

Introduction to Databases for the Web: Pt. 1

By Selena Sol, August 16, 1998

What is a database?

Once upon a time, in the primitive and barbarian days before computers, the amount of information shepherded by a group of people could be collected in the wisdom and the stories of its older members. In this world, storytellers, magicians, and grandparents were considered great and honored storehouses for all that was known.

Apparently, and according to vast archeological data, campfires were used (like command-line middleware) by the younger members of the community to access the information stored in the minds of the elders using API's such as

public String TellUsAboutTheTimeWhen(String s);

And then of course, like a sweeping and rapidly-encompassing viral infection, came agriculture, over-production of foodstuffs, and the origins of modern-day commerce.

Dealing with vast storehouses of wheat, rice, and maize became quite a chore for the monarchs and emperors that developed along with the new economy. There was simply too much data to be managed in the minds of the elders (who by now were feeling the effects of hardware obsolescence as they were being pushed quietly into the background).

And so, in order to store all the new information, humanity invented the technology of writing. And though great scholars like Aristotle warned that the invention of the alphabet would lead to the subtle but total demise of the creativity and sensibility of humanity, data began to be stored in voluminous data repositories, called books.

As we know, eventually books propagated with great speed and soon, whole communities of books migrated to the first real "databases", libraries.

Unlike previous versions of data warehouses (people and books), that might be considered the australopithecines of the database lineage, libraries crossed over into the modern-day species, though they were incredibly primitive of course.

Specifically, libraries introduced "standards" by which data could be stored and retrieved.

After all, without standards for accessing data, libraries would be like my closet, endless and engulfing swarms of chaos. Books, and the data within books, had to be quickly accessible by anyone if they were to be useful.

In fact, the usefulness of a library, or any base of data, is proportional to its data storage and retrieval efficiency. This one corollary would drive the evolution of databases over the next 2000 years to its current state.

Thus, early librarians defined standardized filing and retrieval protocols. Perhaps, if you have ever made it off the web, you will have seen an old library with its cute little indexing system (card catalog) and pointers (Dewey decimal system).

And for the next couple thousand years libraries grew, and grew, and grew along with associated storage/retrieval technologies such as the filing cabinet, colored tabs, and three ring binders.

All this until one day about half a century ago, some really bright folks including Alan Turing, working for the British government were asked to invent an advanced tool for breaking German cryptographic "Enigma" codes.

That day the world changed again. That day the computer was born.

The computer was an intensely revolutionary technology of course, but as with any technology, people took it and applied it to old problems instead of using it to its revolutionary potential.

Almost instantly, the computer was applied to the age-old problem of information storage and retrieval. After all, by World War Two, information was already accumulating at rates beyond the space available in publicly supported libraries. And besides, it seemed somehow cheap and tawdry to store the entire archives of "The Three Stooges" in the Library of Congress. Information was seeping out of every crack and pore of modern day society.

Thus, the first attempts at information storage and retrieval followed traditional lines and metaphors. The first systems were based on discrete files in a virtual library. In this file-oriented system, a bunch of files would be stored on a computer and could be accessed by a computer operator. Files of archived data were called "tables" because they looked like tables used in traditional file keeping. Rows in the table were called "records" and columns were called "fields".

Consider the following example:

First Name	Last Name	Email	Phone
Eric	Tachibana	erict@eff.org	213-456-0987
Selena	Sol	selena@eff.org	987-765-4321
Li Hsien	Lim	hsien@somedomain.com	65-777-9876
Jordan	Ramacciato	nadroj@otherdomain.com	222-3456-123

The "flat file" system was a start. However, it was seriously inefficient.

Essentially, in order to find a record, someone would have to read through the entire file and hope it was not the last record. With a hundred thousands records, you can imagine the dilemma.

What was needed, computer scientists thought (using existing metaphors again) was a card catalog, a means to achieve random access processing, that is the ability to efficiently access a single record without searching the entire file to find it.

The result was the indexed file-oriented system in which a single index file stored "key" words and pointers to records that were stored elsewhere. This made retrieval much more efficient. It worked just like a card catalog in a library. To find data, one needed only search for keys rather than reading entire records.

However, even with the benefits of indexing, the file-oriented system still suffered from problems including:

- Data Redundancy - the same data might be stored in different places
- Poor Data Control - redundant data might be slightly different such as in the case when Ms. Jones changes her name to Mrs. Johnson and the change is only reflected in some of the files containing her data
- Inability to Easily Manipulate Data - it was a tedious and error prone activity to modify files by hand
- Cryptic Work Flows - accessing the data could take excessive programming effort and was too difficult for real-users (as opposed to programmers).

Consider how troublesome the following data file would be to maintain.

Name	Address	Course	Grade
Mr. Eric Tachibana	123 Kensigton	Chemistry 102	C+
Mr. Eric Tachibana	123 Kensigton	Chinese 3	A
Mr. Eric Tachibana	122 Kensigton	Data Structures	B
Mr. Eric Tachibana	123 Kensigton	English 101	A
Ms. Tonya Lippert	88 West 1st St.	Psychology 101	A
Mrs. Tonya Duconvney	100 Capitol Ln.	Psychology 102	A
Ms. Tonya Lippert	88 West 1st St.	Human Cultures	A
Ms. Tonya Lippert	88 West 1st St.	European Governments	A

What was needed was a truly unique way to deal with the age-old problem, a way that reflected the medium of the computer rather than the tools and metaphors it was replacing.

