DBS211

Week 4 - Multiple-Table Queries (Joins) and Views

Table of Contents

Additional Reading Materials

- W3Schools - Joins (multiple sections - Joins through Self Join)
- W3Schools - Views

Welcome to Week 4

Welcome to Week 4! This week we dive into creating queries where the data is located in multiple tables. It is actually rare in the real-world to need any queries where data is entirely contained within a single table. The whole concept of relational databases and data modelling results in data being split into many different tables, it is only natural that the data would need to be put back together again for the public, non-technical user. By the end of this week you will be able to:

- identify and understand ANSI-89 Joins
- create queries that use ANSI-92 Joins and be able to explain how they work
- acknowledge the difference between INNER and OUTER joins and be able to accurately use LEFT, RIGHT and FULL outer joins.
- Explain what a view is, how it is saved, and how it can be used to make working in SQL more efficient
- create and save views and use them in the right context in more complexe SQL SELECT statements.

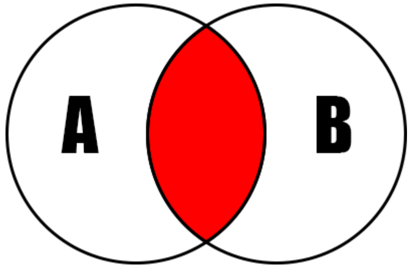
Sample Code

The table creation code for the examples used this week are at the bottom of this page. You should be able to copy and paste that code into SQL Developer, highlight it all and then execute it. This will create the tables and insert the data for the examples below so you can try them out for yourself.

Get Sample Code

Sets & Venn Diagrams

Going back to high school mathematics, many of you may remember a concept you learned called Venn Diagrams. At the time, you might have thought that you would never see them again, but this week we will revisit them from the context of database joins.



In this diagram above we could have two tables with data in them, one table named A, the other named B. The red area would indicate the area where data between the two tables is related. In set operators this was known as A ∩ B (A intersect B). We will use diagrams like this to visualize how the various joins work.

ANSI-89 Joins

ANSI-89 joins are still commonly used today by db developers whom learned it that way and have not changed because they do not see a need to. In the real-world, there are millions of databases in use that have been created over decades of time. This means that there are a significant amount of existing databases that exist that use the older model and they need to be maintained. Therefore, we are covering the topic just enough such that you recognize it when you see it and can work on those databases. However, all work in this course will be done using the current ANSI-92 method unless otherwise stated.

ANSI-89 Joins have been called different things over the years, but the 2 most popular names are "***Simple Joins***" or "***Natural Joins***". The basic syntax is:

```
SELECT <fieldlist>
FROM <table1>, <table2>
WHERE <table1.keyfield> = <table2.keyfield>;
```

As see above, the two table names are both included in the FROM clause, but it is very important to note that a query has no knowledge of any existing relationships, contraints etc. Anything that is required in the output, must be defined in the query. If there is a relationship between the two tables, then it must be described in the statement. The WHERE clause is where this is done by making an equality on the primary key of the parent table and the foreign key of the child table. Meaning, that data will only be retrieved when rows in table 2 have an equivalent value in the key field of table 1.

PLAYERS			
<u>PlayerID</u>	**firstname**	**lastname**	**teamid**
1	John	Smith	10
2	Bob	Marley	10
3	Steven	King	11
4	Jim	Parsons	NULL

TEAMS			
<u>teamID</u>	**teamName**	**shirtColor**	**homeField**
10	Hornets	Yellow	Toronto
11	Falcons	Brown	Barrie
12	Bloopers	Red	Kitchener

In the tables above, you can see that there is a relationship between the two tables on the teamID field. John and Bob play on the Hornets and Steven plays for the Falcons. We also see the Jim is not yet on a team and the Bloopers do not have any players.

