DATABASE DESIGN AND SERVER CONTENT GENERATION

LEARNING HOW TO PREPARE AND PRESENT YOUR DATA

TABLE OF CONTENTS

INTRODUCTION

This document covers a variety of topics that include basic database design theory, database server selection and an introduction to various methods to present your data.

Fundamentals of Relational Database Design

By Paul Litwin

This paper was part of a presentation at a Microsoft TechEd conference in the mid 1990s. It was adapted from Microsoft Access 2 Developer's Handbook, Sybex 1994, by Ken Getz, Paul Litwin and Greg Reddick. Reprinted with permission of the publisher.

While the paper uses Microsoft Access (version 2) for the examples, the vast majority of the discussion applies to any database and holds up pretty well over 11 years after it was written.

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 article 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.

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.

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

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. 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 befor e 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 article.

Relational Database Design

Note: While the examples in this article are centered around Microsoft Access databases, the discussion also applies to any 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.

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 article 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 article 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 article is meant as an informal introduction to database design theory for the database developer.

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.

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: 4 of 6

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.

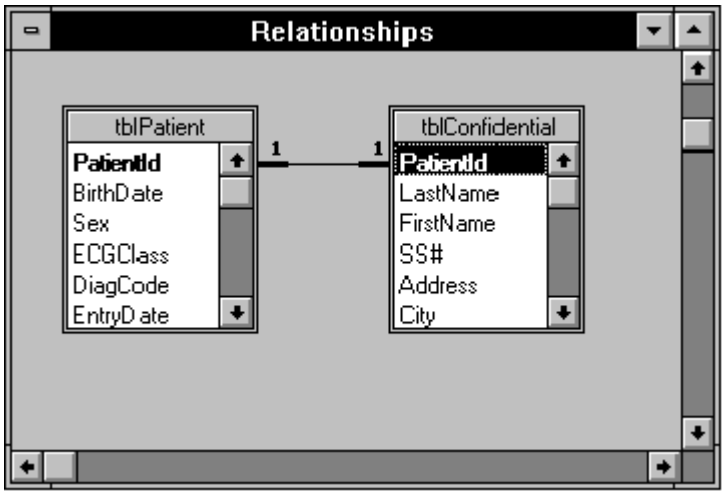

FIGURE 3. THE TABLES TBLPATIENT AND TBLCONFIDENTIAL ARE RELATED IN A ONE-TO-ONE RELATIONSHIP. THE PRIMARY KEY OF BOTH TABLES IS PATIENTID.

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.

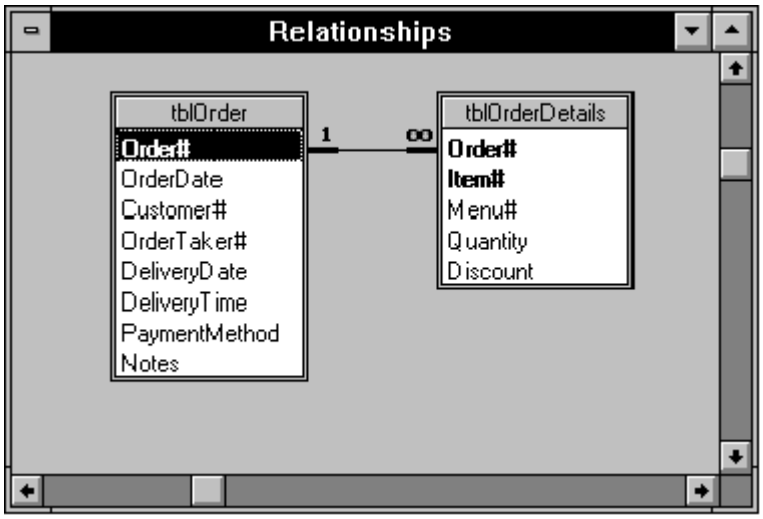

FIGURE 4. THERE CAN BE MANY DETAIL LINES FOR EACH ORDER IN THE PIZZA

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).

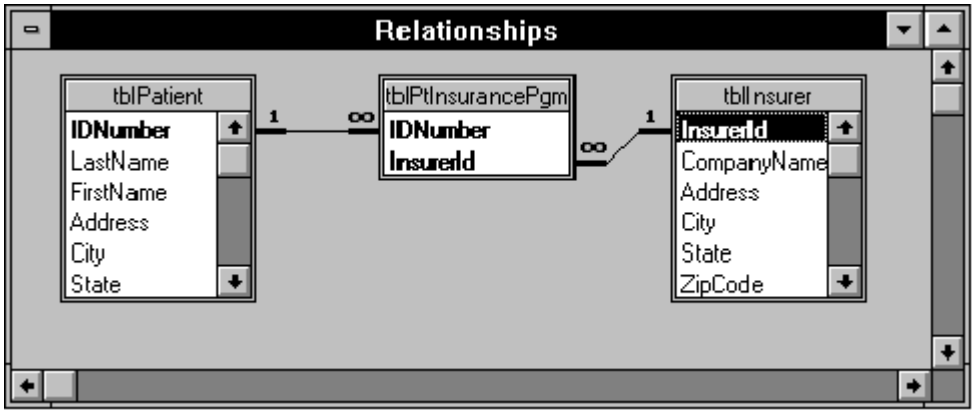

FIGURE 5. A LINKING TABLE, TBLPTINSURANCEPGM, IS USED TO MODEL THE MANY-TO-MANY RELATIONSHIP BETWEEN TBLPATIENT AND TBLINSURER.

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 article, 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 article 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 10' 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).

