**Joins**

To bring together data from more than one relation we have to 'join' the relations using the primary & foreign keys in the relations.

For example:

* List the names and the batting scores of all the players who have a batting score of 100 or more in any match.

To identify who has a batting score of 100 or more in any match we will need a condition in the WHERE clause.

WHERE batting\_score => 100

SQL performs a Cartesian product on all the tables specified in the FROM clause, in this case:

FROM mccPlayer, mccMatch\_performance

To restrict the cartesian product to a join by correctly matching up the batting scores with the right player we need to specify a join condition that matches the primary key and foreign key values that form the relationship between the two tables.

mccMatch\_performance.registration\_number = mccPlayer.registration\_number.

As it is a condition it goes in the WHERE clause, and as both this condition and the batting score condition have to be true we need an AND operator. This leaves the final query looking like:

SELECT name, batting\_score

FROM mccPlayer P, mccMatch\_performance MP

WHERE P.registration\_number = MP.registration\_number

AND batting\_score >= 100;

Remember that because registration\_number appears in both tables we have to be able to distinguish between them by qualifying the attribute name with the table name

Some 2 table join queries for you to try:

1. List the names of the teams, the ground names and street names that MCC have played against at grounds located in the town Tynemouth.
2. What is the name of the player who had a batting score of zero on the 5th July 2013.

Queries on multiple relations use exactly the same principles as a 2 relation query. For example:

* List the names of the grounds that 'Shane Watson' has played at.

SELECT ground\_name

FROM mccPlayer P, mccMatch\_performance MP, mccMatches M

WHERE P.registration\_number = MP.registration\_number

AND MP.match\_date = M.match\_date

AND name = 'Shane Watson';

Follow the primary/foreign key relationships to see how this works. You can think of it as 2 tables being joined together and the result being joined to the third. This procedure can be extended to join as many tables as you want. You always need one fewer join conditions than there are tables to be joined, eg 7 tables to be joined need 6 join conditions.

Notice that I have given each table an alias, eg P for Player, to make it easier to qualify the names of the join attributes.

* List the names of players who live in Durham and who played in the match on the 28th May 2013.

SELECT name

FROM mccPlayer P, mccMatch\_performance MP

WHERE P.registration\_number = MP.registration\_number

AND p\_town = 'Durham'

AND match\_date = 130705;

Some queries for you to try:

1. List the names of players who achieved a batting score of more than 50 in a match at the 'Gasworks' ground.
2. List the grounds (name & town) where the player 'Shane Watson' has achieved a batting score of 150 or more.
3. List the names of players and the games they played (date & opposing team) where their batting score was less than their age.

Joining a table to itself

* List the names of players who live in the same town as 'Shane Watson'.

This query presents a problem a common problem for relational databases. We can easily find out which town Shane Watson lives in;

SELECT p\_town FROM mccPlayer WHERE name = 'Shane Watson';

What we want to do is to take the output, ie the name of the town that Shane Watson lives in, and use this to find rows in the Player relation with the same value for p\_town. We can't use the original Player table & output from the Player table unless we can distinguish between them.

The simplest way of doing this is to use an alias for the Player table, as in:

SELECT Player.name

FROM mccPlayer P, mccPlayer

WHERE P.name = 'Shane Watson'

AND P.p\_town = mccPlayer.p\_town;

You can think of the alias as creating a virtual copy of the Player table called P. The condition P.name = ‘Shane Watson’ restricts the virtual table P to those rows where the condition is true, and the join condition joins the result to the original table 'Player'.

Note that the attribute we are using to perform the join, p\_town, is not a primary key or a foreign key. This is fine. You can use any attribute to perform a join, eg;

SELECT \* FROM mccMatch, Player WHERE Match.ground\_name > Player.p\_street;, but you have to be careful that you don't get rubbish as output, as you would in this case.

1. List the names of grounds that are located in the same town as the 'Leaside' ground.
2. List the names of players who live in the same town as 'Shane Watson', making sure that Shane’s name is not in the output.

Unions

Set operations can only be performed on union compatible relations, ie the relations have the same number of attributes and the corresponding attributes in each relation have the same domain.

They are however very useful for queries such as:

* List the towns where either a player lives or a ground is located.

The towns where players live;

SELECT p\_town FROM mccPlayer;

The towns where ground are located;

SELECT g\_town FROM mccGround;

And a list of both;

SELECT p\_town FROM mccPlayer

UNION

SELECT g\_town FROM mccGround;

List the towns where both a player lives and a ground is located.

SELECT p\_town FROM mccPlayer

INTERSECT

SELECT g\_town FROM mccGround;

The difference operation is not supported in SQLiteStudio but it is in other SQL implementations. Here is how you would use it.

* List the towns where players live but there is no ground located there.

SELECT p\_town FROM mccPlayer

MINUS

SELECT g\_town FROM mccGround;

This last operation, the difference, can useful for queries that involve negation, however SQL provides a much more powerful feature for this kind of query, sub-queries.