Enter the database.

Simply put, a database is a computerized record keeping system. More completely, it is a system involving data, the hardware that physically stores that data, the software that utilizes the hardware's file system in order to 1) store the data and 2) provide a standardized method for retrieving or changing the data, and finally, the users who turn the data into information.

Databases, another creature of the 60s, were created to solve the problems with file-oriented systems in that they were compact, fast, easy to use, current, accurate, allowed the easy sharing of data between multiple users, and were secure.

A database might be as complex and demanding as an account tracking system used by a bank to manage the constantly changing accounts of thousands of bank customers, or it could be as simple as a collection of electronic business cards on your laptop.

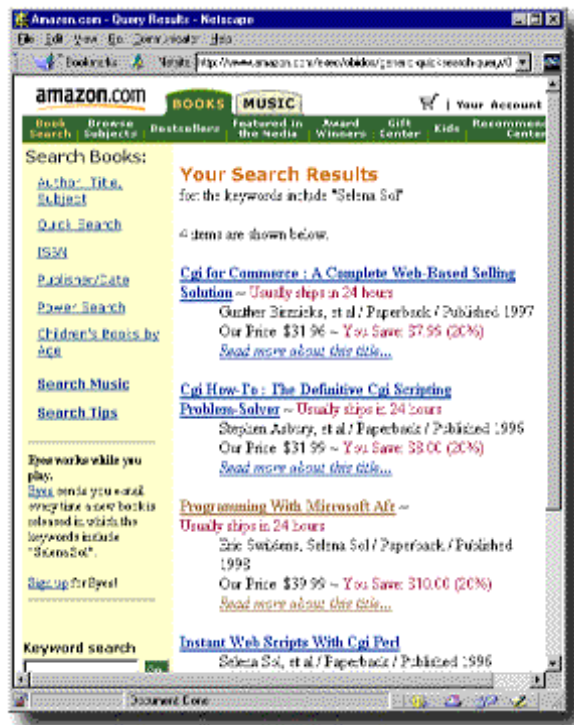
The important thing is that a database allows you to store data and get it or modify it when you need to easily and efficiently regardless of the amount of data being manipulated. What the data is and how demanding you will be when retrieving and modifying that data is simply a matter of scale.

Traditionally, databases ran on large, powerful mainframes for business applications. You will probably have heard of such packages as Oracle 8 or Sybase SQL Server for example.

However with the advent of small, powerful personal computers, databases have become more readily usable by the average computer user. Microsoft's Access is a popular PC-based engine.

More importantly for our focus, databases have quickly become integral to the design, development, and services offered by web sites.

Consider a site like Amazon.com that must be able to allow users to quickly jump through a vast virtual warehouse of books and compact disks.



How could Amazon.com create web pages for every single item in their inventory and how could they keep all those pages up to date. Well the answer is that their web pages are created on-the-fly by a program that "queries" a database of inventory items and produces an HTML page based on the results of that query.

The goal of this tutorial is to give you a rough and ready introduction to databases and give you the tools you need to get to work using the database tools available to you.

We will begin by focusing on some of the more theoretical aspects of databases so that you will have a good feel for the generic subject before we start in on all the specifics.

Types of Databases

These days, when you talk about databases in the wild, you are primarily talking about two types: analytical databases and operational databases.

Let's examine each type.

Analytic Databases

Analytic databases (a.k.a. OLAP- On Line Analytical Processing) are primarily static, read-only databases which store archived, historical data used for analysis. For example, a company might store sales records over the last ten years in an analytic database and use that database to analyze marketing strategies in relationship to demographics.

On the web, you will often see analytic databases in the form of inventory catalogs such as the one shown previously from Amazon.com. An inventory catalog analytical database usually holds descriptive information about all available products in the inventory.

Web pages are generated dynamically by querying the list of available products in the inventory against some search parameters. The dynamically-generated page will display the information about each item (such as title, author, ISBN) which is stored in the database.

Operational Databases

Operational databases (a.k.a. OLTP On Line Transaction Processing), on the other hand, are used to manage more dynamic bits of data. These types of databases allow you to do more than simply view archived data. Operational databases allow you to modify that data (add, change or delete data).

These types of databases are usually used to track real-time information. For example, a company might have an operational database used to track warehouse/stock quantities. As customers order products from an online web store, an operational database can be used to keep track of how many items have been sold and when the company will need to reorder stock.

Database Models

Besides differentiating databases according to function, databases can also be differentiated according to how they model the data.

What is a data model?

Well, essentially a data model is a "description" of both a container for data and a methodology for storing and retrieving data from that container. Actually, there isn't really a data model "thing". Data models are abstractions, oftentimes mathematical algorithms and concepts. You cannot really touch a data model. But nevertheless, they are very useful. The analysis and design of data models has been the cornerstone of the evolution of databases. As models have advanced so has database efficiency.

