Fundamentals of Relational Database Design

Presented by: Paul Litwin

Paul Litwin is an independent developer, editor, writer, and educator. He's the owner of Seattle-based Litwin Consulting, a Microsoft Solution Provider specializing in Windows-based database solutions using Microsoft Access. He regularly teaches Microsoft Access programming to developers for Application Developers Training Company. Paul is also the editor of Smart Access, a monthly newsletter for Microsoft Access developers, as well as the author of the Jet Engine White Paper, and co-author of Microsoft Access 2 Developer's Handbook (Sybex, 1994).

This paper is adapted from Microsoft Access 2 Developer's Handbook, Sybex 1994, by Ken Getz, Paul Litwin and Greg Reddick. Reprinted with permission of the publisher.

Phone: (206) 281-1933 Fax: (206) 281-1669 Internet: 76447.417@compuserve.com

Overview

Database design theory is a topic that many people avoid learning for lack of time. Many others attempt to learn it, but give up because of the dry, academic treatment it is usually given by most authors and teachers. But if creating databases is part of your job, then you're treading on thin ice if you don't have a good solid understanding of relational database design theory.

This paper begins with an introduction to relational database design theory, including a discussion of keys, relationships, integrity rules, and the often-dreaded "Normal Forms." Following the theory, I present a practical step-by-step approach to good database design.

Note Most of the example tables discussed in this paper can be found in the sample Microsoft® Access database included on the Tech•Ed CD-ROM for this session: DESIGN.MDB.

The Relational Model

The relational database model was conceived by E. F. Codd in 1969, then a researcher at IBM. The model is based on branches of mathematics called set theory and predicate logic. The basic idea behind the relational model is that a database consists of a series of unordered tables (or relations) that can be manipulated using non-procedural operations that return tables. This model was in vast contrast to the more traditional database theories of the time that were much more complicated, less flexible and dependent on the physical storage methods of the data.

Note It is commonly thought that the word relational in the relational model comes from the fact that you relate together tables in a relational database. Although this is a convenient way to think of the term, it's not accurate. Instead, the word relational has its roots in the terminology that Codd used to define the relational model. The table in Codd's writings was actually referred to as a relation (a related set of information). In fact, Codd (and other relational database theorists) use the terms relations, attributes and tuples where most of us use the more common terms tables, columns and rows, respectively (or the more physical—and thus less preferable for discussions of database design theory—files, fields and records).

The relational model can be applied to both databases and database management systems (DBMS) themselves. The relational fidelity of database programs can be compared using Codd's 12 rules (since Codd's seminal paper on the relational model, the number of rules has been expanded to 300) for determining how DBMS products conform to the relational model. When compared with other database management programs, Microsoft Access fares quite well in terms of relational fidelity. Still, it has a long way to go before it meets all twelve rules completely.

Fortunately, you don't have to wait until Microsoft Access is perfect in a relational sense before you can benefit from the relational model. The relational model can also be applied to the design of databases, which is the subject of the remainder of this paper.

Relational Database Design

When designing a database, you have to make decisions regarding how best to take some system in the real world and model it in a database. This consists of deciding which tables to create, what columns they will contain, as well as the relationships between the tables. While it would be nice if this process was totally intuitive and obvious, or even better automated, this is simply not the case. A well-designed database takes time and effort to conceive, build and refine.

The benefits of a database that has been designed according to the relational model are numerous. Some of them are:

- Data entry, updates and deletions will be efficient.
- Data retrieval, summarization and reporting will also be efficient.
- Since the database follows a well-formulated model, it behaves predictably.
- Since much of the information is stored in the database rather than in the application, the database is somewhat self-documenting.
- Changes to the database schema are easy to make.

The goal of this paper is to explain the basic principles behind relational database design and demonstrate how to apply these principles when designing a database using Microsoft Access. This paper is by no means comprehensive and certainly not definitive. Many books have been written on database design theory; in fact, many careers have been devoted to its study. Instead, this paper is meant as an informal introduction to database design theory for the database developer.

Note While the examples in this paper are centered around Microsoft Access databases, the discussion also applies to database development using the Microsoft Visual Basic® programming system, the Microsoft FoxPro® database management system, and the Microsoft SQL Server™ client-server database management system.

Tables, Uniqueness and Keys

Tables in the relational model are used to represent "things" in the real world. Each table should represent only one thing. These things (or entities) can be real-world objects or events. For example, a real-world object might be a customer, an inventory item, or an invoice. Examples of events include patient visits, orders, and telephone calls. Tables are made up of rows and columns.

The relational model dictates that each row in a table be unique. If you allow duplicate rows in a table, then there's no way to uniquely address a given row via programming. This creates all sorts of ambiguities and problems that are best avoided. You guarantee uniqueness for a table by designating a primary key—a column that contains unique values for a table. Each table can have only one primary key, even though several columns or combination of columns may contain unique values. All columns (or combination of columns) in a table with unique values are referred to as candidate keys, from which the primary key must be drawn. All other candidate key columns are referred to as alternate keys. Keys can be simple or composite. A simple key is a key made up of one column, whereas a composite key is made up of two or more columns.

The decision as to which candidate key is the primary one rests in your hands—there's no absolute rule as to which candidate key is best. Fabian Pascal, in his book *SQL and Relational Basics*, notes that the decision should be based upon the principles of minimality (choose the fewest columns necessary), stability (choose a key that seldom changes), and simplicity/familiarity (choose a key that is both simple and familiar to users). Let's illustrate with an example. Say that a company has a table of customers called tblCustomer, which looks like the table shown in Figure 1.

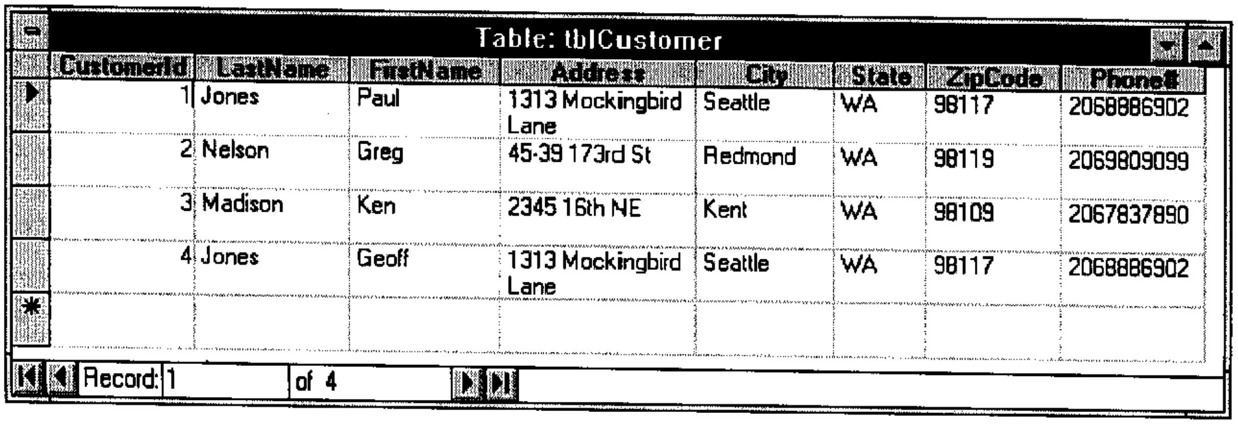


Table: tblCustomer

CustomerId	LastName	FirstName	Address	City	State	ZipCode	Phone#
1	Jones	Paul	1313 Mockingbird Lane	Seattle	WA	98117	2068886902
2	Nelson	Greg	45-39 173rd St	Redmond	WA	98119	2069809099
3	Madison	Ken	2345 16th NE	Kent	WA	98109	2067837890
4	Jones	Geoff	1313 Mockingbird Lane	Seattle	WA	98117	2068886902

Record: 1 of 4

Figure 1. The best choice for primary key for tblCustomer would be CustomerId.

Candidate keys for tblCustomer might include CustomerId, (LastName + FirstName), Phone#, (Address, City, State), and (Address + ZipCode). Following Pascal's guidelines, you would rule out the last three candidates because addresses and phone numbers can change fairly frequently. The choice among CustomerId and the name composite key is less obvious and would involve tradeoffs. How likely would a customer's name change (e.g., marriages cause names to change)? Will misspelling of names be common? How likely will two customers have the same first and last names? How familiar will CustomerId be to users? There's no right answer, but most developers favor numeric primary keys because names do sometimes change and because searches and sorts of numeric columns are more efficient than of text columns in Microsoft Access (and most other databases).

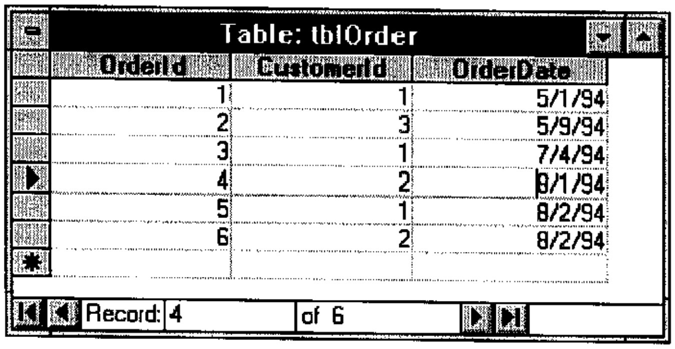
Counter columns in Microsoft Access make good primary keys, especially when you're having trouble coming up with good candidate keys, and no existing arbitrary identification number is already in place. Don't use a counter column if you'll sometimes need to renumber the values—you won't be able to—or if you require an alphanumeric code—Microsoft Access supports only long integer counter values. Also, counter columns only make sense for tables on the one side of a one-to-many relationship (see the discussion of relationships in the next section).

Note In many situations, it is best to use some sort of arbitrary static whole number (e.g., employee ID, order ID, a counter column, etc.) as a primary key rather than a descriptive text column. This avoids the problem of misspellings and name changes. Also, don't use real numbers as primary keys since they are inexact.

Foreign Keys and Domains

Although primary keys are a function of individual tables, if you created databases that consisted of only independent and unrelated tables, you'd have little need for them. Primary keys become essential, however, when you start to create relationships that join together multiple tables in a database. A foreign key is a column in a table used to reference a primary key in another table.

Continuing the example presented in the last section, let's say that you choose CustomerId as the primary key for tblCustomer. Now define a second table, tblOrder, as shown in Figure 2.



OrderId	CustomerId	OrderDate
1	1	5/1/94
2	3	5/9/94
3	1	7/4/94
4	2	8/1/94
5	1	8/2/94
6	2	8/2/94

Figure 2. CustomerId is a foreign key in tblOrder which can be used to reference a customer stored in the tblCustomer table.

CustomerId is considered a foreign key in tblOrder since it can be used to refer to given customer (i.e., a row in the tblCustomer table).

It is important that both foreign keys and the primary keys that are used to reference share a common meaning and draw their values from the same domain. Domains are simply pools of values from which columns are drawn. For example, CustomerId is of the domain of valid customer ID #'s, which in this case might be Long Integers ranging between 1 and 50,000. Similarly, a column named Sex might be based on a one-letter domain equaling 'M' or 'F'. Domains can be thought of as user-defined column types whose definition implies certain rules that the columns must follow and certain operations that you can perform on those columns.

