1

History of Data Access

Over the years, many APIs have been released, all of which work toward the goal of providing universal data access. Universal data access is the concept of having a single code base for accessing data from any source, from any language.

Having universal data access is important for four reasons: First, developers can easily work on applications targeting different data stores without needing to become experts on each one. Second, developers can have a common framework for data access when switching between programming languages, making the transition to new languages easier. This is especially important in the .NET Framework, in which developers are expected to be able to easily switch between VB.NET and C#. Third, it enables developers to more easily write a single application that can be deployed against multiple data stores. Finally, it provides a level of abstraction between the application and direct communication to the database to simplify the code the average developer needs to write.

Microsoft has conducted surveys to determine which key factors companies are looking for in a data access layer. They came back with four main points, which they have tried to implement in their databases and data access components:

- **High performance** — As any developer knows, performance can make or break almost any application. No matter how much a data access layer may simplify accessing the data, it absolutely must perform nearly as well or better than the alternatives before it becomes a viable solution for the majority of applications.
- **High reliability** — If a component consumed by an application is buggy or occasionally stops working, it is perceived by the users as an error in that application. In addition to being a liability and annoyance to the company that implemented the application, it also reflects very poorly on the developer(s) who wrote the application. Any issues, such as memory leaks, that cause unreliable results are unacceptable to the development community. It's also very important to the support personnel that it be fairly maintenance-free. No one wants to have to reboot a server on a regular basis or constantly apply patches just to keep an application running.

- **Vendor commitment —** Without the widespread buy-in of vendors to build drivers/providers for their products, any universal data access model wouldn't be universal. Microsoft could provide the drivers for some of the most common vendor products, but it really takes an open, easily extensible model in order to gain widespread acceptance. No matter how much companies try to avoid it, almost all of them become "locked-in" to at least a handful of vendors. Switching to a vendor that supports the latest data access components is not really an option, so without widespread buy-in from vendors, a data access model cannot succeed.
- **Broad industry support —** This factor is along the same lines as vendor commitment, but includes a wider arena. It takes more than the data access model to be able to easily create good applications with it; it also requires good tools that can work with the data access model. Furthermore, it requires backing by several big players in the industry to reassure the masses. It also requires highly skilled people available to offer training. Finally, of course, it requires willing adoption by the development community so employers can find employees with experience.

Steady progress has been made, improving databases and universal data access over the last few decades. As with any field, it's important to know where we've come from in database and data access technologies in order to understand where the fields are heading. The following section looks at some early achievements.

The Early Days

In the 1950s and early 1960s, data access and storage was relatively simple for most people. While more advanced projects were under development and in use by a limited number of people, the majority of developers still stored data in flat text files. These were usually fixed-width files, and accessing them required no more than the capability to read and write files. Although this was a very simple technique for storing data, it didn't take too long to realize it wasn't the most efficient method in most cases.

CODASYL

As with the Internet, databases as we know them today began with the U.S. Department of Defense. In 1957, the U.S. Department of Defense founded the *Conference on Data Systems Languages*, commonly known as *CODASYL*, to develop computer programming languages. CODASYL is most famous for the creation of the COBOL programming language, but many people don't know that CODASYL is also responsible for the creation of the first modern database.

On June 10, 1963, two divisions of the U.S. Department of Defense held a conference titled "Development and Management of a Computer-Centered Data Base." At this conference, the term *database* was coined and defined as follows:

> **A set of files (tables), where a *file* is an ordered collection of entries (rows) and an *entry* consists of a key or keys and data.**

Two years later, in 1965, CODASYL formed a group called the List Processing Task Force, which later became the Data Base Task Group. The Data Base Task Group released an important report in 1971 outlining the *Network Data Model,* also known as the *CODASYL Data Model* or *DBTG Data Model*. This data model defined several key concepts of a database, including the following:

- A syntax for defining a schema
- A syntax for defining a subschema
- A data manipulation language

These concepts were later incorporated into the COBOL programming language. They also served as a base design for many subsequent data storage systems.