Before the 1980's, the two most commonly used Database Models were the hierarchical and network systems. Let's take a quick look at these two models and then move on to the more current models.

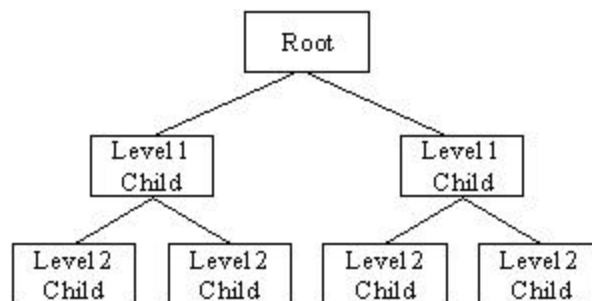
Hierarchical Databases

As its name implies, the Hierarchical Database Model defines hierarchically-arranged data.

Perhaps the most intuitive way to visualize this type of relationship is by visualizing an upside down tree of data. In this tree, a single table acts as the "root" of the database from which other tables "branch" out.

You will be instantly familiar with this relationship because that is how all windows-based directory management systems (like Windows Explorer) work these days.

Relationships in such a system are thought of in terms of children and parents such that a child may only have one parent but a parent can have multiple children. Parents and children are tied together by links called "pointers" (perhaps physical addresses inside the file system). A parent will have a list of pointers to each of their children.



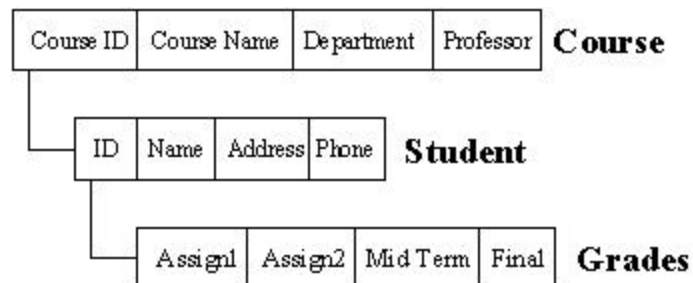
This child/parent rule assures that data is systematically accessible. To get to a low-level table, you start at the root and work your way down through the tree until you reach your target. Of

course, as you might imagine, one problem with this system is that the user must know how the tree is structured in order to find anything!

The hierarchical model however, is much more efficient than the flat-file model we discussed earlier because there is not as much need for redundant data. If a change in the data is necessary, the change might only need to be processed once. Consider the student flat file database example from our discussion of what databases are:

Name	Address	Course	Grade
Mr. Eric Tachibana	123 Kensigton	Chemistry 102	C+
Mr. Eric Tachibana	123 Kensigton	Chinese 3	A
Mr. Eric Tachibana	122 Kensigton	Data Structures	B
Mr. Eric Tachibana	123 Kensigton	English 101	A
Ms. Tonya Lippert	88 West 1st St.	Psychology 101	A
Mrs. Tonya Ducovney	100 Capitol Ln.	Psychology 102	A
Ms. Tonya Lippert	88 West 1st St.	Human Cultures	A
Ms. Tonya Lippert	88 West 1st St.	European Governments	A

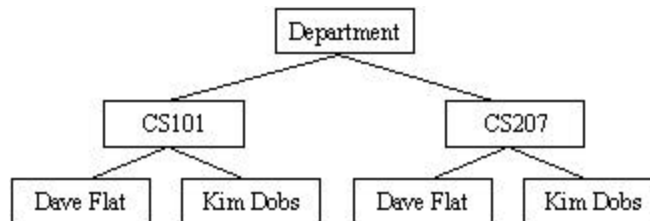
As we mentioned before, this flat file database would store an excessive amount of redundant data. If we implemented this in a hierarchical database model, we would get much less redundant data. Consider the following hierarchical database scheme:



However, as you can imagine, the hierarchical database model has some serious problems. For one, you cannot add a record to a child table until it has already been incorporated into the parent table. This might be troublesome if, for example, you wanted to add a student who had not yet signed up for any courses.

Worse, yet, the hierarchical database model still creates repetition of data within the database. You might imagine that in the database system shown above, there may be a higher level that includes multiple course. In this case, there could be redundancy because students would be enrolled in several courses and thus each "course tree" would have redundant student information.

Redundancy would occur because hierarchical databases handle one-to-many relationships well but do not handle many-to-many relationships well. This is because a child may only have one parent. However, in many cases you will want to have the child be related to more than one parent. For instance, the relationship between student and class is a "many-to-many". Not only can a student take many subjects but a subject may also be taken by many students. How would you model this relationship simply and efficiently using a hierarchical database? The answer is that you wouldn't.



Though this problem can be solved with multiple databases creating logical links between children, the fix is very kludgy and awkward.