Let's write a naturla join to obtain the players names and the team name for the teams they play on.

```
SELECT playerID, firstname || ' ' || lastname PlayerName, teamName
    FROM players, teams;
```

This would output:

PLAYERID	PLAYERNAME	TEAMNAME
1	John Smith	Hornets
2	Bob Marley	Hornets
3	Steven King	Hornets
4	Jim Parsons	Hornets
1	John Smith	Falcons
2	Bob Marley	Falcons
3	Steven King	Falcons
4	Jim Parsons	Falcons
1	John Smith	Bloopers
2	Bob Marley	Bloopers
3	Steven King	Bloopers
4	Jim Parsons	Bloopers

As you can see this output is WRONG, it says that each of the 4 players play one all 3 teams. These results are called the CROSS join results. This is done because the query has no prior knowledge of the relationship and unless this link is described in the query, it gets all possible combinations that could be true. Therefore, it is essential that a WHERE clause be added to describe the equality.

```
SELECT playerID, firstname || ' ' || lastname PlayerName, teamName
    FROM players, teams
    WHERE players.teamID = teams.teamID;
```

Now the output is:

PLAYERID	PLAYERNAME	TEAMNAME
1	John Smith	Hornets
2	Bob Marley	Hornets
3	Steven King	Falcons

From this new query we now have the correct results. Note that Jim is not in the output, as he is not on a team, and the Bloopers team is missing because they do not have any players.

The natural join syntax gets messy when you now need to add conditions or filters in the `WHERE` clause as now `AND` is required. If you have multiple filters, you then have multiple ANDs and ORs which make it more confusing and mistakes are easily made. Additionally, if we consider Order of Execution, remember that the FROM clause executes before the `WHERE` clause. This results in the CROSS join results being obtained first and then eliminating the results that don't match. This could result in memory issues in very large queries and slower processing time compared to the new ANSI-92 standard.

ANSI-92 Joins

In 1992, a new standard format was introduced that would become the standard format across all DBMS versions of SQL. This introduces the JOIN keyword along with ON and USING. We are going to cover several different ways to retrieve data from the tables using inner and outer join types.

The basic syntax of ANSI-92 Joins is:

```
SELECT <fieldlist from fields from either table>
    FROM <table1> <Join Type> JOIN <table2> ON <table1.keyfield> = <table2.keyfield>;
```

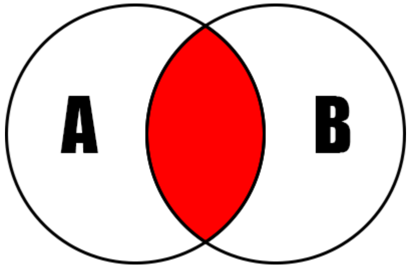
OR

```
SELECT <fieldlist from fields from either table>
    FROM <table1> <Join Type> JOIN <table2> USING (keyfield>);
```

When the JOIN is specified with no prefix qualifier, the default join type is `INNER JOIN`. The first version, using the `ON` keyword can always be used and it is best to know this one as you can create almost any join query using this syntax. The USING syntax has a few limitations, but also has some benefits.

- USING can only be used if the keyfield names in both tables is exactly the same (teamID in our example)
- table aliases can not be used on the tables involved in the USING keyword
- when Using is used, the field indicated is combined into a single field, meaning it can no longer be ambiguous.

Inner Joins



Inner Joins return the results when the equality exactly matches. This may seem obvious, because = means equals, but you will see with outer joins that we can obtain results that don't satisfy the equality. Inner Joins will only return rows where the key value from table 1 exists in table 2 AND the key values in table 2 exist in table 1.

For example:

```
SELECT playerID, firstname || ' ' || lastname PlayerName, teamName
    FROM players INNER JOIN teams ON players.teamID = teams.teamID;
```

This statement has the exact same output as the last query using the Natural join with the `WHERE` equality.

Outer Joins

Outer Joins extend the Inner join by allowing us to retrieve results from one or both tables, even if there is no matching record in the other table. This is a very powerful feature that allows us to get information that there are no matches for instance.

There are 3 types of outer joins available to us in ANSI-92, `LEFT`, `RIGHT` and `FULL`. The concept of Left and Right may be confusing at first, but it is actually really easy once you get it.

