Databases and SQL

Table of Contents

Useful resources....1
Flat files....1
Relational databases....2
Database servers....3
Database systems architecture....4
Normalisation....5
Users, groups and permissions....7
Views....7
Schema and data dictionary....7
Advantages of database use....8
SQL....8
SQL Tutorial....9
Creating the database and tables....10
Inserting new rows....12
Basic queries....13
Queries using WHERE....14
LIKE....14
GROUP BY....15
DROP....15
ORDER BY....15
UPDATE fields values....16
JOIN tables....17
DELETE rows....17
Nested queries....18
Referential integrity....18
DDL and DML....19
Transactions and ACID....19
Commit and roll-back....20
Concurrency and record-locking....22

Useful resources

Khan Academy SQL
MySQL official complete reference
Wikibook on SQL
BCCampus book on database design
SQL syntax wiki
Apache SPARK SQL

Flat files

Flat files are data files made of *records*, with each record divided into *fields*. For example a file for personnel would have one record per employee. The fields might be first name, last name, department, pay grade and so on. One field would be the *key field* - such as employee payroll number. Each employee has a different key field.

Data files like this were standard for mainframe computers during the 1950s and 1960s. The files would probably be on tape and *accessed sequentially*.

Simple flat files still have some uses. For example in a game, saving the game might involve writing the map location, health, weapons and whatever to a file. A simple data file would be good for this purpose.

Relational databases

A database usually means

- Using a *server* - a piece of software dedicated to handling data file access. The server is usually separate from the application which uses it.
- The data is stored in some structure on a *non-volatile medium* - on local disc or possibly in the cloud. Applications do not access these files directly - they use them indirectly, through the server.

Two types of database are *hierarchical* and *network* - but the most common type currently is a *relational database*.

A relational database is made of *tables*. A table is usually about one thing - such as customers, or products, or suppliers, or library books, or students, or classes and so on. A database might have 5 or 10 or a 100 tables.

Each table will have a set of *columns*. For example the columns in a student table might have column headings firstName, lastName, dob (date of birth), phoneNumber and so on. These column names are also called *attributes* or *fields*.

We do not use age as a column, since someone's age changes every day. We use date of birth instead.

Tables also have *rows* or *records*. One row is about one thing. So in a students table, one row is about one student.

A table might have many thousands, or typically millions, of rows.

A table should have a *key field*. This is distinct (different for every row), and is used to identify which thing is which. For example in a

students table, there would probably be a field named studentID, which would identify the student.

We cannot use name as a key field, because two people can have the same name.

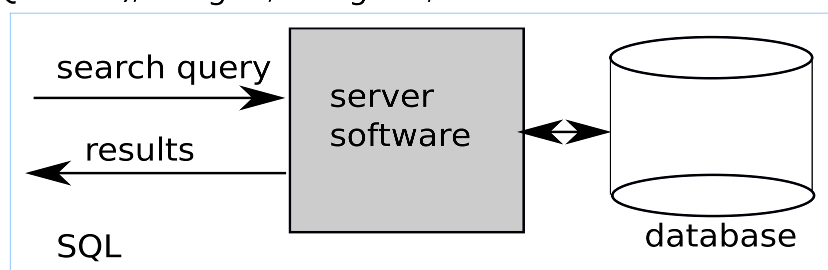
Tables can have several keys, in which case one is the *primary key*.

A key might be a combination of two fields - such as name and date of birth. This is known as a *compound key*.

A field in a table, which is a primary key in another table, is called a *foreign key*. This is explained under normalisation.

Database servers

A database server is a piece of software which provides access to a database (or several databases). These are products from *vendors* such as Oracle, Microsoft (SQLServer), Postgres, MongoDB, and MariaDB. Some companies provide more than one server (for example Oracle provides both Oracle and MySQL). These are mostly free for non-commercial use.