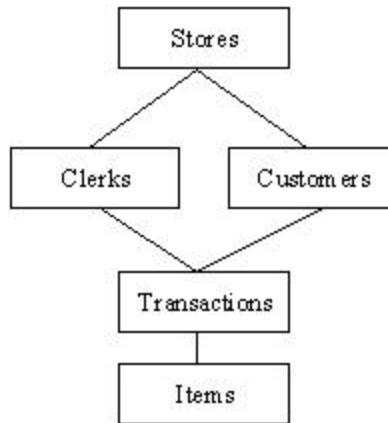
Faced with these serious problems, the computer brains of the world got together and came up with the network model.

Network Databases

In many ways, the Network Database model was designed to solve some of the more serious problems with the Hierarchical Database Model. Specifically, the Network model solves the problem of data redundancy by representing relationships in terms of sets rather than hierarchy. The model had its origins in the Conference on Data Systems Languages (CODASYL) which had created the Data Base Task Group to explore and design a method to replace the hierarchical model.

The network model is very similar to the hierarchical model actually. In fact, the hierarchical model is a subset of the network model. However, instead of using a single-parent tree hierarchy, the network model uses set theory to provide a tree-like hierarchy with the exception that child tables were allowed to have more than one parent. This allowed the network model to support many-to-many relationships.

Visually, a Network Database looks like a hierarchical Database in that you can see it as a type of tree. However, in the case of a Network Database, the look is more like several trees which share branches. Thus, children can have multiple parents and parents can have multiple children.



Nevertheless, though it was a dramatic improvement, the network model was far from perfect. Most profoundly, the model was difficult to implement and maintain. Most implementations of the network model were used by computer programmers rather than real users. What was needed was a simple model which could be used by real end users to solve real problems.

Enter the Relational Database Model.

Relational Databases

Of course in the 80's the "Relational Database Model" became the rage. The Relational Model developed out of the work done by Dr. E. F. Codd at IBM in the late 1960s who was looking for ways to solve the problems with the existing models.

Because he was a mathematician, he naturally built the model on mathematical concepts which he expounded in the famous work called "A Relational Model of Data for Large Shared Databanks".

At the core of the relational model is the concept of a table (also called a relation) in which all data is stored. Each table is made up of records (horizontal rows also known as tuples) and fields (vertical columns also known as attributes).

It is important to note that how or where the tables of data are stored makes no difference. Each table can be identified by a unique name and that name can be used by the database to find the table behind the scenes. As a user, all you need to know is the table name in order to use it. You do not need to worry about the complexities of how the data is stored on the hard drive.

This is quite a bit different from the hierarchical and network models in which the user had to have an understanding of how the data was structured within the database in order to retrieve, insert, update, or delete records from the database.

So how do you find data in a relational database if there is no map (like a hierarchy defined by pointers) to follow?

Well, in the relational model, operations that manipulate data do so on the basis of the data values themselves. Thus, if you wish to retrieve a row from a table for example, you do so by comparing the value stored within a particular column for that row to some search criteria.

For example, you might say (not getting into syntax yet) "Give me all the rows from the 'STUDENTS' table which have 'Selena' in the 'FIRST_NAME' column."

The database might return a list which looks essentially like this:

Selena	Sol	SID-001	213-456-7890
Selena	Roberts	SID-268	818-934-5069
Selena	Smith	SID-991	310-234-6475

You could then use the data from a retrieved row to query another table. For example, you might say "Okay, I want to know what grade 'Selena Sol' received in 'Underwater Basket Weaving 101'. So I will now use the 'Student ID' number from my previous query as the keyword in my next query. I want the row in the 'Underwater Basket Weaving Course' table where student ID equals 'SID-001'.

This data access methodology makes the relational model a lot different from and better than the earlier database models because it is a much simpler model to understand. This is probably the main reason for the popularity of relational database systems today.

Another benefit of the relational system is that it provides extremely useful tools for database administration. Essentially, tables can not only store actual data but they can also be used as the means for generating meta-data (data about the table and field names which form the database structure, access rights to the database, integrity and data validation rules etc).

Thus everything within the relational model can be stored in tables. This means that many relational systems can use operations recursively in order to provide information about the database. In other words, a user can query information concerning table names, access rights, or some data and the results of these queries would then be presented to the user in the form of a table.

This makes database administration as easy as usage!

Client/Server Databases

As we said before, most databases that you will come across these days will be relational databases. However, there are many types of relational databases and not all of them will be useful for web applications.

In particular, it will be the client/server databases rather than the stand-alone packages that you will use for the web.

A client/server database works like this: A database server is left running 24 hours a day, and 7 days a week. Thus, the server can handle database requests at any hour. Database requests come in from "clients" who access the database through its command line interface or by connecting to a database socket. Requests are handled as they come in and multiple requests can be handled at one time.

For web applications which must be available for world wide time zone usage, it is essential to build upon a client/server database which can run all the time.

For the most part, these are the only types of databases that Internet Service Providers will even provide. However if you are serving web pages yourself, you should consider many of the excellent freeware, shareware or commercial products around. I myself like postgres for UNIX since I prefer a UNIX-based web server. However, there are plenty of good applications for PC and Mac as well.