IMS

During the same period CODASYL was creating the Network Data Model, another effort was under way to create the first hierarchical database. During the space race, North American Rockwell won the contract to launch the first spacecraft to the moon. In 1966, members of IBM, North American Rockwell, and Caterpillar Tractor came together to begin the design and development of the Information Control System (ICS) and Data Language/I (DL/I). This system was designed to assist in tracking materials needed for the construction of the spacecraft.

The ICS portion of this system was the database portion responsible for storing and retrieving the data, while the DL/I portion was the query language needed to interface with it. In 1968, the IBM portion of this system (ICS) was renamed to *Information Management System*, or *IMS*. Over time, the DL/I portion was enhanced to provide features such as message queuing, and eventually became the transaction manager portion of IMS. IMS continued to evolve and was adopted by numerous major organizations, many of which still use it today.

Relational Databases

Both the Network Data Model from CODASYL and IMS from IBM were major steps forward because they marked the paradigm shift of separating data from application code, and they laid the framework for what a database should look like. However, they both had an annoying drawback: They expected programmers to navigate around the dataset to find what they wanted — thus, they are sometimes called *navigational databases*.

In 1970, Edgar Codd, a British computer scientist working for IBM, released an important paper called "A Relational Model of Data for Large Shared Data Banks" in which he introduced the *relational model*. In this model, Codd emphasized the importance of separating the raw, generic data types from the machine-specific data types, and exposing a simple, high-level query language for accessing this data. This shift in thinking would enable developers to perform operations against an entire data set at once instead of working with a single row at a time.

Within a few years, two systems were developed based on Codd's ideas. The first was an IBM project known as *System R*; the other was *Ingres* from the University of California at Berkeley. During the course of development for IBM's System R, a new query language known as *Structured Query Language (SQL)* was born. While System R was a great success for proving the relational database concept and creating SQL, it was never a commercial success for IBM. They did, however, release SQL/DS in 1980, which was a huge commercial success (and largely based on System R).

The Ingres project was backed by several U.S. military research agencies and was very similar to System R in many ways, although it ran on a different platform. One key advantage that Ingres had over System R that led to its longevity was the fact that the Ingres source code was publicly available, although it was later commercialized and released by Computer Associates in the 1980s.

Over the next couple of decades, databases continued to evolve. Modern databases such as Oracle, Microsoft SQL Server, MySQL, and LDAP are all highly influenced by these first few databases. They have improved greatly over time to handle very high transaction volume, to work with large amounts of data, and to offer high scalability and reliability.

The Birth of Universal Data Access

At first, there were no common interfaces for accessing data. Each data provider exposed an API or other means of accessing its data. The developer only had to be familiar with the API of the data provider he or she used. When companies switched to a new database system, any knowledge of how to use the old system became worthless and the developer had to learn a new system from scratch. As time went on, more data providers became available and developers were expected to have intimate knowledge of several forms of data access. Something needed to be done to standardize the way in which data was retrieved from various sources.

ODBC

Open Database Connectivity (ODBC) helped address the problem of needing to know the details of each DBMS used. ODBC provides a single interface for accessing a number of database systems. To accomplish this, ODBC provides a driver model for accessing data. Any database provider can write a driver for ODBC to access data from their database system. This enables developers to access that database through the ODBC drivers instead of talking directly to the database system. For data sources such as files, the ODBC driver plays the role of the engine, providing direct access to the data source. In cases where the ODBC driver needs to connect to a database server, the ODBC driver typically acts as a wrapper around the API exposed by the database server.

With this model, developers move from one DBMS to another and use many of the skills they have already acquired. Perhaps more important, a developer can write an application that doesn't target a specific database system. This is especially beneficial for vendors who write applications to be consumed by multiple customers. It gives customers the capability to choose the back-end database system they want to use, without requiring vendors to create several versions of their applications.