Microsoft Access supports domains only partially. For example, Microsoft Access will not let you create a relationship between two tables using columns that do not share the same datatype (e.g., text, number, date/time, etc.). On the other hand, Microsoft Access will not prevent you from joining the Integer column EmployeeAge from one table to the Integer column YearsWorked from a second table, even though these two columns are obviously from different domains.

Relationships

You define foreign keys in a database to model relationships in the real world. Relationships between real-world entities can be quite complex, involving numerous entities each having multiple relationships with each other. For example, a family has multiple relationships between multiple people—all at the same time. In a relational database such as Microsoft Access, however, you consider only relationships between pairs of tables. These tables can be related in one of three different ways: one-to-one, one-to-many or many-to-many.

One-to-One Relationships

Two tables are related in a one-to-one (1–1) relationship if, for every row in the first table, there is at most one row in the second table. True one-to-one relationships seldom occur in the real world. This type of relationship is often created to get around some limitation of the database management software rather than to model a real-world situation. In Microsoft Access, one-to-one relationships may be necessary in a database when you have to split a table into two or more tables because of security or performance concerns or because of the limit of 255 columns per table. For example, you might keep most patient information in tblPatient, but put especially sensitive information (e.g., patient name, social security number and address) in tblConfidential (see Figure 3). Access to the information in tblConfidential could be more restricted than for tblPatient. As a second example, perhaps you need to transfer only a portion of a large table to some other application on a regular basis. You can split the table into the transferred and the non-transferred pieces, and join them in a one-to-one relationship.

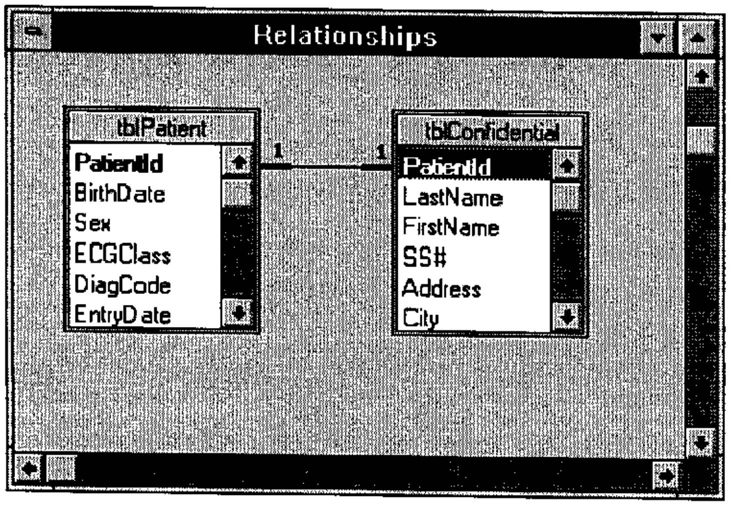


Figure 3. The tables tblPatient and tblConfidential are related in a one-to-one relationship. The primary key of both tables is PatientId.

Tables that are related in a one-to-one relationship should always have the same primary key, which will serve as the join column.

One-to-Many Relationships

Two tables are related in a one-to-many (1–M) relationship if for every row in the first table, there can be zero, one, or many rows in the second table, but for every row in the second table there is exactly one row in the first table. For example, each order for a pizza delivery business can have multiple items. Therefore, tblOrder is related to tblOrderDetails in a one-to-many relationship (see Figure 4). The one-to-many relationship is also referred to as a parent-child or master-detail relationship. One-to-many relationships are the most commonly modeled relationship.

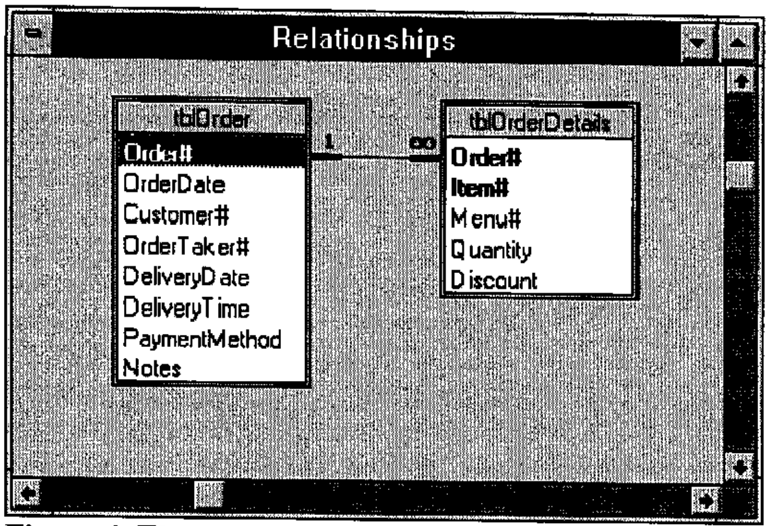


Figure 4. There can be many detail lines for each order in the pizza delivery business, so tblOrder and tblOrderDetails are related in a one-to-many relationship.

One-to-many relationships are also used to link base tables to information stored in lookup tables. For example, tblPatient might have a short one-letter DischargeDiagnosis code, which can be linked to a lookup table, tlkpDiagCode, to get more complete Diagnosis descriptions (stored in DiagnosisName). In this case, tlkpDiagCode is related to tblPatient in a one-to-many relationship (i.e., one row in the lookup table can be used in zero or more rows in the patient table).

Many-to-Many Relationships

Two tables are related in a many-to-many (M–M) relationship when for every row in the first table, there can be many rows in the second table, and for every row in the second table, there can be many rows in the first table. Many-to-many relationships can't be directly modeled in relational database programs, including Microsoft Access. These types of relationships must be broken into multiple one-to-many relationships. For example, a patient may be covered by multiple insurance plans and a given insurance company covers multiple patients. Thus, the tblPatient table in a medical database would be related to the tblInsurer table in a many-to-many relationship. In order to model the relationship between these two tables, you would create a third, linking table, perhaps called tblPtInsurancePgm that would contain a row for each insurance program under which a patient was covered (see Figure 5). Then, the many-to-many relationship between tblPatient and tblInsurer could be broken into two one-to-many relationships (tblPatient would be related to tblPtInsurancePgm and tblInsurer would be related to tblPtInsurancePgm in one-to-many relationships).

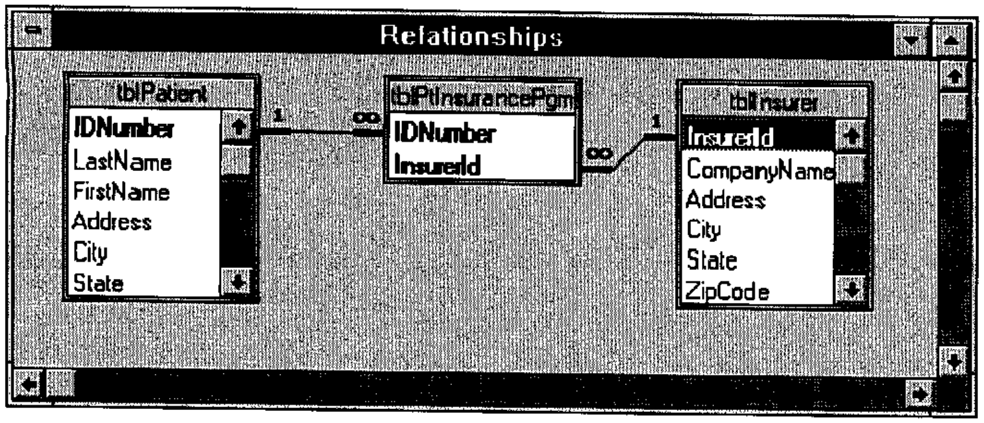


Figure 5. A linking table, tblPtInsurancePgm, is used to model the many-to-many relationship between tblPatient and tblInsurer.

In Microsoft Access, you specify relationships using the Edit–Relationships command. In addition, you can create ad-hoc relationships at any point, using queries.

Normalization

As mentioned earlier in this paper, when designing databases you are faced with a series of choices. How many tables will there be and what will they represent? Which columns will go in which tables? What will the relationships between the tables be? The answers each to these questions lies in something called normalization. Normalization is the process of simplifying the design of a database so that it achieves the optimum structure.

Normalization theory gives us the concept of normal forms to assist in achieving the optimum structure. The normal forms are a linear progression of rules that you apply to your database, with each higher normal form achieving a better, more efficient design. The normal forms are:

- First Normal Form
- Second Normal Form
- Third Normal Form
- Boyce Codd Normal Form
- Fourth Normal Form
- Fifth Normal Form

In this paper I will discuss normalization through Third Normal Form.

Before First Normal Form: Relations

The Normal Forms are based on relations rather than tables. A relation is a special type of table that has the following attributes:

1. They describe one entity.
2. They have no duplicate rows; hence there is always a primary key.
3. The columns are unordered.
4. The rows are unordered.

Microsoft Access doesn't require you to define a primary key for each and every table, but it strongly recommends it. Needless to say, the relational model makes this an absolute requirement. In addition, tables in Microsoft Access generally meet attributes 3 and 4. That is, with a few exceptions, the manipulation of tables in Microsoft Access doesn't depend upon a specific ordering of columns or rows. (One notable exception is when you specify the data source for a combo or list box.)

For all practical purposes the terms table and relation are interchangeable, and I will use the term table in the remainder of this chapter. It's important to note, however, that when I use the term table, I actually mean a table that also meets the definition of a relation.

First Normal Form

First Normal Form (1NF) says that all column values must be atomic. The word atom comes from the Latin *atomis*, meaning indivisible (or literally "not to cut"). 1NF dictates that, for every row-by-column position in a given table, there exists only one value, not an array or list of values. The benefits from this rule should be fairly obvious. If lists of values are stored in a single column, there is no simple way to manipulate those values. Retrieval of data becomes much more laborious and difficult to generalize. For example, the table in Figure 6, tblOrder1, used to store order records for a hardware store, would violate 1NF:

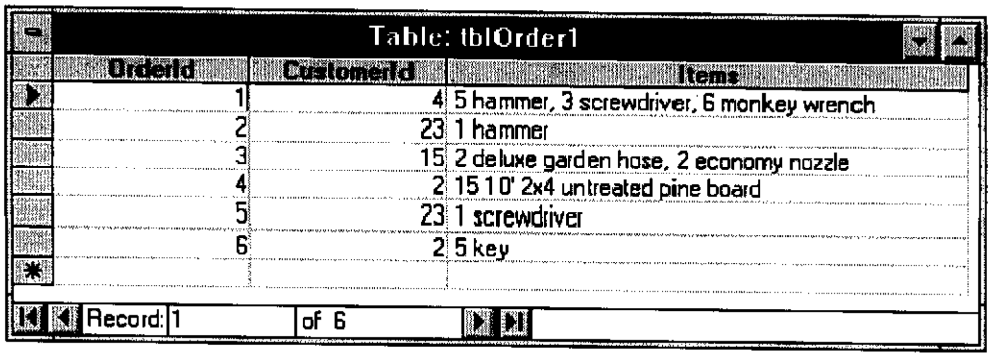


Table: tblOrder1

OrderId	CustomerId	Items
1	4	5 hammer, 3 screwdriver, 6 monkey wrench
2	23	1 hammer
3	15	2 deluxe garden hose, 2 economy nozzle
4	2	15 1 0' 2x4 untreated pine board
5	23	1 screwdriver
6	2	5 key

Record: 1 of 6

Figure 6. tblOrder1 violates First Normal Form because the data stored in the Items column is not atomic.