Good Database Design

It is most likely that as a web developer, you will be working with one of the modern relational databases and that you will be able to work in conjunction with an existing database administrator. That is, this tutorial is limited to the "use" of databases rather than to the creation and administration of them. In fact, the creation and administration of databases is a topic well beyond the scope of this tutorial and probably well beyond the scope of your job. After all, database administration is its own entire job description.

However, we have been spending a lot of time going through general database theory because although you may not be designing databases yourself, in order to take the most advantage of them, it is important that you understand how they work. Likewise, it is important that you have a feel for good database design. After all, a database's usefulness is directly proportional to the sleekness and efficiency of its design. And your ability to take advantage of a database is directly proportional to your ability to decipher and internalize that design.

When thinking about good database design, it is important that you keep data retrieval, storage and modification efficiency in mind. It will pay off one thousand fold if you take a week or two to simply play with different arrangements of data. You will find that certain table structures will provide easier and more intuitive access than others.

Tables should describe only one subject, have distinct fields, contain no redundant data, and have a field with unique values so that the table can be related to others.

You should also keep in mind future expansion of the database and make sure that your design is easily extensible.

Typically, you will go through a requirements phase in which you should simply sit with the problem, interview users, and achieve an intuition about the data and the project.

Next, you should spend time modeling the data, preferably using some standard methodology like ER Diagramming. However, even if you do not model in any traditional way, you can still play with different ideas and think about the pros and cons.

Finally, you should try out your ideas and hone them through limited trials.

Hopefully, you will also choose a database with full- functionality such as security and concurrency control (making sure that two users cannot simultaneously change a record). There are many excellent choices available in the market today from freeware to commercial products.

Of course, as we said above, you will probably be coming onto a project with an already existing database. This is the case for most web developers. In this case, you should work closely with the database administrator to define the database.

At the very least you should sit down over a brew one evening and discuss the database design thoroughly.

Talking to Databases

Once the database is up and running and populated with data, you will need a way of talking to it. Essentially, there are two ways of doing that: connecting to the database directly using a command shell or by connecting to the database over the network using sockets (if that database allows such interaction).

Command shells are pretty common and come bundled with just about every database. These are usually simple filters which allow you to log on to the database, execute commands and receive output. They can either be very simple command-line shells or nice, fanciful graphical user interfaces. Whichever the case, command shells provide access to the database from the machine on which the database is running. We will see more examples of this methodology when we discuss CGI-based web databases in part three.

Socket-based middleware does the exact same thing, but over a network. Thus, with a socket-based system, I could send commands to my database in Los Angeles while I am working in Singapore by perhaps using TCP/IP over the internet to connect to my database. We will see more examples of this when we look at JDBC-based web databases in part four.

You will almost certainly want to get access to a database and its command shell just to try things out during this tutorial. Hopefully your ISP will provide a database to test, but if not, you can probably install a database like Microsoft Access on your local system just to play with SQL commands.

Of course, whether you choose a command shell or a socket based connection, you will need to know the language of the database in order to send commands. For this tutorial, we are going to focus on SQL which is actually the de facto language of databases today.

SQL comes in many flavors depending on the proprietary database system implementing it. However, regardless of the proprietary extensions, the core of the language is the same in every environment. Fortunately, for most web uses, generic SQL will be more than sufficient.

SQL protects us from the implementation of data storage and retrieval. Essentially, so long as we use the standard commands such as:

```
SELECT * FROM database1
```

It is up to the database itself to figure out what to do. We don't need to know any of the internal workings of the database or how it stores data on the file system.

The Basics of the SQL Database

As we said, SQL (Structured Query language) is the language of choice for most modern multi-user, relational databases. That is because SQL provides the syntax and idioms (language) you need to talk to (query) relational databases in a standardized, cross-platform/product way (structured).

The beauty of SQL is that it idiomizes the relational model. Rather than refer to data as a set of pointers, SQL provides predefined procedures to allow you to use any value in a table to relate other tables in a database. So long as a database is structured using the relational model, SQL will be a natural fit because SQL was designed to make sense in a relational system. SQL by its very design is a language that can be used to talk about relating tables.

For the rest of Part One and Two, we will examine how you will use SQL to access relational databases. However, first we should say a little bit about the structure of SQL databases before we plunge into usage.

SQL databases (most modern relational databases), as you will recall, are composed of a set of row/column-based "tables", indexed by a "data dictionary". To access data in the tables, you simply use SQL to navigate the system and produce "views" based on search criteria defined in the SQL query.

Okay, that was quite a bit of jargon all at once. Let's step back for a moment and look at each of these terms.

SQL Tables

We have already discussed the concept of tables in the last part, but let's just refresh our memory in terms of how tables relate to SQL. A table is a systematic way to store data. For the most part, a table is just like a spreadsheet. Tables are composed of rows (records) and each row is composed of columns (fields).

Employee Table			
Employee ID Number	Employee Name	Employee Phone	Salary
001	Lim Li Chuen	654-3210	90,000 USD
002	Lim Sing Yuen	651-0987	40,000 USD
003	Loo Soon Keat	123-4567	50,000 USD

How the tables are stored by the database you are using does not really make a difference for you. The beauty of SQL is that it works independently of the internal structure of the database. The tables could be stored as simple flat files on a local PC or as complex, networked, compressed, encrypted and proprietary data structures.

