

Databases

Database Basics

A database is a simple, yet flexible and powerful tool for storing and retrieving data. Every company, every website, has lots of data. The more of your data that you keep in your database - the better. Far from being a tool only useful to big businesses, even if you just want a simple guest book or page hit counter, a database is perfect. As you'll see in this tutorial, using a database is easy and rewarding.

Whichever database you use - it'll be a *relational* database. This is the industry standard design these days. Relational databases use the principles of [set theory](#). Set theory is a field of mathematics that describes how to deal with sets of data. Relational databases are quite intuitive and easy to understand.

This chapter describes how databases work. All the main SQL commands are explained later. Once you understand them, you'll be able to access *all* the features of the database in the way it was designed to be used..

Here are the basics:

- All data is held in **tables**.
- A table has **columns** (along the top) and **rows**.
- You create the tables you need. You define the **table names**.
- You define what the **column names** are in each table.
- You define what **type** of data the columns are...

Data Types:

There are a number of different data types available which represent the different types of data you find in real life. There are analogous types in *all* databases and programming languages. Each has variations, but they're all fundamentally the same. They are:

- **Numerical Types.** i.e. Numbers. There are fundamentally two types: integer and float. Integers are whole numbers (i.e. 1, 2, 100, 999999). Floats are numbers with decimal places (i.e. (1.1, 22.5, 3.1415927).
- **String Types.** i.e. Text. There are two types here: Fixed length, and variable length. 'char' is the only fixed length type in MySQL - from 1-255 characters. 'varchar' is a variable length field that can be 1-255 characters. There are several 'text' types of varying lengths in MySQL.
- **Date and Time Types** For storing dates & times.
- **Binary Data** This is arbitrary data, could be images, programs absolutely anything.

For a definitive list of data types, you should refer to the user manual for your database. Exact type definitions are database specific, and subject to change.

- You often need to specify the **precision** of the columns too. This is the number of digits / decimal places / characters etc. This is shown in brackets after the type, e.g. `varchar(50)` is a string of up to 50 characters.

Example 'Orders' Database Table...

OrderID (integer) (primary key)	Date (date)	Product (varchar(50))	Price (float(8.2))	CustomerID (integer)
1	3-4-2001	BookA	12.95	1
2	4-4-2001	BookB	8.95	1
3	6-4-2001	BookA	12.95	2

...there would be one row per sale.

- You define which column is the **primary key**. This is the column that is *guaranteed different* for all the rows in the table. In the example table above, it would be the 'order_id' column.
- You or your programs **populate** the tables with **rows of data**.
- Tables are **related** to each other by the *columns they have in common*. (Hence *relational* database!)

e.g. If you have a 'customers' and a 'sales' table, then you'd have a 'customer_id' column which is in *both* tables!

Example 'Customers' Database Table Structure...

CustomerID	Name	Address	Email
1	Bob	3 The Cottages...	bob@warmmail.com
2	Fred	128 Long Ave...	f22@mymail.com
3	Bill	2 Broad St...	billyb@warmmail.com

...there would be one row per customer.

- You can *interact* with the database in at least two different ways:
 1. You can modify table or submit query into a program with graphical user interface (GUI), such as Microsoft Access, or type commands into a character-based *user-interface program, such as MySQL*, and see a response. e.g. you'd type a `'create table...'` command to create a table, and then see a `'...table created ok!'` response.
 2. You access the database from a program you've written. The language you use has to have special functions to:
 - Connect to the database

- Submit a query
- Get the results back, line by line.
- Disconnect from the database

There are other methods for getting data in and out of databases including:

- Report Generators and 'Data Mining' software.
- Programs to load / save entire databases.
- Programs that automatically synchronize two or more databases.

Indexes

All Relational Databases use **indexes**. Similar to the index in a book, indexes provide a quick way to find the exact data item you want.

Imagine you have a database of 100,000 customers, and you want to find just one. If you just read the 'customers' table from start to finish until you find the one your searching for, you could end up having to read all 100,000 records. This would be very slow. Most relational databases use a [b-tree](#) index structure. This is a clever algorithm that guarantees that you can find a data item by reading *at most* 3 rows from the index. Databases commonly have millions of rows - so you can see the necessity for indexes!

Indexes are a large part of databases and their design. Defining a column as the primary key *implicitly* creates an index. If you have a primary key on a table - it has an index.

You can add a number of indexes to each table you have. You'd use the `create index` command - more later...

Indexes are *used* automatically by the database itself when you issue a **query** (ask for data). It uses the index to find the data in the table . For example, we want to get a customer's details from the example 'customers' table above...

If we submit the following SQL query, the database will use the index it created for our primary key column 'customer_id', and get everything for customer 1:

```
select * from customers where customer_id = 1;
```

The database uses the index because it *can* use it. The query contains the 'customer_id' so it can look in the index and find the location of customer '1'.

However, if you only have a street address for the customer, you have to write the following query to get the same data, and the database can't use the primary key index because it doesn't have a customer_id:

```
select * from customers where addr1 = "3 The Cottages";
```

If we do this query often, we can add an index on the 'addr1' column to the 'customers' table. Then the database will be able to use that index.

If there's no index on the column in the query, the database will have to go through the whole table! This is called a **full table scan**

Today database software (e.g.,MySQL) is blindingly fast and will do a full table scan, even on large tables, in a very short time. It's barely worth indexing tables of one hundred rows or

less. However, when you've got more than a few hundred rows in a table, you should definitely start to think about putting indexes on your most queried columns.

INFO: When you issue a query to the database, the '**Query Optimiser**' first decides how it will execute the query - which order to join the tables, which indexes to use etc. You can provide hints to some databases to tell them which indexes to use.

Very small tables don't benefit from being indexed. If you had a 'customers' table with just a few customers in it, then a full table scan would be quicker than going via the index.

As a general rule you should provide a primary key for all tables, and only provide additional indexes when you have hundreds of rows, *and* you know that you're going to be running queries that select data using a column *other than* the primary key.

© TRK Hosting 2003, with modification by Leith Chan