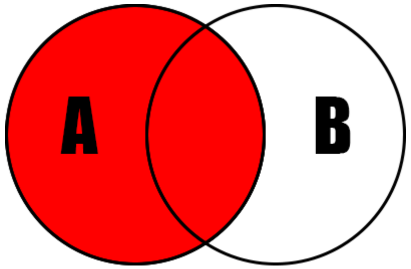
```
table1 JOIN table2
```

Using the above join, table1 is considered the LEFT table because it is before the `JOIN` keyword. This then means that table2 would be the `RIGHT` table because it is located after the JOIN keyword. The reason we do not say left or right of the join is because it can be written in different ways. For example:

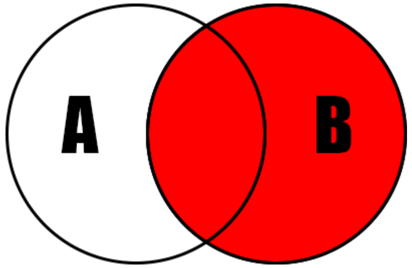
```
table1
    JOIN
table2
```

Even though this appears like table2 is left of the JOIN, it is still the right table because it is after. For readability purposes, it is helpful in the tables and `JOIN` keyword are written on one line.

Left or Right Joins



A `LEFT OUTER JOIN` would return ALL the rows from the left table regardless if a matching record exists in the right table, but will still only return rows from the right table if there is an exact matching record in left table.



A `RIGHT OUTER JOIN` would return ALL the rows from the right table regardless if a matching record exists in the left table, but will still only return rows from the left table if there is an exact matching record in right table.

Example

```
SELECT playerID, firstname || ' ' || lastname PlayerName, teamName
    FROM players LEFT OUTER JOIN teams ON players.teamID = teams.teamID;

    -- OR

SELECT playerID, firstname || ' ' || lastname PlayerName, teamName
    FROM teams RIGHT OUTER JOIN players ON players.teamID = teams.teamID;
```

These two statements above will return the exact same results, as we swapped the tables and changed left to right.

PLAYERID	PLAYERNAME	TEAMNAME
1	John Smith	Hornets
2	Bob Marley	Hornets
3	Steven King	Falcons
4	Jim Parsons	(null)

In this result, you see that all the matching records are present, but in addition, player 4 is included even though there is no team that is linked to him. Notice that the value of the teamname can not be retrieved, because there is no team that matches, and therefore the value returned for teamname is `NULL`.

Example 2

```
SELECT playerID, firstname || ' ' || lastname PlayerName, teamName
    FROM players RIGHT OUTER JOIN teams ON players.teamID = teams.teamID;

    -- OR

SELECT playerID, firstname || ' ' || lastname PlayerName, teamName
    FROM teams LEFT OUTER JOIN players ON players.teamID = teams.teamID;
```

returns...

PLAYERID	PLAYERNAME	TEAMNAME
1	John Smith	Hornets
2	Bob Marley	Hornets
3	Steven King	Falcons
(null)		Bloopers

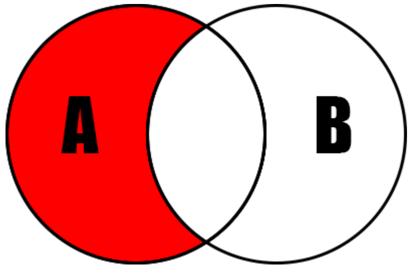
In this result, you see that all the matching records are present, but in addition, the Bloopers team is included even though there is no player linked to that team. Notice that the value of the playerid can not be retrieved, because there is no player that matches, and therefore the value returned for playerid, firstname or lastname is `NULL`.

It is interesting to note, that although firstname and lastname would be NULL in this case, the PlayerName column returns a blank and NOT a NULL. This is because playername is a calculated field using a string function, therefore NULLs are eliminated and a zero length string is returned.

Inverse Outer Joins

Let us say we are the league administration and it is important that we know which teams do not yet have players or players that are not yet on a team. Knowing that this information is included in the outer joins we just covered, we can use these results and by adding an additional filter can eliminate the matching records.