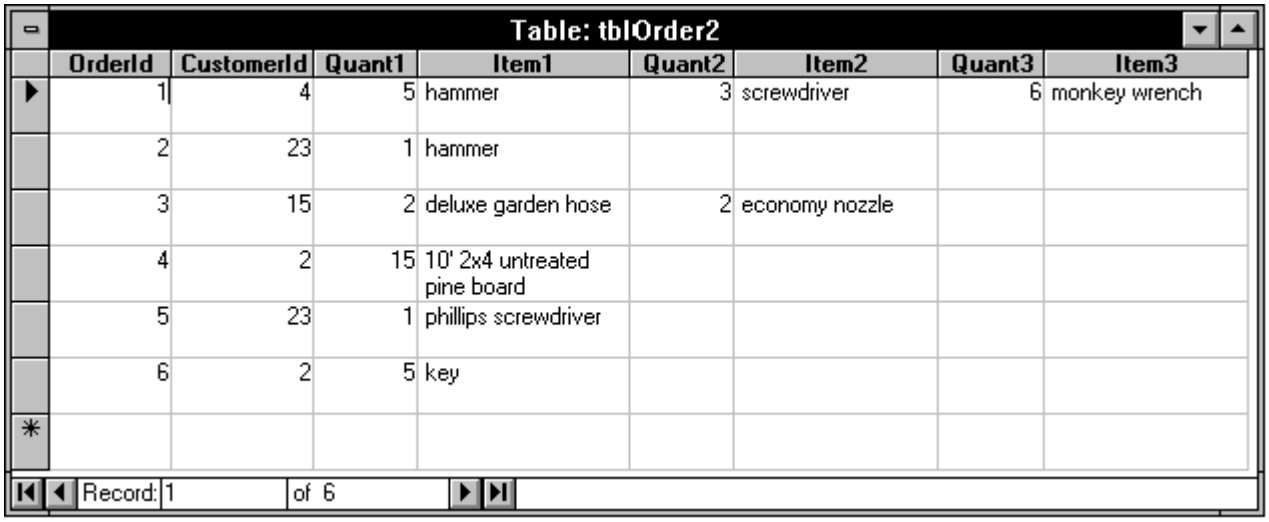

Table: tblOrder2

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

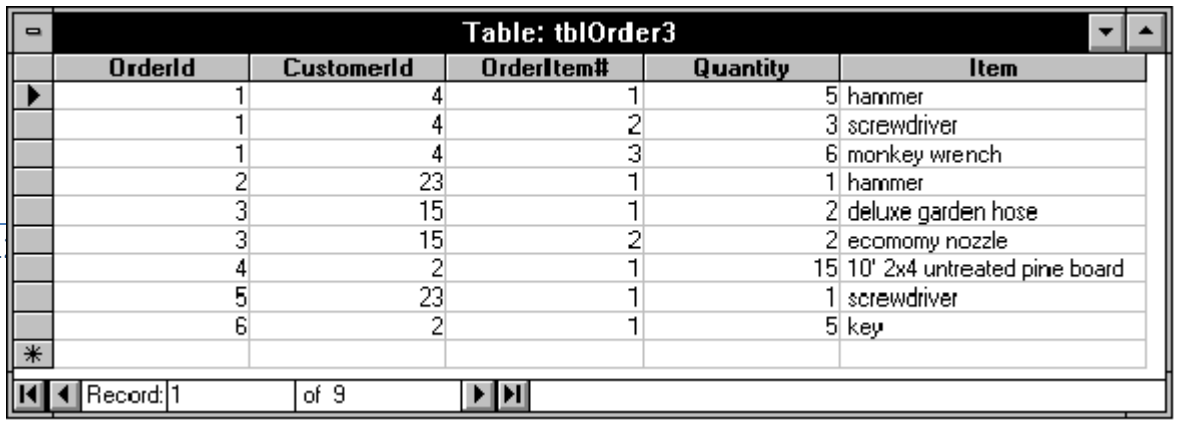

Table: tblOrder3

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	economy nozzle
4	2	1	15	10' 2x4 untreated pine board
5	23	1	1	screwdriver
6	2	1	5	key

Record: 1 of 9

OrderItem#.

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.

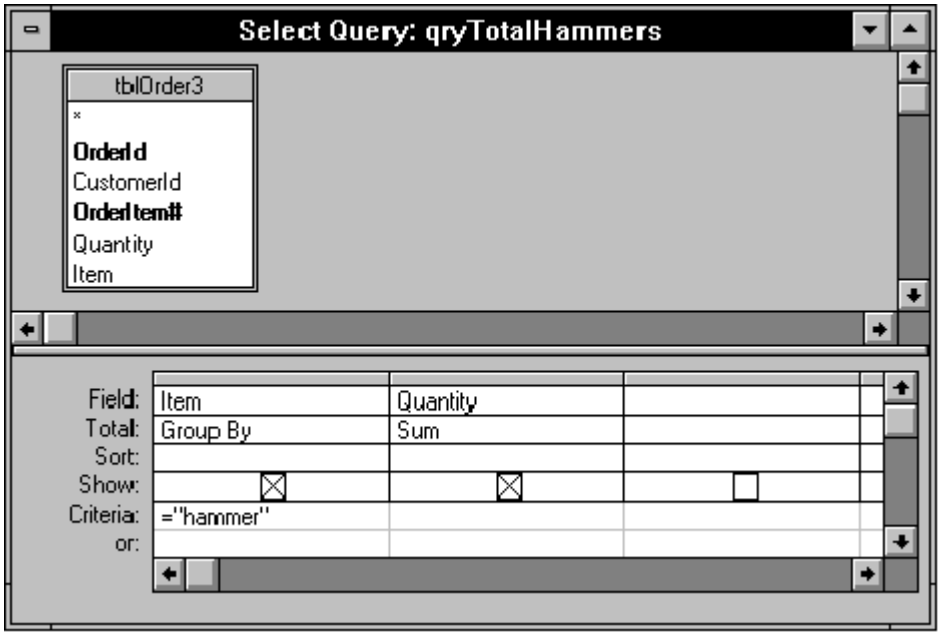

Figure 9. Since tblOrder3 is in First Normal Form, you can easily construct a Totals query to determine the total number of hammers ordered by customers.

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.

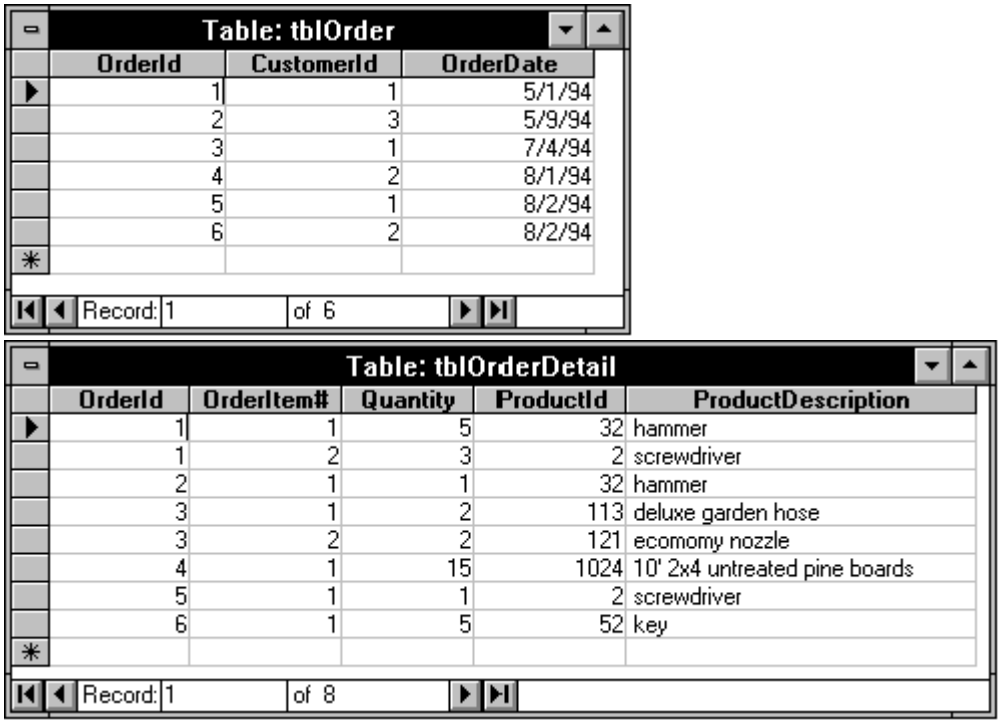

Table: tblOrder

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.

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.

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.

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.

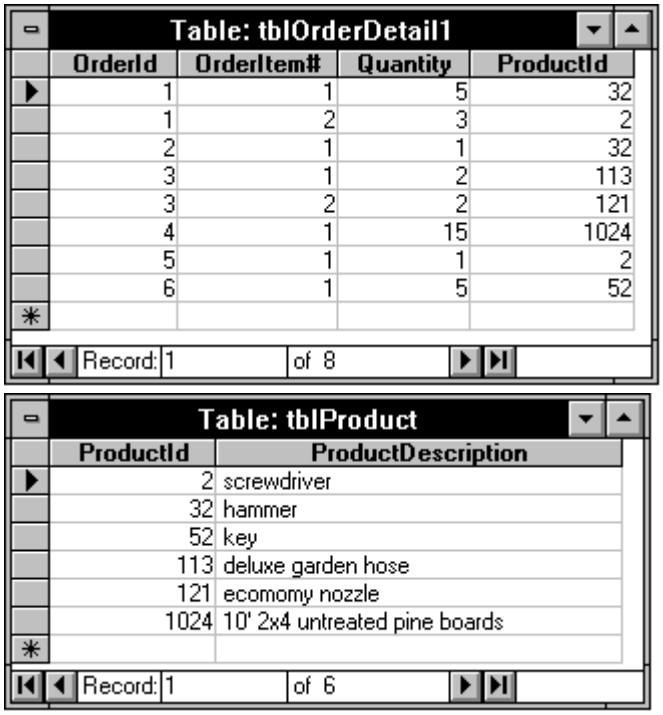

Table: tblOrderDetail1

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

Record: 1 of 8

Table: tblProduct

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

Record: 1 of 6

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, 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 Boyce/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 article, 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 article, 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.

