Syllabus Content

1.8.1 Database Management Systems (DBMS)

- understanding the limitations of using a file-based approach for the storage and retrieval of data
- describe the features of a relational database & the limitations of a file-based approach
- show understanding of the features provided by a DBMS to address the issues of:
  - data management, including maintaining a data dictionary
  - data modeling
  - logical schema
  - data integrity
  - data security, including backup procedures and the use of access rights to individuals/groups of users
- show understanding of how software tools found within a DBMS are used in practice:
  - developer interface
  - query processor11
- show that high-level languages provide accessing facilities for data stored in a Database

1.8.2 Relational database modeling

- show understanding of, and use, the terminology associated with a relational database model: entity, table, tuple, attribute, primary key, candidate key, foreign key, relationship, referential integrity, secondary key and indexing
- produce a relational design from a given description of a system
- use an entity-relationship diagram to document a database design
- show understanding of the normalisation process: First (1NF), Second (2NF) and Third Normal Form (3NF)
- explain why a given set of database tables are, or are not, in 3NF
- make the changes to a given set of tables which are not in 3NF to produce a solution in 3NF, and justify the changes made

File-based Systems

A flat file database is a type of database that stores data in a single table. This is unlike a relational database, which makes use of multiple tables and relations.

- Flat-File databases hold all of their data in **one table only**.
- They are only suitable for **very simple databases**.

The patient database is an example of a flat-file as all of the information is stored in one single table:

Flat-File (one table)

Patient Id	Name	D.o.B	Gender	Phone	Doctor Id	Doctor	Room
134	Jeff	4-Jul-1993	Male	7876453	01	Dr Hyde	03
178	David	8-Feb-1987	Male	8635467	02	Dr Jekyll	06
198	Lisa	18-Dec-1979	Female	7498735	01	Dr Hyde	03
210	Frank	29-Apr-1983	Male	7943521	01	Dr Hyde	03
258	Rachel	8-Feb-1987	Female	8367242	02	Dr Jekyll	06

Limitations of a Flat-File Database

The problems with using a flat-file databases are as follows:

- **Duplicated Data** is often **unnecessarily** entered.
- **Database space** is wasted with this duplicated data.
- **Duplicated Data** takes a long time to enter and update (unnecessarily).

What is Data Redundancy?

Data Redundancy is where you store the **same data many times** (duplicate data) in your table.

This repeated data needs to be **typed in over and over again** which takes a long time.

For example:-
The patients database contains several entries of duplicate data:

- **Doctor Id**
- **Dr. Hyde**
- **Room 03**

Patient Id	Name	D.o.B	Gender	Phone	Doctor Id	Doctor	Room	
134	Jeff	4-Jul-1993	Male	7876453	01	Dr Hyde	03	Duplicate
178	David	8-Feb-1987	Male	8635467	02	Dr Jekyll	06	
198	Lisa	18-Dec-1979	Female	7498735	01	Dr Hyde	03	Duplicate
210	Frank	29-Apr-1983	Male	7943521	01	Dr Hyde	03	
258	Rachel	8-Feb-1987	Female	8367242	02	Dr Jekyll	06	

REMEMBER!
Data that is duplicated unnecessarily within a database is bad practice. If we had 100 patients who were all assigned Dr Hyde. His **Doctor Id**, **Name** and **Room Number** would have to be **entered 100 separate times**.

Also if Dr Hyde left the doctors surgery, we would have to update the new doctors details for every patient in the database.

Key Words:
Relationships, Primary Key, Foreign Key, Common Field

What is the solution to Data Redundancy?

To avoid the data redundancy with flat-file databases is to create a **relational database**.

NOTE!
Here I have moved all the **repeating data** into **a table of its own**.

Now I have a **patient table** (for patient details) and a **doctor table** (for doctor details)

Relational Databases

- Relational Databases use **two or more tables** linked together (to form a relationship).
- Relational Databases do not store all the data in the same table.
- Repeated data is moved into it's **own table** as shown in the image below:

Patient Table

Patient Id	Name	D.o.B	Gender	Phone	Doctor Id
134	Jeff	4-Jul-1993	Male	7876453	01
178	David	8-Feb-1987	Male	8635467	02
198	Lisa	18-Dec-1979	Female	7498735	01
210	Frank	29-Apr-1983	Male	7943521	01
258	Rachel	8-Feb-1987	Female	8367242	02

Doctor Table

Doctor Id	Doctor	Room
01	Dr Hyde	03
02	Dr Jekyll	06

What is a relationship?

- A relationship is formed when our **two tables are joined together**.
- Relationships make use of **key fields** and **primary keys** to allow the **two tables to communicate** with each other and **share their data**.
- Key fields are identified using a primary key as shown in the image below:

Key Field

Patient Table

Patient Id	Name	D.o.B	Gender	Phone	Doctor Id
134	Jeff	4-Jul-1993	Male	7876453	01
178	David	8-Feb-1987	Male	8635467	02
198	Lisa	18-Dec-1979	Female	7498735	01
210	Frank	29-Apr-1983	Male	7943521	01
258	Rachel	8-Feb-1987	Female	8367242	02

Key Field

Doctor Table

Doctor Id	Doctor	Room
01	Dr Hyde	03
02	Dr Jekyll	06

- Once the tables are linked together each one can read data from the other.
- This means that we only need to enter the details of each doctor once instead of many separate entries.

How do you form the relationship? (link the tables)

- In order to link the tables we need to use a **common field**.
- A common field is **data that appears in BOTH tables**.
- If you look at the image below you will see that the common field in the patient database is **Doctor Id**:

Patient Table

Patient Id	Name	D.o.B	Gender	Phone	Doctor Id
134	Jeff	4-Jul-1993	Male	7876453	01
178	David	8-Feb-1987	Male	8635467	02
198	Lisa	18-Dec-1979	Female	7498735	01
210	Frank	29-Apr-1983	Male	7943521	01
258	Rachel	8-Feb-1987	Female	8367242	02