Example: Create a query that lists the teams that do not yet have any players linked to them:

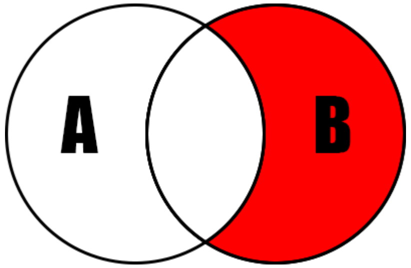


```
SELECT teamID, teamName
FROM teams LEFT OUTER JOIN players USING (teamID)
WHERE playerID IS NULL;
```

by using the LEFT join, all rows from the teams table are retrieved, regardless if a matching player exists, and only those players returned that are on a team are returned. Knowing that the extra team rows returned can not retrieve player information and therefore inserts NULL values, we can add an additional filter of WHERE playerID IS NULL, eliminating the records that do match.

TEAMID	TEAMNAME
12	Bloopers

Another Example: Create a query that lists all players not currently on a team!

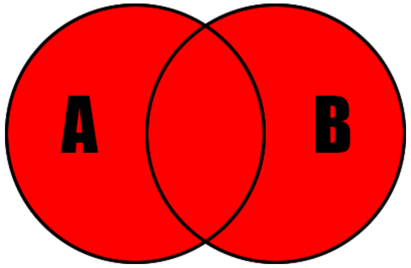


```
SELECT playerID, firstname, lastname
FROM teams RIGHT OUTER JOIN players USING (teamID)
WHERE teamID IS NULL;
```

The right join ensures that all players are returned and the use of the teamid being null gives us only those players whom are not on a team.

PLAYERID	FIRSTNAME	LASTNAME
4	Jim	Parsons

Full Outer Joins



A third type of outer join combines the results from both left and right joins together. A full join will return:

- All records that have matching records in the other table
- All records from the LEFT table regardless if there is a matching record in the RIGHT table

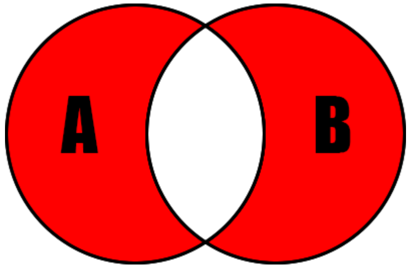
- All records from the RIGHT table regardless if there is amatching record in the LEFT table
- eliminates duplicate records such that matching records are only returned once.

```
SELECT playerID, firstname, lastname, teamName
FROM teams FULL OUTER JOIN players USING (teamID)
WHERE teamID IS NULL;
```

PLAYERID	FIRSTNAME	LASTNAME	TEAMNAME
1	John	Smith	Hornets
2	Bob	Marley	Hornets
3	Steven	King	Falcons
4	Jim	Parsons	(null)
(null)	(null)	(null)	Bloopers

Inverse Full Joins

Lastly, we can view all players not on a team, and all teams that do not have a player in the same resultant set and not show any matched records.



```
SELECT playerID, firstname, lastname, teamName
FROM teams FULL OUTER JOIN players USING (teamID)
WHERE teamID IS NULL;
```

PLAYERID	FIRSTNAME	LASTNAME	TEAMNAME
4	Jim	Parsons	(null)
(null)	(null)	(null)	Bloopers

Joins With More Than 2 Tables.

Now we need to introduce a 3rd table to the mix.

PLAYERS			
PlayerID	**firstname**	**lastname**	**teamid**
1	John	Smith	10
2	Bob	Marley	10
3	Steven	King	11
4	Jim	Parsons	NULL

TEAMS			
teamID	**teamName**	**shirtColor**	**homeField**
10	Hornets	Yellow	Toronto

teamID	teamName	shirtColor	homeField
11	Falcons	Brown	Barrie
12	Bloopers	Red	Kitchener

FIELDS		
FieldName	**Address**	**Manager**
Toronto	22 Queen St. E.	Doug Ford
Barrie	380 Bayview Ave	Richard Alexander
Kitchener	220 Weber St. S	Marco Albana
Waterloo	576 King St. N	Paul Caruso

So now we need to ask the question, create a query that outputs the ***player names*** and the ***address*** that they need to go to for practice. We now requried fields from 2 different tables that do not have a way to join them, there are no matching fields. However, we know players play on teams and teams play at field locations, so if we use all 3 tables, then we can get the answer.

```
SELECT firstname, lastname, address
FROM players JOIN teams USING (teamID)
    JOIN fields ON teams.homefield = fields.fieldname;
```