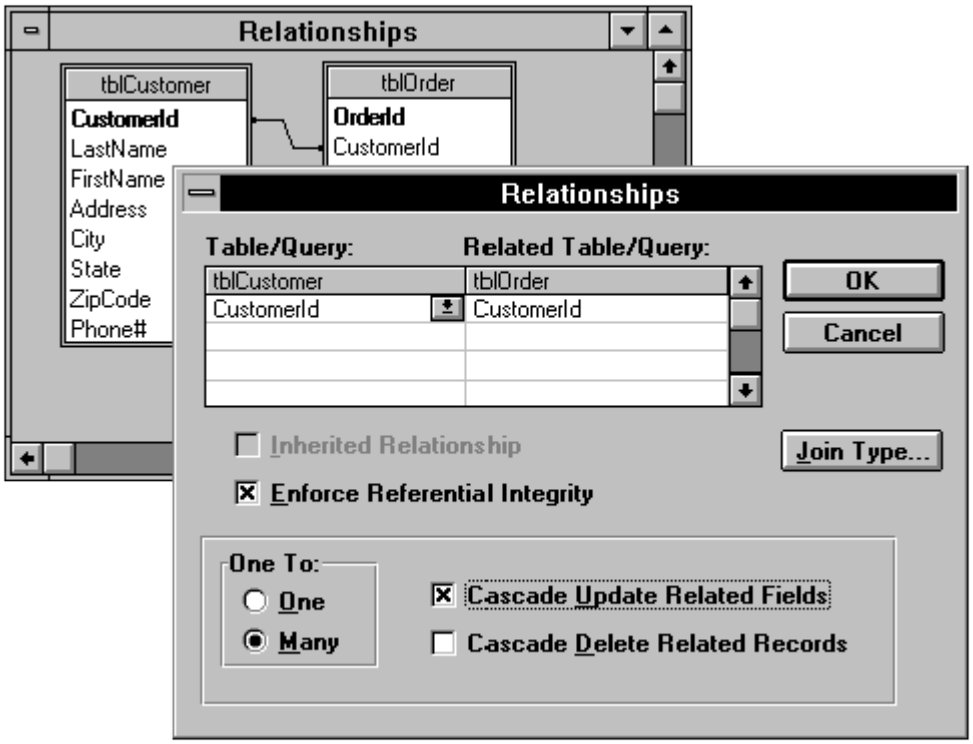

Figure 13. Specifying a relationship with referential integrity between the tblCustomer and tblOrder tables using the Edit | Relationships command. Updates of CustomerID in tblCustomer will be cascaded to tblOrder. Deletions of rows in tblCustomer will be disallowed if rows in tblOrders would be orphaned.

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.

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 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....").

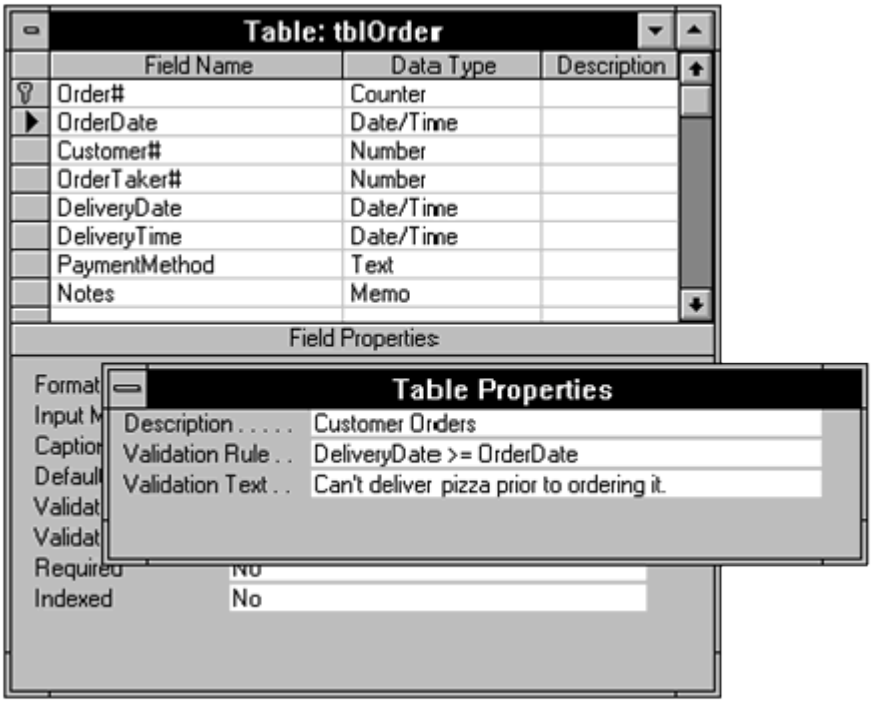

FIGURE 15. A TABLE VALIDATION RULE HAS BEEN CREATED TO REQUIRE THAT DELIVERIES BE MADE ON OR AFTER THE DATE THE PIZZA WAS ORDERED.

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 article, 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 article 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.

CHOOSING YOUR DATABASE

By Rich Simpson

Software development isn't an exact science. The same application can be developed many different ways using different tools and still achieve the same functionality. However, choosing the right software tools, the right algorithms and a good data storage structure can quickly determine how fast a developer gets there and how well the software performs. Custom software development is basically the learning and understanding of a company's workflow and the data tracked, then developing a software application to support that workflow. Some key reasons to automate may be to make it easier to find certain information about customers and products or to make the crunching of large amounts of data into simple summarized reports for management or to make an older application Y2K compliant. Whatever the reasons are to automate a certain job's workflow or data management, choosing the right tool can make a large impact on the project in both time and money. Different software development tools are better at developing particular types of software applications. Most business applications involve the entering and tracking of data specific to a business. Therefore, this article covers software development tools that are used primarily for database management. Besides just having many different software tools there are also many languages as well. So, choosing the right one can be the most crucial decision made at the start of any project. This article does not focus on the different languages and their effects on a particular project but looks at different database engines and technologies and their capabilities and limitations.

The question is how do you choose the right one? There are basically two types of database tools on the market: File/Server based and Client/Server based. File/Server database examples are Access, FileMaker Pro, FoxPro, Visual FoxPro, Paradox. These tools do not require dedicated database servers and can be run on a local workstation or have the database files shared from a file server. Examples of Client/Server type databases are Oracle, Microsoft SQL Server, Sybase and Interbase. These databases require a dedicated file server to run the database Server on and then Client applications running on the workstations to access the data from the server. In some cases you may have to choose more than one development tool or more than one operating system. For example if you are going to be supplying data to local network workstations and to remote users on the web, then several tools will be necessary to complete the project. Another item that can be critical to your decision is what languages and tools are your staff familiar with. Learning a new language or a new programming tool can slow a project down significantly. The following chart provides a brief overview displaying published limits and more realistic usable limits of some common database development tools on the market

today. A more in depth discussion regarding the information presented is discussed below.