You'd have a difficult time retrieving information from this table, because too much information is being stored in the Items field. Think how difficult it would be to create a report that summarized purchases by item.

1NF also prohibits the presence of repeating groups, even if they are stored in composite (multiple) columns. For example, the same table might be improved upon by replacing the single Items column with six columns: Quant1, Item1, Quant2, Item2, Quant3, Item3 (see Figure 7).

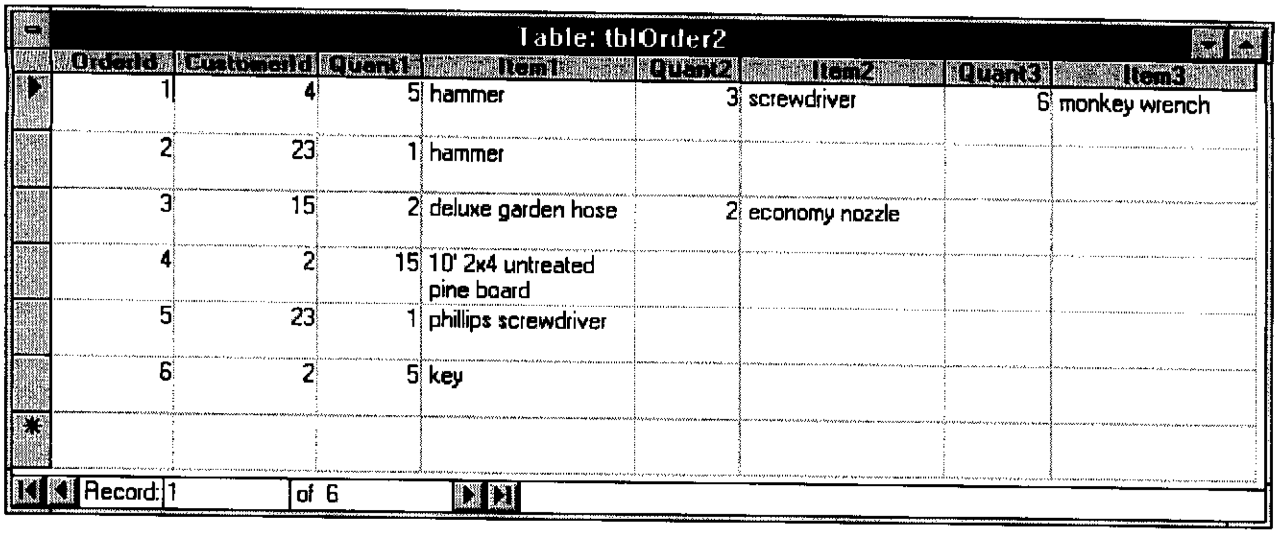


Table: tblOrder2

OrderId	CustomerId	Quant1	Item1	Quant2	Item2	Quant3	Item3
1	4	5	hammer	3	screwdriver	6	monkey wrench
2	23	1	hammer				
3	15	2	deluxe garden hose	2	economy nozzle		
4	2	15	10' 2x4 untreated pine board				
5	23	1	phillips screwdriver				
6	2	5	key				

Record: 1 of 6

Figure 7. A better, but still flawed, version of the Orders table, tblOrder2. The repeating groups of information violate First Normal Form.

While this design has divided the information into multiple fields, it's still problematic. For example, how would you go about determining the quantity of hammers ordered by all customers during a particular month? Any query would have to search all three Item columns to determine if a hammer was purchased and then sum over the three quantity columns. Even worse, what if a customer ordered more than three items in a single order? You could always add additional columns, but where would you stop? Ten items, twenty items? Say that you decided that a customer would never order more than twenty-five items in any one order and designed the table accordingly. That means you would be using 50 columns to store the item and quantity information per record, even for orders that only involved one or two items. Clearly this is a waste of space. And someday, someone would want to order more than 25 items.

Tables in 1NF do not have the problems of tables containing repeating groups. The table in Figure 8, tblOrder3, is 1NF since each column contains one value and there are no repeating groups of columns. In order to attain 1NF, I have added a column, OrderItem#. The primary key of this table is a composite key made up of OrderId and OrderItem#.

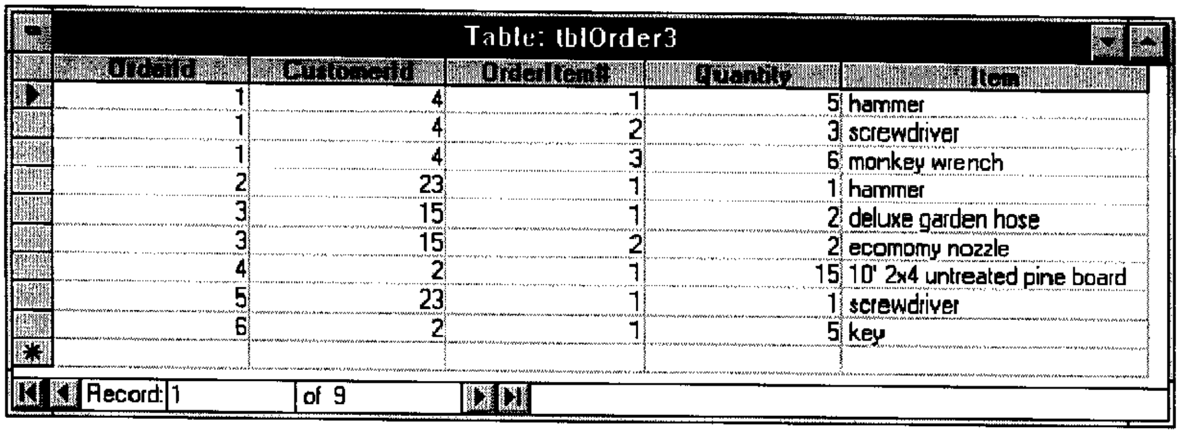


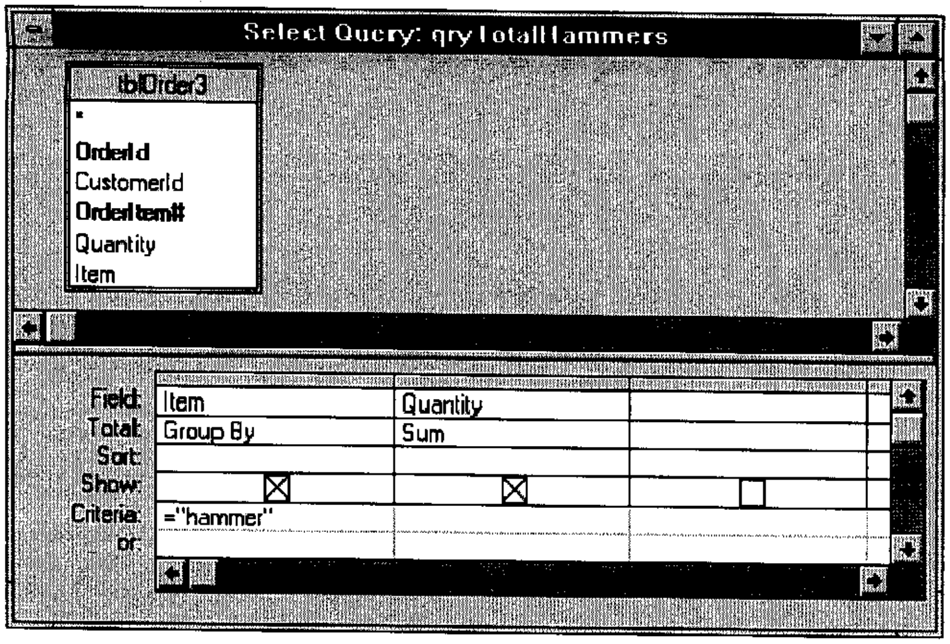
Table: tblOrder3

OrderId	CustomerId	OrderItem#	Quantity	Item
1	4	1	5	hammer
1	4	2	3	screwdriver
1	4	3	6	monkey wrench
2	23	1	1	hammer
3	15	1	2	deluxe garden hose
3	15	2	2	ecomomy nozzle
4	2	1	15	10' 2x4 untreated pine board
5	23	1	1	screwdriver
6	2	1	5	key

Record: 1 of 9

Figure 8. The tblOrder3 table is in First Normal Form.

You could now easily construct a query to calculate the number of hammers ordered. The query in Figure 9 is an example of such a query.


Select Query: qryTotalHammers

tblOrder3
*
OrderId
CustomerId
OrderItem#
Quantity
Item

Field:	Item	Quantity	
Total:	Group By	Sum	
Sort:			
Show:	☒	☒	☐
Criteria:	="hammer"		
or:			

Figure 9. Since tblOrder3 is in First Normal Form, you can easily construct a Totals query to determine the total number of hammers ordered by customers.

Second Normal Form

A table is said to be in Second Normal Form (2NF), if it is in 1NF and every non-key column is fully dependent on the (entire) primary key. Put another way, tables should only store data relating to one "thing" (or entity) and that entity should be described by its primary key.

The table shown in Figure 10, tblOrder4, is slightly modified version of tblOrder3. Like tblOrder3, tblOrder4 is in First Normal Form. Each column is atomic, and there are no repeating groups.