ODBC was a huge leap forward and helped to greatly simplify database-driven application development. It does have some shortfalls, though. First, it is only capable of supporting relational data. If you need to access a hierarchical data source such as LDAP, or semi-structured data, ODBC can't help you. Second, it can only handle SQL statements, and the result must be representable in the form of rows and columns. Overall, ODBC was a huge success, considering what the previous environment was like.

OLE-DB

Object Linking and Embedding Database (OLE-DB) was the next big step forward in data providers, and it is still widely used today. With OLE-DB, Microsoft applied the knowledge learned from developing ODBC to provide a better data access model. OLE-DB marked Microsoft's move to a COM-based API, which made it easily consumable by most programming languages, and the migration to a 32-bit OS with the release of Windows 95.

As with any code, ODBC became bulky through multiple revisions. The OLE-DB API is much cleaner and provides more efficient data access than ODBC. Oddly enough, the only provider offered with its initial release was the ODBC provider. It was just a wrapper of the ODBC provider and offered no performance gain. The point was to get developers used to the new API while making it possible to access any existing database system they were currently accessing through ODBC. Later, more efficient providers were written to access databases such as MS SQL Server directly, without going through ODBC.

OLE-DB Providers

OLE-DB is also much less dependent upon the physical structure of the database. It supports both relational and hierarchical data sources, and does not require the query against these data sources to follow a SQL structure. As with ODBC, vendors can create custom providers to expose access to their database system. Most people wouldn't argue with the belief that it is far easier to write an OLE-DB provider than an ODBC driver. A provider needs to perform only four basic steps:

1. Open the session.
2. Process the command.
3. Access the data.
4. Prepare a rowset.

OLE-DB Consumers

The other half of the OLE-DB framework is the OLE-DB consumer. The consumer is the layer that speaks directly to the OLE-DB providers, and it performs the following steps:

1. Identify the data source.
2. Establish a session.
3. Issue the command.
4. Return a rowset.

Figure 1-1 shows how this relationship works.

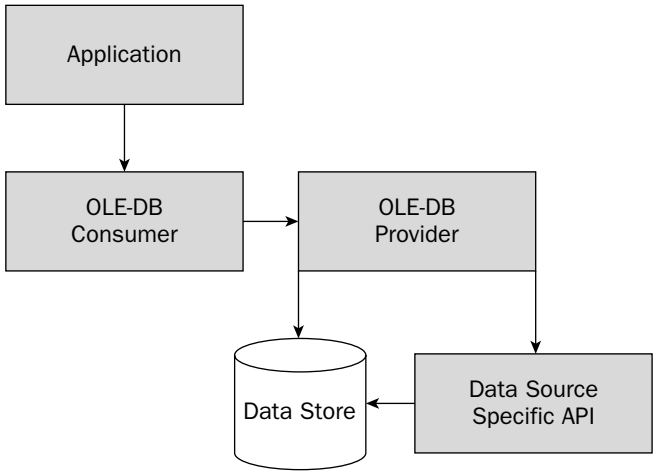

Figure 1-1

Data Access Consumers

Developers who use languages that support pointers — such as C, C++, VJ++, and so on — can speak directly to the ODBC and OLE-DB APIs. However, developers using a language such as Visual Basic need another layer. This is where the data access consumers such as DAO, RDO, ADO, and ADO.NET come into play.

DAO

With the release of Visual Basic 2.0, developers were introduced to a new method for accessing data, known as *Data Access Objects (DAO)*. This was Microsoft's first attempt to create a data consumer API. Although it had very humble beginnings, and when first released only supported forward-only operations against ODBC data sources, it was the beginning of a series of libraries that would lead developers closer to the ideal of Universal Data Access. It also helped developers using higher-level languages such as Visual Basic to take advantage of the power of ODBC that developers using lower-level languages such as C were beginning to take for granted.