All you need to know is the table's name. If you know the name, you can use SQL to call up the table.

We'll look at manipulating tables in detail a bit later. But first, let's look at the data dictionary.

The Data Dictionary

How does the database know where all of these tables are located? Well, behind the scenes, the database maintains a "data dictionary" (a.k.a. catalog) which contains a list of all the tables in the database as well as pointers to their locations.

Essentially, the data dictionary is a table of tables containing a list of all the tables in the database, as well as the structure of the tables and often, special information about the database itself.

When you use SQL to talk to the database and provide a table name, the database looks up the table you referred to in the data dictionary. Of course, you needn't worry about the data dictionary; the database does all the searching itself. As we said before, you just need to know the name of the table you want to look at.

It is interesting to note that because the data dictionary is a table, in many databases, you can even query the data dictionary itself to get information about your environment. This can often be a very useful tool when exploring a new database.

Okay, so how do you actually grab table data using the data dictionary? Well, in an SQL database you create "views". Let's examine views a bit.

Views

When you submit a query to an SQL database using SQL, the database will consult its data dictionary and access the tables you have requested data from. It will then put together a "view" based upon the criteria you have defined in your SQL query.

A "view" is essentially a dynamically generated "result" table that is put together based upon the parameters you have defined in your query. For example, you might instruct the database to give you a list of all the employees in the `EMPLOYEES` table with salaries greater than 50,000 USD per year. The database would check out the `EMPLOYEES` table and return the requested list as a "virtual table".

Similarly, a view could be composed of the results of a query on several tables all at once (sometimes called a "join"). Thus, you might create a view of all the employees with a salary of greater than 50K from several stores by accumulating the results from queries to the `EMPLOYEES` and `STORES` databases. The possibilities are limitless.

By the way, many databases allow you to store "views" in the data dictionary as if they were physical tables.

Basics of an SQL Query

As we have already alluded to, a "query" is a structured request to the database for data. At its core, a query is something like, "Hey, give me a list of all the clients in the `CLIENTS` table who live in the 213 area code!"

Or, in more specific terms, a query is a simple statement (like a sentence) which requests data from the database.

Much as is the case with English, an SQL statement is made up of subjects, verbs, clauses, and predicates.

Let's take a look at the statement made above. In this case, the subject is "hey you database thing". The verb is "give me a list". The clause is "from the `CLIENTS` table". Finally, the predicate is "who live in the 213 area code."

We'll explain the code later, but let me show you what the above statement might look like in SQL:

```
SELECT * FROM CLIENTS WHERE area_code = 213
```

- `SELECT` = VERB = give me a list
- `FROM CLIENTS` = CLAUSE = from the `CLIENTS` table
- `area_code = 213` = PREDICATE = who live in the 213 area code

Data Types

Okay, we are about to go into the details of SQL queries, but before that we should say one last thing about SQL database structures. Specifically, most databases store their data in terms of data types. Defining data types allows the database to be more efficient and helps to protect you against adding bad data to your tables.

There are several standard data types including

Type	Alias	Description
CHARACTER	CHAR	Contains a string of characters. Usually, these fields will have a specified maximum length that is defined when the table is created.
NUMERIC	NONE	Contains a number with a specified number of decimal digits and scale (indicating a power to which the value should be multiplied) defined at the table creation.
DECIMAL	DEC	Similar to NUMERIC except that it is more proprietary.
INTEGER	INT	Only accepts integers
SMALLINT	NONE	Same as INTEGER except that precision must be smaller than INT precisions in the same table.
FLOAT	NONE	Contains floating point numbers
DOUBLE PRECISION	NONE	Like FLOAT but with greater precision

It is important to note that not all databases will implement the entire list and that some will implement their own data types such as calendar or monetary types. Some fields may also allow a `NULL` value in them even if `NULL` is not exactly the correct type.

Okay, we will explain data types when we actually start using them, so for now, let's go on to some real examples of doing things with SQL. Let's log on to a database and start executing queries using SQL.

Logging on to the Database

It is important to note that for most databases, you will actually need to log on. That is, most databases implement a security system that gives various users different privileges to do different things such as `READ ONLY` or `READ/WRITE`.

Most likely, your database administrator will have provided you with a login name and a password. She will also have either provided you with the net address of the database so you can log in remotely, or has given you a direct command line shell.

Once you are logged in to a database, you can begin to populate the database with data or extract already existing data. Let's look first at populating the database.

A Sample Database

Okay, let's define a simple relational database that we can use to practice with...