In this sample, we first joined players and teams, and then joined the result to fields. Note that the first join creates a new temporary table, that includes all fields from both players and teams tables, and then we join the third table, fields, to the new temporary table. *It is NOT table 1 joins to table 2 and table 2 joins to table 3.* The results are:

FIRSTNAME	LASTNAME	ADDRESS

FIRSTNAME	LASTNAME	ADDRESS
John	Smith	22 Queen St. E.
Bob	Marley	22 Queen St. E.
Steven	King	380 Bayview Ave

Let us now introduce outer joins.

Example:: create a query that outputs the ***player names, team name*** and the ***address*** that they need to go to for practice, but include all fields regardless of teams associated with them.

```
SELECT firstname, lastname, teamname, address
FROM players JOIN teams USING (teamID)
    RIGHT OUTER JOIN fields ON teams.homefield = fields.fieldname;
```

In order to return all the fields, we need to do a right join on the join that includes fields. Because we only wanted matching teams and players, the first join could simply be an inner join.

FIRSTNAME	LASTNAME	TEAMNAME	ADDRESS
John	Smith	Hornets	22 Queen St. E.
Bob	Marley	Hornets	22 Queen St. E.
Steven	King	Falcons	380 Bayview Ave
(null)	(null)	(null)	220 Weber St. S
(null)	(null)	(null)	576 King St. N

> It is important to note that if the order of the tables in the FROM clause was different, for example fields was the firsts table, then the join types would have to chage to match the order the tables were entered.

Example: create a query that outputs the ***player names, team name*** and the ***address*** that they need to go to for practice, but include **all players** regardless of teams associated with them.

```
SELECT firstname, lastname, address
FROM players LEFT OUTER JOIN teams USING (teamID)
    LEFT OUTER JOIN fields ON teams.homefield = fields.fieldname;
```

In this case, the table that we needed all records returned from was the first table. Therefore, the first join needed to be a left join to carry those records forward. Now the first join created a temp table in memory that contains all fields from both tables. Therefore the player information now resides in the temp table which is joined to the 3rd, fields, table. Because we still need to keep allplayers, a second left join is required to maintain that data. An inner or right join would have eliminated players without teams.

FIRSTNAME	LASTNAME	ADDRESS
Bob	Marley	22 Queen St. E.
John	Smith	22 Queen St. E.
Steven	King	380 Bayview Ave
Jim	Parsons	(null)

Ambiguous Fields and Aliases

When including multiple tables in a query, having multiple fields with the same name is common. These are referred to as ambiguous fields. There must be a way to identify which field is the one being referred to. The simplest way to do this is by using table aliases.

```
SELECT firstname, lastname, teamID, teamname
FROM players INNER JOIN teams ON players.teamID = teams.teamid;
```

results in the following error:

```
ORA-00918: column ambiguously defined
00918. 00000 -  "column ambiguously defined"
```

because the teamid field is located in both the players and teams tables and the SELECT clause asks for the teamid without specifying which one is requested. This brings up 2 questions:

1. How do we identify the one needed, and
2. how do we know which one we need.

To answer the first question, we use the table.field notation, and is easier typed using abbreviated table aliases.

```
SELECT firstname, lastname, teams.teamID, teamname
FROM players INNER JOIN teams ON players.teamID = teams.teamid;

    -- OR using aliases

SELECT firstname, lastname, t.teamID, teamname
FROM players p INNER JOIN teams t ON p.teamID = t.teamid;
```

both methods clearly indicate that we want the values for teamid from the teams table.

Which one is the right one?

The answer to this question is the ever popular "It Depends" answer. It depends on the type of join being used and the desired results. Let's quickly look at a few examples based on Join Type:

For **INNER joins**, the only results that can be displayed are where they related field matches in both tables. Be definition the two values will be identical and therefore it does not matter which field is used for inner joins.