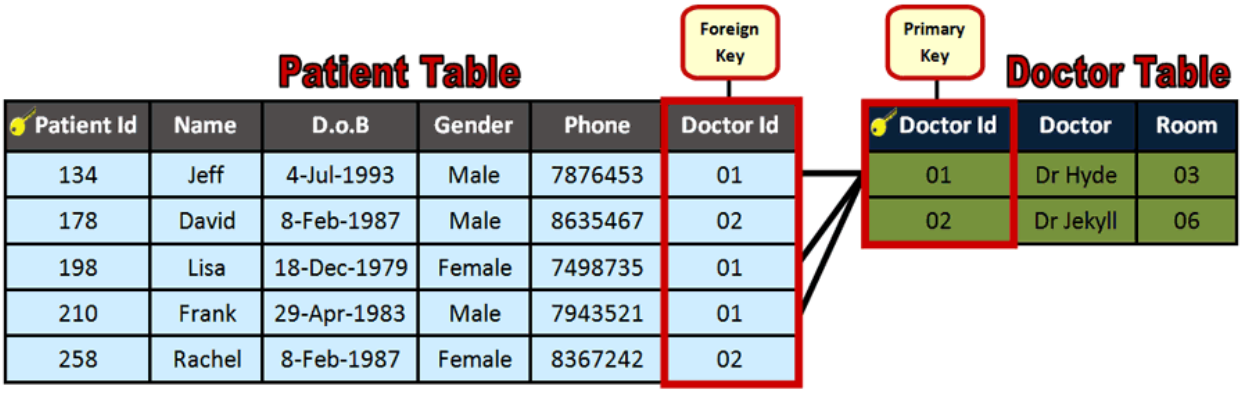
Doctor Table

Doctor Id	Doctor	Room
01	Dr Hyde	03
02	Dr Jekyll	06

What is a foreign key and what are they used for?

- A foreign key is a **regular field in one table** which is being used as the **key field in another table**.
- Foreign keys are used to **provide the link** (relationship) between the tables.

For example:-
In our patient database, **Doctor Id** is a **key field in the Doctor Table** but is also being used in the **Patient Table as a foreign key**:



Patient Id	Name	D.o.B	Gender	Phone	Doctor Id
134	Jeff	4-Jul-1993	Male	7876453	01
178	David	8-Feb-1987	Male	8635467	02
198	Lisa	18-Dec-1979	Female	7498735	01
210	Frank	29-Apr-1983	Male	7943521	01
258	Rachel	8-Feb-1987	Female	8367242	02

Doctor Id	Doctor	Room
01	Dr Hyde	03
02	Dr Jekyll	06

- The foreign key (Doctor Id in the patient table) can then be used to match to the primary key (Doctor Id in the doctor table) and share the correct data.
- **For example**:-
  A patient with a **Doctor Id of 01** will be **automatically assigned** to **Doctor Hyde and Room 03**.

REMEMBER!
Now that we have linked our two tables we can update our doctor information very quickly.

For example:-
If Dr Hyde changed his **room number to 02**, we only need to **change this information once** in the doctor table. The new room would **automatically be assigned to every patient** who was under the care of Dr Hyde.

Take another example for comparison between a Flat-File and a relational database.

For example, a library database could have three tables:

1. **Customers** - when a customer joins the library a record is created. It stores Details such as name and address and includes a unique Customer ID. This will be stored in a table along with the details of other customers.
2. **Books** - each book in the library has a record. It stores details about the book, such as the author and title and includes a unique book ID. This will be stored in a table along with the details of other books.
3. **Lending** - when a customer borrows a book, the lending table stores the customer's unique ID and the book's unique ID in a record. The record could also include additional information such as when the book was borrowed and when it's due back.

The **customer ID** and **book ID** are both examples of key fields.

Flat file database

Book	Customer name	Customer address	Date borrowed	Date due	Overdue?
Aesop's Fables	A Manning	2 Main St	20-Jun	05-Jul	N
War and Peace	T Brown	34 High St	15-Jun	30-Jun	N
DIY Disasters	T Handless	6 Glebe Cr	05-Jun	20-Jun	Y
Great Expectations	T Brown	34 High St	21-Jun	04-Jul	N

Relational database with three file tables

Books

Book ID	Book
245Y	Aesop's Fables
105C	War and Peace
50P	DIY Disasters
1006T	Great Expectations

Customers

Customer ID	Customer name	Customer address
10023	A Manning	2 Main Street
11656	T Brown	34 High Street
98636	T Handless	6 Glebe Crescent

Lending

Customer ID	Book ID	Date borrowed	Date due	Overdue?
10023	245Y	20-Jun	05-Jul	N
11656	105C	15-Jun	30-Jun	N
98636	50P	05-Jun	20-Jun	Y
10023	1006T	21-Jun	04-Jul	N

Primary keys

The primary key of a table within a relational database is a field which uniquely identifies each record in the table. It can be pre-existing data, for example National Insurance Number.

Alternatively it can be generated specifically for use in the database, eg admission number for a school student.

The primary key of a table may also be linked to a foreign key. This allows two tables to be linked. Special care must be taken while inserting data and removing data from the foreign key

column, as a careless deletion or insertion might destroy the relationship between the two tables.

For example, in the library example, tables for **books** and **customers** can be linked to the table **lending** by introducing foreign keys that refer to the book ID in the **books** table and customer ID in the **customers** table.

The customer ID column exists in both **customers** and **lending**tables, while the Book ID column exists in both the **books** and **lending** tables.

These become foreign keys, referring to the primary keys in the **customers** and **books** tables.

Surrogate key

When an entity (table) does not have a naturally occurring primary key it is possible to create a new field that will serve as the primary key. It is not uncommon to use an autonumber field to automatically allocate a unique number to each record in a table. Use of autonumber usually happens when there is no other unique value that would naturally exist in a table and is an example of making use of a surrogate key.

Composite key

A composite key is a specific type of primary key which uses the contents of two or more fields to create a unique value.

Consider the number of times certain footballers scored a goal during a tournament.

Team	Squad number	Goals
Aberdeen	9	4
Hearts	8	3
Celtics	8	5
Queen of the South	11	6
Aberdeen	8	5

In the above example a single field would not create a set of unique values, eg there are two Aberdeen players, and there are three players with the squad number 8.

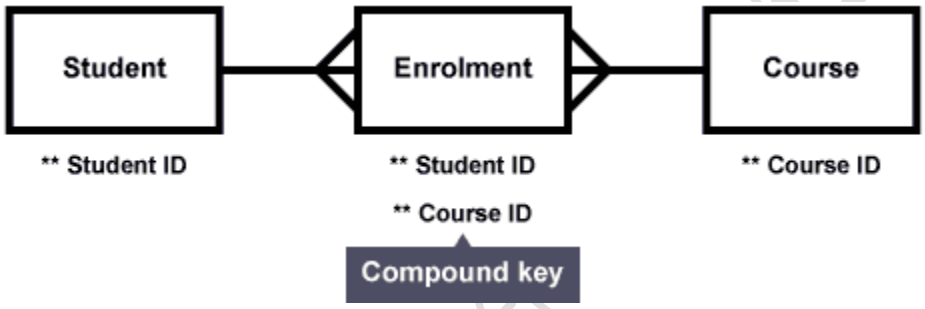
However, if you combine the **Team** and **Squad number** fields, a primary key is created, allowing each player to be uniquely identified. This is an example of a composite key.