We will define a database called "MY_COMPANY" with four tables, "CLIENTS", "EMPLOYEES", "PRODUCTS", and "SALES". These tables will look something like the following:

EMPLOYEES Table			
EMP_NUM	EMP_NAME	EMP_COMMISSION	EMP_SALARY
001	Lim Li Chuen	10%	90,000 USD
002	Lim Sing Yuen	20%	40,000 USD
003	Loo Soon Keat	20%	50,000 USD

CLIENTS Table						
C_NUM	C_NAME	C_ADDR	C_CITY	C_STATE	C_ZIP	C_PHONE
001	Jason Lim	100 W 10th St	LA	CA	90027	456-7890
002	Rick Tan	21 Jack St	LA	CA	90031	649-2038
003	Stephen Petersen	1029#A Kent Ave.	LA	CA	90102	167-3333

PRODUCTS Table		
P_NUM	P_QUANTITY	P_PRICE
001	104	99.99
002	12	865.99
003	2000	50.00

SALES Table					
S_NUM	P_NUM	S_QUANTITY	S_AMOUNT	E_NUM	C_NUM
001	001	1	99.99	101	102
002	001	2	199.98	102	101
003	002	1	865.99	101	103

Creating Databases

Okay, as we have said before, it would be too difficult for us to cover how to install and configure all of the myriad of relational databases around, so it is your job to get something installed on your local system or to arrange with your systems administrator to give you access to an existing database system.

However, for the purposes of example, I am going to use Microsoft Access as an example of an SQL database. Microsoft Access comes with Microsoft Office and is a good database to practice with since it is available for Mac, Windows and UNIX and is a relational-based database that understands SQL.

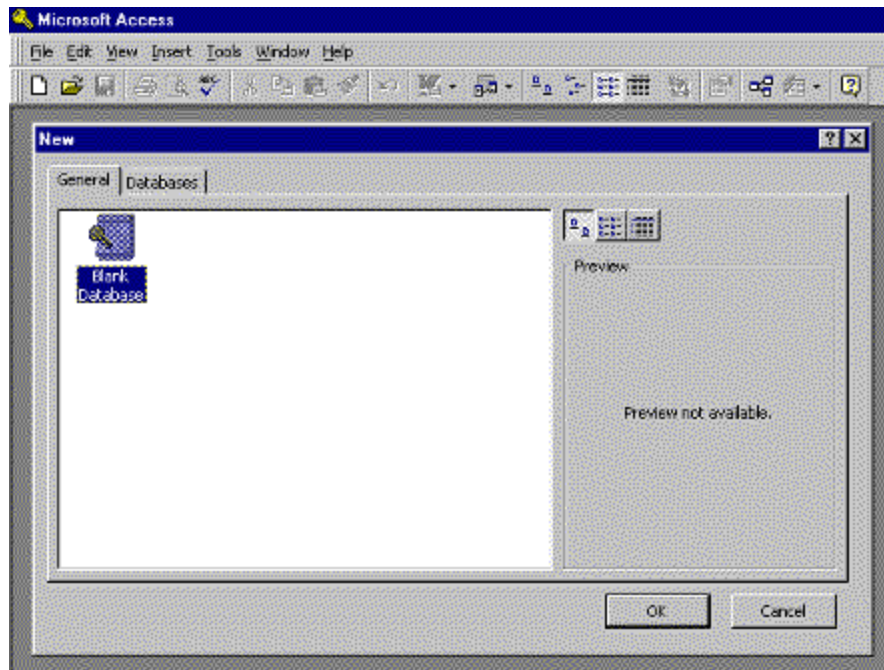
Note: Stephan Wik pointed out that I made a blunder here. Apparently, Access is not a Mac-friendly App. Doh! Microsoft blows it again. Stephen also noted that Mac Dbs include Filemaker, Butler and Oracle. Fortunately, Oracle supports DBI. I am not sure about Filemaker and Butler, so you would have to contact those companies and ask them for the Perl DBI driver.

Of course, I wouldn't necessarily use Access for a web application, because...well...because it is a Microsoft product. But it is pretty useful for practicing and demonstrating SQL since it is pretty ubiquitous. Regardless, in these days of mature software applications, whichever database you use, will have a process that is similar to the one I will describe for Access.

Specifically, to get a database working, you will 1) install the application on the host computer (insert disk A, click install.exe), 2) create a database according to the instructions of the database application you install, 3) populate your database with tables, and 4) populate your tables with data.

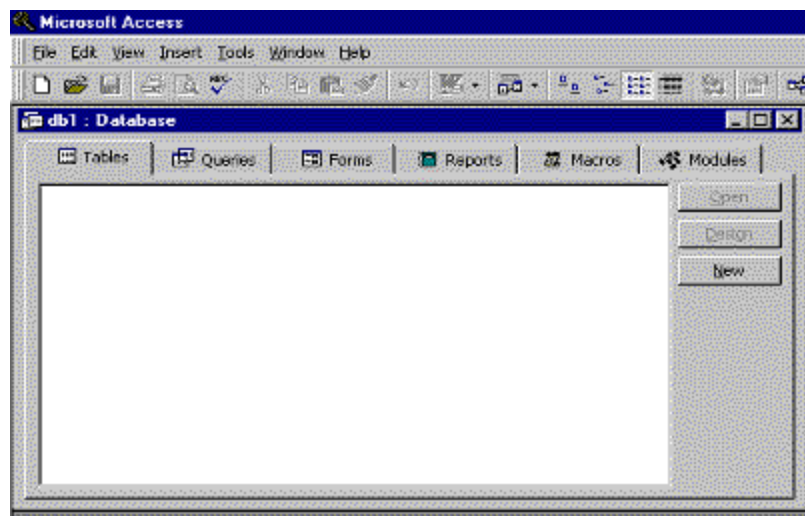
In the case of Access and many other database systems around these days, the process of creating databases and tables is pretty much handled by Wizards and GUI tools. Thus, you rarely need to use SQL for such operations. More likely, you would choose something like "FILE|NEW DATABASE" from the main menu.