When performing Left, Right and Full outer joins, it is possible that one version of the field will have values in it and the other one could be a NULL value. The choice of which field then depends on the question: where we wanted to show all records from the left table, the field from the left table would likely be the right result to show. And if a right join was used, it is likely the field from the right table should be used.

```
SELECT playerid, firstname, lastname, p.teamid pteamid, t.teamid tteamid, teamname
FROM players p RIGHT JOIN teams t ON p.teamid = t.teamid;
```

PLAYERID	FIRSTNAME	LASTNAME	PTEAMID	TTEAMID	TEAMNAME
1	John	Smith	10	10	Hornets
2	Bob	Marley	10	10	Hornets
3	Steven	King	11	11	Falcons
(null)	(null)	(null)	(null)	12	Bloopers

In the above query and results, by showing both teamid's, it is clear which one you would choose for different output requirements. If you needed the teamID for the team without players, you would have to choose t.teamid.

Inverse Joins and Ambiguous Fields

This question becomes more complicated when we are using joins to find NOT answers. Let us return to the question: List the teams that do NOT have players.

```
SELECT t.teamid, teamname
FROM teams t LEFT JOIN players p ON p.teamid = t.teamid
WHERE p.teamid IS NULL;
```

In this example, we want to output the actual teamid, which is only available from the teams table. But, when we are filtering the results to eliminate teams with players, we must use p.teamid as it will contain the NULL values used for comparison.

Views

What is a view? A view is an object stored in the database that can be a named SELECT statement. You are able to create a query and save it as an object in the database. When a view is executed, it produces results in real-time (i.e. at the time of execution).

Views vs Tables

Tables are database objects that have structure, data rows and columns and most importantly stores data. Views DO NOT store data and are nothing more than an SQL statement stored as a string object. I repeat, a view is nothing more than a string, that happens to be a SELECT statement most of the time. When a table is requested, it simply retrieves the current data in the table. When a view is requested, it executes the premade SQL statement and results in a real-time table of results. These results can be the final results, or can be used as the source in further SQL statements.

Why Views?

Views are a very powerful tool in a database schema. One way to think of a view is as an object orientated object that enhances the usability and flexibility of data retrieval.

- The ability to create complex SELECT statements for recall at any time is huge time saving tool, eliminating the need to recreate complex statements multiple times. (for example, creating a query to answer the question, how much money did we make yesterday is likely asked every day. Creating a view that can be run each day makes daily routines extremely easy)
- Views can be used to build base queries: to put tables back together through complex joins and then use the view as the source of a new SELECT statement to perform only the filtering and sorting.
- Views are NOT tables, but can be treated as if they were tables in SQL queries.

How?

Views are created by the DDL statement CREATE:

```
CREATE VIEW <viewname> AS
    <some SELECT statement>;
```

to view the results of the View's SELECT statement, use a SELECT statement with the view as the data source:

```
SELECT * FROM <viewname>
```

Examples:

```
CREATE VIEW vwPlayersOnTeams AS
    SELECT playerid, firstname, lastname, p.teamid pteamid, t.teamid tteamid, teamname
    FROM players p FULL JOIN teams t ON p.teamid = t.teamid;
```

Creates a base view that can be used further to answer many of the questions we have seen in examples above:

```
SELECT * FROM vwPlayersOnTeams;
```

PLAYERID	FIRSTNAME	LASTNAME	PTEAMID	TTEAMID	TEAMNAME
1	John	Smith	10	10	Hornets
2	Bob	Marley	10	10	Hornets
3	Steven	King	11	11	Falcons
4	Jim	Parsons	(null)	(null)	(null)
(null)	(null)	(null)	(null)	12	Bloopers

Now let's answer some questions:

Question: List all players on teams and what the team name is!

```
SELECT * FROM vwPlayersOnTeams
       WHERE playerID IS NOT NULL AND teamname IS NOT NULL;
```

Question: List all players NOT on teams!

```
SELECT * FROM vwPlayersOnTeams
       WHERE teamname IS NULL;
```

Question: List all teams the DO NOT have players!

```
SELECT * FROM vwPlayersOnTeams
       WHERE playerID IS NULL;
```