DAO was based on the JET engine, which was largely designed to help developers take advantage of the desktop database application Microsoft was about to release, Microsoft Access. It served to provide another layer of abstraction between the application and data access, making the developer's task simpler. Although the initial, unnamed release with Visual Basic 2.0 only supported ODBC connections, the release of Microsoft Access 1.0 marked the official release of DAO 1.0, which supported direct communication with Microsoft Access databases without using ODBC. Figure 1-2 shows this relationship.

DAO 2.0 was expanded to support OLE-DB connections and the advantages that come along with it. It also provided a much more robust set of functionality for accessing ODBC data stores through the JET engine. Later, versions 2.5 and 3.0 were released to provide support for ODBC 2.0 and the 32-bit OS introduced with Windows 95.

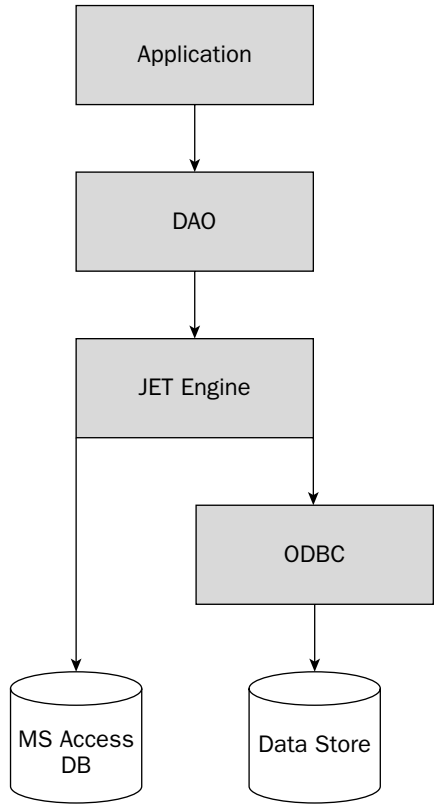

Figure 1-2

The main problem with DAO is that it can only talk to the JET engine. The JET engine then communicates with ODBC to retrieve the data. Going through this extra translation layer adds unnecessary overhead and makes accessing data through DAO slow.

RDO

Remote Data Objects (RDO) was Microsoft's solution to the slow performance created by DAO. For talking to databases other than Microsoft Access, RDO did not use the JET engine like DAO; instead, it communicated directly with the ODBC layer. Figure 1-3 shows this relationship.

Removing the JET engine from the call stack greatly improved performance to ODBC data sources. The JET engine was only used when accessing a Microsoft Access Database. In addition, RDO had the capability to use client-side cursors to navigate the records, as opposed to the server-side cursor requirements of DAO. This greatly reduced the load on the database server, enabling not only the application to perform better, but also the databases on which that application was dependant.

RDO was primarily targeted toward larger, commercial customers, many of whom avoided DAO due to the performance issues. Instead of RDO replacing DAO, they largely co-existed. This resulted for several reasons: First, users who developed smaller applications, where performance wasn't as critical, didn't want to take the time to switch over to the new API. Second, RDO was originally only released with the Enterprise Edition of Visual Basic, so some developers didn't have a choice. Third, with the release of

ODBCDirect, a DAO add-on that routed the ODBC requests through RDO instead of the JET engine, the performance gap between the two became much smaller. Finally, it wasn't long after the release of RDO that Microsoft's next universal access API was released.