Compound key

A **compound key** is similar to a **composite key** in that two or more fields are needed to create a unique value. However, a compound key is created when two or more primary keys from different tables are present as foreign keys within an entity. The foreign keys are used together to uniquely identify each record.

Compound keys are always made up of two or more primary keys from other tables. In their own tables, both of these keys uniquely identify data but in the table using the compound key they are both needed to uniquely identify data.

For example, a database about school may already contain, a student table with student number as the primary key. There may be a second table for each course with a primary key called course number. Class registers could be held in a table called enrolment, with the unique identifier for enrolment in a class being the combination of the student id and the course id.



Types of relationships

The relationship between two linked tables can be described in one of three ways:

- One-to-one
- One-to-many
- Many-to-many

One-to-one cardinality

One-to-one relationships occur when there is a direct one to one link between data held on two different tables. For example, each pupil in Scotland has their own unique Scottish Candidate Number. It is not possible for a pupil to have more than one number and it is not possible for the same number to be allocated to two different pupils. This is a one-to-one relationship:

Pupil entity

Candidate number	First name	Surname	Date of Birth
SCN2312345	Harry	Wilson	04/03/2000
SCN4565432	Jenny	McMillan	17/11/1998
SCN5565532	Hamish	Moore	30/07/1999

Candidate Number entity

Candidate number	Date of registration
SCN2312345	15/08/2012
SCN4565432	13/08/2010
SCN5565532	19/08/2011

There is a direct one-to-one relationship between a pupil and a candidate number. One to one relationships are illustrated by the following notation(s):

This is how the relationship would be shown on an entity relationship diagram:

One-to-one relationships are usually unnecessary, as combining the data held in both entities is often possible without resulting in any duplication of data.

In this example the data could be combined into one entity.

Candidate Entity

Candidate number	First name	Surname	Date of Birth	Date of Registration
SCN2312345	Harry	Wilson	04/03/2000	15/08/2012
SCN4565432	Jenny	McMillan	17/11/1998	13/08/2010
SCN5565532	Hamish	Moore	30/07/1999	19/08/2011

Candidate Number is the primary key in the Candidate entity.

One-to-many cardinality

One-to-many relationships exist where one instance of an entity can exist lots of times in another entity.

A good example relates to Pastoral Care or Guidance teachers in school. Here is an entity holding details on each Pastoral Care/Guidance teacher.

Pastoral Care Teacher entity

PastoralID	First name	Surname	House group
PC001	Maria	McLuskey	Kelso
PC002	Craig	Bell	Iona
PC003	Sean	Blake	Melrose

PastoralID is the primary key in the Pastoral Care Teacher entity.

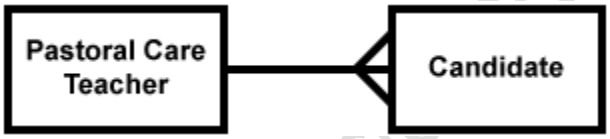
One Pastoral Care/Guidance teacher will have many pupils in their care. If we add a field to our 'Candidate' entity, we can see that one Pastoral Care teacher can be responsible for many pupils. We have added the primary key from the pastoral care teacher entity. When we add a primary key from another table, it becomes a foreign key in this table.

Updated candidate entity

Candidate number	First name	Surname	Date of Birth	Date of Registration	Pastoral Care Teacher
SCN2312345	Harry	Wilson	04/03/2000	15/08/2012	PC002
SCN4565432	Jenny	McMillan	17/11/1998	13/08/2010	PC001
SCN5565532	Hamish	Moore	30/07/1999	19/08/2011	PC001

One pastoral care teacher (PC001 - Miss McLuskey) has many pupils. The above table shows both Hamish and Jenny have Miss McLuskey as their Pastoral Care teacher. There is a one-to-many relationship between the Pastoral Care Teacher entity and the candidate entity. One-to-many relationships are illustrated by the following notation(s):

This is how the relationship would be shown on an entity relationship diagram:

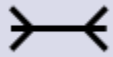


Many-to-many cardinality

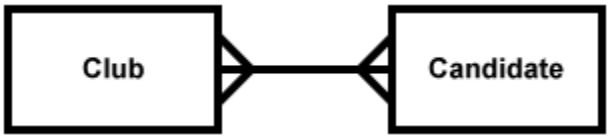
Most schools offer a variety of extracurricular clubs. Clubs can accommodate many pupils and pupils can attend many clubs. This is an example of a many-to-many relationship.

Many-to-many relationships are not good when designing a relational database as they can lead to unnecessary storage of the same data more than once (duplicated) and can make it difficult to avoid errors such as update, deletion or insertion anomalies.

Many-to-many relationships should not be present in a relational database but can be represented on entity relationship diagrams using the following notation(s):

Many-to-many M:N or 

The many-to-many relationship between clubs and pupils would be represented as follows on an entity relationship diagram:



For many-to-many relationships it is necessary to create a third entity, called an associate entity. The new table/entity will have two one-to-many relationships with both of the current entities.

Resolving many-to-many relationships

In this example a third entity called membership could be created. A compound key would be used to uniquely identify each record in the new table.

Candidate

Candidate number	First name	Surname	Date of Birth	Date of Registration	Pastoral Care Teacher
SCN2312345	Harry	Wilson	04/03/2000	15/08/2012	PC002
SCN4565432	Jenny	McMillan	17/11/1998	13/08/2010	PC001
SCN5565532	Hamish	Moore	30/07/1999	19/08/2011	PC001

Membership

Candidate Number	Club ID
SCN2312345	001
SCN2312345	003
SCN4565432	003
SCN5565532	002

Club

Club ID	Name	Teacher	Day	Location
001	Chess	Mr Fergusson	Wednesday	F101
002	Dodgeball	Miss Jones	Friday	Games Hall
003	Volleyball	Mrs Ali	Monday	Gym

The primary key from the Candidate table (Candidate number) and the primary key from the Club table (Club ID) are present as foreign keys in the Membership table. They combine to create a compound key which can be used to uniquely identify each member of each club. We have created two one to many relationships to remove the many to many relationship from the database. The relationships between the entities would look like this on an entity relationship diagram:

The relationships are now:

- One pupil (candidate) can have many memberships
- One club can have many members

The many-to-many relationship has been replaced by two one-to-many relationships. There is a fully worked example of how to identify the relationship between entities and create entity relationship diagrams in the **Design Notation** learner guide.