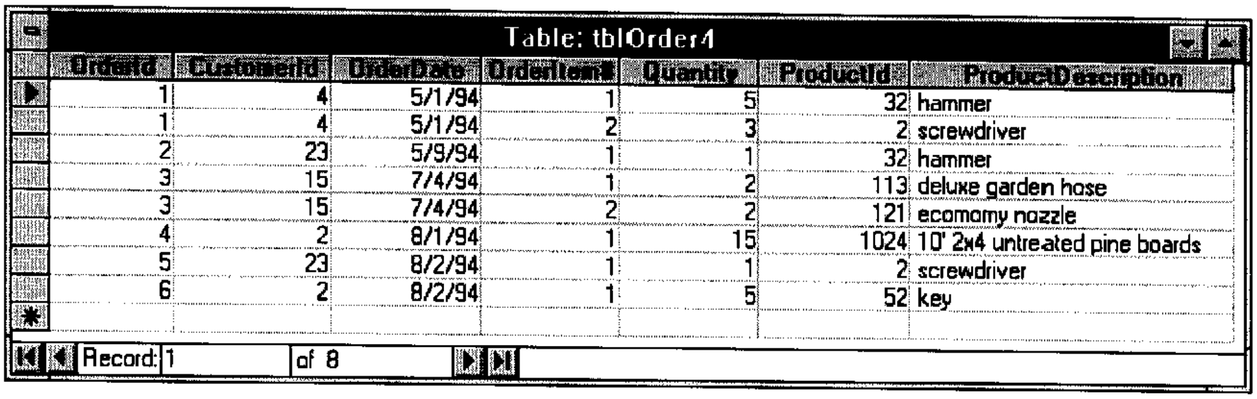


Table: tblOrder4

OrderId	CustomerId	OrderDate	OrderItem#	Quantity	ProductId	ProductDescription
1	4	5/1/94	1	5	32	hammer
1	4	5/1/94	2	3	2	screwdriver
2	23	5/9/94	1	1	32	hammer
3	15	7/4/94	1	2	113	deluxe garden hose
3	15	7/4/94	2	2	121	ecomomy nozzle
4	2	8/1/94	1	15	1024	10' 2x4 untreated pine boards
5	23	8/2/94	1	1	2	screwdriver
6	2	8/2/94	1	5	52	key

Record: 1 of 8

Figure 10. The tblOrder4 table is in First Normal Form. Its primary key is a composite of OrderId and OrderItem#.

To determine if tblOrder4 meets 2NF, you must first note its primary key. The primary key is a composite of OrderId and OrderItem#. Thus, in order to be 2NF, each non-key column (i.e., every column other than OrderId and OrderItem#) must be fully dependent on the primary key. In other words, does the value of OrderId and OrderItem# for a given record imply the value of every other column in the table? The answer is no. Given the OrderId, you know the customer and date of the order, *without* having to know the OrderItem#. Thus, these two columns are not dependent on the *entire* primary key which is composed of both OrderId and OrderItem#. For this reason tblOrder4 is not 2NF.

You can achieve Second Normal Form by breaking tblOrder4 into two tables. The process of breaking a non-normalized table into its normalized parts is called decomposition. Since tblOrder4 has a composite primary key, the decomposition process is straightforward. Simply put everything that applies to each *order* in one table and everything that applies to each *order item* in a second table. The two decomposed tables, tblOrder and tblOrderDetail, are shown in Figure 11.

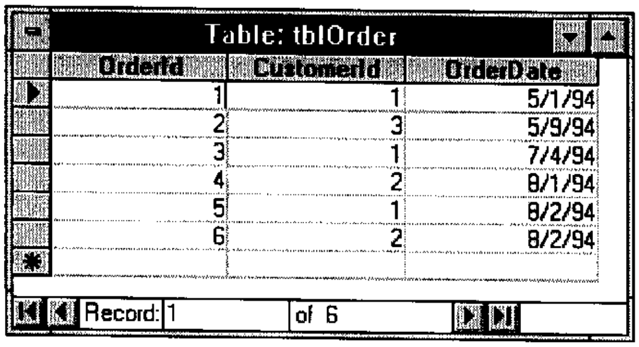


Table: tblOrder

OrderId	CustomerId	OrderDate
1	1	5/1/94
2	3	5/9/94
3	1	7/4/94
4	2	8/1/94
5	1	8/2/94
6	2	8/2/94

Record: 1 of 6

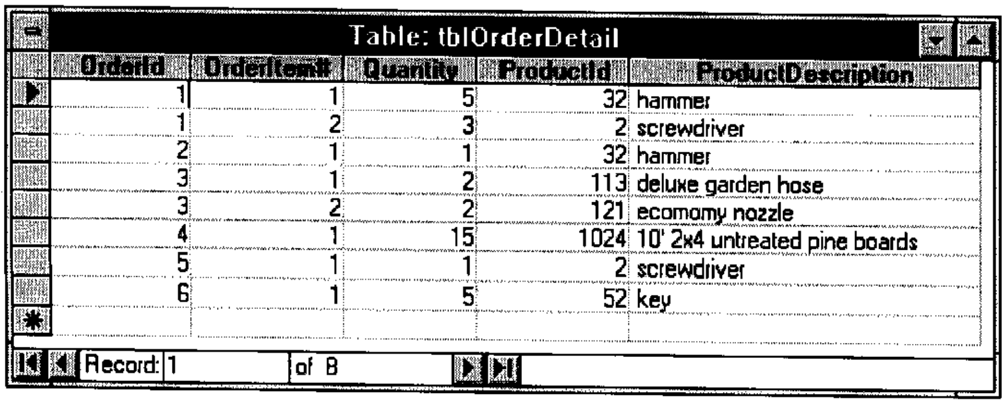


Table: tblOrderDetail

OrderId	OrderItem#	Quantity	ProductId	ProductDescription
1	1	5	32	hammer
1	2	3	2	screwdriver
2	1	1	32	hammer
3	1	2	113	deluxe garden hose
3	2	2	121	ecomomy nozzle
4	1	15	1024	10' 2x4 untreated pine boards
5	1	1	2	screwdriver
6	1	5	52	key

Record: 1 of 8

Figure 11. The tblOrder and tblOrderDetail tables satisfy Second Normal Form. OrderId is a foreign key in tblOrderDetail that you can use to rejoin the tables.

Two points are worth noting here.

- When normalizing, you don't throw away information. In fact, this form of decomposition is termed *non-loss decomposition* because no information is sacrificed to the normalization process.
- You decompose the tables in such a way as to allow them to be put back together again using queries. Thus, it's important to make sure that tblOrderDetail contains a foreign key to tblOrder. The foreign key in this case is OrderId which appears in both tables.

Third Normal Form

A table is said to be in Third Normal Form (3NF), if it is in 2NF and if all non-key columns are mutually independent. An obvious example of a dependency is a calculated column. For example, if a table contains the columns Quantity and PerItemCost, you could opt to calculate and store in that same table a TotalCost column (which would be equal to Quantity*PerItemCost), but this table wouldn't be 3NF. It's better to leave this column out of the table and make the calculation in a query or on a form or a report instead. This saves room in the database and avoids having to update TotalCost, every time Quantity or PerItemCost changes.

Dependencies that aren't the result of calculations can also exist in a table. The tblOrderDetail table from Figure 11, for example, is in 2NF because all of its non-key columns (Quantity, ProductId and ProductDescription) are fully dependent on the primary key. That is, given an OderID and an OrderItem#, you know the values of Quantity, ProductId and ProductDescription. Unfortunately, tblOrderDetail also contains a dependency among two if its non-key columns, ProductId and ProductDescription.

Dependencies cause problems when you add, update, or delete records. For example, say you need to add 100 detail records, each of which involves the purchase of screwdrivers. This means you would have to input a ProductId code of 2 *and* a ProductDescription of "screwdriver" for each of these 100 records. Clearly this is redundant. Similarly, if you decide to change the description of the item to "No. 2 Phillips-head screwdriver" at some later time, you will have to update all 100 records. Another problem arises when you wish to delete all of the 1994 screwdriver purchase records at the end of the year. Once all of the records are deleted, you will no longer know what ProductId of 2 is, since you've deleted from the database both the history of purchases and the fact that ProductId 2 means "No. 2 Phillips-head screwdriver." You can remedy each of these anomalies by further normalizing the database to achieve Third Normal Form.

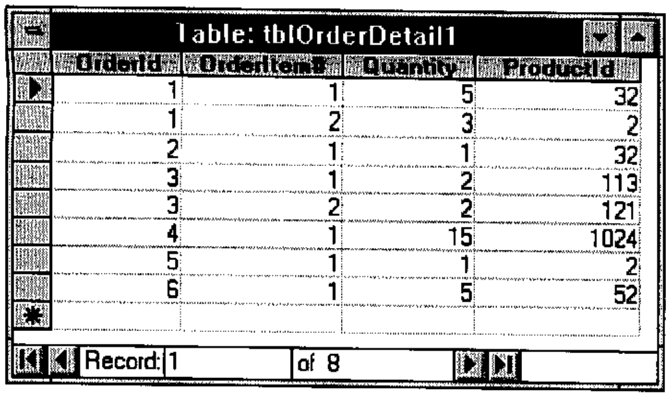
Note An Anomaly is simply an error or inconsistency in the database. A poorly designed database runs the risk of introducing numerous anomalies. There are three types of anomalies:

• **Insertion**—an anomaly that occurs during the insertion of a record. For example, the insertion of a new row causes a calculated total field stored in another table to report the wrong total.

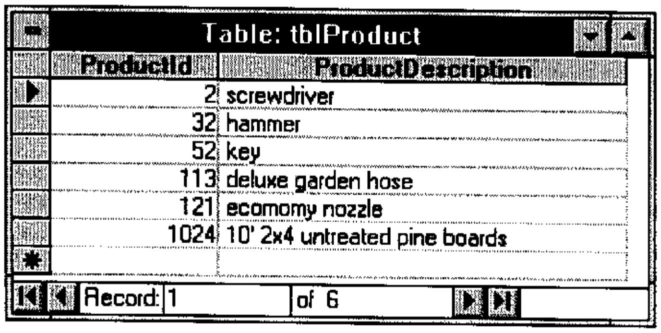
• **Deletion**—an anomaly that occurs during the deletion of a record. For example, the deletion of a row in the database deletes more information than you wished to delete.

• **Update**—an anomaly that occurs during the updating of a record. For example, updating a description column for a single part in an inventory database requires you to make a change to thousands of rows.

The tblOrderDetail table can be further decomposed to achieve 3NF by breaking out the ProductId–ProductDescription dependency into a lookup table as shown in Figure 12. This gives you a new order detail table, tblOrderDetail1 and a lookup table, tblProduct. When decomposing tblOrderDetail, take care to put a copy of the linking column, in this case ProductId, in both tables. ProductId becomes the primary key of the new table, tblProduct, and becomes a foreign key column in tblOrderDetail1. This allows you to easily join together the two tables using a query.



OrderId	OrderItem#	Quantity	ProductId
1	1	5	32
1	2	3	2
2	1	1	32
3	1	2	113
3	2	2	121
4	1	15	1024
5	1	1	2
6	1	5	52



ProductId	ProductDescription
2	screwdriver
32	hammer
52	key
113	deluxe garden hose
121	ecomomy nozzle
1024	10' 2x4 untreated pine boards

Figure 12. The tbOrderDetail1 and tblProduct tables are in Third Normal Form. The ProductId column in tblOrderDetail1 is a foreign key referencing tblProduct.

Higher Normal Forms

After Codd defined the original set of normal forms it was discovered that Third Normal Form, as originally defined, had certain inadequacies. This led to several higher normal forms, including the Boyce/Codd, Fourth and Fifth Normal Forms. I will not be covering these higher normal forms, instead I direct you to the books listed at the end of this paper. Still, several points are worth noting here:

- Every higher normal form is a superset of all lower forms. Thus, if your design is in Third Normal Form, by definition it is also in 1NF and 2NF.
- If you've normalized your database to 3NF, you've likely also achieved Bocye/Codd Normal Form (and maybe even 4NF or 5NF).
- To quote C.J. Date, the principles of database design are "nothing more than *formalized common sense*."
- Database design is more art than science.