As you can see, these queries are much easier than the ones we created above in the JOINS section. Although, it is important to note that the View still contains a join. In fact, the viewname can be replaced by the actual SELECT statement in the view and it would product the same results.

```
SELECT * FROM vwPlayersOnTeams
       WHERE playerID IS NULL;

            -- is the same as

SELECT * FROM
    (SELECT playerid, firstname, lastname, p.teamid pteamid, t.teamid tteamid, teamname
    FROM players p FULL JOIN teams t ON p.teamid = t.teamid)
WHERE playerID IS NULL;
```

Using Views in Joins

As a view can be treated as a table in the context of SQL, it is therefore possible to use a join between a table and a view.

```
CREATE VIEW vwPlayerTeams AS
        SELECT *
        FROM players JOIN teams USING (teamID);

SELECT firstname, lastname, teamname, Address, Manager
FROM vwPlayerTeams v JOIN fields f ON v.homefield = f.fieldname;
```

FIRSTNAME	LASTNAME	TEAMNAME	ADDRESS	MANAGER
John	Smith	Hornets	22 Queen St. E.	Doug Ford
Bob	Marley	Hornets	22 Queen St. E.	Doug Ford
Steven	King	Falcons	380 Bayview Ave	Richard Alexander

Modifying a View

It is often the case during development that a view needs to be changed. Although you could DROP the view and the recreate it, there is an alternative using the CREATE OR REPLACE statement. The general syntax is:

```
CREATE OR REPLACE VIEW viewname AS
    SELECT .....
```

Summary

This was a pretty big week, and probably one of the most important weeks. Joins are a fundamental part of relational databases and SQL. Almost every query written in the real worl will involve at least one join, therefore they are an essential part of SQL and computer programming. Views are used every day for adding an eliment of object oriented programming to SQL and database access.

In this week you learned to:

- identify and understand ANSI-89 Joins
- create queries that use ANSI-92 Joins and be able to explain how they work
- acknowledge the difference between INNER and OUTER joins and be able to accurately use LEFT, RIGHT and FULL outer joins.
- Explain what a view is, how it is saved, and how it can be used to make working in SQL more efficient
- create and save views and use them in the right context in more complexe SQL SELECT statements.

Practice Exercises

- ANSI-89 Joins
- Inner and Outer Joins
- Joins With More Than 2 Tables
- Views

Sample Database CREATION statements

If you want to practice the statements in this weeks lesson, then run the following script first, which will create the tables and insert the data as described above.

```
-- Sample Tables for DBS211.ca website samples
-- ---------------------------------------------
```

```
CREATE TABLE teams (
    teamID INT PRIMARY KEY,
    teamName varchar(15),
    shirtColor varchar(10),
    homeField varchar(15)
);

INSERT ALL
    INTO teams VALUES (10, 'Hornets', 'Yellow', 'Toronto')
    INTO teams VALUES (11, 'Falcons', 'Brown', 'Barrie')
    INTO teams VALUES (12, 'Bloopers', 'Red', 'Kitchener')
SELECT * FROM dual;
COMMIT;

CREATE TABLE players (
    playerID INT PRIMARY KEY,
    firstName varchar(20),
    lastName varchar(20),
    teamID INT );

INSERT ALL
    INTO players VALUES (1, 'John', 'Smith', 10)
    INTO players VALUES (2, 'Bob', 'Marley', 10)
    INTO players VALUES (3, 'Steven', 'King', 11)
    INTO players VALUES (4, 'Jim', 'Parsons', NULL)
SELECT * FROM dual;
COMMIT;

CREATE TABLE fields (
    fieldname varchar(15) PRIMARY KEY,
    Address varchar(50),
    Manager varchar(25));

INSERT ALL
    INTO fields VALUES ('Toronto', '22 Queen St. E.', 'Doug Ford')
    INTO fields VALUES ('Barrie', '380 Bayview Ave', 'Richard Alexander')
    INTO fields VALUES ('Kitchener', '220 Weber St. S', 'Marco Albano')
    INTO fields VALUES ('Waterloo', '576 King St. N', 'Pauyl Caruso')
SELECT * FROM dual;

COMMIT;
```