Degrees of Relationship (Cardinality)

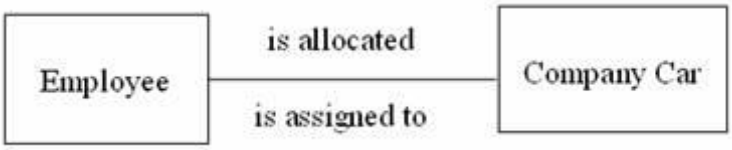
The **degree of relationship** (also known as cardinality) is the number of occurrences in one entity which are associated (or linked) to the number of occurrences in another. There are three degrees of relationship, known as:

1. one-to-one (1:1)
2. one-to-many (1:M)
3. many-to-many (M:N)

The latter one is correct, it is M:N and **not** M:M.

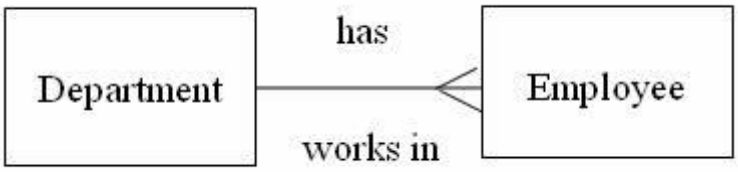
One-to-one (1:1)

This is where one occurrence of an entity relates to only one occurrence in another entity.
A one-to-one relationship rarely exists in practice, but it can. However, you may consider combining them into one entity.
For example, an employee is allocated a company car, which can only be driven by that employee. Therefore, there is a one-to-one relationship between employee and company car.



One-to-Many (1:M)

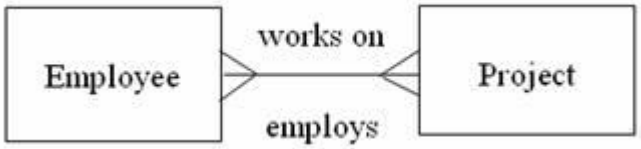
Is where one occurrence in an entity relates to many occurrences in another entity.
For example, taking the employee and department entities shown on the previous page, an employee works in one department but a department has many employees.
Therefore, there is a one-to-many relationship between department and employee.



Many-to-Many (M:N)

This is where many occurrences in an entity relate to many occurrences in another entity.
The normalisation process discussed earlier would prevent any such relationships but the definition is included here for completeness.

As with one-to-one relationships, many-to-many relationships rarely exist. Normally they occur because an entity has been missed.
For example, an employee may work on several projects at the same time and a project has a team of many employees.
Therefore, there is a many-to-many relationship between employee and project.



However, in the normalisation process this many-to-many is resolved by the entity Project Team.

Definitions:

entity something about which data is collected, stored, and maintained

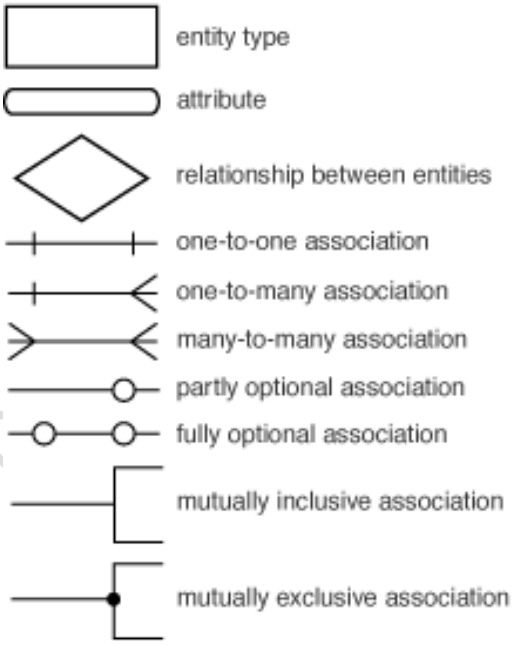
attribute a characteristic of an entity

relationship an association between entities

entity type a class of entities that have the same set of attributes

record an ordered set of attribute values that describe an instance of an entity type

Symbols:



Examples:

One A is associated with one B:

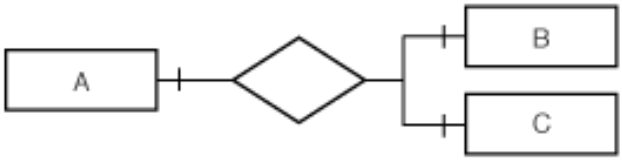
One A is associated with one or more B's:

One or more A's are associated with one or more B's:

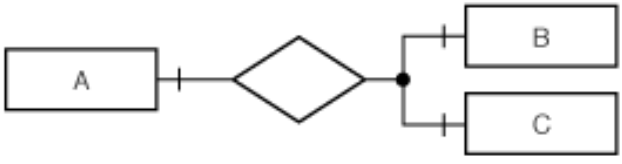
One A is associated with zero or one B:

One A is associated with zero or more B's:

One A is associated with one B and one C:



One A is associated with one B or one C (but not both):

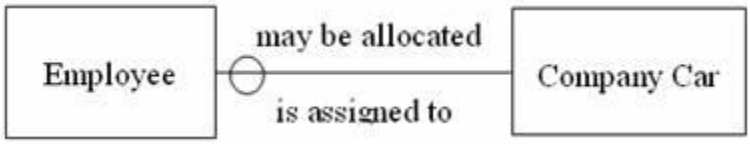


Optional Relationships

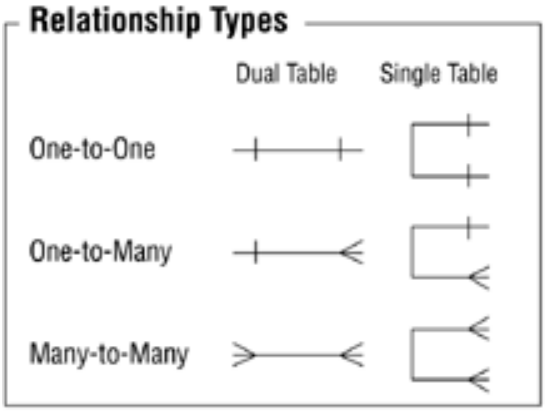
A relationship may also be optional. Either end of the relationship can include zero occurrences as an option. This is defined by the business rules of the system being implemented.
Taking the three examples above, the business rules may allow for the following.