This last item needs to be emphasized. While it's relatively easy to work through the examples in this paper, the process gets more difficult when you are presented with a business problem (or another scenario) that needs to be computerized (or downsized). I have outlined an approach to take later in this paper, but first the subject of integrity rules will be discussed.

Integrity Rules

The relational model defines several integrity rules that, while not part of the definition of the Normal Forms are nonetheless a necessary part of any relational database. There are two types of integrity rules: general and database-specific.

General Integrity Rules

The relational model specifies two general integrity rules. They are referred to as general rules, because they apply to all databases. They are: entity integrity and referential integrity.

The entity integrity rule is very simple. It says that primary keys cannot contain null (missing) data. The reason for this rule should be obvious. You can't uniquely identify or reference a row in a table, if the primary key of that table can be null. It's important to note that this rule applies to both simple and composite keys. For composite keys, none of the individual columns can be null. Fortunately, Microsoft Access automatically enforces the entity integrity rule for you. No component of a primary key in Microsoft Access can be null.

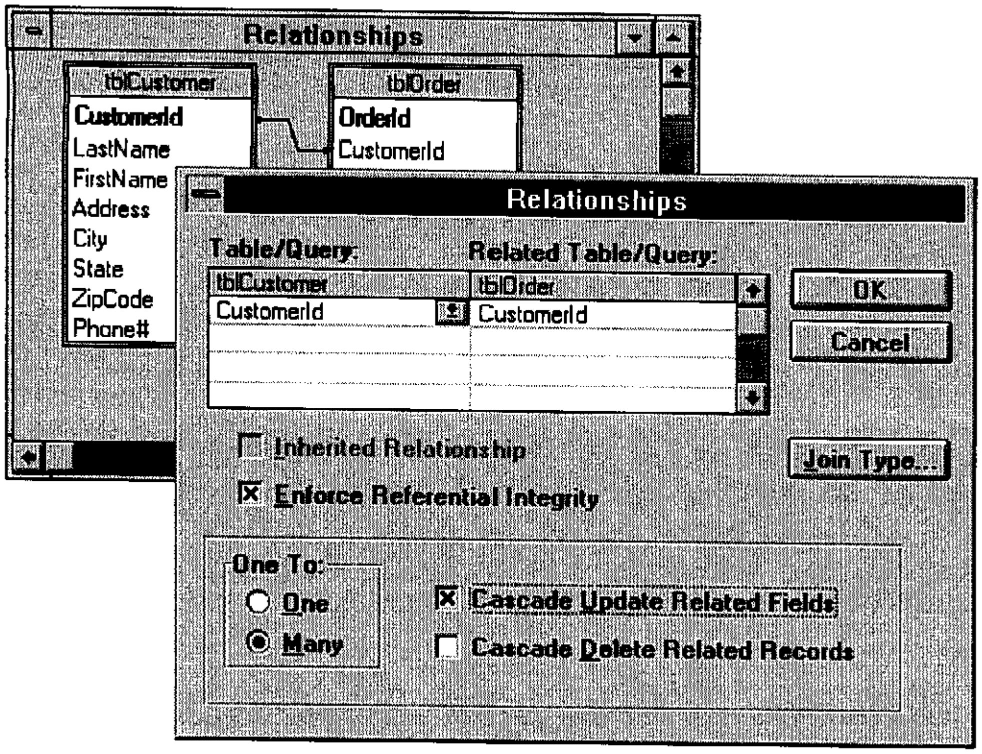
The referential integrity rule says that the database must not contain any unmatched foreign key values. This implies that:

- A row may not be added to a table with a foreign key unless the referenced value exists in the referenced table.
- If the value in a table that's referenced by a foreign key is changed (or the entire row is deleted), the rows in the table with the foreign key must not be "orphaned."

In general, there are three options available when a referenced primary key value changes or a row is deleted. The options are:

- **Disallow**. The change is completely disallowed.
- **Cascade**. For updates, the change is cascaded to all dependent tables. For deletions, the rows in all dependent tables are deleted.
- **Nullify**. For deletions, the dependent foreign key values are set to Null.

Microsoft Access allows you to disallow or cascade referential integrity updates and deletions using the Edit|Relationships command (see Figure 13). Nullify is not an option.



tblCustomer: CustomerId, LastName, FirstName, Address, City, State, ZipCode, Phone#

tblOrder: OrderId, CustomerId

Table/Query:	Related Table/Query:
tblCustomer	tblOrder
CustomerId	CustomerId

[ ] Inherited Relationship

[✓] Enforce Referential Integrity

One To: ( ) One [✓ Many]

[✓] Cascade Update Related Fields

[ ] Cascade Delete Related Records

Figure 13. Specifying a relationship with referential integrity between the tblCustomer and tblOrder tables using the Edit|Relationships command. Updates of CustomerId in tblCustomer will be cascaded to tblOrder. Deletions of rows in tblCustomer will be disallowed if rows in tblOrders would be orphaned.

Note When you wish to implement referential integrity in Microsoft Access, you must perform one additional step outside of the Edit|Relationships dialog: in table design, you must set the Required property for the foreign key column to Yes. Otherwise, Microsoft Access will allow your users to enter a Null foreign key value, thus violating strict referential integrity.

Database-Specific Integrity Rules

All integrity constraints that do not fall under entity integrity or referential integrity are termed database-specific rules or business rules. These type of rules are specific to each database and come from the rules of the business being modeled by the database. It is important to note that the enforcement of business rules is *as* important as the enforcement of the general integrity rules discussed in the previous section.

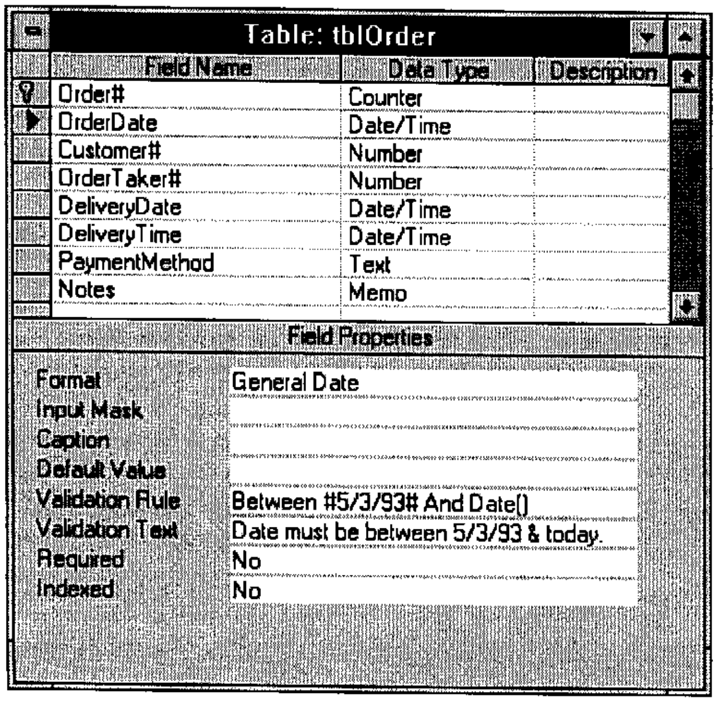
Note Rules in Microsoft Access 2.0 are now enforced at the engine level, which means that forms, action queries and table imports can no longer ignore your rules. Because of this change, however, column rules can no longer reference other columns or use domain, aggregate, or user-defined functions.

Without the specification and enforcement of business rules, bad data will get in the database. The old adage, "garbage in, garbage out" applies aptly to the application (or lack of application) of business rules. For example, a pizza delivery business might have the following rules that would need to be modeled in the database:

- Order date must always be between the date the business started and the current date.
- Order time and delivery time can be only during business hours.

- Delivery time must be greater than or equal to Order time.
- New orders cannot be created for discontinued menu items.
- Customer zip codes must be within a certain range—the delivery area.
- The quantity ordered can never be less than 1 or greater than 50.
- Non-null discounts can never be less than 1 percent or greater than 30 percent.

Microsoft Access 2.0 supports the specification of validation rules for each column in a table. For example, the first business rule from the above list has been specified in Figure 14.



Field Name	Data Type	Description
Order#	Counter	
OrderDate	Date/Time	
Customer#	Number	
OrderTaker#	Number	
DeliveryDate	Date/Time	
DeliveryTime	Date/Time	
PaymentMethod	Text	
Notes	Memo	

Format	General Date
Input Mask	
Caption	
Default Value	
Validation Rule	Between #5/3/93# And Date()
Validation Text	Date must be between 5/3/93 & today.
Required	No
Indexed	No

Figure 14. A column validation rule has been created to limit all order dates to some time between the first operating day of the business (5/3/93) and the current date.

Microsoft Access 2.0 also supports the specification of a global rule that applies to the entire table. This is useful for creating rules that cross-reference columns as the example in Figure 15 demonstrates. Unfortunately, you're only allowed to create one global rule per table, which could make for some awful validation error messages (e.g., "You have violated one of the following rules: 1. Delivery Date > Order Date. 2. Delivery Time > Order Time....").

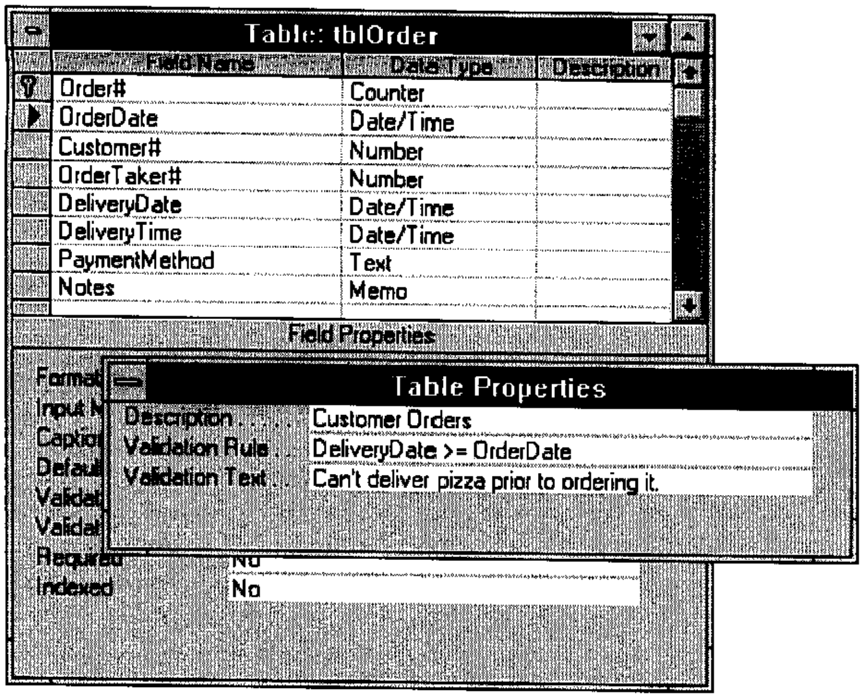


Table: tblOrder

Field Name	Data Type	Description
Order#	Counter	
OrderDate	Date/Time	
Customer#	Number	
OrderTaker#	Number	
DeliveryDate	Date/Time	
DeliveryTime	Date/Time	
PaymentMethod	Text	
Notes	Memo	

Field Properties

Required	No
Indexed	No

Table Properties

Description	Customer Orders
Validation Rule	DeliveryDate >= OrderDate
Validation Text	Can't deliver pizza prior to ordering it.