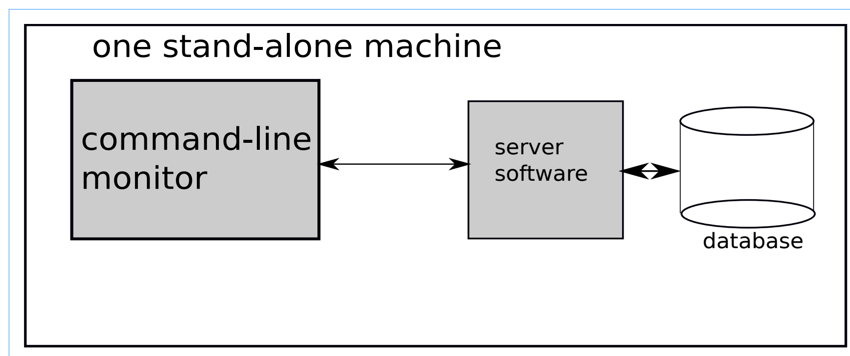
The database server receives commands usually in a language known as *SQL* (structured query language) and returns results, for example the results of a database search.

The database might consist of many files on disc. Application developers do not know, or care. Only people who write database servers think about that. Applications do not access the database directly, only through the server. This is an example of *abstraction*. We do not care how Oracle actually codes their databases - just that it works.

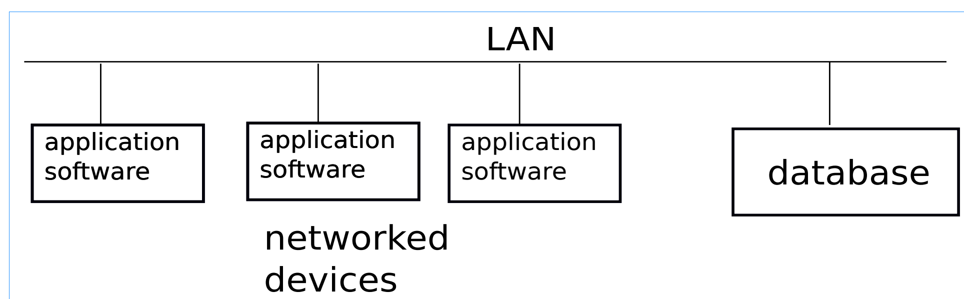
We might have several databases on one hardware machine. Or for very large systems, one database might be spread over several machines in a server farm.

Database systems architecture

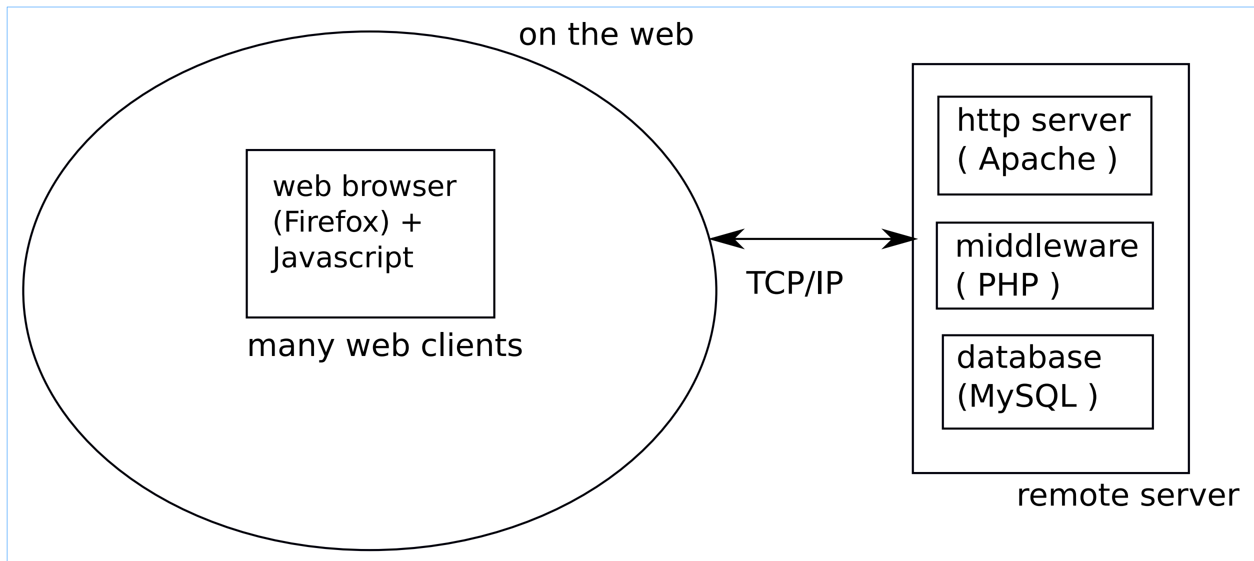
This means how and where the server, database and user are connected. The simplest is for the user to use a 'monitor', a piece of software which can connect to a server and in which SQL commands can be typed in and results displayed, in a single device, as in the tutorial in this document:



But we often want the data to be shared by several users. One possibility is a set of devices connected over a *LAN*. This might be used for example in a book lending library:



or we might have a web-based client-server architecture, such as for e-commerce. The items in brackets (like Firefox) are just examples of possibilities -



The remote server (hardware) is running:

- a *web server* like Apache
- some *middleware* scripts, using PHP or ASP or JSP and so on
- a database server (maybe *MySQL*) accessing a database.