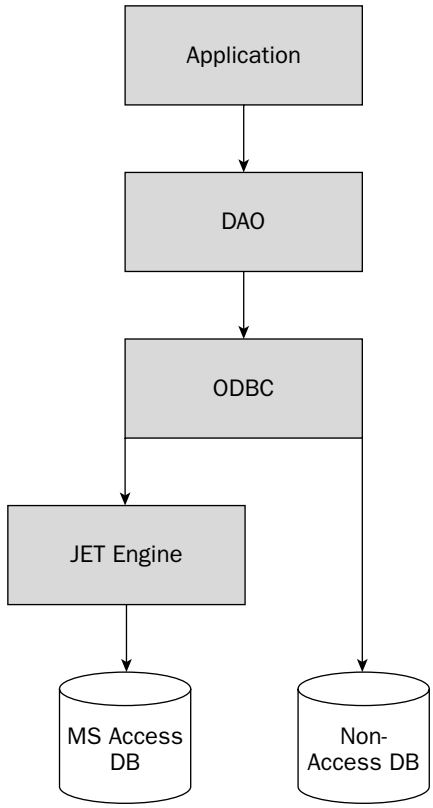

Figure 1-3

ADO

Microsoft introduced ActiveX Data Objects (ADO) primarily to provide a higher-level API for working with OLE-DB. With this release, Microsoft took many of the lessons from the past to build a lighter, more efficient, and more universal data access API. Unlike RDO, ADO was initially promoted as a replacement for both DAO and RDO. At the time of its release, it (along with OLE-DB) was widely believed to be a universal solution for accessing any type of data — from databases to e-mail, flat text files, and spreadsheets.

ADO represented a major shift from previous methods of data access. With DAO and RDO, developers were expected to navigate a tree of objects in order to build and execute queries. For example, to execute a simple insert query in RDO, developers couldn't just create an `rdoQuery` object and execute it. Instead, they first needed to create the `rdoEngine` object, then the `rdoEnvironment` as a child of it, then an `rdoConnection`, and finally the `rdoQuery`. It was a very similar situation with DAO. With ADO,

however, this sequence was much simpler. Developers could just create a `command` object directly, passing in the connection information and executing it. For simplicity and best practice, most developers would still create a separate `command` object, but for the first time the object could stand alone.

As stated before, ADO was primarily released to complement OLE-DB; however, ADO was not limited to just communicating with OLE-DB data sources. ADO introduced the provider model, which enabled software vendors to create their own providers relatively easily, which could then be used by ADO to communicate with a given vendor's data source and implement many of the optimizations specific to that data source. The ODBC provider that shipped with ADO was one example of this. When a developer connected to an ODBC data source, ADO would communicate through the ODBC provider instead of through OLE-DB. More direct communication to the data source resulted in better performance and an easily extensible framework. Figure 1-4 shows this relationship.

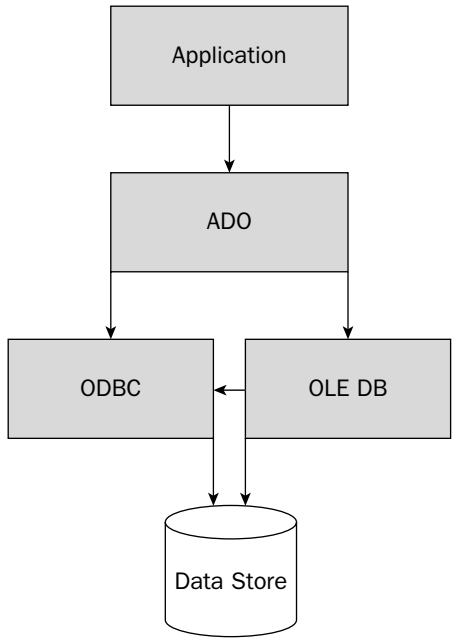

Figure 1-4

In addition to being a cleaner object model, ADO also offered a wider feature set to help lure developers away from DAO and RDO. These included the following:

- **Batch Updating** — For the first time, users enjoyed the capability to make changes to an entire recordset in memory and then persist these changes back to the database by using the `UpdateBatch` command.
- **Disconnected Data Access** — Although this wasn't available in the original release, subsequent releases offered the capability to work with data in a disconnected state, which greatly reduced the load placed on database servers.
- **Multiple Recordsets** — ADO provided the capability to execute a query that returns multiple recordsets and work with all of them in memory. This feature wasn't even available in ADO.NET until this release, now known as *Multiple Active Result Sets (MARS).*

In addition to all of the great advancements ADO made, it too had some shortcomings, of course. For example, even though it supported working with disconnected data, this was somewhat cumbersome. For this reason, many developers never chose to use this feature, while many others never even knew it existed. This standard practice of leaving the connection open resulted in heavier loads placed on the database server.

