SQL Course

An Introduction to Simple Joins

- Introduce the concept of an Equi-Join.
- Note some important considerations when joining tables.
- Illustrate the use of multi-table queries with a few examples.

A join based on an exact match between two columns is more precisely called an **Equi-Join**. Joins can also be based on other kinds of column comparisons. As all of the data in a relational database is stored in its columns as explicit data values, all possible relationships between tables can be formed by matching the contents of related columns. The join thus provides a powerful facility for exercising the data relationships in a database.

The syntax for an equi-join follows :

> **SELECT** *select_list* **FROM** *table_1, table_2* [,*table_3* ...] **WHERE** *[table_1.]column* **=** *[table_2.]column*;

When the join operator is given as an **=**, the join is called an equi-join. The search condition compares columns from two different tables. We call these two columns the matching columns for the two tables. Like all search conditions, this one restricts the rows that appear in the query results.
The most common multi-table queries involve two tables that have a natural parent/child relationship. This is usually achieved using a primary key/foreign key relationship. SQL does not require that the matching columns be included in the results of a multi table query.

The search condition that specifies the matching columns in a multi-table query can be combined with other search conditions to further restrict the contents of the query results. To join tables based on a parent/child relationship where the relationship is specified by more than one column you must join the tables over these multiple columns. Multi column joins are less common than single column joins, and are usually found in queries involving compound keys.

SQL can combine data from three or more tables using the same basic technique used for two table queries. Look at the following example to see how this is achieved:

> **SELECT** *select_list* **FROM** *table_1, table_2, table_3* **WHERE** *[table_1.]column* **=** *[table_2.]column* **AND** *[table_2.]column* **=** *[table_3.]column*;

The vast majority of multi table queries are based on parent/child relationships, but SQL does not require that the matching columns be related as a foreign key and primary key. Any pair of columns from two tables can serve as matching columns, provided they have comparable data types. Matching columns like this, generate a many to many relationship between the two tables.

- Joins that match primary keys to foreign keys always create one to many parent/child relationships.
- Other joins may also produce one to many relationships if one of the matching columns has unique values for all rows in the table.
- In general, joins on arbitrary matching columns generate many to many relationships.

Non equi-joins are when the tables are joined by another comparison operator other than the equality sign.

What happens when the connecting columns contain a **NULL** value ?

- Most SQL implementations are clear about what happens when nulls are present in columns being joined. If there are nulls in the connecting columns of tables being joined, the nulls will never join because nulls represent unknown or inapplicable values.
- Few SQL implementations support the join on null values. DB2 is one of the few. There is ongoing debate about how a system should deal with joins on null values. The consequence of not supporting a join on nulls is that some needed or useful information would not appear in query results.

The multi table queries do not usually require any special SQL syntax or language features beyond those described for single table queries. However some multi table queries cannot be expressed without the additional SQL language features described as follows :
Qualified Column Names
These are needed to eliminate ambiguous column references. This happens when two tables share the same column name and that column is specified in the select list. This leads SQL to produce an error. To eliminate this ambiguity, you must use a qualified column name to identify the column. This is achieved by specifying the name of the table followed by the column name separated by a full stop. The table specified in the qualified column name must, of course, match one of the tables specified in the **FROM** clause.
All column selection (*)
In a multi table query, the * selects all columns from all tables in the **FROM** clause. Many SQL dialects treat the asterix as a special kind of wild-card column name that is expanded into a list of columns. In these dialects, the asterix, can be qualified with a table name. Although this is permitted in most SQL dialects, it is not permitted by the ANSI/ISO standard.
Self Joins
Some multi table queries involve a relationship that a table has with itself. SQL uses an imaginary duplicate table approach to join a table to itself. Instead of actually duplicating the contents of the table, SQL simply lets you refer to it by a different name, called a table alias. The **FROM** clause assigns an alias to each copy of the table by specifying the alias name immediately after the actual table name. When a **FROM** clause contains a table alias, the alias must be used to identify the table in qualified column references.

Table Aliases or Correlation Names
A Correlation name (or alias) allows a convenient shorthand for a table name.

- The correlation name is paired with the table name in the **FROM** clause.
- Correlation names also allow more advanced queries in conjunction with sub queries.
- **SELECT** *af.call_sign, af.departure_time*
  FROM *aircraft_flight af*;
  Displays the call sign and departure time of all flights.

Table aliases are required in queries involving self joins but they can also be used in any query. For example if your query refers to another users table, or if the name of a table is very long, the table name can become tedious to type as a column qualifier. If a table alias is specified, it becomes the table tag, otherwise the tables name, exactly as it appears in the **FROM** clause, becomes the tag.
The SQL2 standard optionally allows the keyword **AS** to appear between a table name and

table alias. While this makes the **FROM** clause easier to read, it may not be yet supported by all implementations of SQL.

The following example retrieves the aircraft name, flight no, departure details and arrival location for all aircraft flights :

> **SELECT** *aircraft.aircraft_name, flight.flight_no, flight.departure_date, flight.departure_time, flight.departure_location, flight.arrival_location* **FROM** *aircraft, aircraft_flight, flight* **WHERE** *aircraft_flight.call_sign* **=** *aircraft.call_sign* **AND** *aircraft_flight.flight_no* **=** *flight.flight_no* **AND** *aircraft_flight.departure_date* **=** *flight.departure_date***;**

We can join tables using more than one criteria, by including another clause in the **SELECT** statement.

The following example will return only airplanes of model 'Boeing 737-200' that are registered for flights :

> **SELECT** *aircraft.aircraft_name, aircraft.model, aircraft.no_club_seats, aircraft.no_economy_seats, aircraft_flight.call_sign, aircraft_flight.flight_no, aircraft_flight.departure_date* **FROM** *aircraft, aircraft_flight* **WHERE** *aircraft_flight.call_sign* **=** *aircraft.call_sign* **AND** *aircraft.model* **=** 'Boeing 737-200'**;**