Figure 15. A table validation rule has been created to require that deliveries be made on or after the date the pizza was ordered.

Although Microsoft Access business-rule support is better than most other desktop DBMS programs, it is still limited (especially the limitation of one global table rule), so you will typically build additional business rule logic into applications, usually in the data entry forms. This logic should be layered on top of any table-based rules and can be built into the application using combo boxes, list-boxes and option groups that limit available choices, form-level and field-level validation rules, and event procedures. These application-based rules, however, should be used only when the table-based rules cannot do the job. The more you can build business rules in at the table level, the better, because these rules will always be enforced and will require less maintenance.

A Practical Approach to Database Design

As mentioned earlier in this paper, database design is more art than science. While it's true that a properly designed database should follow the normal forms and the relational model, you still have to come up with a design that reflects the business you are trying to model. Relational database design theory can usually tell you what *not* to do, but it won't tell you where to start or how to manage your business. This is where it helps to understand the business (or other scenario) you are trying to model. A well-designed database requires business insight, time, and experience. Above all, it shouldn't be rushed.

To assist you in the creation of databases, I've outlined the following 20-step approach to sound database design:

1. Take some time to learn the business (or other system) you are trying to model. This will usually involve sitting down and meeting with the people who will be using the system and asking them lots of questions.
2. On paper, write out a basic mission statement for the system. For example, you might write something like "This system will be used to take orders from customers and track orders for accounting and inventory purposes." In addition, list out the requirements of the system. These

requirements will guide you in creating the database schema and business rules. For example, create a list that includes entries such as "Must be able to track customer address for subsequent direct mail."

3. Start to rough out (on paper) the data entry forms. (If rules come to mind as you lay out the tables, add them to the list of requirements outlined in step 2.) The specific approach you take will be guided by the state of any existing system.

    - If this system was never before computerized, take the existing paper-based system and rough out the table design based on these forms. It's very likely that these forms will be non-normalized.
    - If the database will be converted from an existing computerized system, use its tables as a starting point. Remember, however, that it's very likely that the existing schema will be non-normalized. It's much easier to normalize the database *now* rather than later. Print out the existing schema, table by table, and the existing data entry forms to use in the design process.
    - If you are *really* starting from scratch (e.g., for a brand new business), then rough out on paper what forms you envision filling out.
4. Based on the forms, you created in step 3, rough out your tables on paper. If normalization doesn't come naturally (or from experience), you can start by creating one huge, non-normalized table per form that you will later normalize. If you're comfortable with normalization theory, try and keep it in mind as you create your tables, remembering that each table should describe a single entity.
5. Look at your existing paper or computerized reports. (If you're starting from scratch, rough out the types of reports you'd like to see on paper.) For existing systems that aren't currently meeting the user needs, it's likely that key reports are missing. Create them now on paper.
6. Take the roughed-out reports from step 5 and make sure that the tables from step 4 include this data. If information is not being collected, add it to the existing tables or create new ones.
7. On paper, add several rows to each roughed-out table. Use real data if at all possible.
8. Start the normalization process. First, identify candidate keys for every table and using the candidates, choose the primary key. Remember to choose a primary key that is minimal, stable, simple, and familiar. Every table must have a primary key! Make sure that the primary key will guard against all present *and* future duplicate entries.
9. Note foreign keys, adding them if necessary to related tables. Draw relationships between the tables, noting if they are one-to-one or one-to-many. If they are many-to-many, then create linking tables.
10. Determine whether the tables are in First Normal Form. Are all fields atomic? Are there any repeating groups? Decompose if necessary to meet 1NF.
11. Determine whether the tables are in Second Normal Form. Does each table describe a single entity? Are all non-key columns fully dependent on the primary key? Put another way, does the primary key imply all of the other columns in each table? Decompose to meet 2NF. If the table has a composite primary key, then the decomposition should, in general, be guided by breaking the key apart and putting all columns pertaining to each component of the primary key in their own tables.
12. Determine if the tables are in Third Normal Form. Are there any computed columns? Are there any mutually dependent non-key columns? Remove computed columns. Eliminate mutual dependent columns by breaking out lookup tables.
13. Using the normalized tables from step 12, refine the relationships between the tables.
14. Create the tables using Microsoft Access (or whatever database program you are using). If using Microsoft Access, create the relationships between the tables using the Edit|Relationships command. Add sample data to the tables.

15. Create prototype queries, forms, and reports. While creating these objects, design deficiencies should become obvious. Refine the design as needed.
16. Bring the users back in. Have them evaluate your forms and reports. Are their needs met? If not, refine the design. Remember to re-normalize if necessary (steps 8-12).
17. Go back to the table design screen and add business rules.
18. Create the final forms, reports, and queries. Develop the application. Refine the design as necessary.
19. Have the users test the system. Refine the design as needed.
20. Deliver the final system.

This list doesn't cover every facet of the design process, but it's useful as a framework for the process.

Breaking the Rules: When to Denormalize

Sometimes it's necessary to break the rules of normalization and create a database that is deliberately less normal than it otherwise could be. You'll usually do this for performance reasons or because the users of the database demand it. While this won't get you any points with database design purists, ultimately you have to deliver a solution that satisfies your users. If you do break the rules, however, and decide to denormalize you database, it's important that you follow these guidelines:

- Break the rules deliberately; have a good reason for denormalizing.
- Be fully aware of the tradeoffs this decision entails.
- Thoroughly document this decision.
- Create the necessary application adjustments to avoid anomalies.

This last point is worth elaborating on. In most cases, when you denormalize, you will be required to create additional application code to avoid insertion, update, and deletion anomalies that a more normalized design would avoid. For example, if you decide to store a calculation in a table, you'll need to create extra event procedure code and attach it to the appropriate event properties of forms that are used to update the data on which the calculation is based.

If you're considering denormalizing for performance reasons, don't always assume that the denormalized approach is the best. Instead, I suggest you first fully normalize the database (to Third Normal Form or higher) and then denormalize only if it becomes necessary for reasons of performance.

If you're considering denormalizing because your users think they need it, investigate why. Often they will be concerned about simplifying data entry, which you can usually accomplish by basing forms on queries while keeping your base tables fully normalized.

Here are several scenarios where you might choose to break the rules of normalization:

- You decide to store an indexed computed column, Soundex, in tblCustomer to improve query performance, in violation of 3NF (because Soundex is dependent on LastName). The Soundex column contains the sound-alike code for the LastName column. It's an indexed column (with duplicates allowed) and is calculated using a user-defined function. If you wish to perform searches on the Soundex column with any but the smallest tables, you'll find a significant performance advantage to storing the Soundex column in the table and indexing this computed column. You'd likely use an event procedure attached to a form to perform the Soundex calculation and store the result in the Soundex column. To avoid update anomalies, you'll want to ensure that this column cannot be updated by the user and that it is updated every time LastName changes.

- In order to improve report performance, you decide to create a column named TotalOrderCost that contains a sum of the cost of each order item in tblOrder. This violates 2NF because TotalOrderCost is dependent on the primary key of tblOrderDetail, not on tblOrder's primary key. TotalOrderCost is calculated on a form by summing the column TotalCost for each item. Since you often create reports that need to include the total order cost, but not the cost of individual items, you've broken 2NF to avoid having to join these two tables every time this report needs to be generated. As in the last example, you have to be careful to avoid update anomalies. Whenever a record in tblOrderDetail is inserted, updated, or deleted, you will need to update tblOrder, or the information stored there will be erroneous.
- You decide to include a column, SalesPerson, in the tblInvoice table, even though SalesId is also included in tblInvoice. This violates 3NF because the two non-key columns are mutually dependent, but it significantly improves the performance of certain commonly run reports. Once again, this is done to avoid a join to the tblEmployee table, but introduces redundancies and adds the risk of update anomalies.

Summary

This paper has covered the basics of database design in the context of Microsoft Access. The main concepts covered were:

- The relational database model was created by E.F. Codd in 1969 and is founded on set theory and logic.
- A database designed according to the relational model will be efficient, predictable, well performing, self-documenting and easy to modify.
- Every table must have a primary key, which uniquely identifies rows in the table.
- Foreign keys are columns used to reference a primary key in another table.
- You can establish three kinds of relationships between tables in a relational database: one-to-one, one-to-many or many-to-many. Many-to-many relationships require a linking table.
- Normalization is the process of simplifying the design of a database so that it achieves the optimum structure.
- A well-designed database follows the Normal Forms.
- The entity integrity rule forbids nulls in primary key columns.
- The referential integrity rule says that the database must not contain any unmatched foreign key values.
- Business rules are an important part of database integrity.
- A well-designed database requires business insight, time, and experience.
- Occasionally, you made need to denormalize for performance.

Database design is an important component of application design. If you take the time to design your databases properly, you'll be rewarded with a solid application foundation on which you can build the rest of your application.

Appendix

Resources

For a more detailed discussion of database design, I suggest one of the following texts.

These three books are not design books, per se. They are books on Microsoft Access that include good chapters on database design:

- *Microsoft Access 2 Developer's Handbook.* Ken Getz, Paul Litwin, and Greg Reddick. Sybex 1994. ISBN# 0-7821-1327-3. *(This paper was adapted from the design chapter of this book.)*
- *Running Microsoft Access.* John Viescas. Microsoft Press® 1994. ISBN# 1-55615-592-1.
- *Using Microsoft Access 2 for Windows®, Special Edition.* Roger Jennings. Que 1994. ISBN# 1-56529-628-1.

The following report presents a good introduction to database design using Microsoft Access:

- *Database Design with Microsoft Access.* Michael J. Hernandez. Pinnacle Publishing 1994.

These are excellent books on the relational model:

- *An Introduction to Database Systems.* Vol I. CJ Date. Addison Wesley.
- *Database Processing: Fundamentals, Design, and Implementation.* David M Kroenke. MacMillan.
- *The Relational Model, Version 2.* EF Codd. Addison Wesley.
- *SQL and Relational Basics.* Fabian Pascal. M&T Books.

Understanding SQL JOINS

By The Disenchanted Developer

Table of Contents

Parlez–Vous SQL?

A few years ago, I decided to visit Paris for a short two–week holiday. I didn't know French, had never visited the country before, and only knew that it was famous for its wines and cheeses. However, I naively assumed that I'd have no trouble getting around – after all, I reasoned, Paris was a cosmopolitan city, and English was bound to be one of the more common languages there.

As you might imagine, my naïve optimism came crashing down to earth the moment I got off the plane. All the signs were in French, everyone seemed to be gesticulating frantically, and I couldn't understand a single word of what the people around me were saying. Sign language got me a cab, and a hotel booking slip got me to a bed – but I quickly realized that if I was going to survive in the city, I'd need to learn a little French...and fast!