The developers who did choose to close the connection immediately after retrieving the data faced another problem: having to continually create and destroy connections in each method that needed to access data. This is a very expensive operation without the advantages of connection pooling that ADO.NET offers; and as a result, many best practice articles were published advising users to leave a single connection object open and forward it on to all the methods that needed to access data.

ADO.NET

With the release of the .NET Framework, Microsoft introduced a new data access model, called *ADO.NET*. The ActiveX Data Object acronym was no longer relevant, as ADO.NET was not ActiveX, but Microsoft kept the acronym due to the huge success of ADO. In reality, it's an entirely new data access model written in the .NET Framework.

ADO.NET supports communication to data sources through both ODBC and OLE-DB, but it also offers another option of using database-specific data providers. These data providers offer greater performance by being able to take advantage of data-source-specific optimizations. By using custom code for the data source instead of the generic ODBC and OLE-DB code, some of the overhead is also avoided. The original release of ADO.NET included a SQL provider and an OLE-DB provider, with the ODBC and Oracle providers being introduced later. Many vendors have also written providers for their databases since. Figure 1.5 shows the connection options available with ADO.NET.

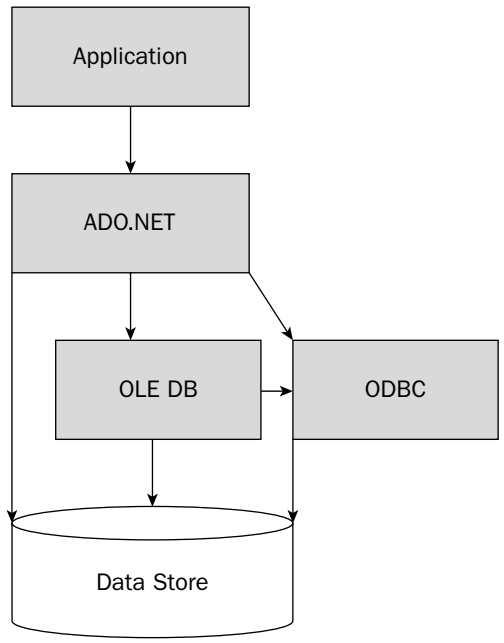

Figure 1-5

With ADO.NET, the days of the recordset and cursor are gone. The model is entirely new, and consists of five basic objects:

- **Connection —** The `Connection` object is responsible for establishing and maintaining the connection to the data source, along with any connection-specific information.
- **Command —** The `Command` object stores the query that is to be sent to the data source, and any applicable parameters.
- **DataReader —** The `DataReader` object provides fast, forward-only reading capability to quickly loop through the records.
- **DataSet —** The `DataSet` object, along with its child objects, is what really makes ADO.NET unique. It provides a storage mechanism for disconnected data. The `DataSet` never communicates with any data source and is totally unaware of the source of the data used to populate it. The best way to think of it is as an in-memory repository to store data that has been retrieved.
- **DataAdapter —** The `DataAdapter` object is what bridges the gap between the `DataSet` and the data source. The `DataAdapter` is responsible for retrieving the data from the `Command` object and populating the `DataSet` with the data returned. The `DataAdapter` is also responsible for persisting changes to the `DataSet` back to the data source.

ADO.NET made several huge leaps forward. Arguably, the greatest was the introduction of truly disconnected data access. Maintaining a connection to a database server such as MS SQL Server is an expensive operation. The server allocates resources to each connection, so it's important to limit the number of simultaneous connections. By disconnecting from the server as soon as the data is retrieved, instead of when the code is done working with that data, that connection becomes available for another process, making the application much more scalable.