The middle-ware consists of a set of scripts, with names like login.php. The web server is configured to pass requests for these to a PHP interpreter (for example), which can execute them. Those scripts can connect and access the database. Results of the SQL are converted to html and passed to the server, which sends them back to the client web browser.

Normalisation

This is about database design. Here is an example problem. Suppose a supermarket has a database with a table of products:

ProductID	Description	SupplierName	SupplierAddress
2365	Beans	Acme Supplies	1 Acacia Avenue
5677	Celery	Bona Foods	21 High Street
1432	Nuts	Bona Foods	21 High Street
1567	Lettuce	Acme Supplies	1 Acacia Avenue
3412	Bananas	Acme Supplies	11 Acacia Avenue
2566	Coconuts	Bona Foods	21 High Street

There are two issues. One is *data duplication*. We repeat the supplier address, obviously wasting space. The second is *data inconsistency*. The 5th row has a different address for Acme Supplies. There must be an error somewhere.

We can fix this with a different database design. We have a table of products:

ProductID	Description	SupplierID
2365	Beans	1
5677	Celery	2
1432	Nuts	2
1567	Lettuce	1
3412	Bananas	1
2566	Coconuts	2

and a supplier table:

SupplierID	Name	Address
1	Acme Supplies	1 Acacia Avenue
2	Bona Foods	21 High Street

The products table has primary key ProductID. The supplier table has primary key SupplierID. The tables are linked through the SupplierID in the products table. This is a *foreign key* - a field in a table which is a primary key in another table.

This solves both problems. We have no space wasted through data duplication. And we have no possibility of inconsistent copies, because we have no copies. If we need to update, say, a supplier address, we only need do it in one place.

This is the idea of *normalisation* - the best design of tables in a database.

Different normal forms are defined:

First normal form (1NF) in each table cell there is only 1 value. For example we could have a suppliers table with a cell containing a list of all product IDs that they supply, separated by commas. But that list, in one cell, would be several data values, so that is not 1NF.

Second normal form (2NF) - in 1NF and no composite primary keys. A composite key has 2 or more fields. These are logically separate, and should be in separate tables.

Third normal form (3NF) - in 2NF, and *no non-key field depends on another*. In our first version, for example, the supplier address obviously depends on the supplier name. So it was not in 3NF. We fixed it by putting suppliers in a separate table.

There are other forms beyond the 3rd.

Databases should at least be in 3NF.

Users, groups and permissions

A database will have a set of users, each with a unique user name, and identified by a password. This is controlled by the database server.

Each user will have a set of *permissions* relating to what they can access and alter, in terms of read access, write access, delete, alter table structure and so on.

A *user group* is a group of users with the same access rights. These might be normal users, super users and admins, for example.

Views

Imagine a doctor's clinic. The receptionist needs to see patient contact details - but their medical history should be hidden. But the doctor must be able to see the medical history, and is not interested in where they live.

This is supported by the idea of a *view*. A view picks out selected fields from tables, not every column. Different user groups see different views - not simply tables.

Schema and data dictionary

A database schema is the structure of a database - what tables there are, what fields are in which table, how there are linked and so on. A schema is *meta-data* - data about data.

A data dictionary is related to a schema, and lists which tables each data item is stored in.

Advantages of database use

Over simple data files:

- Avoid data duplication and multiple updates through *normalisation*
- Fast - servers will use techniques such as *binary search on indexes* and *caching* for optimised speed
- *Tools* for common operations like backup, export, import, mirroring
- Matching *user needs* - different groups of users see the data they require through views
- *Security* - different user groups have appropriate access permissions
- *Standard* -concepts and SQL

SQL

SQL is Structured Query Language.