Database Development Tool	Maximum Records	Maximum File Size	Simultaneous Users	Cost Per User	Server Cost	Built In Data Integrity	Target Market
Microsoft Access	100,000+ performance degrades	1GB (Access 97) 2GB (Access 2000)	50 - 250 (5 – 10 for usable performance)	no	no	no	End Users
MySQL, PostGreSQL, FireBird	Billions	Terabytes	Unlimited	no	no	yes	Developers
Microsoft SQL Server, Oracle, Sybase, Interbase	Billions	Terabytes	Unlimited	yes	yes	yes	Developers

How much data will you need to store?

Different databases can handle different amounts of data better. For example Microsoft Access is fairly easy to use and simple databases can be maintained without an extensive knowledge of the language. It can handle up to 1GB of data in a single MDB file in Access 97 or a 2GB MDB in Access 2000. However, most developers agree that Access is good for projects of up to about 100,000 records of data at which performance begins to degrade. FileMaker Pro, Microsoft FoxPro, Microsoft Visual FoxPro and other xBase languages are good at handling millions of records but are limited to a maximum single table file size of 2GB. These databases use separate table files for storing different information and those files are each limited to 2GB. Therefore one table can hold 2GB of customer data and another table 2GB of invoices and so on. This allows for very large databases to be developed consisting of many related tables. For most small to medium sized businesses this amount of data is sufficient. Businesses have been using applications written using these tools since the mid 1980's. For instance the Chunnel in England runs a Visual FoxPro database application managing many gigabytes of data. However, for maximum scalability and maximum database sizes a Client/Server database such as Oracle, Microsoft SQL Server, Inprise's Interbase or Sybase should be used. Databases designed on Client/Server platforms can be scaled across several drives to handle databases into the Terabytes.

How many users will need to access the data simultaneously?

Another important deciding factor when choosing a database tool is how many users will need to have simultaneous access to the database files. If you need to implement a database application requiring more than 5 to 10 simultaneous users, then Access is not a good choice. Access's Jet database engine is not designed to support a large number of users simultaneously updating the data files. Microsoft white papers mention that the Jet database can handle up to 255 users reading the data. However, if you are writing an application that will be performing any transactions the practical limit is between 50 and 250 users. However, in real life situations other developers have discovered performance begins to degrade after about 5 - 10 users. FoxPro, Visual FoxPro and other shared database engines are good at handling up to 100 - 200 users or more before significant performance degradation occurs. A lot of the performance using these tools depends on the file server performance and the network performance as well as the development techniques used. If you need to support several hundred to several thousand users then a SQL server database engine will better serve your project. These database servers can support a large number of users and can handle heavy loads much better. The speed of Client/Server databases are not as fast as what can be achieved using Visual FoxPro for a small number of users (2 – 100), but as the number of users increase they will begin to outperform File/Server based applications.

WHAT SIZE BUDGET DO YOU HAVE FOR THE PROJECT?

The File/Server based database tools don't require you to purchase a separate database server as the Client/Server databases do. Therefore, in many instances if a developer is writing an application for your company, you will not need to purchase any additional software to run the application. If your application is being developed to run in a Client/Server environment such as Microsoft SQL Server or Oracle, you will need to pay for a front end development tool or have a developer write the front end. Therefore the cost of a Client/Server solution will be much higher than a File/Server based solution. This cost can add up very quickly. Most Client/Server database engines cost approximately $1200 - $1500 for a single server with a 5-user license. In addition to purchasing the Client/Server database engine, you must also purchase user licenses for each additional user that will be simultaneously accessing the database files on the server. The typical cost per user is between $75 - $300 depending on the number of user licenses being purchased and the product being used. The Client/Server database engine and user costs do not include the cost of a front-end development tool. These tools can be Microsoft Visual Basic, C++, Inprise's Delphi, Microsoft Visual FoxPro, Microsoft Access or a number of other software tools that support ODBC connectivity. The front-end is the Client application that will be running on the end user's desktop and accessing the Client/Server database. Be prepared to also invest in purchasing a separate high performance file server to run your Client/Server database engine on. In the case of Client/Server applications the database server executes all of the queries on the server and sends the results back

to the workstation that requested the data. This means that the database server is handling the majority of the workload as compared to File/Server based applications where the query is processed by the workstation which has to pull the data across the network first in order to process it.

The other area for budgeting in database development is for the man hours required to develop the actual application. This will actually comprise most of the cost for building a database application. It typically takes several hundred to many thousands of hours to develop complex database applications. Time is money and if your developers are more familiar with the tool that they are using they will typically be able to more productive rather than having to learn a new tool while also doing the development on the project. Usually learning a new tool while developing a new project causes many more rewrites of the application as better ways are discovered to do things.

DO YOU NEED SECURE DATA OR DATA INTEGRITY?

Data can never be 100% secure from prying eyes or theft. Of the File/Server database tools that I am familiar with, Access is one of them that does support password level access to the database files. However, nothing is going to prohibit a user from copying the files to a local hard drive or to a data cartridge and running off with the files. You can add some sense of security to accessing your data files by using a secured server such as Windows NT or Linux. However, a Windows 95 or Windows 98 workstation acting as a file server is not considered a secure server unless it is locked up in a room with no access to the workstation itself. Even, if you have a Windows 95 or Windows 98 system password protected acting as a file server, there is nothing to stop a user from pressing the Cancel button at the user/password prompt and getting access to the hard drive or from booting on a floppy disk. Once the user has access to the hard drive that user can then copy database files off to a floppy or removable data cartridge. However, Client/Server databases are stored on a secure server. A user must log into the database server in order to gain access to the data files. The user cannot simply browse over to the file server containing the database files and simply open them using a file viewer or some other tool as can be done with File/Server database files. There are ways to protect the data inside of a File/Server database by encrypting the actual data within the database files. However, encrypting data does cause some performance hits and also causes much more effort for the programmer since any viewing of data has to be decrypted before displaying the data. This includes forms and reports. There are some third party tools that can do the encryption/decryption in the background to alleviate this problem but you are again facing some performance slow downs.

Data integrity is an issue of how good is the data in case of a power failure or the server locking up or a user rebooting in the middle of a transaction or editing data. None of the File/Server database tools have transaction logging and the ability to roll back a transaction or edit if it has not been completely saved to the disk. In the case of a File/Server database your best resort to data corruption is to restore from a backup such as a tape or disk cartridge. Client/Server databases have transaction logging and roll back capabilities and in the event of a power outage or system lockup the database server will remove any incomplete transactions from the databases prior to the problem. Therefore for mission critical applications Client/Server technologies lead the pack in this area.

How long will it take to implement the solution?

This can be a very difficult question to answer. There are so many factors that come into play in the development of an application that it is always hard to nail down. In fact, almost every application ever developed is constantly in a state of flux because requirements keep changing during the development effort as new ideas or new information is discovered. Obviously, the simpler the application and the simpler the tool is the shorter the development time will be. This is not always true though. Some of the development tools like Visual FoxPro are object oriented and can help reduce coding time with reusable code. However, a lot of time is invested in the development of those reusable tools before they can be used down the road. Again there are always tradeoffs. The complexity of the application will play a significant role in the amount of time it will take to develop. In addition the skill level of the developers and their familiarity with the tools being used will also play a role in the time spent developing the application. Another reason many projects take longer than expected are due to re-writes of the application as the end users make changes as to what they want. Many times a simple database structure change or new requirement can cause 100's of hours of rewrites. This is why up front planning and information gathering is very important and critical to the completion of any project.