Another feature of ADO.NET that greatly improved performance was the introduction of connection pooling. Not only is maintaining a connection to the database an expensive operation, but creating and destroying that connection is also very expensive. Connection pooling cuts down on this. When a connection is destroyed in code, the Framework keeps it open in a pool. When the next process comes around that needs a connection with the same credentials, it retrieves it from the pool, instead of creating a new one.

Several other advantages are made possible by the `DataSet` object. The `DataSet` object stores the data as XML, which makes it easy to filter and sort the data in memory. It also makes it easy to convert the data to other formats, as well as easily persist it to another data store and restore it again.

ADO.NET 2.0

Data access technologies have come a long way, but even with ADO.NET, there's still room to grow. The transition to ADO.NET 2.0 is not a drastic one. For the most part, Microsoft and the developers who use ADO.NET like it the way it is. In the 2.0 Framework, the basic design is the same, but several new features have been added to make common tasks easier, which is very good for backward compatibility. ADO.NET 2.0 should be 100% backwardly compatible with any ADO.NET 1.0 code you have written.

With any 2.0 product, the primary design goal is almost always to improve performance. ADO.NET 1.0 does not perform poorly by any means, but a few areas could use improvement, including XML serialization and connection pooling, which have been reworked to provide greater performance.

In the 2.0 Framework, Microsoft has also been able to improve performance by introducing several new features to reduce the number of queries that need to be run and to make it easier to run multiple queries at once. For example, the bulk insert feature provides the capability to add multiple rows to a database with a single query, instead of the current method of inserting one at a time. This can greatly reduce the amount of time it takes to insert a large number of rows.

Another example is the capability to be notified when data changes and to expire the cache only when this happens. This eliminates the need to periodically dump and reload a potentially large amount of data just in case something has changed. The introduction of *Multiple Active Result Sets (MARS)* provides the capability to execute multiple queries at once and receive a series of results. Removing the back and forth communication that is required by executing one query at a time and waiting for the results greatly improves the performance of an application that needs this functionality. If you prefer to do other work while waiting for your data to return, you also have the option of firing an asynchronous command. This has been greatly simplified in the 2.0 Framework.

Another major design goal is to reduce the amount of code necessary to perform common tasks. The buzz phrase we all heard with the release of .NET Framework 1.0 was "70 percent less code" than previous methods. The goal with the .NET 2.0 Framework is the same: to reduce the amount of code needed for common tasks by 70% over .NET 1.0. We'll leave the decision as to whether this goal was met or not to you, but after reading this book and using ADO.NET for awhile, you should notice a significant decrease in the amount of code needed to write your application.

The rest of the enhancements are primarily new features. For example, there is now a database discovery API for browsing the schema of a database. Also offered is the option of writing provider-independent database access code. This is very beneficial if you sell applications to customers who want to run it against numerous data sources. Keep in mind that the queries you write still must match that provider's syntax.

Summary

Now that you know some of the history behind how technologies such as ADO.NET and Microsoft SQL Server have evolved, you should have a clearer vision of where these technologies are heading. Throughout this book, we will cover the new features of these technologies in great depth and lay out the roadmap describing where many of them are heading. This release is just another major stepping-stone on the path to efficient universal data access.

For More Information

To complement the information in this chapter, take a look at the following resources:

- ***Funding a Revolution: Government Support for Computing Research,*** by the Computer Science and Telecommunications Board (CSTB), National Research Council. Washington, D.C.: National Academy Press, 1999. `www.nap.edu/execsumm/0309062780.html`.
- **Network (CODASYL) Data Model (Course Library) —** `http://coronet.iicm.edu/wbtmaster/allcoursescontent/netlib/library.htm`
- **"Technical Note — IMS Celebrates 30 Years as an IBM Product,"** by Kenneth R. Blackman. `www.research.ibm.com/journal/sj/374/blackman.html`.