With a little help from some friendly students, I quickly learnt the basics – "bon jour" when you meet someone, "au revoir" when you part, "merci" when you want to thank someone and – if all else fails – "parlez–vouz anglais?". By the end of my holiday, I understood a great deal more of what was happening around me, and was even able to conduct rudimentary conversations with the locals.

What does this have to do with anything? Quite a lot.

You see, learning a programming language is a lot like learning a spoken language. You start off knowing almost nothing and slowly work your way into a situation where you're comfortable with using it to accomplish basic tasks. You then build on this basic knowledge in order to acquire greater proficiency with the language, until your versatility and complete comfort with the language's most arcane nuances impress even your closest friends.

If you've been following this column over the past few months, you've probably picked up the basics of SQL, the standard language used to communicate with databases; I've used it frequently when discussing database–driven, dynamic Web applications. Even if you started out as a novice, you're probably now fairly comfortable using SQL to retrieve data from a database or insert new records into it. And so, it's time to begin the second phase of your education – learning some of the more advanced things you can do with SQL.

That's where this article comes in. Over the next few pages, I will be introducing you to joins, which have a reputation for being complex, difficult to understand and frightening. Most of this is just nasty rumour – they're actually simple, extremely friendly and nowhere near as frightening as some of the bills I got in Paris. Let me show you.

Meeting The Family

Before we begin, I'd like to introduce you to the three tables I'll be using in this tutorial. Say hello to table "a".

```
+----+------+
| a1 | a2 |
+----+------+
| 10 | u |
| 20 | v |
| 30 | w |
| 40 | x |
| 50 | y |
| 60 | z |
+----+------+
6 rows in set (0.05 sec)
```

Next up, drop by table "b",

```
+----+------+
| b1 | b2 |
+----+------+
| 10 | p |
| 20 | q |
+----+------+
2 rows in set (0.06 sec)
```

Finally, last but not least, table "c".

```
+-----+------+
| c1 | c2 |
+-----+------+
| 90 | m |
| 100 | n |
| 110 | o |
+-----+------+
3 rows in set (0.05 sec)
```

As you can see, I spent a lot of time naming them, so I expect you to be properly appreciative of my efforts.

These tables have been constructed in MySQL 4.x, a free (and very powerful) open–source RDBMS. Due to differences in capabilities, the techniques outlined in this article may not work on other RDBMS; you should refer to the documentation that comes with each system for accurate syntax.

I've deliberately kept these tables simple, so that you have as little difficulty as possible understanding the examples. At a later stage, once the basic joins are clear to you, I'll replace these simple tables with more complex, real–world representations, so that you understand some of the more arcane applications of joins.

Keeping It Simple

We'll start with something simple. The general format for a SELECT statement is:

```
SELECT <column list> FROM <table list> WHERE <condition list>;
```

Here's an example of how it can be used – the following SQL query retrieves all the records in table "a".

```
SELECT * FROM a;
```

Here's the output:

```
+----+------+
| a1 | a2 |
+----+------+
| 10 | u |
| 20 | v |
| 30 | w |
| 40 | x |
| 50 | y |
| 60 | z |
+----+------+
```

I can filter out some of these records by adding a WHERE clause:

```
SELECT * FROM a WHERE a1 > 20;
```

This returns

```
+----+------+
| a1 | a2 |
+----+------+
| 30 | w |
| 40 | x |
| 50 | y |
| 60 | z |
+----+------+
```

And I can even restrict the number of columns shown, by specifying a list of column names instead of the all–purpose asterisk:

```
SELECT a2 FROM a WHERE a1 > 20;
```

Which gives me:

```
+-------+
| a2  |
+-------+
| w  |
| x  |
| y  |
| z  |
+-------+
```

Crossed Wires

The examples on the previous page dealt with a single table, so the question of a join never arose. But often, using a single table returns an incomplete picture; you need to combine data from two or more tables in order to see a more accurate result.

That's where joins come in – they allow you to combine two or more tables, and to massage the data within those tables, in a variety of different ways. For example,

```
SELECT * FROM a,b;
```

Here's the output:

```
+----+------+----+------+
| a1 | a2 | b1 | b2 |
+----+------+----+------+
| 10 | u | 10 | p |
| 20 | v | 10 | p |
| 30 | w | 10 | p |
| 40 | x | 10 | p |
| 50 | y | 10 | p |
| 60 | z | 10 | p |
| 10 | u | 20 | q |
| 20 | v | 20 | q |
| 30 | w | 20 | q |
| 40 | x | 20 | q |
| 50 | y | 20 | q |
| 60 | z | 20 | q |
+----+------+----+------+
12 rows in set (0.00 sec)
```

And that's your very first join!

In this case, columns from both tables are combined to produce a resultset containing all possible combinations. This kind of join is referred to as a "cross join", and the number of rows in the joined table will be equal to the product of the number of rows in each of the tables used in the join. You can see this from the example above – table "a" has 6 rows, table "b" has 2 rows, and so the joined table has 6x2 = 12 rows.

There are two basic types of SQL joins – the "inner join" and the "outer join". Inner joins are the most common – the one you just saw was an example of an inner join – and also the most symmetrical, since they require a match in each table which forms a part of the join. Rows which do not match are excluded from the final resultset.

As you might imagine, a cross join like the one above can have huge implications for the performance of your database server. In order to illustrate, look what happens when I add a third table to the join above:

```
SELECT * FROM a,b,c;
```

Here's the output:

```
+----+------+----+------+-----+------+
| a1 | a2 | b1 | b2 | c1 | c2 |
+----+------+----+------+-----+------+
| 10 | u | 10 | p | 90 | m |
| 20 | v | 10 | p | 90 | m |
| 30 | w | 10 | p | 90 | m |
| 40 | x | 10 | p | 90 | m |
| 50 | y | 10 | p | 90 | m |
| 60 | z | 10 | p | 90 | m |
| 10 | u | 20 | q | 90 | m |
| 20 | v | 20 | q | 90 | m |
| 30 | w | 20 | q | 90 | m |
| 40 | x | 20 | q | 90 | m |
| 50 | y | 20 | q | 90 | m |
| 60 | z | 20 | q | 90 | m |
| 10 | u | 10 | p | 100 | n |
| 20 | v | 10 | p | 100 | n |
| 30 | w | 10 | p | 100 | n |
| 40 | x | 10 | p | 100 | n |
| 50 | y | 10 | p | 100 | n |
| 60 | z | 10 | p | 100 | n |
| 10 | u | 20 | q | 100 | n |
| 20 | v | 20 | q | 100 | n |
| 30 | w | 20 | q | 100 | n |
| 40 | x | 20 | q | 100 | n |
| 50 | y | 20 | q | 100 | n |
| 60 | z | 20 | q | 100 | n |
| 10 | u | 10 | p | 110 | o |
| 20 | v | 10 | p | 110 | o |
| 30 | w | 10 | p | 110 | o |
| 40 | x | 10 | p | 110 | o |
| 50 | y | 10 | p | 110 | o |
| 60 | z | 10 | p | 110 | o |
| 10 | u | 20 | q | 110 | o |
| 20 | v | 20 | q | 110 | o |
| 30 | w | 20 | q | 110 | o |
| 40 | x | 20 | q | 110 | o |
| 50 | y | 20 | q | 110 | o |
| 60 | z | 20 | q | 110 | o |
+----+------+----+------+-----+------+
36 rows in set (0.05 sec)
```

Though each of the tables used in the join contains less than ten records each, the final joined result contains 6x2x3=36 records. This might not seem like a big deal when all you're dealing with are three tables containing a total of 11 records, but imagine what would happen if you had three tables, each containing 100 records, and you decided to cross join them...

Finding Common Ground

Since a cross join produces a huge resultset, it is considered a good idea to attach a WHERE clause to the join to filter out some of the records. Here's an example:

```
SELECT * FROM a,b WHERE a1 > 30;
```

This returns

```
+----+------+----+------+
| a1 | a2 | b1 | b2 |
+----+------+----+------+
| 40 | x | 10 | p |
| 50 | y | 10 | p |
| 60 | z | 10 | p |
| 40 | x | 20 | q |
| 50 | y | 20 | q |
| 60 | z | 20 | q |
+----+------+----+------+
6 rows in set (0.06 sec)
```

A variant of the cross join is the "equi–join", where certain fields in the joined tables are equated to each other. In this case, the final resultset will only include those rows from the joined tables which have matches in the specified fields.

```
SELECT * FROM a,b WHERE a1 = b1;
```

This returns

```
+----+------+----+------+
| a1 | a2 | b1 | b2 |
+----+------+----+------+
| 10 | u | 10 | p |
| 20 | v | 20 | q |
+----+------+----+------+
2 rows in set (0.00 sec)
```

Here, too, you can use a WHERE clause to further narrow the resultset:

```
SELECT * FROM a,b WHERE a1 = b1 AND a1 = 20;
```

This returns

```
+----+------+----+------+
| a1 | a2 | b1 | b2 |
+----+------+----+------+
| 20 | v | 20 | q |
+----+------+----+------+
1 row in set (0.00 sec)
```

One Step Left...

Let's move on to outer joins. You'll remember, from a few pages back, that inner joins are symmetrical – in order to be included in the final resultset, rows must match in all joined tables. Outer joins, on the other hand, are asymmetrical – all rows from one side of the join are included in the final resultset, regardless of whether or not they match rows on the other side of the join.

This might seem a little complicated, but an example should soon clear that up. Consider the following SQL query:

```
SELECT * FROM a LEFT JOIN b ON a1 = b1;
```

In English, this translates to "select all the rows from the left side of the join (table a) and, for each row selected, either display the matching value from the right side (table b) or display an empty row". This kind of join is known as a "left join" or, sometimes, a "left outer join".

Here's the result:

```
+----+------+------+------+
| a1 | a2 | b1 | b2 |
+----+------+------+------+
| 10 | u | 10 | p |
| 20 | v | 20 | q |
| 30 | w | NULL | NULL |
| 40 | x | NULL | NULL |
| 50 | y | NULL | NULL |
| 60 | z | NULL | NULL |
+----+------+------+------+
6 rows in set (0.06 sec)
```

As you can see, all the rows from the left side of the join – table "a" – appear in the final resultset. Those which have a corresponding value on the right side – table "b" – as per the match criteria a1 = b1 have that value displayed; the rest have a null row displayed.

This kind of join comes in very handy when you need to see which values from one table are missing in another table – all you need to do is look for the null rows. Just from a quick glance at the example above, it's fairly easy to see that rows 30–60 are present in table "a", but absent in table "b". This technique also comes in handy when you're looking for corrupted, or "dirty", data.