Hopefully this article has provided you with some usable guidelines before you begin your next database project.

A comprehensive comparison of various database servers can be found on Wikipedia.
http://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems

SQL INJECTION ATTACKS ON DATABASES

By Mike Chapple, About.com Guide

The vast majority of databases in use today have some form of web interface, allowing internal and/or external users easy access through familiar browser software. If you're security-conscious, you've undoubtedly spent a significant amount of time setting appropriate security permissions on your databases and web servers. Have you also considered the security of the code that powers the database-web interface?

One common type of database attack, the SQL Injection, allows a malicious individual to execute arbitrary SQL code on your serverLet's take a look at how it works by analyzing a very simple web application that processes customer orders. Suppose Acme Widgets has a simple page for existing customers where they simply enter their customer number to retrieve all of their current order information. The page itself might be a basic HTML form that contains a textbox called CustomerNumber and a submit button. When the form is submitted, the following SQL query is executed:

```
SELECT *
FROM Orders
WHERE CustomerNumber = CustomerNumber
```

The results of this query are then displayed on the results page. During a normal customer inquiry, this form works quite well. Suppose John visits the page and enters his customer ID (14). The following query would retrieve his results:

```
SELECT *
FROM Orders
WHERE CustomerNumber = 14
```

However, the same code can be a dangerous weapon in the hands of a malicious user. Imagine that Mal comes along and enters the following data in the CustomerNumber field: “14; DROP TABLE Orders”. This would cause the following query to execute:

```
SELECT *
FROM Orders
WHERE CustomerNumber = 14; DROP TABLE Orders
```

Obviously, this is not a good thing! There are several steps that you can take to protect your server against SQL Injection attacks:

- Implement parameter checking on all applications. For example, if you're asking someone to enter a customer number, make sure the input is numeric before executing the query. You may wish to go a step further and perform additional checks to ensure the customer number is the proper length, valid, etc.
- Limit the permissions of the account that executes SQL queries. The rule of least privilege applies. If the account used to execute the query doesn't have permission to drop tables, the table dropping will not succeed!
- Use stored procedures (or similar techniques) to prevent users from directly interacting with SQL code.

As with many security principles, an ounce of prevention is worth a pound of cure. Take the time to verify the code running on your servers before disaster strikes!

DATABASE SECURITY ISSUES: INFERENCE

By Mike Chapple, About.com Guide

Databases introduce a number of unique security requirements for their users and administrators. On one hand, databases are designed to promote open and flexible access to data. On the other hand, it's this same open access that makes databases vulnerable to many kinds of malicious activity. This article is the first in a series that will look at a number of database-specific security concerns and guide you as you attempt to steer your databases clear of these obstacles.

One of the main issues faced by database security professionals is avoiding inference capabilities. Basically, inference occurs when users are able to piece together information at one security level to determine a fact that should be protected at a higher security level. It's best explained through a practical example.

Imagine that you are the database administrator for a military transportation system. You have a table named cargo in your database that contains information on the various cargo holds available on each outbound airplane. Each row in the table represents a single shipment and lists the contents of that shipment and the flight identification number. The flight identification number may be cross-referenced with other tables to determine the origin, destination, flight time and similar data. The cargo table appears as follows:

Flight ID	Cargo Hold	Contents	Classification
1254	A	Boots	Unclassified
1254	B	Guns	Unclassified
1254	C	Atomic Bomb	Top Secret
1254	D	Butter	Unclassified

Suppose that General Jones (who has a Top Secret security clearance) comes along and requests information on the cargo carried by flight 1254. The general would (correctly) see all four shipments. On the other hand, if Private Smith (who has no security clearance) requests the data, the private would see the following table:

Flight ID	Cargo Hold	Contents	Classification
1254	A	Boots	Unclassified
1254	B	Guns	Unclassified
1254	D	Butter	Unclassified

This correctly implements the security rules that prohibit someone from seeing data classified above their security level. However, assume that there is a unique constraint on flight ID and cargo hold (to prevent scheduling two shipments for the same hold). When Private Jones sees that nothing is scheduled for hold C on flight 1254, he might attempt to insert a new record to transport some vegetables on that flight. However, when he attempts to insert the record, his insert will fail due to the unique constraint. At this point, Private Jones has all the data he needs to infer that there is a secret shipment on flight 1254. He could then cross-reference the flight information table to find out the source and destination of the secret shipment and various other information.

This leads to a natural question -- what can you do about inference? Basically, you have two options. First, you can include the classification column in the unique constraint. This technique, known as polyinstantiation, allows different records to exist in the same table at various security levels. Private Jones would never learn of the Top Secret shipment. On the other hand, he might wind up consequently double-booking the cargo hold, leaving a truckload of vegetables stranded at the airport. The second option is to simply leave the tables as-is. Private Jones will know that a classified shipment is taking place but won't have access to the contents of the shipment. Neither solution is ideal, but both represent the types of trade-offs that must be made to balance security and practicality.

Tutorial: How to Build a Full-Featured Login System

Posted on www.tutorialized.com by Pat in PHP, Tutorials, mySQL on Nov 26th, 2010

In this tutorial I will be showing you how to make a simple login system consisting of a login page, register page, forgotten password page, email activation, logout page and finally a "users online" page. I made this tutorial to mainly target new-to-PHP developers, due to the fact when I started I noticed the lack in quantity of basic login systems. Therefore, I decided to make one myself giving high quality advice on how to make your first login system with a "users online" script!

1. Making a basic stylesheet

We are going to create a very basic CSS stylesheet just to add a little bit of design and tidy up the way this login system looks. So too start off with open your text editor and we can begin making our styelsheet.

CSS

```
body {
    font-family: arial;
    font-size: 10pt;
}
table {
    font-size: 10pt;
    margin: 0 auto;
}
#border {
    border: 2px solid #999;
    background: #CCC;
    padding: 15px;
    margin: 0 auto;
    width: 300px;
}
```

Save this file as style.css so we can link back to it whenever we need to. There we have our simple stylesheet! Now we can begin making our pages without having to worry too much about making them look reasonably good.

2. Creating the Login Page

Okay so we have a stylesheet defined, now it's time to get things displaying on our pages. Open a new file in your text editor, this is going to be our login.php page!

HTML

```
<html>
    <head>
        <title>Login with Users Online Tutorial</title>
        <link rel="stylesheet" type="text/css" href="style.css" />
    </head>
    <body>
    <form action="login.php" method="post">
        <div id="border">
            <table cellpadding="2" cellspacing="0" border="0">
                <tr>
                    <td>Username:</td>
                    <td><input type="text" name="username" /></td>
                </tr>
                <tr>
                    <td>Password:</td>
                    <td>
                        <input type="password" name="password" />
                    </td>
                </tr>
                <tr>
                    <td colspan="2" align="center">
                        <input type="submit" name="submit" value="Login" />
                    </td>
                    </tr>
                    <tr>
                        <td align="center" colspan="2">
                            <a href="register.php">Register</a> |
                            <a href="forgot.php">Forgot Pass</a></td>
                    </tr>
                </table>
            </div>
        </form>
    </body>
</html>
```

At the moment you will notice that it doesn't work. This is because we have not told the page what to do if the form is submitted.