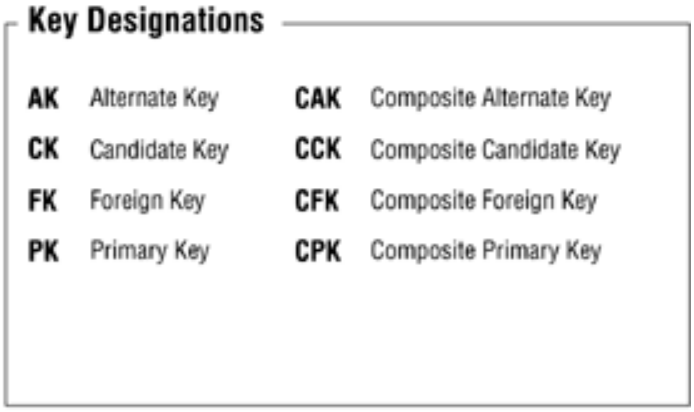
- Not all employees are allocated a company car.
- A car is defined as a pool car and not allocated to a specific employee.
- A new department is created but, as yet, there are no employees working within it.
- A new project is defined but as yet the team has not been established.
- A new employee starts within the company but, as yet, is not assigned to a project.

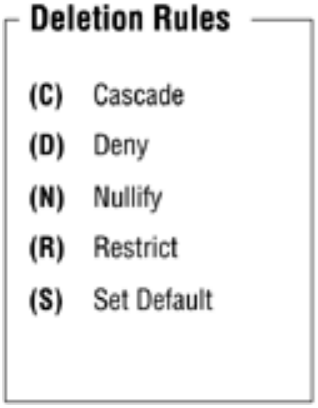
Taking the first business rule, graphically this can be shown as:

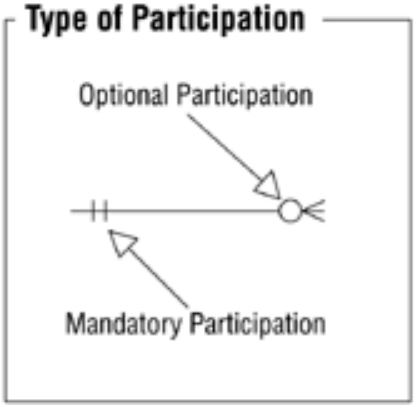


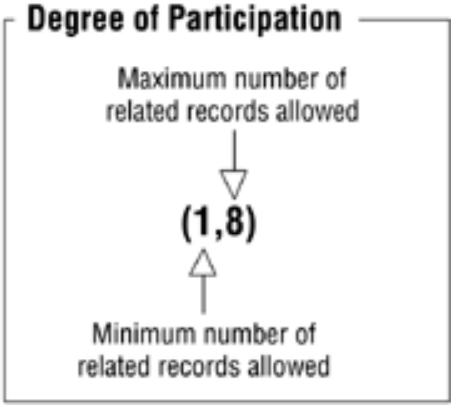
The circle (O) represents optionality.

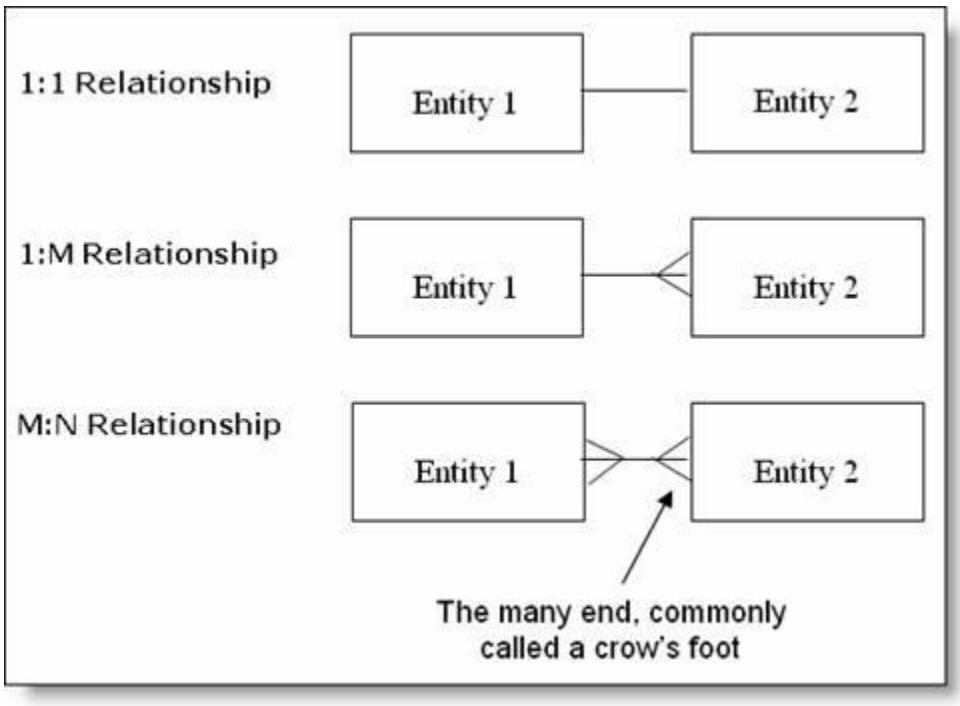






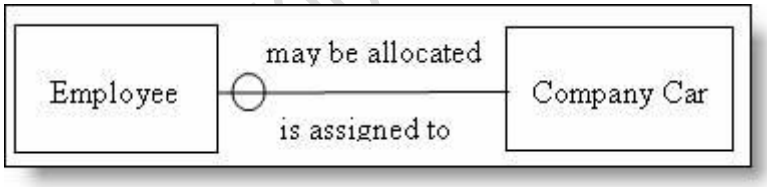




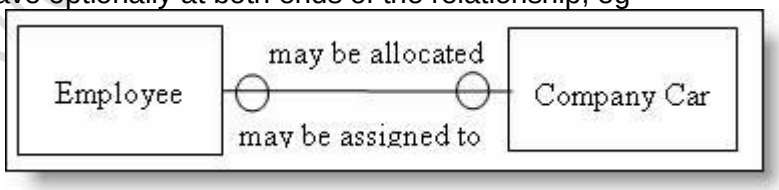


More Symbols

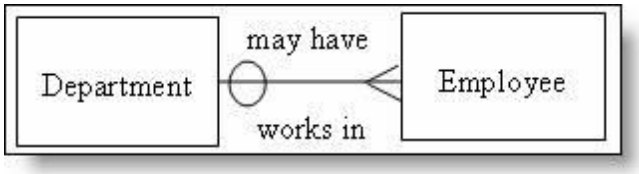
An optional relationship is denoted by a circle (O) placed at the optional end of the relationship, ie the 'may' part of the name. Using our previous example:



