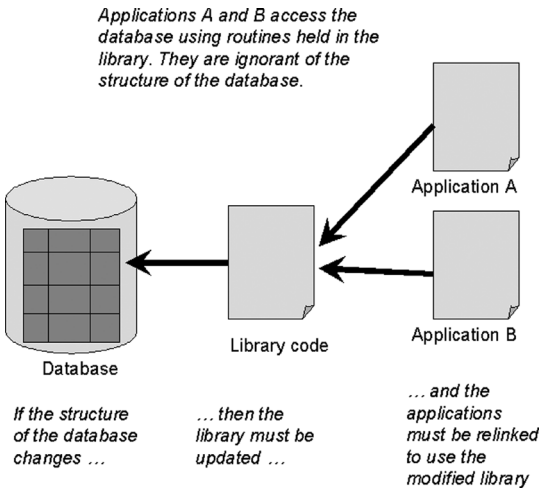
Databases—Ancient and Modern

If you already know all about relational databases, Structured Query Language (SQL), and Microsoft's strategy for universal data access, feel free to skip this lesson. For the rest of you out there—here's a little bit of history.

What is a Database Anyway?

A database is an organized collection of related data. The data needs to be organized in some way so you know how to gain access to it and find what you are looking for. The information in the database should also be relevant to other items in the database—you might record the names and address of customers that you have sold cars to in your sales database, but probably not the inside-leg measurements of customers' pet camels! (Strictly speaking, you might be tempted to argue that a database needs to be neither organized nor contain related data, but this makes it more of a "datadump" than a database).

Databases take many forms. You use databases all the time—the telephone directory and the television listings guide are two examples. In the world of computers, you are more likely to be concerned with electronic databases. Originally, a database was just a single file that had a predefined structure. A programmer who knew the structure could write programs to read the file, and find and update data. However, there is a problem with this approach. If the structure of the file changed, for example you wanted to record additional information not catered to by the field layout in the file, you would have to modify the program that reads and writes the file as well. If several programs shared the same data file, they would all need to be updated. You could put the data access code that your programs used in a separate library and just update that, but it is still not an ideal situation and you would still need to relink your programs to use the new version of the library.

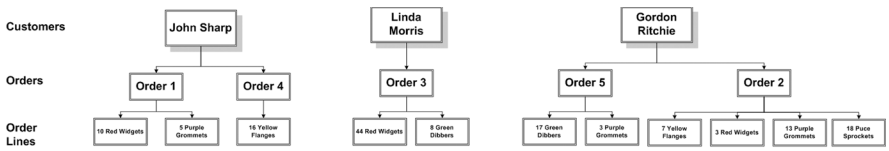


Some common standards and libraries evolved over the years for defining how data should be structured. In the 1950s, 1960s, and 1970s, the most common formats were the Hierarchical and the Network (or CODASYL) structures. These are jointly known as navigational systems.

A hierarchical database defines a tree structure. Related records are held as child nodes in the tree structure. The example shown in the following illustration shows part of a sales database—customers place orders, and an order comprises one or more order lines. Given a customer, you can quickly find all the orders for that customer, or given an order, you can locate the order lines.

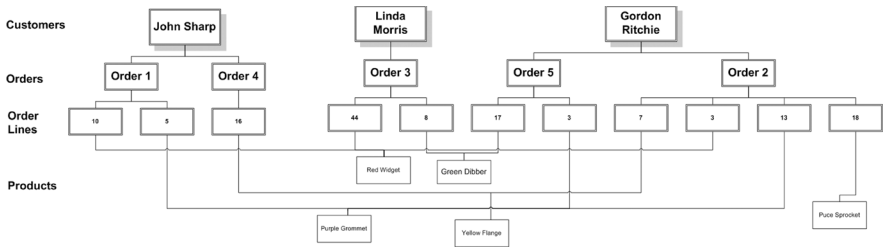
CODASYL—What's in a Name?