In fact, you can even save your eyeballs the trouble of scanning the output – just let SQL do it for you, with an additional WHERE clause:

```
SELECT a1 FROM a LEFT JOIN b ON a1 = b1 WHERE b1 IS NULL;
```

Here's the output:

```
+----+
| a1 |
+----+
| 30 |
| 40 |
| 50 |
| 60 |
+----+
4 rows in set (0.05 sec)
```

...Two Steps Right

Just as there's a left join, there's also a "right join", which does exactly what a left join does, but in reverse. Consider the following query,

```
SELECT * FROM a RIGHT JOIN c ON a1=c1;
```

which translates to "select all the rows from the right side of the join (table c) and, for each row selected, either display the matching value from the left side (table a) or display an empty row" and displays

```
+------+------+-----+------+
| a1   | a2   | c1  | c2   |
+------+------+-----+------+
| NULL | NULL | 90  | m    |
| NULL | NULL | 100 | n    |
| NULL | NULL | 110 | o    |
+------+------+-----+------+
3 rows in set (0.06 sec)
```

All the rows from the right side of the join – table "c" – appear in the final resultset. Those which have a corresponding value on the left side – table "a" – as per the match criteria a1 = c1 have that value displayed; the rest have a null row displayed.

The Bookworm Turns

So that's the theory – now let's see how it plays out in real life. Consider the following tables, which contain data for a small lending library:

```
+----+-----------------------------------+
| id | title |
+----+-----------------------------------+
| 1 | Mystic River |
| 2 | Catch-22 |
| 3 | Harry Potter and the Goblet of Fire |
| 4 | The Last Detective |
| 5 | Murder on the Orient Express |
+----+-----------------------------------+
```

This table, named "books", contains a complete list of books available for lending. Each book has a unique ID.

```
+-----+-------+
| id | name |
+-----+-------+
| 100 | Joe |
| 101 | John |
| 103 | Dave |
| 109 | Harry |
| 110 | Mark |
| 113 | Jack |
+-----+-------+
```

This table, named "members", contains a list of all the members of the library. Like the books, each member also has a unique ID.

```
+-----------+---------+
| member_id | book_id |
+-----------+---------+
| 101 | 1 |
| 103 | 3 |
| 109 | 4 |
+-----------+---------+
```

As members borrow books from the library, the "data" table, above, is updated with information on which book has been lent to which member.

Using these tables and whatever you've just learned about joins, it's possible to write queries that answer the most common questions asked of the system by its operators. For example, let's suppose I wanted a list of all the current members:

```
SELECT * FROM members;

+-----+-------+
| id | name |
+-----+-------+
| 100 | Joe |
| 101 | John |
| 103 | Dave |
| 109 | Harry |
| 110 | Mark |
| 113 | Jack |
+-----+-------+
```

How about a list of all the books currently in the library catalog?

```
SELECT * FROM books;

+----+------------------------------------+
| id | title |
+----+------------------------------------+
| 1 | Mystic River |
| 2 | Catch-22 |
| 3 | Harry Potter and the Goblet of Fire |
| 4 | The Last Detective |
| 5 | Murder on the Orient Express |
+----+------------------------------------+
```

How about an enquiry on Dave's account, to see which books he's currently checked out?

```
SELECT books.title FROM data,books WHERE data.book_id =
books.id AND
data.member_id = 103;

+------------------------------------+
| title |
+------------------------------------+
| Harry Potter and the Goblet of Fire |
+------------------------------------+
```

Can we find out which members *don't* have any books checked out to them (maybe these are inactive accounts)?

```
SELECT members.id, members.name FROM members LEFT JOIN data ON
data.member_id = members.id WHERE data.book_id IS NULL;

+-----+------+
| id | name |
+-----+------+
| 100 | Joe |
| 110 | Mark |
| 113 | Jack |
+-----+------+
```

How about a list of all the books that are currently in stock (not checked out to anyone)?

```
SELECT books.title FROM books LEFT JOIN data ON data.book_id =
books.id
WHERE data.book_id IS NULL

+------------------------------+
| title |
+------------------------------+
| Catch-22 |
| Murder on the Orient Express |
+------------------------------+
```

Now, how about a list of all the books currently checked out, together with the name of the member they've been checked out to?

```
SELECT members.name, books.title FROM data, members, books
WHERE
data.member_id = members.id AND data.book_id = books.id;

+-------+--------------------------------------+
| name | title |
+-------+--------------------------------------+
| John | Mystic River |
| Dave | Harry Potter and the Goblet of Fire |
| Harry | The Last Detective |
+-------+--------------------------------------+
```

You could also accomplish the above with the following query (an outer join instead of an inner join):

```
SELECT members.name, books.title FROM books,members LEFT JOIN
data ON
data.book_id = books.id WHERE data.member_id = members.id AND
data.book_id IS NOT NULL;
```

As you can see, joins allow you to extract all kinds of different information from a set of tables. Go ahead and try it for yourself – think of a question you would like to ask this system, and then write a query to answer it.

Up A Tree

In addition to inner and outer joins, SQL also allows a third type of join, known as a "self join". Typically, this type of join is used to extract data from a table whose records contain internal links to each other. Consider the following table, which illustrates what I mean:

```
+----+-----------------------+--------+
| id | label | parent |
+----+-----------------------+--------+
| 1 | Services | 0 |
| 2 | Company | 0 |
| 3 | Media Center | 0 |
| 4 | Your Account | 0 |
| 5 | Community | 0 |
| 6 | For Content Publishers | 1 |
| 7 | For Small Businesses | 1 |
| 8 | Background | 2 |
| 9 | Clients | 2 |
| 10 | Addresses | 2 |
| 11 | Jobs | 2 |
| 12 | News | 2 |
| 13 | Press Releases | 3 |
| 14 | Media Kit | 3 |
| 15 | Log In | 4 |
| 16 | Columns | 5 |
| 17 | Colophon | 16 |
| 18 | Cut | 16 |
| 19 | Boombox | 16 |
+----+-----------------------+--------+
```

This is a simple menu structure, with each record identifying a unique node in the menu tree. Each record has a unique record ID, and also contains a parent ID – these two IDs are used to define the parent–child relationships between the branches of the menu tree. So, if I were to represent the data above hierarchically, it would look like this:

```
<root>
|-Services
|-For Content Publishers
|-For Small Businesses
|-Company
|-Background
|-Clients
|-Addresses
|-Jobs
|-News
```

```
|-Media Center
|-Press Releases
|-Media Kit
|-Your Account
|-Log In
|-Community
|-Columns
|-Colophon
|-Cut
|-Boombox
```

Now, let's suppose I need to display a list of all the nodes in the tree, together with the names of their parents. What I'm looking for is a resultset resembling this:

```
+--------------+-------------------------+
| parent_label | child_label |
+--------------+-------------------------+
| Services | For Content Publishers |
| Company | Background |
| Your Account | Log In |
+--------------+-------------------------+
```

If you think about it, you'll see that there's no easy way to obtain this resultset – since all the data is in a single table, a simple SELECT won't work, and neither will one of those complicated outer joins. What I really need here is something called a self join, which allows me to create a second, virtual copy of the first table, and then use a regular inner join to map the two together and get the output I need.

Here's the query I would use:

```
SELECT a.label AS parent_label, b.label AS child_label FROM
menu AS a,
menu AS b WHERE a.id = b.parent;
```

Here's the output:

```
+--------------+-------------------------+
| parent_label | child_label |
+--------------+-------------------------+
| Services | For Content Publishers |
| Services | For Small Businesses |
| Company | Background |
| Company | Clients |
| Company | Addresses |
| Company | Jobs |
| Company | News |
| Media Center | Press Releases |
| Media Center | Media Kit |
| Your Account | Log In |
| Community | Columns |
| Columns | Colophon |
| Columns | Cut |
| Columns | Boombox |
+--------------+------------------------+
```

Exactly what I need!

Most of the magic here lies in the table aliasing – I've created two copies of the "menu" table, and aliased them as "a" and "b" respectively. This will result in the following two "virtual" tables.

```
+----+-------------------------+---------+
| TABLE a |
+----+-------------------------+---------+
| id | label | parent |
+----+-------------------------+---------+
| 1 | Services | 0 |
| 2 | Company | 0 |
| 3 | Media Center | 0 |
| 4 | Your Account | 0 |
| 5 | Community | 0 |
| 6 | For Content Publishers | 1 |
| 7 | For Small Businesses | 1 |
| 8 | Background | 2 |
| 9 | Clients | 2 |
| 10 | Addresses | 2 |
| 11 | Jobs | 2 |
| 12 | News | 2 |
| 13 | Press Releases | 3 |
| 14 | Media Kit | 3 |
| 15 | Log In | 4 |
| 16 | Columns | 5 |
| 17 | Colophon | 16 |
| 18 | Cut | 16 |
| 19 | Boombox | 16 |
+----+-------------------------+---------+

+----+-------------------------+---------+
| TABLE b |
+----+-------------------------+---------+
| id | label | parent |
+----+-------------------------+---------+
| 1  | Services | 0 |
| 2  | Company | 0 |
| 3  | Media Center | 0 |
| 4  | Your Account | 0 |
| 5  | Community | 0 |
| 6  | For Content Publishers | 1 |
| 7  | For Small Businesses | 1 |
| 8  | Background | 2 |
| 9  | Clients | 2 |
| 10 | Addresses | 2 |
| 11 | Jobs | 2 |
| 12 | News | 2 |
| 13 | Press Releases | 3 |
| 14 | Media Kit | 3 |
| 15 | Log In | 4 |
| 16 | Columns | 5 |
| 17 | Colophon | 16 |
| 18 | Cut | 16 |
| 19 | Boombox | 16 |
+----+-----------------------+--------+
```

Once these two tables have been created, it's a simple matter to join them together, using the node IDs as the common column, and to obtain a list of child and parent labels in the desired format.

A Long Goodbye

And that's about it for the moment. In this article, I helped you dip your toes into the deeper waters of SQL, showing you how you could use SQL to massage your resultsets so that you only see the data you want to see. I showed you a basic inner join, the cross join, which will have your database crying for Mommy if you use it too often, and the equi-join, which you can use to filter our unwanted results from your inner join.

Next, I moved on to outer joins, demonstrating how the left and right outer joins can be used to compare tables and isolate the missing or common elements. Finally, I wrapped things up with the self join, a very neat little concept that is sure to come in useful when you're dealing with a single table containing linked records.

In order to avoid having the theory overwhelm you, I also showed you how the various types of joins may be used in real-world situations, to answer basic questions that arise when dealing with linked tables in an RDBMS. Hopefully, the two real-world examples in this article fully demonstrated the value of SQL joins, and also destroyed some of the myths associated with their ease of use (or lack thereof).

This isn't all, though – SQL is very popular on the Web, and there are reams of information out there which will teach you how to manipulate inner and outer joins to do ever more complicated acrobatics. Here are a few good links to get you started:

Speaking SQL, at http://www.devshed.com/Server_Side/MySQL/Speak/

The MySQL manual, at http://www.mysql.com/doc

The W3School's tutorial on joins, at http://www.w3schools.com/sql/sql_join.asp

CNET's tutorial on joins, at http://asia.cnet.com/itmanager/netadmin/0,39006400,39042301,00.htm

Until next time...au revoir!

Note: All examples in this article have been tested on Linux/i586 with MySQL 4. Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!