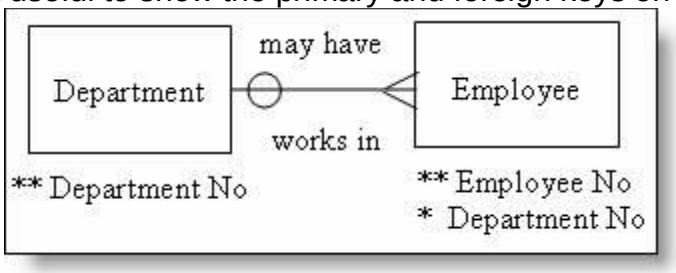
An employee may be allocated a company car.
A company car is always assigned to an employee.
It is possible to have optionally at both ends of the relationship, eg



An employee may be allocated a company car.
A company car may be assigned to an employee (eg a pool car).
Finally, the relationship is named. A relationship should be given two names, the name of the relationship between the first entity and the second and, conversely, the name of the relation between the second entity and the first, eg:



This is read clockwise and states: A department may have one or more employees. This optionality allows for a newly created department which, currently, does not have any employees.
An employee **works in** one department.
You may also find it useful to show the primary and foreign keys on the data model.



However the data model can get quite 'busy' with relationship names and keys and, consequently, difficult to read. In this case you may consider omitting naming of the keys and any obvious relationship names.

Normalisation of Database

Database Normalisation is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anamolies. It is a multi-step process that puts data into tabular form by removing duplicated data from the relation tables.
Normalization is used for mainly two purpose,

- Eliminating redundant (useless) data.
- Ensuring data dependencies make sense i.e data is logically stored.

Problem Without Normalization

Without Normalization, it becomes difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anamolies are very frequent if Database is not Normalized.

Database Normalization Examples -

Assume a video library maintains a database of movies rented out. Without any normalization, all information is stored in one table as shown below.

Full Names	Physical Address	Movies rented	Salutation	Category
Janet Jones	First Street Plot No 4	Pirates of the Caribbean, Clash of the Titans	Ms.	Action, Action
Robert Phil	3rd Street 34	Forgetting Sarah Marshal, Daddy's Little Girls	Mr.	Romance, Romance
Robert Phil	5th Avenue	Clash of the Titans	Mr.	Action

Here you see **Movies Rented column has multiple values**.

Database Normal Forms

Now let's move into 1st Normal Forms

1NF (First Normal Form) Rules

- Each table cell should contain a single value.
- Each record needs to be unique.

The above table in 1NF-

Full Names	Physical Address	Movies rented	Salutation
Janet Jones	First Street Plot No 4	Pirates of the Caribbean	Ms.
Janet Jones	First Street Plot No 4	Clash of the Titans	Ms.
Robert Phil	3rd Street 34	Forgetting Sarah Marshal	Mr.
Robert Phil	3rd Street 34	Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

Table 1: In 1NF Form

1NF Example

Let's move into second normal form 2NF

2NF (Second Normal Form) Rules

- Rule 1- Be in 1NF
- Rule 2- Single Column Primary Key

It is clear that we can't move forward to make our simple database in 2nd Normalization form unless we partition the table above.

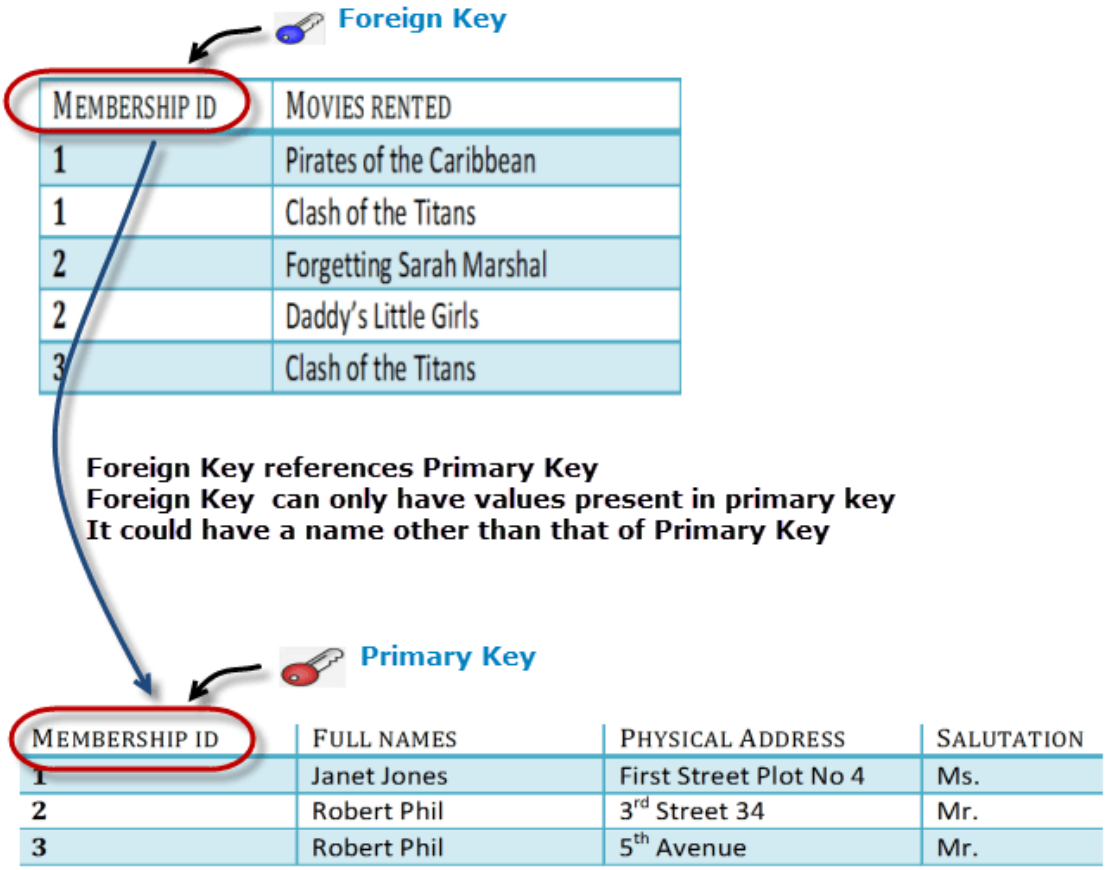
MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

