — overnight!

The computer jargon term commonly used to assess the potential future growth of a computer system is scalability. Is it scalable?

- *Future changes can be accommodated for, but potential structural changes can be difficult to allow for* — Parts of the various different types of database models naturally allow extension and enhancement. Some parts do not allow future changes easily. Some arguments for future growth state that more granularity and normalization are essential to allow for future growth, whereas other opinions can state exactly the opposite. This objective can often depend on company requirements. The problem with allowing for future growth in a database model is that it is much easier to allow for database size growth by adding new data. Adding new metadata

structures is also not necessarily a problem. On the contrary, changing existing structures can cause serious problems, particularly where relationships between tables change, and even sometimes simply adding new fields to tables. Any table changes can affect applications. The best way to deal with this issue is to code applications generically, but generic coding can affect overall performance. Other ways are to black box SQL database access code either in applications or the database.

The term "black box" implies chunks of code that can function independently, where changes made to one part of a piece of software will not affect others.

- Minimize dependence between applications and database model structures if you expect change. This makes it easier to change and enhance both database model and application code in the future.

Changes to underlying database model structure can cause huge maintenance costs. Minimizing dependence between application database access code and database model structures might help this process, but this can result in inefficient generic coding. No matter what, database model changes nearly always result in unpleasant application code changes. The important point is to build the application properly as well as the database model. Changes are unavoidable in applications when a database model is altered, but they can be adequately planned for.

Catering to all these objectives could cause you a real headache in designing your database model. They are only guidelines with possibilities both good and bad, and then all only potentially arising at one point or another. However, the positive results from using good database model design objectives are as follows:

- From an operational perspective, the most important objective is fulfilling the needs of applications. OLTP applications require rapid response times on small transactions and high concurrency levels — in other words, lots and lots of users, all doing the same stuff and at exactly the same time. A data warehouse has different requirements, of course, and a hybrid type of database a mixture of both.
- Queries should be relatively easy to code without producing errors because of lack of data integrity or poor table design. Table and relationship structures must be correct.
- The easier applications can be built, the better. In general, the less co-dependence between database model and application, the better. In tightly controlled OLTP application environments where no ad-hoc activity is permitted, this is easy. Where end-users are allowed to interact more directly with the database such as in a data warehouse, this becomes more difficult.
- Changing data and metadata is always an issue, and from an operational perspective, data changes are more important. Changing table structures would be nice if it were always easy, but metadata changes tend to affect applications adversely no matter how unglued applications and database structures are.

Strive for the best you can in the given circumstances, budget, and requirements.

That's ideally where you want to be when your database model design is built, implemented, and applications using your database are running and performing their tasks up to the operational expectations of the business. In other words, you are in business and business has improved substantially both in turnover and efficiency after your company has invested large sums of money in computerization.

Looking at Methods of Database Design

So far, you have looked at why a design process is required and why you need to define objectives to give the design process a goal at which to aim. So, the question you might be asking is how do you go about designing a database model?

There are various methodologies available for designing database models. Each of these different approaches consists of a number of steps. The following sequence of steps to database model design seems the most sensible for a book such as this.

- *Requirements analysis* — Collect information about the nature of the data, features required, and any specialized needs such as expected output responses. This step covers what is needed, so simply analyze it and write it down. Talk to the customer and company employees to get a better idea of exactly what they need.
- *Conceptual design* — This is where you get to use the fancy graphical tools and draw the pretty pictures — Entity Relationship Diagrams (ERDs). This step includes creation of tables, fields within those tables, and relationships between the tables. This step also includes normalization. Later chapters describe all aspects of conceptual design. Figure 1-10 shows a simple ERD for an online store selling books.
- *Logical design* — Create database language commands to generate table definitions. Some tools used for creating ERDs allow generation of data definition language (DDL) scripting; however, they are likely to generate generic scripts. Be sure that you check anything generated before executing in any specific database engine.

Data definition language (DDL) is made of the commands used to change metadata in a database, such as creating tables, changing tables, and dropping tables.

- *Physical design* — Adjust database language commands to alter the database model for the underlying physical attributes of tables. For example, you might want to store large binary objects in separate, underlying files to that of standard relational record-field data.
- *Tuning phase* — This step includes items such as appropriate indexing, further normalization, or even denormalization, security features, and anything else not covered by the previous steps.

These separate steps are interchangeable, repeatable, iterative, and really *anything-able,* according to various different approaches used for different database engines and different database designer personal preferences. Some designers may even put some of these steps into single steps and divide others up into more detailed sets of subset steps. In other words, this is all open to interpretation. The only thing I do insist that should be universal is that you draw the ERDs and build tables well before you build metadata table creation code, placing visual design prior to physical implementation.

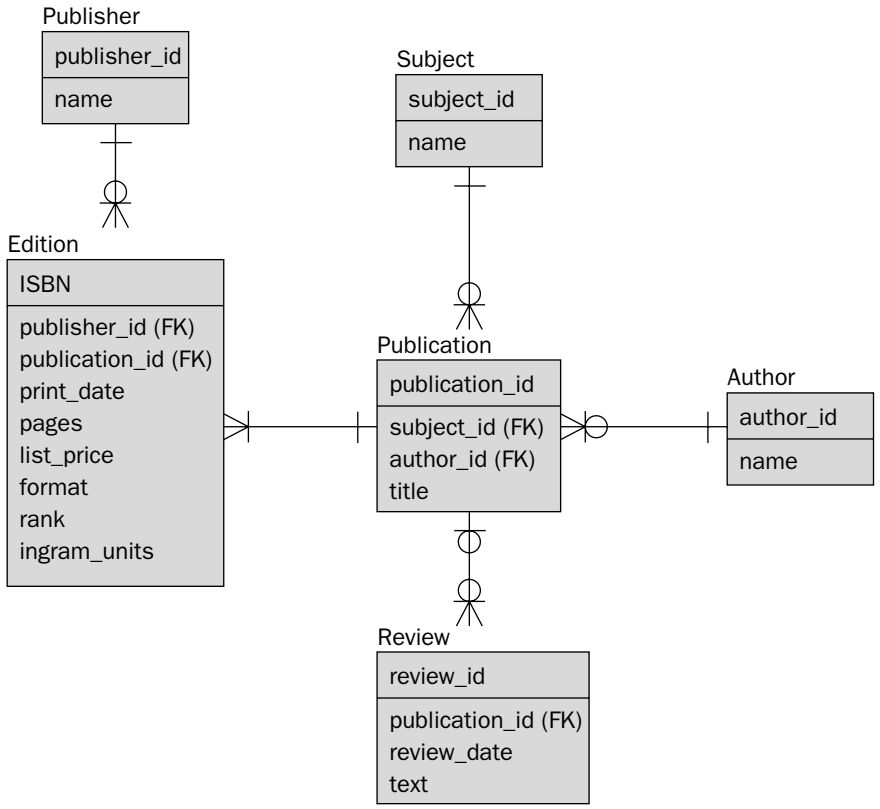

Figure 1-10: A simple online bookstore database model ERD

Summary

In this chapter, you learned about:

- The differences between a database, a database model, and an application
- The hierarchical and network database models
- The relational database model
- The object and object-relational database models
- Why different database models evolved
- The relational database model is the best all round option available
- Database design depends on applications
- Database types
- Database design objectives and methods

The next chapter discusses database modeling in the workplace, examining topics such as business rules, people, and unfavorable scenarios.