It is a standard, with the first version approved by ANSI in 1986, and the latest by ISO in 2019. Database vendors mostly comply with it, with variations.

SQL is a *declarative language*, not *procedural*. For example

SELECT name FROM suppliers

is a command to fetch the name column from the suppliers table. But it does not say *how* to do this.

SQL is not a complete language - it does not have loops for example. Vendors have extended it to complete procedural languages, such as PL/SQL from Oracle and T/SQL from Microsoft.

But a more common approach is to enable general purpose languages - such as Java and Python - to connect to a database server and send SQL queries and receive returned responses.

SQL Tutorial

This is a tutorial on some basic SQL commands. It creates a very simple database with two tables, inserts some data, and does some basic SQL queries.

It uses a MySQL server:

```
walter@mint2 ~ $ mysql -V
mysql  Ver 14.14 Distrib 5.7.32, for Linux (x86_64) using  EditLine wrapper
```

running on Linux Debian Mint

```
walter@mint2 ~ $ hostnamectl
   Static hostname: mint2
         Icon name: computer-desktop
           Chassis: desktop
        Machine ID: 5ab3c275b7304ed3b8aeef9ffcc37eb4
           Boot ID: ea71f080deaa45648b2175886d07267b
  Operating System: Linux Mint 18.1
            Kernel: Linux 4.4.0-194-generic
      Architecture: x86-64
```

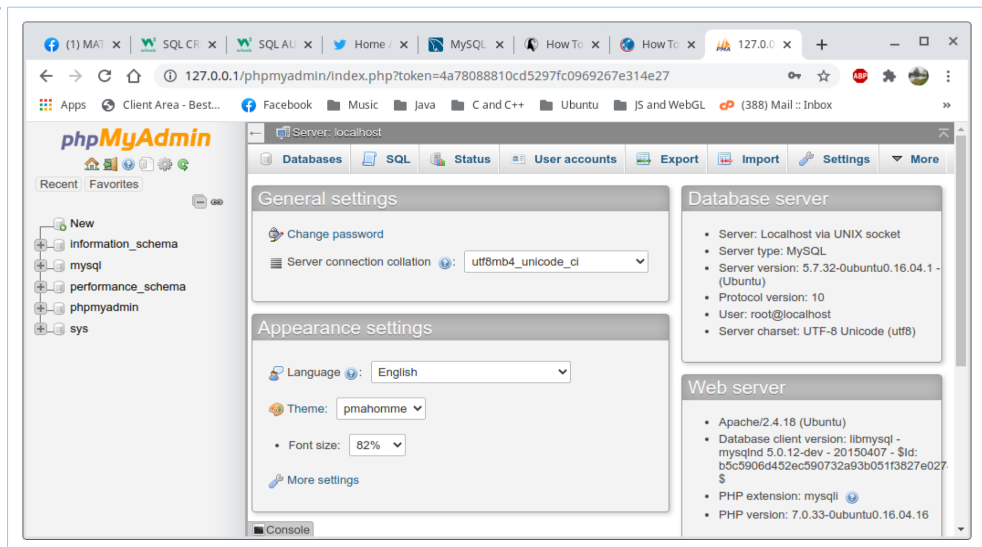
But SQL is a standard, so the commands should work on nearly all systems - this is the idea of a standard.

We are issuing commands using the mysql command-line monitor, part of the standard MySQL installation:

```
walter@mint2 ~ $ sudo mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6
Server version: 5.7.32-0ubuntu0.16.04.1 (Ubuntu)

Copyright (c) 2000, 2020, Oracle and/or its affiliates....
```

There are other ways to do this. One example is phpmyadmin:



This is a web-based interface, which enables the user to enter commands through a GUI interface, which it then translates into SQL commands and sends them to the database server. But since the point of this is to learn basic SQL, we use the command line version.

In practice we are more likely to have these SQL commands issued by some application, written in a language such as Java or Python. The SQL queries would return a recordset, which might be displayed or further processed. But it is easier to first learn SQL alone, and then embed it in another language.

This is just a brief introduction to basic SQL syntax.

Creating the database and tables

What databases do we have at the moment?