Table 1

MEMBERSHIP ID	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

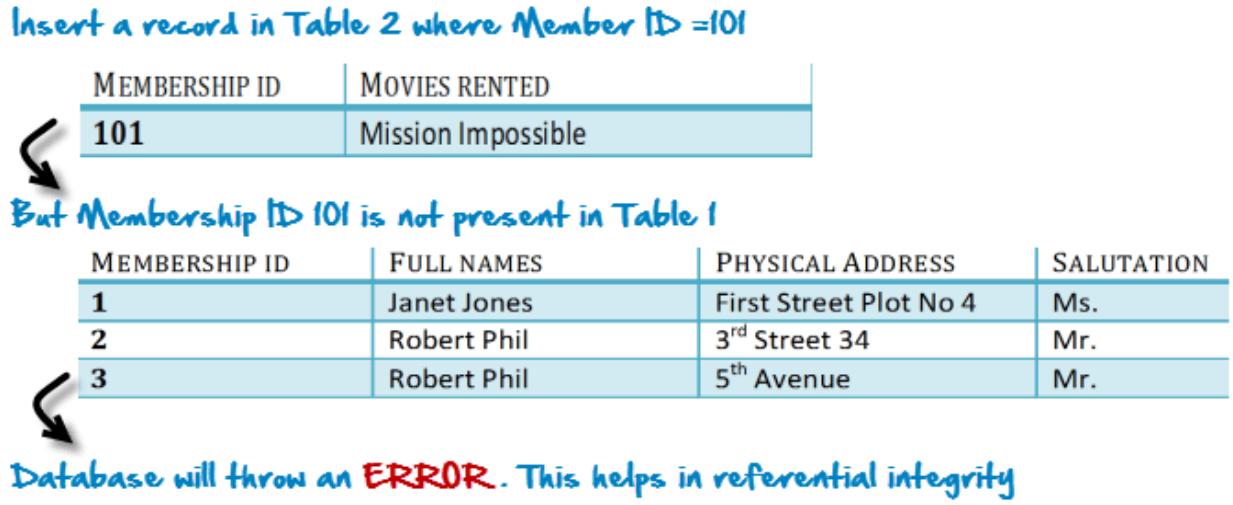
Table 2

We have divided our 1NF table into two tables viz. Table 1 and Table2. Table 1 contains member information. Table 2 contains information on movies rented.
We have introduced a new column called Membership_id which is the primary key for table 1. Records can be uniquely identified in Table 1 using membership id



Why do you need a foreign key?
Suppose an idiot inserts a record in Table B such as
You will only be able to insert values into your foreign key that exist in the unique key in the parent table.

This helps in referential integrity.



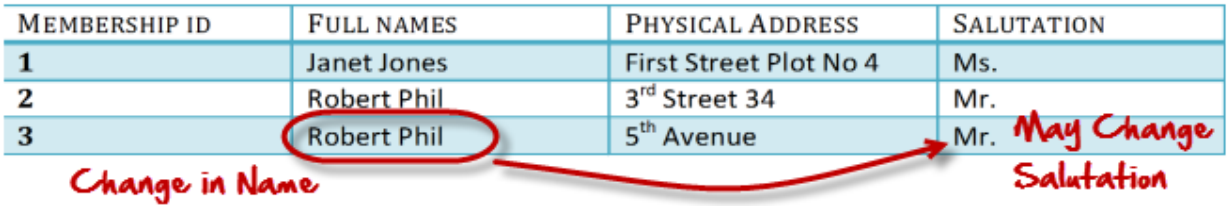
The above problem can be overcome by declaring membership id from Table2 as foreign key of membership id from Table1

Now, if somebody tries to insert a value in the membership id field that does not exist in the parent table, an error will be shown!

What are transitive functional dependencies?

A transitive functional dependency is when changing a non-key column, might cause any of the other non-key columns to change

Consider the table 1. Changing the non-key column Full Name may change Salutation.



Let's move into 3NF

3NF (Third Normal Form) Rules

- **Rule 1- Be in 2NF**
- **Rule 2- Has no transitive functional dependencies**

To move our 2NF table into 3NF, we again need to again divide our table.

3NF Example

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION ID
1	Janet Jones	First Street Plot No 4	2
2	Robert Phil	3rd Street 34	1
3	Robert Phil	5th Avenue	1

TABLE 1

MEMBERSHIP ID	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

Table 2

SALUTATION ID	SALUTATION
1	Mr.
2	Ms.
3	Mrs.
4	Dr.

Table 3

We have again divided our tables and created a new table which stores Salutations.

There are no transitive functional dependencies, and hence our table is in 3NF

In Table 3 Salutation ID is primary key, and in Table 1 Salutation ID is foreign to primary key in Table 3

Now our little example is at a level that cannot further be decomposed to attain higher forms of normalization. In fact, it is already in higher normalization forms. Separate efforts for moving into next levels of normalizing data are normally needed in complex databases. However, we will be discussing next levels of normalizations in brief in the following.

Further normalization forms are not in syllabus, but just explaining them briefly below

Boyce-Codd Normal Form (BCNF)
Even when a database is in 3rd Normal Form, still there would be anomalies resulted if it has more than one **Candidate** Key.

Sometimes is BCNF is also referred as **3.5 Normal Form.**

4NF (Fourth Normal Form) Rules
If no database table instance contains two or more, independent and multivalued data describing the relevant entity, then it is in 4th Normal Form.

5NF (Fifth Normal Form) Rules
A table is in 5th Normal Form only if it is in 4NF and it cannot be decomposed into any number of smaller tables without loss of data.

6NF (Sixth Normal Form) Proposed
6th Normal Form is not standardized, yet however, it is being discussed by database experts for some time. Hopefully, we would have a clear & standardized definition for 6th Normal Form in the near future...

That's all to Normalization!!!

The advantage of removing transtive dependency is,

- Amount of data duplication is reduced.
- Data integrity achieved.