CODASYL stands for the Conference on Data Systems Languages. In the 1950s, the conference brought together representatives of the United States Department of Defense, government officials, and members of the business and commercial world who had a common interest in database processing. The initial aim was to propose a general purpose programming language for creating applications. This was achieved with the publication of the first COBOL proposals. However, CODASYL continued to function and, in 1965, the CODASYL List Processing Task Force was formed to closely examine navigational databases and make recommendations for extending COBOL to manipulate databases more easily. This eventually led to the creation of the Data Base Task Force and the development of the Network model.



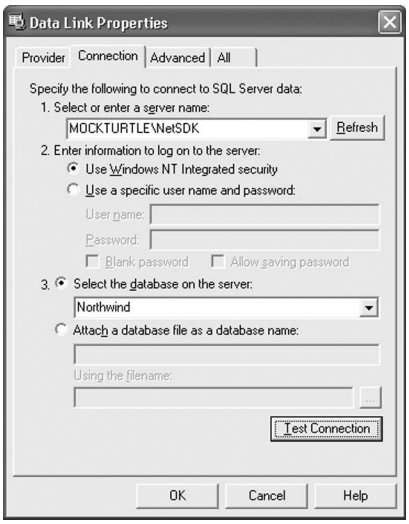
Hierarchical databases are easy to define and understand but they are not very flexible. The sales database is very good for queries such as "Tell me all the orders for Gordon Ritchie." However, it is highly inefficient for queries such as "Which customers have placed orders for red widgets?" You would have to navigate through each customer and examine their orders and order lines one by one. The Network model was developed by the CODASYL Data Base Task Force to combat this problem.

The term "network," used in this context, refers to the way in which the data items in a database were linked (or networked) together, allowing a program to navigate from one item of data to all other related items. The following illustration shows the same sales database structured as a Network database.



You can see that a new entity, Products, has been introduced together with the multitude of links connecting products to their related order lines. This allows you to run the "Which customers have placed orders for red widgets?" query much more quickly. However, this structure is also not very flexible. If you want to run other queries, you might have to define additional links between entities, and it all gets quite difficult to maintain and understand.

In the late 1960s and early 1970s, Edgar (Ted) Codd, a mathematician by trade, took matters into his own hands and developed the Relational Model. The Relational Model removed the idea of statically-linked data entities and replaced them with dynamically-deduced relationships. You specified how data in the various entities were related when you ran queries. This allowed incredible flexibility. Most developers these days are familiar with the Relational Model—entities are implemented as two-dimensional tables containing rows and columns. The following illustration shows the Sales database implemented as relational tables.



The majority of modern database servers are based on the Relational Model, including Microsoft SQL Server. Updates to the Relational Model over the years have added useful features such as referential integrity (for example, making sure you could not create orders for non-existent customers) and primary keys (which prevent you from accidentally adding the same data twice), but the underlying principles have not changed.

Understanding Structured Query Language

Alongside the Relational Model, Ted Codd also devised a language for manipulating relational data. This language went through several changes and eventually became Structured Query Language, or SQL as it is more commonly known. SQL was designed to be "English-like" (in the same way that COBOL was also designed to be "English-like"), the intention being to make it easy to read and write even for people who were not necessarily programmers. SQL contains three categories of commands—data manipulation language (DML), which you use for querying, inserting, updating, and deleting data in tables; data definition language (DDL), which you use for creating and managing tables and other database objects; and data control language (DCL), which you use for controlling access and security in databases. In this book, you will only use DML because the tables you will be using already exist.

There are four fundamental operations in DML. You can:

1 Query data by using a *SELECT* statement. For example:

```
SELECT CustomerName
FROM Customers
WHERE CustomerID = 1
```

This statement retrieves the contents of the CustomerName column from Customers table for all customers whose CustomerID is 1. In this case, this only fetches John Sharp. If there were more than one matching row (for example, the *WHERE* clause could have specified CustomerID > 1), Linda Morris and Gordon Ritchie would have been fetched instead.

tip

SQL is not case sensitive, but it is often a good idea to type SQL reserved words (such as *SELECT*, *DELETE*, and *FROM*) in uppercase letters so you (or whoever will be maintaining your code) can quickly read the statement and follow what it is doing at a later date.