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| phpmyadmin         |
| sys                |
+--------------------+
5 rows in set (0.00 sec)
```

The server stores data (such as users) in databases, and this is what these are.

We can create a new database:

```
mysql> create database supermarket;
Query OK, 1 row affected (0.01 sec)
```

and say we want to use it:

```
mysql> use supermarket;
Database changed
```

Initially there are no tables:

```
mysql> show tables;
Empty set (0.02 sec)
```

We create a table:

```
mysql> CREATE TABLE Products (
    ->     productID int NOT NULL AUTO_INCREMENT,
    ->     description varchar(255) NOT NULL,
    ->     inStock int,
    ->     supplierID int,
    ->     PRIMARY KEY (productID)
    -> );
Query OK, 0 rows affected (0.49 sec)
```

This creates a table named Products. It has 4 *fields*, named productID, description, inStock and supplierD.

The field 'description' has *type varchar*(255), which means it is a character string with variable length up to 255 characters. The other 3 are type 'int' - integers.

productID and description are *NOT NULL*. This means they are not allowed to be blank. If we try to insert a new product without a description, this will fail.

productID is *AUTO_INCREMENT*. This means it will automatically increase by 1 every time we add a new product. This means the productID will be distinct, different for every product, and we do not need to remember all the product IDs used up to now.

The *primary key* of this table is productID. This must be unique, and adding a new product with the same ID as an existing one will fail. Searching the table with this field will be very fast.

Create a second table:

```
mysql> CREATE TABLE Suppliers (
    ->     supplierID int NOT NULL AUTO_INCREMENT,
    ->     name varchar(255) NOT NULL,
    ->     address varchar(255),
```

```
    ->     PRIMARY KEY (supplierID)
    -> );
Query OK, 0 rows affected (1.08 sec)

mysql> show tables;
+-----------------------+
| Tables_in_supermarket |
+-----------------------+
| Products              |
| Suppliers             |
+-----------------------+
2 rows in set (0.00 sec)

mysql>
```

Some possible data types are:

CHAR(n) - character string with length fixed at n

VARCHAR(n) - string with variable length up to n

BOOLEAN - true, false and unknown (not Oracle)

INTEGER - integer

REAL - real number

DATE

TIME

Date and time vary with vendor. In Oracle DATE includes data and time, and has no TIME type).

BLOB - binary large object. A use for this is for example storing an image or sound in a database.

Inserting new rows

```
mysql> INSERT INTO Suppliers (name, address) VALUES ("Acme foods", "12 High
Street");
Query OK, 1 row affected (0.09 sec)

mysql> INSERT INTO Suppliers (name, address) VALUES ("Bona Supplies", "102
Main Road");
Query OK, 1 row affected (0.08 sec)
```

Check the result:

```
mysql> SELECT * FROM Suppliers;
+------------+---------------+----------------+
| supplierID | name          | address        |
+------------+---------------+----------------+
|          1 | Acme foods    | 12 High Street |
```

```
|          2 | Bona Supplies | 102 Main Road  |
+------------+---------------+----------------+
2 rows in set (0.00 sec)
```

INSERT INTO takes a list for field names (name, address) and matching values ("Acme foods", "12 High Street").

We did not give values for the supplierID field, because this is set to be auto-increment, and the server has assigned values 1 and 2 to this column.

We insert some new products in the same way:

```
mysql> INSERT INTO Products (description, inStock, supplierID) VALUES
("Cornflakes", "200",1);
Query OK, 1 row affected (0.03 sec)
```

and get:

```
mysql> SELECT * FROM Products;
+-----------+-------------+---------+------------+
| productID | description | inStock | supplierID |
+-----------+-------------+---------+------------+
|         1 | Beans       |     100 |          1 |
|         2 | Cornflakes  |     200 |          1 |
|         3 | Rice        |     300 |          1 |
|         4 | Potatoes    |      30 |          1 |
|         5 | Noodles     |     300 |          2 |
|         6 | Burgers     |      30 |          2 |
|         7 | Salmon      |      30 |          2 |
|         8 | Celery      |     300 |          2 |
+-----------+-------------+---------+------------+
8 rows in set (0.00 sec)
```

Basic queries

To fetch all rows, and all columns, we can say

```
mysql> SELECT * FROM Products;
```

where the * means 'all fields'.

We can just fetch selected fields by naming them:

```
mysql> SELECT productID, description  FROM Products;
+-----------+-------------+
| productID | description |
+-----------+-------------+
|         1 | Beans       |
|         2 | Cornflakes  |
|         3 | Rice        |
|         4 | Potatoes    |
|         5 | Noodles     |
|         6 | Burgers     |
```