Complex database operations

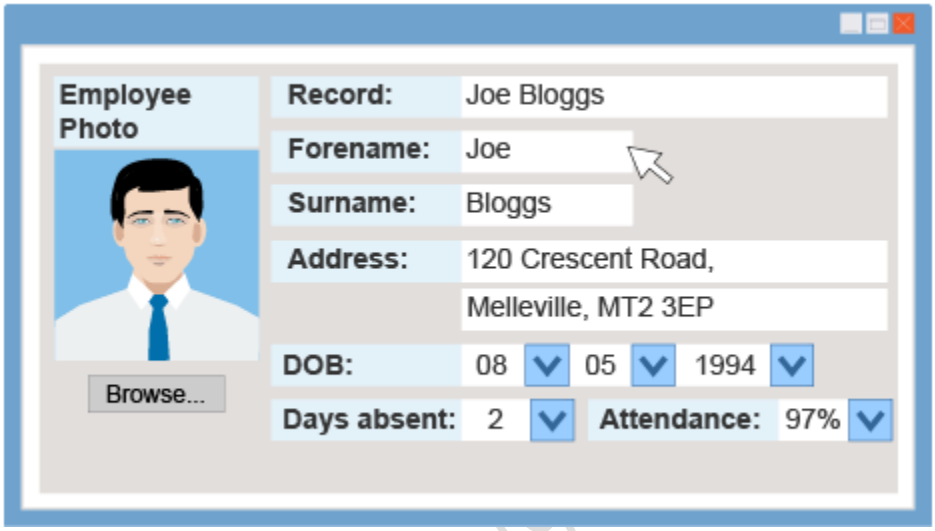
Database operations allow us to retrieve information from a database more efficiently, amend its contents or present/output the contents of that database in a variety of ways. Most commonly this is by use of:

- **forms**
- **queries**
- **reports**

Forms

Forms are created with the purpose of generating a user friendly interface for data entry. Forms may use programmable buttons that perform various commands for viewing, entering, and editing data in the tables.

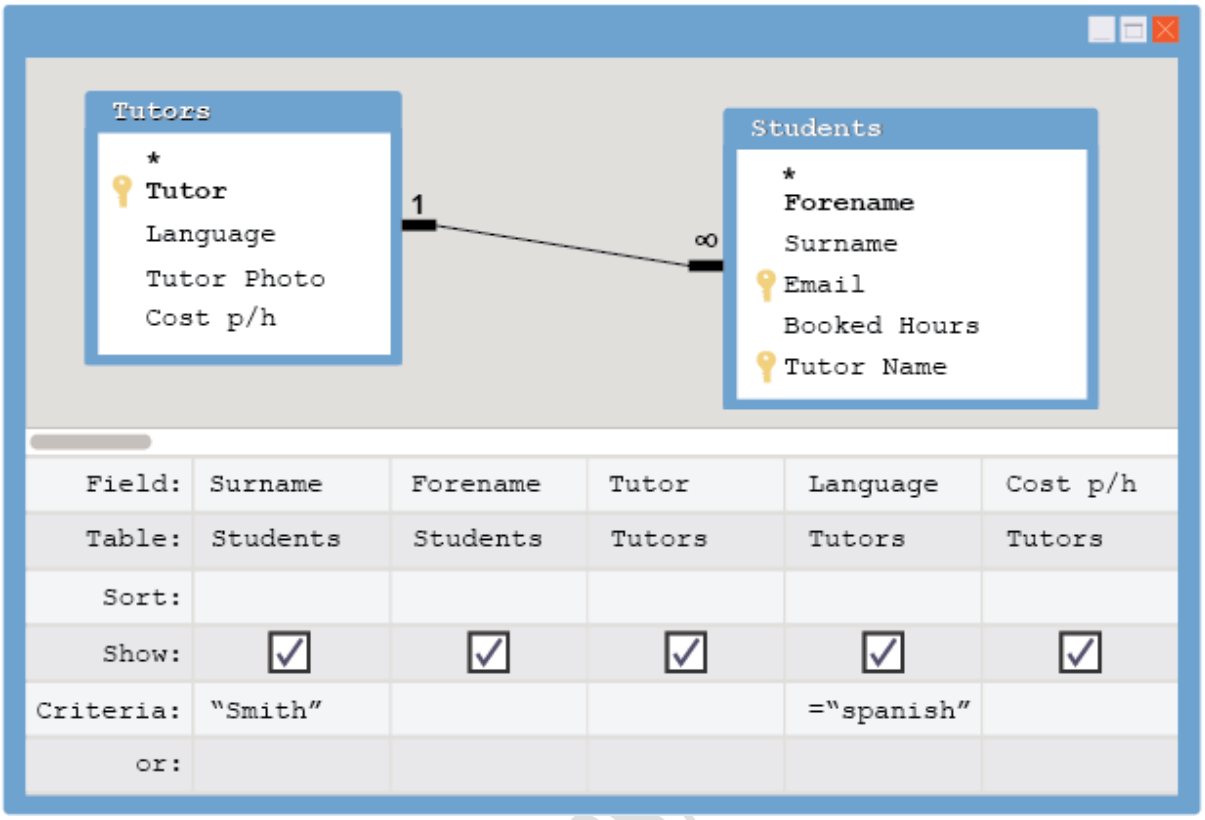
Forms also allow the database creator to control how other users interact with the data in the database. For example, you can create a form that shows only certain fields and allows only certain operations to be performed. This helps protect data and to ensure that the data is entered properly.



Queries

Queries allow users to search and sort data held in a database. There are two different ways to create queries. The first is to use the built in query generator that comes with most database software.

The example below shows a complex query searching two tables to find any student named Smith who is taught Spanish. This query has been created using an in built query generator.



More advanced users may want to create some code to perform queries. SQL is an example of a language used to create queries within database applications. The same query in SQL could be written as:

```
SELECT surname, forename, tutor, language, cost p/h
FROM Students, Tutors;
WHERE Students.Surname ='Smith' AND Tutors.Language = 'spanish';
```

Learning SQL is not required but having an understanding of how the database system generates queries can be used to enhance your knowledge and ability to use computational thinking.

Complex database operations

Calculations

Queries also allow users to perform calculations when using in built query generators or SQL. Many database packages include functions to let users quickly use these calculations.

Common calculations include:

SUM	Will add within a column/field
AVERAGE	Will find the average value in a column/field
COUNT	Counts the number of occurrences in a column/field
MAXIMUM	Finds the maximum value in a column/field
MINIMUM	Find the minimum value in a column/field

Summary fields are also a feature of some database packages that allow the result of calculations to appear on reports.

Reports

Reports allow data to be summarised and output from a database. Reports are often constructed based on the results of queries. Reports can be set up to display only specific information rather than the full result of a query or all of the data held in tables.

Each report can be formatted to present the information in the most readable way possible. A report can be run at any time, and will always reflect the current data in the database. Reports are usually formatted to be printed out, but they can also be viewed on the screen, exported to another program, or sent as an attachment to an e-mail message.

It is important to be able to look at a report and determine the fields used in its creation. These fields are often used within a query, with the results of the query used to generate a report.

References:

http://www.sqa.org.uk/e-learning/SoftDevRDS02CD/page_55.htm#Act11

https://www.ictlounge.com/html/types_of_databases.htm

https://www.guru99.com/database-normalization.html