

APPENDIX A

Relational Database Technology: A Crash Course

ADO.NET can be used to access data from any data source: relational databases, object databases, flat files, and text files. The vast majority of web applications, however, will access data from a relational database such as SQL Server. While one can certainly write an entire book on relational databases and another on SQL, the essentials of these technologies are not hard to understand.



All of the examples in this appendix assume you are working with SQL Server and that the flavor of SQL you are using is T-SQL. Users of other relational databases will find that the lessons learned here transfer well to their environment, but be especially careful with applications like Access that use a different variation of SQL.

A *database* is a repository of data. A *relational database* organizes your data into tables that are “related” to one another. For example, one table might contain a customer’s information and a second table might contain information about orders. The tables are related to one another because each customer has certain orders, and each order is owned by an individual customer.

Similarly, you might have a table of cars and a second table of car parts. Each part can be in one or more cars, and each car is made up of parts. Or, you might have a table for bugs and a table for developers. Each bug is owned by one developer, and each developer has a list of bugs he owns.

Tables, Records, and Columns

The principal division of a database is into tables. Tables, like classes, should describe one logical entity and all of what you know about that entity.

Every table in a relational database is organized into rows, where each row represents a single record. The rows are organized into columns. All the rows in a table have the same column structure. For example, the Bugs table described in

Appendix B (and used in Chapter 11) might have columns for the bugID, the ID of the person reporting the bug, the date the bug was reported, the status of the bug, and so forth.



It is common to make an analogy between tables and classes, and between rows and objects. The Bugs table, for example, tells you a great deal about the contents of a Bug, just as a Bug class tells you about the state and structure of a Bug. Each row in the Bug table describes a particular Bug, much as an object does.

This analogy is compelling, but limited. There is only an imperfect match between relational databases and objects, and one of the challenges facing an object-oriented programmer is overcoming the design differences between the object model, on the one hand, and the database model, on the other.

Relational databases are very good at defining the relationship among objects, but are not good at capturing the behavior of the types described in the table. The “impedance mismatch” between relational databases and object-oriented programs has led some developers to try to create object databases. While this has met with some success, the vast majority of data is still stored in relational databases because of their great flexibility, performance, and ability to be searched quickly and easily.

Typically, the interface between the back-end relational database and the objects in the application is managed by creating a database interface layer of objects that negotiate between the creation of objects and the storage of information in the database tables.

Table Design

To understand the issues in table design, consider the Bug database described in Chapter 11. You need to know who reported each bug, and it would be very useful to know the email address, phone number, and other identifying information about each person as well.

You can imagine a form in which you display details about a given bug, and in that detail page you offer the email address and phone number of the “reporter” so that the developer working on the bug can contact that person.

You could store the identifying information with each bug, but that would be very inefficient. If John Doe reported 50 bugs, you’d rather not repeat John Doe’s email address and phone number in 50 records. It’s also a data maintenance nightmare. If John Doe changes his email address and phone number, you’d have to make the change in 50 places.

Instead, you’ll create a second table called People, in which each row represents a single person. In the People table there will be a column for the PersonID. Each person will have a unique ID, and that field will be marked as the *primary key* for the

Person table. A primary key is the column or combination of columns that uniquely identifies a record in a given table.

The Bugs table will use the PersonID as a *foreign key*. A foreign key is a column (or combination of columns) that is a primary (or otherwise unique) key from a different table. The Bug table uses the PersonID, which is the primary key in People, to identify which person reported the bug. If you need later to determine the email address for that person, you can use the PersonID to look up the Person record in the People table and that will give you all the detailed information about that person.

By “factoring out” the details of the person’s address into a Person table, you reduce the redundant information in each Bug record. This process of taking out redundant information from your tables is called *normalization*.

Normalization

Normalization not only makes your use of the database more efficient, it reduces the likelihood of data corruption. If you kept the person’s email address both in the People table and also in the Bug table, you would run the risk that a change in one table might not be reflected in the other. Thus, if you changed the person’s email address in the Person table, that change might not be reflected in every row in the Bugs table (or it would be a lot of work to make sure that it was reflected). By keeping only the PersonID in Bugs, you are free to change the email address or other personal information in People, and the change will automatically be reflected for each bug.