```
|         7 | Salmon      |
|         8 | Celery      |
+-----------+-------------+
8 rows in set (0.03 sec)
```

Queries using WHERE

We often have tables with millions of rows, and we rarely want to fetch all of them. We can use a *WHERE clause* to filter only certain rows:

```
mysql> SELECT *  FROM Products WHERE supplierID=1;
+-----------+-------------+---------+------------+
| productID | description | inStock | supplierID |
+-----------+-------------+---------+------------+
|         1 | Beans       |     100 |          1 |
|         2 | Cornflakes  |     200 |          1 |
|         3 | Rice        |     300 |          1 |
|         4 | Potatoes    |      30 |          1 |
+-----------+-------------+---------+------------+
4 rows in set (0.05 sec)
```

As well as =, we can use > and other relational operators:

```
mysql> SELECT *  FROM Products WHERE inStock<100;
+-----------+-------------+---------+------------+
| productID | description | inStock | supplierID |
+-----------+-------------+---------+------------+
|         4 | Potatoes    |      30 |          1 |
|         6 | Burgers     |      30 |          2 |
|         7 | Salmon      |      30 |          2 |
+-----------+-------------+---------+------------+
3 rows in set (0.00 sec)
```

and compound queries:

```
mysql> SELECT *  FROM Products WHERE inStock<100 AND supplierID=1;
+-----------+-------------+---------+------------+
| productID | description | inStock | supplierID |
+-----------+-------------+---------+------------+
|         4 | Potatoes    |      30 |          1 |
+-----------+-------------+---------+------------+
1 row in set (0.01 sec)
```

We can use AND, OR and NOT.

LIKE

We can have a WHERE clause which matches some pattern using LIKE:

```
mysql> select * from Products where description like "%o%";
+-----------+-------------+---------+------------+
```

```
| productID | description | inStock | supplierID |
+-----------+-------------+---------+------------+
|         2 | Cornflakes  |     200 |          1 |
|         5 | Noodles     |     300 |          2 |
|         7 | Salmon      |      30 |          2 |
+-----------+-------------+---------+------------+
3 rows in set (0.00 sec)
```

a % means zero or more characters, so %o% is any string containing the letter 'o'. An underscore _ means any single character. In some dialects, * means zero or more characters.

GROUP BY

Suppose we want to know how many products each supplier supplies:

```
mysql> SELECT COUNT(productID), supplierID FROM Products GROUP BY supplierID;
+------------------+------------+
| COUNT(productID) | supplierID |
+------------------+------------+
|                3 |          1 |
|                4 |          2 |
+------------------+------------+
2 rows in set (0.01 sec)
```

GROUP BY puts together rows with the same field value.

COUNT() counts rows. We can also say SUM and AVG for average, for numeric types.

DROP

This deletes a database and its contents:

```
mysql> DROP DATABASE test;
ERROR 1008 (HY000): Can't drop database 'test'; database doesn't exist
```

Use with *extreme care*.

ORDER BY

In a relational database, the rows and columns are in effect not in any order. This means the columns might be in some order left to right, or the server might keep different columns in different files. Or it might change the order for more efficient access. Same for the rows. It may access rows through different indexes, which make it seem the rows are in different orders at the same time.

But we can tell the server to make a search, then fetch the rows in some order:

```
mysql> SELECT *  FROM Products ORDER BY description;
+-----------+-------------+---------+------------+
| productID | description | inStock | supplierID |
+-----------+-------------+---------+------------+
|         1 | Beans       |     100 |          1 |
|         6 | Burgers     |      30 |          2 |
|         8 | Celery      |     300 |          2 |
|         2 | Cornflakes  |     200 |          1 |
|         5 | Noodles     |     300 |          2 |
|         4 | Potatoes    |      30 |          1 |
|         3 | Rice        |     300 |          1 |
|         7 | Salmon      |      30 |          2 |
+-----------+-------------+---------+------------+
8 rows in set (0.06 sec)
```

Exactly what 'ORDER BY' means depends on the data type. For ints they will be in numerical order, and for strings it will be alphabetical (and in more detail, it depends on the *collation order*, including what *character set* is used, such as Unicode, and what *encoding*, such as UTF8).

We can also say ASCENDING or DESCENDING