Planning

Now let's do some planning before we dive into the PHP. We need to ask ourselves "What is the page going to be checking when the form is submitted?". For the login page here is a list of what we are going to be checking -

- That both the username and password boxes have been filled in
- That the username supplied exists in our database
- That if the username exists in our database, the password matches the one for the username
- Finally, that the user has activated their account

If the PHP can answer yes to all four of those points, then log the user in. Now in those four points you will notice there was a database mentioned. We are going to be using a MySQL database to store all of the information about each of our users. So before we get started on out PHP we need to make this database. At this point a bit more planning is needed. We need to decide what information we need to store about the users, what types of data are we storing, do we need a default value etc etc. Here is my plan below -

- We need to store a username for the user which will be a varchar
- We need a password to which will also be a varchar
- We will need an email for our email activation function this can be varchar too
- A field telling is if the account has been activated or not, this will be an integer
- A field giving information about whether the user is online or not, this will be an integer
- Finally, a field giving us a time the user registered, this is also an integer

Building the Database

Now from this we can see exactly how to build our table in our database. First create a database called loginTut. Then in this database we want to run the SQL I have provided below -

MYSQL

```
CREATE TABLE IF NOT EXISTS `users` (
`id` int(11) NOT NULL auto_increment,
`username` varchar(32) NOT NULL,
`password` varchar(32) NOT NULL,
`online` int(20) NOT NULL default '0',
`email` varchar(100) NOT NULL,
`active` int(1) NOT NULL default '0',
`rtime` int(20) NOT NULL default '0',
PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

Now we have a table to store all the information we need about our users, let's add a user for testing purposes. To do this run the SQL provided below -

MYSQL

```
INSERT INTO `users` (`id`, `username`, `password`, `online`, `email`, `active`, `rtime`)
VALUES
(1, 'testing', 'testing', 0, 'fake@noemail.co.uk', 0, 0);
```

So we now have one user with the username testing, the password testing and email fake@noemail.co.uk. Now we can get to the PHP and make out login form work!

ADDING THE PHP

First things first we need to think about security and how secure is this login form going to be. To help prevent SQL Injection which is a very common form of database hacking we are going to make a function that will protect all strings stored in the database. This we will put in an external file called functions.php. Here is the source -

PHP

```
<?php

function protect($string){
    $string = trim(strip_tags(addslashes($string)));
    return $string;
}

?>
```

This function will trim our string (cut off any white space at the beginning or end of the string), strip tags (remove all html and PHP tags in the string), and then add slashes to the string escaping speech marks (') and quotation marks (").

BACK TO LOGIN.PHP

Now we have a place to store and check user information from, a function to protect strings being passed to the database, and a nice looking layout for our login page! Below you can see the commented code for our login.php file with the newly added PHP.

```
<?php
//allow sessions to be passed so we can see if the user is logged in
session_start();
ob_start();

//connect to the database so we can check, edit, or insert data to our users table
$con = mysql_connect('localhost', 'root', '') or die(mysql_error());
$db = mysql_select_db('loginTut', $con) or die(mysql_error());

//include out functions file giving us access to the protect() function made earlier
include "./functions.php";

?>
<html>
    <head>
        <title>Login with Users Online Tutorial</title>
        <link rel="stylesheet" type="text/css" href="style.css" />
    </head>
    <body>
        <?php

        //If the user has submitted the form
        if($_POST['submit']){
            //protect the posted value then store them to variables
            $username = protect($_POST['username']);
            $password = protect($_POST['password']);

            //Check if the username or password boxes were not filled in
            if(!$username || !$password){
                //if not display an error message
                echo "<center>You need to fill in a <b>Username</b> and a
<b>Password</b>!</center>";
            }else{
                //if the were continue checking

                //select all rows from the table where the username matches the one entered by
the user
                $res = mysql_query("SELECT * FROM `users` WHERE `username` = '".$username."'");
                $num = mysql_num_rows($res);

                //check if there was not a match
                if($num == 0){
                    //if not display an error message
                    echo "<center>The <b>Username</b> you supplied does not exist!</center>";
                }else{
                    //if there was a match continue checking

                    //select all rows where the username and password match the ones submitted
by the user
                    $res = mysql_query("SELECT * FROM `users` WHERE `username` =
'".$username."' AND `password` = '".$password."'");
                    $num = mysql_num_rows($res);
```

```
                    //check if there was not a match
                    if($num == 0){
                        //if not display error message
                        echo "<center>The <b>Password</b> you supplied does not match the one
for that username!</center>";
                    }else{
                        //if there was continue checking

                        //split all fields fom the correct row into an associative array
                        $row = mysql_fetch_assoc($res);

                        //check to see if the user has not activated their account yet
                        if($row['active'] != 1){
                            //if not display error message
                            echo "<center>You have not yet <b>Activated</b> your
account!</center>";
                        }else{
                            //if they have log them in

                            //set the login session storing there id - we use this to see if
they are logged in or not
                            $_SESSION['uid'] = $row['id'];
                            //show message
                            echo "<center>You have successfully logged in!</center>";

                            //update the online field to 50 seconds into the future
                            $time = date('U')+50;
                            mysql_query("UPDATE `users` SET `online` = '".$time."' WHERE `id` =
'".$_SESSION['uid']."'");

                            //redirect them to the usersonline page
                            header('Location: usersOnline.php');
                        }
                    }
                }
            }
        }

        ?>
        <form action="login.php" method="post">
            <div id="border">
                <table cellpadding="2" cellspacing="0" border="0">
                    <tr>
                        <td>Username:</td>
                        <td><input type="text" name="username" /></td>
                    </tr>
                    <tr>
                        <td>Password:</td>
                        <td><input type="password" name="password" /></td>
                    </tr>
                    <tr>
                        <td colspan="2" align="center"><input type="submit" name="submit"
value="Login" /></td>
                    </tr>
                    <tr>
```

```
                    <td align="center" colspan="2"><a href="register.php">Register</a> | <a
href="forgot.php">Forgot Pass</a></td>
                  </tr>
               </table>
            </div>
         </form>
    </body>
</html>
<?
ob_end_flush();
?>
```

Most of this is explained by the commenting but one part I didn't explain is the online field. When you successfully login, we updated the online field to 50 seconds ahead of now. The date('U') function gives us a the amount of seconds since January 1 1970 00:00:00 GMT (Unix epoch). This means that date('U') will never get smaller, the value will always increase. If we set the online field to 50 seconds ahead of now then when the Users Online page is loaded we can check to find all the users where the online value is more than the time when the page is loaded, if this is the case then display each of their names.

Now feel free to test your login page. Make sure that all the checks are performed correctly and that once successfully logged in, you get redirected to the non existing users online page. You can also check to see if it has successfully updated the online field by checking your users table!

3. Creating the Register Page

What good is a login page without a register page? Not much at all so I think that will be the next step for us to take. Creating the register page is going to be very similar to creating our login page. We need to do some basic check to see if the username wanted is already taken, but there's nothing new happening there. Below you can see the commented register page code -

PHP

```
<?php
//allow sessions to be passed so we can see if the user is logged in
session_start();

//connect to the database so we can check, edit, or insert data to our users table
$con = mysql_connect('localhost', 'root', '') or die(mysql_error());
$db = mysql_select_db('loginTut', $con) or die(mysql_error());

//include out functions file giving us access to the protect() function
include "./functions.php";

?>
```

```
<html>
    <head>
        <title>Login with Users Online Tutorial</title>
        <link rel="stylesheet" type="text/css" href="style.css" />
    </head>
    <body>
        <?php