Just as VB and C# programmers want the compiler to catch bugs at compile time rather than at runtime, database programmers want the database to help them avoid data corruption. A compiler helps avoid bugs by enforcing the rules of the language. For example, in C# you can’t use a variable you’ve not defined. SQL Server and other modern relational databases help you avoid bugs by enforcing constraints that you create. For example, the People database marks the PersonID as a primary key. This creates a primary key constraint in the database, which ensures that each PersonID is unique. If you were to enter a person named Jesse Liberty with the PersonID of LIBE, and then you were to try to add Stacey Liberty with a PersonID of LIBE, the database would reject the second record because of the primary key constraint. You would need to give one of these people a different, and unique, personID.

Declarative Referential Integrity

Relational databases use *Declarative Referential Integrity* (DRI) to establish constraints on the relationships among the various tables. For example, you might declare a constraint on the Bug table that dictates that no Bug may have a PersonID unless that PersonID represents a valid record in People. This helps you avoid two types of mistakes. First, you cannot enter a record with an invalid PersonID. Second, you cannot delete a Person record if that PersonID is used in any Bug. The integrity of your data and the relationships among records is thus protected.

SQL

The language of choice for querying and manipulating databases is *Structured Query Language*, often referred to as SQL. SQL is often pronounced “sequel.” SQL is a declarative language, as opposed to a procedural language, and it can take a while to get used to working with a declarative language if you are used to languages like VB or C#.

Most programmers tend to think in terms of a sequence of steps: “Find me all the bugs, then get the reporter’s ID, then use that ID to look up that user’s records in People, then get me the email address.” In a declarative language, you declare the entire query, and the query engine returns a set of results. You are not thinking about a set of steps; rather, you are thinking about designing and “shaping” a set of data. Your goal is to make a single declaration that will return the right records. You do that by creating temporary “wide” tables that include all the fields you need and then filtering for only those records you want. “Widen the Bugs table with the People table, joining the two on the PersonID, then filter for only those that meet my criteria.”

The heart of SQL is the *query*. A query is a statement that returns a set of records from the database. For example, you might like to see all of the BugIDs and Bug Descriptions in the Bugs table whose status is Open. To do so you would write:

```
Select BugID, BugDescription from Bugs where status = 'open'
```

SQL is capable of much more powerful queries. For example, suppose the Quality Assurance manager would like to know the email address for everyone who has reported a high-priority bug that was resolved in the past ten days. You might create a query such as:

```
Select emailAddress from Bugs b  
join People p on b.personID = p.personID  
where b.priority='high'  
and b.status in ('closed', 'fixed', 'NotABug')  
and b.dateModified < DateAdd(d,-10,GetDate( ))
```



GetDate returns the current date, and DateAdd returns a new date computed by adding or subtracting an interval from a specified date. In this case, you are returning the date computed by subtracting ten days from the current date.

At first glance, you appear to be selecting the email address from the Bugs table, but that is not possible because the Bugs table does not have an email address. The key phrase is:

```
Bugs b join People p on b.personID = p.personID
```

It is as if the join phrase creates a temporary table that is the width of both the Bugs table and the People table joined together. The on keyword dictates how the tables are joined. In this case, the tables are joined on the personID: each record in Bugs (represented by the alias b) is joined to the appropriate record in People (represented by the alias p) when the personID fields match in both records.

Joining Tables

When you join two tables you can say either “get every record that exists in either,” (this is called an *outer join*) or you can say, as we’ve done here, “get only those records that exist in both tables” (called an *inner join*).



Inner joins are the default, and so writing join is the same as writing inner join.

The inner join shown above says: get only the records in People that match the records in Bugs by having the same value in the PersonID field (on `b.PersonID = p.PersonID`).

The where clause further constrains the search to those records whose priority is high, whose status is one of the three that constitute a resolved Bug (closed, fixed, or not a bug), and that were last modified within the past ten days.

Using SQL to Manipulate the Database

SQL can be used not only for searching for and retrieving data but also for creating, updating, and deleting tables and generally managing and manipulating both the content and the structure of the database. For example, you can update the Priority of a bug in the Bugs table with this statement:

```
Update Bugs set priority = 'high' where BugID = 101
```

For a full explanation of SQL and details on using it well, take a look at *Transact-SQL Programming*, by Kevin Kline, Lee Gould, and Andrew Zanevsky (O’Reilly).