```
mysql> SELECT *  FROM Products ORDER BY inStock DESC;
+-----------+-------------+---------+------------+
| productID | description | inStock | supplierID |
+-----------+-------------+---------+------------+
|         3 | Rice        |     300 |          1 |
|         5 | Noodles     |     300 |          2 |
|         8 | Celery      |     300 |          2 |
|         2 | Cornflakes  |     200 |          1 |
|         1 | Beans       |     100 |          1 |
|         4 | Potatoes    |      30 |          1 |
|         6 | Burgers     |      30 |          2 |
|         7 | Salmon      |      30 |          2 |
+-----------+-------------+---------+------------+
8 rows in set (0.02 sec)
```

UPDATE fields values

We can change field values using UPDATE:

```
mysql> UPDATE Products SET inStock = inStock+10 WHERE productID=8;
Query OK, 1 row affected (0.11 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM Products WHERE productID =8;
+-----------+-------------+---------+------------+
| productID | description | inStock | supplierID |
+-----------+-------------+---------+------------+
```

```
|         8 | Celery       |     310 |          2 |
+-----------+-------------+---------+------------+
1 row in set (0.02 sec)
```

JOIN tables

In a normalized design we usually need to access more than table to get the information we want. Usually this means 'joining'two or more tables.

For example, suppose we need to know the address of the supplier of product id 2 (to order some more, for example):

```
mysql> select description, address from Products join Suppliers on
Products.supplierID=Suppliers.supplierID where productID=2;
+-------------+----------------+
| description | address        |
+-------------+----------------+
| Cornflakes  | 12 High Street |
+-------------+----------------+
1 row in set (0.00 sec)
```

Because the field name 'supplierID is used in both tables, we need to say something like Products.supplierID to specify the supplierID field in table Products.

DELETE rows

This can be done like

```
mysql> DELETE FROM Products WHERE productID=4;
Query OK, 1 row affected (0.15 sec)
```

But

- The deletion cannot be undone
- In an information system, deleting data is usually a bad design.

For example we might delete a product in a supermarket because it is discontinued and no longer sold. But that deletion means data on sales of that product are no longer valid, and old orders to re-stock the product now refer to something which no longer exists. So we have paid for something, but we do not know what.

Capturing data costs money. Deleting data is in effect deleting money.

In this example a better design would be to have a field in the Products table for 'current', with boolean type. When we discontinue a product we simply set its current field to be false.

Nested queries

Usually a query applies to a table, as in

select something from some table

But it is possible for a query to use the results of another query. For example

```
mysql> select description
    -> from Products
    -> where productID IN
    -> ( SELECT productID
    -> FROM Products
    -> WHERE supplierID=2);
+-------------+
| description |
+-------------+
| Noodles     |
| Burgers     |
| Salmon      |
| Celery      |
+-------------+
4 rows in set (0.05 sec)
```

Referential integrity

In our supermarkets database we had rows like:

```
+-----------+-------------+---------+------------+
| productID | description | inStock | supplierID |
+-----------+-------------+---------+------------+
|         3 | Rice        |     300 |          1 |
```

Suppose there is no row in the suppliers table with ID 1? That would mean the data was invalid, either in the products table or in the suppliers table.

We can make the server prevent such invalidity, by having a *constraint* in the table definition, like