        //Check to see if the form has been submitted
        if(isset($_POST['submit'])){

            //protect and then add the posted data to variables
            $username = protect($_POST['username']);
            $password = protect($_POST['password']);
            $passconf = protect($_POST['passconf']);
            $email = protect($_POST['email']);

            //check to see if any of the boxes were not filled in
            if(!$username || !$password || !$passconf || !$email){
                //if any weren't display the error message
                echo "<center>You need to fill in all of the required filds!</center>";
            }else{
                //if all were filled in continue checking

                //Check if the wanted username is more than 32 or less than 3 charcters long
                if(strlen($username) > 32 || strlen($username) < 3){
                    //if it is display error message
                    echo "<center>Your <b>Username</b> must be between 3 and 32 characters
long!</center>";
                }else{
                    //if not continue checking

                    //select all the rows from out users table where the posted username
matches the username stored
                    $res = mysql_query("SELECT * FROM `users` WHERE `username` =
'".$username."'");
                    $num = mysql_num_rows($res);

                    //check if theres a match
                    if($num == 1){
                        //if yes the username is taken so display error message
                        echo  "<center>The <b>Username</b> you have chosen is already
taken!</center>";
                    }else{
                        //otherwise continue checking

                        //check if the password is less than 5 or more than 32 characters long
                        if(strlen($password) < 5 || strlen($password) > 32){
                            //if it is display error message
                            echo "<center>Your <b>Password</b> must be between 5 and 32
characters long!</center>";
                        }else{
                            //else continue checking

                            //check if the password and confirm password match
                            if($password != $passconf){
```

```
                                //if not display error message
                                echo "<center>The <b>Password</b> you supplied did not math the
confirmation password!</center>";
                            }else{
                                //otherwise continue checking

                                //Set the format we want to check out email address against
                                $checkemail = "/^[a-z0-9]+([_\\.-][a-z0-9]+)*@([a-z0-9]+([\.-
][a-z0-9]+)*)+\\.[a-z]{2,}$/i";

                                //check if the formats match
                                 if(!preg_match($checkemail, $email)){
                                        //if not display error message
                                     echo "<center>The <b>E-mail</b> is not valid, must be
name@server.tld!</center>";
                                 }else{
                                        //if they do, continue checking

                                        //select all rows from our users table where the emails
match
                                        $res1 = mysql_query("SELECT * FROM `users` WHERE `email`
= '".$email."'");
                                        $num1 = mysql_num_rows($res1);

                                        //if the number of matchs is 1
                                        if($num1 == 1){
                                                //the email address supplied is taken so display
error message
                                                echo "<center>The <b>E-mail</b> address you
supplied is already taken</center>";
                                        }else{
                                                //finally, otherwise register there account

                                                //time of register (unix)
                                        $registerTime = date('U');

                                        //make a code for our activation key
                                        $code = md5($username).$registerTime;

                                        //insert the row into the database
                                                $res2 = mysql_query("INSERT INTO `users`
(`username`, `password`, `email`, `rtime`)
VALUES('".$username."','".$password."','".$email."','".$registerTime."')");

                                                //send the email with an email containing the
activation link to the supplied email address
                                                mail($email, $INFO['chatName'].' registration
confirmation', "Thank you for registering to us ".$username.",\n\nHere is your activation
link. If the link doesn't work copy and paste it into your browser address
bar.\n\nhttp://www.yourwebsitehere.co.uk/activate.php?code=".$code, 'From:
noreply@youwebsitehere.co.uk');

                                                //display the success message
                                                echo "<center>You have successfully registered,
please visit you inbox to activate your account!</center>";
                                        }
```

```
                                }
                            }
                        }
                    }
                }
            }
        }

        ?>
        <div id="border">
            <form action="register.php" method="post">
                <table cellpadding="2" cellspacing="0" border="0">
                    <tr>
                        <td>Username: </td>
                        <td><input type="text" name="username" /></td>
                    </tr>
                    <tr>
                        <td>Password: </td>
                        <td><input type="password" name="password" /></td>
                    </tr>
                    <tr>
                        <td>Confirm Password: </td>
                        <td><input type="password" name="passconf" /></td>
                    </tr>
                    <tr>
                        <td>Email: </td>
                        <td><input type="text" name="email" size="25"/></td>
                    </tr>
                    <tr>
                        <td colspan="2" align="center"><input type="submit" name="submit"
value="Register" /></td>
                    </tr>
                    <tr>
                        <td colspan="2" align="center"><a href="login.php">Login</a> | <a
href="forgot.php">Forgot Pass</a></a></td>
                    </tr>
                </table>
            </form>
        </div>
    </body>
</html>
```

New Functions

This file contains some new things you may not be familiar with, therefore I will go over everything. Firstly, the strlen() function, this returns the number of characters in a string allowing us to check how long strings are. Then the preg_match() function, this checks to see if the formatting of a string matches the formatting you specify (in this case being an email format). Finally the mail() function, this sends an email from the server to any email of your choice, containing anything you want. You should save this file as register.php

Now you can test you register page, you can see when you enter your correct email address you will receive an email with an activation link contained inside. You can also see that a row containing the data filled into the form is entered into the users table. The value of active is 0 showing that this account has not yet been activated!

4. Activate Your Account Page

This is only a small page with very little code required, but it's still very important and plays a huge role in a secure login system. The source for this page is shown below – activate.php

PHP

```
<?php
//allow sessions to be passed so we can see if the user is logged in
session_start();

//connect to the database so we can check, edit, or insert data to our users table
$con = mysql_connect('localhost', 'root', '') or die(mysql_error());
$db = mysql_select_db('loginTut', $con) or die(mysql_error());

//include out functions file giving us access to the protect() function made earlier
include "./functions.php";

?>
<html>
    <head>
        <title>Login with Users Online Tutorial</title>
        <link rel="stylesheet" type="text/css" href="style.css" />
    </head>
    <body>
        <?php

        echo md5('other');
        //get the code that is being checked and protect it before assigning it to a variable
        $code = protect($_GET['code']);

        //check if there was no code found
        if(!$code){
            //if not display error message
            echo "<center>Unfortunatly there was an error there!</center>";
        }else{
            //other wise continue the check

            //select all the rows where the accounts are not active
            $res = mysql_query("SELECT * FROM `users` WHERE `active` = '0'");

            //loop through this script for each row found not active
            while($row = mysql_fetch_assoc($res)){
                //check if the code from the row in the database matches the one from the user
                if($code == md5($row['username']).$row['rtime']){
                    //if it does then activate there account and display success message
                    $res1 = mysql_query("UPDATE `users` SET `active` = '1' WHERE `id` =
'".$row['id']."'");
                    echo "<center>You have successfully activated your account!</center>";
```

```
                }
            }
        }

        ?>
        </div>
    </body>
</html>
```

There are two new things in this file, we use the GET method instead of POST and also we use a while() loop. The get method simply gets data from the address bar at the top of the user's browser (in this case being the code sent with the email to their email address). The while() loop is perfecting for checking through multiple rows of data selected from the database (in this case to see if there is a match with the codes).

Overview so Far

So far you should've learned many new things if your new to PHP and successfully created a half of a login system. The pages completed so far are -

- style.css
- functions.php
- login.php
- register.php
- activate.php

Some useful functions used so far are -

- mysql_connect() – Connect to a mysql database
- mysql_select_db() – Select the database that we should work with
- mysql_query() – Send querys to the database to get, insert or edit data
- trim() – Cut unwanted white space of the beginning and end of a string
- strip_tags() – Remove html and PHP tags from a string
- addslashes() – Add slashes to s string allowing quotes and speech marks to be used safely
- strlen() – Get the number of characters in a string
- preg_match() – Preg match is to match the formatting of a string
- mail() – Send mail from the server to the specified email address
- md5() – This calculates the md5 hash of a string

5. Forgotten Your Password?

Next up is our forgotten password page. If the user forgets their password, we can email it to them now we know that they supplied a real email address because of the activation. So without further ado here's the commented code for forgot.php -

PHP

