# Databases

## Relational Databases

A **database** is a collection of data stored in an organised and logical way. Data are stored in **tables** and tables are made up of **records** (rows) which can have 1 more **attributes** (columns). More complex **relational databases** have multiple tables linked together by shared attributes called a key.

Relational databases make it easier to search and find information that you want.

Relational databases reduce the amount of duplication (redundancy) of data.

| | |
|---|---|
| **Record** | An object about which data can be collected (eg person or item) |
| **Attribute** | A characteristic of a record |
| **Primary Key** | All tables have a primary key to uniquely identify each record. This is also known as entity identifier |
| **Foreign Key** | These are primary keys that are held in other tables to cross reference data between tables allowing tables to be linked together |
| **Composite Primary Key** | A primary key that has more than one attribute in order to uniquely identify the entity |

## Entity relationships

**One-to-one**


A phone can only have one number and a number can only be allocated to one phone

**One-to-many**


A player can only belong to one football club, but a club can have many players

**Many-to-Many**


A recipe can have many ingredients and an ingredient can be in many recipes

Entity descriptions have the following form:
`Entity(Attribute1, Attribute 2, ..)`

Example
`Customer(CustomerID,CustomerFirstName, ..)`

We define the primary key by underlining that attribute
`Entity(Attribute1, Attribute 2, …..)`

Example:
`Customer(CustomerID,CustomerFirstName, ..)`

We underline all the attributes that make up the composite primary key
`Entity(Attribute1, Attribute 2, …..)`

## Normalisation

**Purpose of Normalisation**
- To simplify the data structure
- To reduce data redundancy and to remove repeating data
- To be able to update, remove and search the database
- To reduce the size of the database

**First Normal form (1NF)**
The data are atomic. Atomic means the data cannot be broken down into smaller components. There are no repeating groups of attributes.
- Remove repeating attributes (columns) from table.
- Each record (row) needs a unique identifier

**Second normal form (2NF)**
- Need to be in 1NF (so data are atomic and no repeating groups of attributes)
- No partial key dependencies of non-key attributes

For each non key attribute we determine whether it is dependent on any of the attributes that make up the primary key

**Third normal form (3NF)**
- Need to be in 1NF (so data are atomic and no repeating groups of attributes)
- Need to be in 2NF (no partial key dependencies of non-key attributes)
- No dependencies on non-key attributes

## Structured Query Language

**Define a database table**

```
CREATE TABLE Entity(
      attribute1 DATATYPE PRIMARY KEY,
      attribute2 DATATYPE,
      ..);
```

Datatypes include: `TEXT, INTEGER, FLOAT, BOOLEAN, DATE`

*Example*

```
CREATE TABLE books(
      bookID INTEGER PRIMARY KEY,
      title TEXT,
      author TEXT,
      year INTEGER,
      publisher TEXT,
      genre TEXT);
```

**Insert Statement -** INSERT INTO is a commonly used command in SQL for adding new records to database tables.

```
INSERT INTO Entity (attribute 1, attribute 2, ..) VALUES
(value 1, value 2, ..);
```

*Example*

```
INSERT INTO books (bookID, title, author, genre) VALUES
(1, 'Harry Potter and the Order of the Phoenix', 'JK
Rowling', 'Children');
```

**Delete Statement -** To delete a record we specify which record(s) from which table we wish to remove.

```
DELETE FROM books WHERE author='JK Rowling';
```

**Update Statement -** makes changes to a record that is already in a table we can use the UPDATE statement.

```
UPDATE books SET publisher="HarperCollins" WHERE
title="War Horse"
```

**Select Statement -** To retrieve data from the table we can use the SELECT statement.

```
SELECT * from table;
```
Example: `SELECT * from books;`

The * is the wild card and means select everything from the books table.

We can also choose the attributes that we wish to retrieve

```
SELECT attribute 1, attribute 2 from table;
```
Example: `SELECT title, author from books;`

We can avoid selecting repeating data and can select distinct data using:
`SELECT DISTINCT author FROM books;`

We can sort the output of our SELECT statement by using the ORDER BY clause. ASC and DESC refer to sorting ascending and descending alphabetically and numerically.

```
SELECT title, author FROM books ORDER BY title ASC;
SELECT title, author FROM books ORDER BY title DESC;
```

The WHERE clause is used to filter records so that we do not apply a statement to a whole table.
`SELECT * FROM books WHERE author="Michael Morpurgo"`

When selecting the data from multiple tables we need to specify the name of the table from which each attribute we are wishing to retrieve. We need to specify that we only wish to select the records where the primary key and foreign key match.

```
SELECT books.title, authors.name, from authors, books
WHERE authors.author_ID=books.author_ID;
```

We can also use the INNER JOIN clause to select data from a pair of tables that have the same values to get the same result.

SELECT books.title, authors.name from authors INNER JOIN books ON authors.author_ID=books.author_ID;

## Client Server database

Client server databases allow simultaneous access from many clients over a network. The difficulty arises when more that client wishes to access and modify the same data at the same time. Access can be managed in the following ways:

**Record locks** - A record lock only allows records to be modified by one client at a time. The records remain locked to all other clients so they cannot be modified simultaneously by these clients. Once the client has completed the session then the records can be unlocked until another client starts to edit the data.

**Serialisation** - Serialisation only allows one transaction (read and write operation) at a time (in serial) from multiple connected clients, so that transactions from different clients do not interfere with one another.

**Timestamp ordering** – read and write operations will have a timestamp. Each data item in the database also has a read and write timestamp of their most recent access. The purpose is not to lose data by overwriting. Some rules include: The transactions with earlier timestamp will be processed first. If there is a transaction which tries to write to a database with a later timestamp, the transaction will not be allowed.

**Commitment ordering** - Some transactions will have priority over others according to their dependencies as well as their timestamps.