```
CREATE TABLE Products (
    ->     productID int NOT NULL AUTO_INCREMENT,
    ->     description varchar(255) NOT NULL,
    ->     inStock int,
    ->     supplierID int,
    ->     PRIMARY KEY (productID),
    ->     CONSTRAINT supCheck FOREIGN KEY (supplierID) REFERENCES Suppliers
(supplierID)
    -> );
```

Then when we insert data into the Products table the server will check there is a matching supplier in the suppler table.

This gives us *referential integrity* - that a foreign key in a table refers to a row which actually exists in the other table.

DDL and DML

SQL has two parts - a data definition language *DDL*, and a data manipulation language *DML*.

The DDL lets us create and alter a *database structure*. So CREATE and ALTER and DROP are DDL commands.

SELECT, DELETE, UPDATE fetch or alter the *data in a database*, and so are DML commands.

Transactions and ACID

This is about making sure a database system works correctly.

An example problem would be as follows. A customer transfers some money between accounts remotely, online. The following sequence happens

1. The customer logs on
2. He removes $1000 from account A
3. He adds $1000 to account B
4. He logs out

But between steps 2 and 3 there is a power failure and the bank's database is off. The customer has lost $1000 with no record of where it has gone.

A database *transaction* is a sequence of steps which must be handled as a single item. The transaction must work despite hardware and software failures, power loss, network interruption and so on.

ACID is an *acronym* to support this - Atomicity, Consistency, Isolation and Durability.

Atomicity - means treating the transaction as a single atom - a unit which should not be split into steps. In our account transfer example, we must have either a debit and a credit together, or neither. See *commit and roll-back*

Consistency - means the data meets some *validation rules*, for what valid data should comply with. In our example a rule might be -

sum of accounts = previous sum + deposits in - transfers out

There were no transfers in or out, so the sum of the accounts should be unchanged.

Isolation - each transaction should take place by itself. In real databases many users are doing transactions at the same time. These should not corrupt each other. See *concurrency*.

Duration - transaction changes should last, an dbe permanent. For example a databse change might be confirmed to the user, but in fact is held cached in memory and not yet written to disc. Then there is a power failure and so the change is lost.

Commit and roll-back

This is a way to get atomicity - the A in ACID. Instead of carrying out individual commands, we have a work unit which is either

```
begin work
.. SQL commands
commit
```

or

```
begin work
.. SQL commands
roll-back
```

A commit means the database changes are *made permanent*. A roll-back means the *changes are discarded* and the database is restored to its state before the begin work.

For example - the initial state:

```
mysql> select * from Products;
+-----------+-------------+---------+------------+
| productID | description | inStock | supplierID |
+-----------+-------------+---------+------------+
|         1 | Beans       |     100 |          1 |
|         2 | Cornflakes  |     200 |          1 |
|         3 | Rice        |     300 |          1 |
|         5 | Noodles     |     300 |          2 |
|         6 | Burgers     |      30 |          2 |
|         7 | Salmon      |      30 |          2 |
|         8 | Celery      |     310 |          2 |
+-----------+-------------+---------+------------+
7 rows in set (0.17 sec)
```

Start a transaction, and reduce all stock levels by 20

```
mysql> begin work;
Query OK, 0 rows affected (0.00 sec)

mysql> update Products set inStock = instock-20;
Query OK, 7 rows affected (0.31 sec)
Rows matched: 7  Changed: 7  Warnings: 0

mysql> select * from Products;
+-----------+-------------+---------+------------+
| productID | description | inStock | supplierID |
+-----------+-------------+---------+------------+
|         1 | Beans       |      80 |          1 |
|         2 | Cornflakes  |     180 |          1 |
|         3 | Rice        |     280 |          1 |
|         5 | Noodles     |     280 |          2 |
|         6 | Burgers     |      10 |          2 |
|         7 | Salmon      |      10 |          2 |
|         8 | Celery      |     290 |          2 |
+-----------+-------------+---------+------------+
7 rows in set (0.00 sec)
```

Now rollback because there was some problem - discard changes

```
mysql> rollback;
Query OK, 0 rows affected (0.07 sec)

mysql> select * from Products;
+-----------+-------------+---------+------------+
| productID | description | inStock | supplierID |
+-----------+-------------+---------+------------+
```

```
|          1 | Beans         |      100 |          1 |
|          2 | Cornflakes    |      200 |          1 |
|          3 | Rice          |      300 |          1 |
|          5 | Noodles       |      300 |          2 |
|          6 | Burgers       |       30 |          2 |
|          7 | Salmon        |       30 |          2 |
|          8 | Celery        |      310 |          2 |
+------------+---------------+----------+------------+
7 rows in set (0.00 sec)
```

Concurrency and record-locking

A database is usually used by more than one user at the same time. This is called *concurrent access*.

It might result in problems. For example this might happen in an ecommerce example

user A checks the stock level of a product	10 in stock
user B checks stock level	10 in stock
user A buys 10 items	stock level is 0
user B buys 10 items	stock level is -10
database state invalid. User B does not get his goods	

Preventing this achieves isolation - the I in ACID.

One method is *locking* - data is locked when it is being accessed, and locked data cannot be accessed by another user.

One simple technique is file locking - a file in use is locked and cannot be accessed by others. Other users must wait until the lock is released.

A better technique is record-locking - just one row is locked, not the entire file. This is better because users accessing other rows are not affected and do not need to wait.