```
<?php
//allow sessions to be passed so we can see if the user is logged in
session_start();

//connect to the database so we can check, edit, or insert data to our users table
$con = mysql_connect('localhost', 'root', '') or die(mysql_error());
$db = mysql_select_db('loginTut', $con) or die(mysql_error());

//include out functions file giving us access to the protect() function made earlier
include "./functions.php";

?>
<html>
    <head>
        <title>Login with Users Online Tutorial</title>
        <link rel="stylesheet" type="text/css" href="style.css" />
    </head>
    <body>
        <?php

        //Check to see if the forms submitted
        if($_POST['submit']){
            //if it is continue checks

            //store the posted email to variable after protection
            $email = protect($_POST['email']);

            //check if the email box was not filled in
            if(!$email){
                //if it wasn't display error message
                echo "<center>You need to fill in your <b>E-mail</b> address!</center>";
            }else{
                //else continue checking

                //set the format to check the email against
                $checkemail = "/^[a-z0-9]+([_\\.-][a-z0-9]+)*@([a-z0-9]+([\.-][a-z0-
9]+)*)+\\.[a-z]{2,}$/i";

                //check if the email doesnt match the required format
                 if(!preg_match($checkemail, $email)){
                    //if not then display error message
                     echo "<center><b>E-mail</b> is not valid, must be
name@server.tld!</center>";
                 }else{
                    //otherwise continue checking
```

```
                    //select all rows from the database where the emails match
                    $res = mysql_query("SELECT * FROM `users` WHERE `email` = '".$email."'");
                    $num = mysql_num_rows($res);

                    //check if the number of row matched is equal to 0
                    if($num == 0){
                        //if it is display error message
                        echo "<center>The <b>E-mail</b> you supplied does not exist in our
database!</center>";
                    }else{
                        //otherwise complete forgot pass function

                        //split the row into an associative array
                        $row = mysql_fetch_assoc($res);

                        //send email containing their password to their email address
                        mail($email, 'Forgotten Password', "Here is your password:
".$row['password']."\n\nPlease try not too lose it again!", 'From:
noreply@yourwebsitehere.co.uk');

                        //display success message
                        echo "<center>An email has been sent too your email address containing
your password!</center>";
                    }
                }
            }
        }

        ?>
        <div id="border">
            <form action="forgot.php" method="post">
                <table cellpadding="2" cellspacing="0" border="0">
                    <tr>
                        <td>Email: </td>
                        <td><input type="text" name="email" /></td>
                    </tr>
                    <tr>
                        <td colspan="2" align="center"><input type="submit" name="submit"
value="Send" /></td>
                    </tr>
                    <tr>
                        <td colspan="2" align="center"><a href="register.php">Register</a> | <a
href="login.php">Login</a></a></td>
                    </tr>
                </table>
            </form>
        </div>
    </body>
</html>
```

This page consists of nothing new therefore I will spend less time looking over it. One thing I do want to mention is that if you haven't noticed because we have been

including our css file into every page the layout we are using for each page is staying very similar keeping a nice smart design throughout the whole website.

The next and final page we will be doing in this tutorial will be slightly different. This page has the check to see if the user is logged in or not, and in this case displays all the users online at that current moment (or to be precise within the past 50 seconds).

6. The Users Online Page

Okay so we have made it to the section of the website you need to be logged in to view. As I mentioned before this one is going to be slightly different to the others because of the fact that we need to check if the user is logged in or not. If they are not logged in and try to view the page we have a few options we can do. The first being we can display an error message saying something along the lines of "You need to be logged in to view this page!", or we can redirect them back to the login page. For this tutorial I think I'm going to use the error message method.

So here is the usersOnline.php page's source -

PHP

```
<?php
//allow sessions to be passed so we can see if the user is logged in
session_start();

//connect to the database so we can check, edit, or insert data to our users table
$con = mysql_connect('localhost', 'root', '') or die(mysql_error());
$db = mysql_select_db('loginTut', $con) or die(mysql_error());

//include out functions file giving us access to the protect() function made earlier
include "./functions.php";

?>
<html>
    <head>
        <title>Login with Users Online Tutorial</title>
        <link rel="stylesheet" type="text/css" href="style.css" />
    </head>
    <body>
        <?php

        //if the login session does not exist therefore meaning the user is not logged in
        if(strcmp($_SESSION['uid'],"") == 0){
            //display and error message
            echo "<center>You need to be logged in to user this feature!</center>";
        }else{
            //otherwise continue the page

            //this is out update script which should be used in each page to update the users
online time
```

```
            $time = date('U')+50;
            $update = mysql_query("UPDATE `users` SET `online` = '".$time."' WHERE `id` =
'".$_SESSION['uid']."'");
            ?>
            <div id="border">
                <table cellpadding="2" cellspacing="0" border="0" width="100%">
                    <tr>
                        <td><b>Users Online:</b></td>
                        <td>
                        <?php

                        //select all rows where there online time is more than the current time
                        $res = mysql_query("SELECT * FROM `users` WHERE `online` >
'".date('U')."'");

                        //loop for each row
                        while($row = mysql_fetch_assoc($res)){
                            //echo  each username found to be online with a dash to split them
                            echo $row['username']." - ";
                        }

                        ?>
                        </td>
                    </tr>
                    <tr>
                        <td colspan="2" align="center"><a href="logout.php">Logout</a></td>
                    </tr>
                </table>
            </div>
            <?php

        //make sure you close the check if their online
        }

        ?>
    </body>
</html>
```

As I mentioned, you can see this page is slightly different. Not only do we ensure that they are logged in, but we update the online time keeping the online field ahead of the current time. Each time a page is loaded with that script, it will update to put them online. Now we have one more final page to do and then we are done. Once a user has logged in, he needs to be able to log out!

LOGOUT.PHP

This has to be considered the easiest page to make which I am sure most of you are glad to hear. Now here is the commented code for the logout.php file -

PHP

```
<?php
//allow sessions to be passed so we can see if the user is logged in
session_start();

//connect to the database so we can check, edit, or insert data to our users table
$con = mysql_connect('localhost', 'root', '') or die(mysql_error());
$db = mysql_select_db('loginTut', $con) or die(mysql_error());

//include out functions file giving us access to the protect() function made earlier
include "./functions.php";

?>
<html>
    <head>
        <title>Login with Users Online Tutorial</title>
        <link rel="stylesheet" type="text/css" href="style.css" />
    </head>
    <body>
        <?php

        //check if the login session does no exist
        if(strcmp($_SESSION['uid'],"") == 0){
            //if it doesn't display an error message
            echo "<center>You need to be logged in to log out!</center>";
        }else{
            //if it does continue checking

            //update to set this users online field to the current time
            mysql_query("UPDATE `users` SET `online` = '".date('U')."' WHERE `id` =
'".$_SESSION['uid']."'");

            //destroy all sessions canceling the login session
            session_destroy();

            //display success message
            echo "<center>You have successfully logged out!</center>";
        }

        ?>
    </body>
</html>
```