2 Remove data by using a *DELETE* statement. For example:

```
DELETE FROM OrderLines
WHERE OrderLineID = 3
```

This statement removes OrderLine 3 from the OrderLines table in the database.

3 Modify data by using an *UPDATE* statement. For example:

```
UPDATE OrderLines
SET Volume = 100
WHERE OrderLineID = 5
```

This statement changes the volume ordered for order line 5.

4 Add new data by using an *INSERT* statement. For example:

```
INSERT INTO Products (ProductID, ProductName)
VALUES(6, "Pink Doofer")
```

This statement adds a new row to the Products table.

There is not enough space in this book to go through each of these different statements in detail. Instead, you will learn (or, if you already know SQL, be reminded of) the essential aspects of each statement as you use them in the exercises that follow.

Adopting a Standard

As well as making it easier to understand how to manipulate data in a database, another purpose of SQL is related to standards. Prior to SQL, each database manufacturer tended to implement its own programming interface that developers had to learn to access the database. This made the programs that used databases non-portable—if you wanted to change to a different database manufacturer, you would have to rewrite much of your code.

In the mid 1980s, the International Organization for Standardization (ISO) decided to adopt SQL as a standard. This meant that database vendors would all implement SQL as the standard application programming interface (API) for their systems and bring order to the previously chaotic world of databases. The original SQL standard contained a few "holes." Since the late 1980s, additional SQL standards have been proposed and adopted, adding more features to SQL and closing some of these holes. There are still some areas of database management that are not covered by SQL but, generally, most database vendors implement a very similar dialect of SQL.

Implementing Universal Data Access with ADO

The main issue with SQL is that it is a very high-level language. SQL statements have to be parsed and checked by database management systems, converted into low-level machine code, and executed, in much the same way that programs written in C# have to be compiled before you can run them. Originally, if you wanted to use SQL from a program written in another language, you had to "embed" the SQL in your code and run a preprocessor that would convert the embedded SQL into constructs that the programming language compiler could understand. To standardize operations at this lower level, the SQL Access Group (or SAG) set out to define a Call Level Interface (a programmers' API that could be used by programming languages and that would be understood by database management systems). As a result of this work, open database connectivity (ODBC) was born. Although ODBC was not strictly SAG's offering—it was developed by Microsoft, which was part of SAG—it became the de facto standard.

ODBC was fine at the time but it had its limitations. By the mid 1990s, Microsoft realized that there were other nonrelational databases (or data sources, as Microsoft prefers to call them) that did not fit into the ODBC model of processing. Microsoft therefore decided to create a new API that encompassed relational and non-relational databases and the concept of universal data access was born. Microsoft's architecture was called OLE DB (pronounced "Olay DeeBee") and comprised two parts. The upper part defined a "consumer"

interface and the lower part defined a "provider" interface. The bit in the middle was the OLE DB infrastructure supplied by Microsoft that connected the two halves together.

The idea was that data source vendors would provide access to their systems and implement the provider interface. Microsoft itself created an OLE DB provider that could convert OLE DB requests into ODBC calls, allowing continued (although a little slower) access to relational databases, even if the corresponding vendor had not yet implemented an OLE DB provider. Developers writing programs that needed to access a database could use the consumer interface. The consumer interface was still a bit low-level for many programmers, so Microsoft built another, more abstract layer on top, called ActiveX Data Objects or ADO.

One interesting aspect of ADO was that you could use it from Microsoft Visual Basic very easily, which made it very successful. (ODBC was more difficult to use because the API was really designed for C programmers, and the OLE DB consumer interface was designed for C++ programmers.) Abstractions in ADO included objects for handling connections to the data source, commands (possibly but not necessarily SQL statements), and Recordsets (data in a tabular format). ADO also exposed events, allowing a program to submit a command to a data source and then do something else while the command was processed. When the command completed, the program would be notified using an event, and the program could determine whether the command was completed successfully or not. Events were used in other parts of ADO as well.