Below is an example of the Wizard used by Microsoft Access. In this case, we have chosen "File|New Database" from the main menu at the top of the application window and then we double click "Blank Database". Of course, although Access and other database systems offer template databases for your convenience, we will create our own for practice.



In Access, as will be the case in most databases, once you create a database, you will then be asked to define the database structure.

Specifically, you will be asked to define some tables in your database (and perhaps other more advanced tools like Macros, Indexes, Views, Forms, Queries, etc). Below is a screen shot from Access that offers a series of database definition tabs.



To create a table, simply choose "New" from the "Tables" tab and follow instructions for defining your fields. Nothing could be simpler. Hopefully, now that you understand what the database is doing in the background, you will be easily able to understand what you need to do to get it working.

However, even though GUIs are pretty swank these days, it is probably a good idea to learn the SQL to which is being used in the background to create the database. Specifically, you use the `CREATE` command to create a database such as in the following example

```
CREATE DATABASE DATABASE_NAME;
```

We might use the following code to create a database called `MY_COMPANY`.

```
CREATE DATABASE MY_COMPANY;
```

Creating Tables

Once you have created your database you can then start populating it with tables. In the case of Access, as you saw on the last page, creating tables is as easy as clicking "New" in the table tab of the "Database" tab.

However, you should know that in the background, Access, and other GUI database systems are using the `CREATE TABLE` command to create a new table.

This command looks like the following:

```
CREATE TABLE TABLE_NAME (COLUMN_NAME DATA_TYPE, COLUMN_NAME DATA_TYPE,
COLUMN_NAME DATA_TYPE) IN DATABASE DATABASE_NAME
```

For example, you might see the following SQL code to create a table called `PRODUCTS` with three columns in the `MY_COMPANY` database we just created. Note that the three columns would be `P_NUM` which would be an integer value and could not be null, the `P_QUANTITY` which would also accept integers as values, and the `P_PRICE` column which would accept decimal numbers with 8 digits before and 2 digits after the decimal point.

```
CREATE TABLE PRODUCTS (P_NUM INT NOT NULL, P_QUANTITY INT, P_PRICE
DECIMAL(8,2)) IN DATABASE MY_COMPANY;
```

Notice that as we mentioned before, when you create a table, you must specify the data type for each column. Notice also that you may use the "NOT NULL" keyword to tell the database that it should not allow any NULL values to be added to the column.

As a final note, I would like to mention that you can also typically create Views, Indexes, and Synonyms, however, those topics are beyond the scope of this tutorial since you will most likely not be doing database administration types of activities. For most web development work, it is simply enough to define some tables.

Deleting Databases and Tables

It is also simple to delete tables and databases using GUI tools or SQL. In the case of a GUI, you will simply select a table or database from your main menu and delete it as you would a file in a file system manager.

In SQL, you would simply use the `DELETE` or `DROP` commands depending on if you were deleting a whole database or just a table in a single database.

In the case of deleting a whole database, you will use the `DELETE` command as follows:

```
DELETE DATABASE DATABASE_NAME ;
```

The following example would delete the database `MY_COMPANY`:

```
DELETE DATABASE MY_COMPANY ;
```

In the case of a table, you use the `DROP` command:

```
DROP TABLE TABLE_NAME ;
```

such as:

```
DROP TABLE EMPLOYEES ;
```

Essentially when you use delete and drop, you are modifying the database management system's data dictionary. It shouldn't have to be said, but I will say it...**BE CAREFUL WHEN DELETING OR DROPPING!**

Altering a Table

Finally, you should know that it is possible to "alter" a table after it has been created using either a standard GUI tool or by using the `ALTER SQL` command as follows:

```
ALTER TABLE TABLE_NAME DROP COLUMN_NAME, COLUMN_NAME ADD COLUMN_NAME
DATA_TYPE, COLUMN_NAME DATA_TYPE RENAME COLUMN_NAME NEW NAME MODIFY
COLUMN_NAME DATA_TYPE ;
```

such as the following case in which we alter the table named `EMPLOYEES` by dropping the `E_GENDER` Column and adding an `E_ZIP` column which will accept `INTEGERS` and which must be filled in for every new employee added to the table, and the `E_MIDDLE_INIT` column which will accept a single character as a value.

```
ALTER TABLE EMPLOYEES DROP E_GENDER ADD E_ZIP INTEGER NOT NULL,
E_MIDDLE_INIT CHAR (1) ;
```