Database Designs

Section 4: Subqueries, Constraints & Advanced Concepts

4.1 Subqueries Introductor

So far we have explored how to query from tables and virtual table to generate a results table. SQL Subqueries is a fairly advanced concept which allows us to query from a SQL query i.e. a query within a query.

So far we have seen queries as standalone commands that fetch data from a database; however, in reality, queries are generally plug-and-play - what do we mean by this? Plug-and-play means the ability to use queries in places where you would not expect them to be used, this is because the results of queries are tables.
Tables can be real tables, tables that are generated by joins or tables that are a result of queries. Therefore, queries are plug-and-play into other pieces of SQL.

A query is a command that returns a table (columns and rows). If you imagine the result of a query as a table by itself, then this mentality will open all the possibilities that you can do with the results of a query. For example:

We could calculate the Union, Intersection or Differences of two queries.
We could use one query inside another (via Subqueries).
We could use a subquery to populate a table via an **INSERT**.

To conclude, a subquery is nothing more than a query of which the results are used in another query, otherwise there is no logical differences between a subquery and a normal standalone query.

4.2 Union, Union All, Intersect and Except

We can calculate the Union, Intersection and Difference between two queries provided they have the same columns i.e. the number, order and type of the columns are identical across the two queries.

To demonstrate the **UNION**, **UNION ALL**, **INTERSECT** and **EXCEPT** set operators subqueries, below are two tables QUERY_ONE and QUERY_TWO.
The QUERY_ONE table row data is coloured turquoise while the QUERY_TWO table row data is coloured peach. The common data rows of both tables are coloured in yellow.

This will help clearly illustrate how SQL creates the subquery tables using each Set operators and how we can use subqueries to perform more advanced queries and open our minds to all the possibilities SQL offers using the plug-and-play mentality.

QUERY_ONE			
String	Number	Date	Boolean
🟦	🟦	🟦	🟦
🟨	🟨	🟨	🟨
🟦	🟦	🟦	🟦
🟦	🟦	🟦	🟦

QUERY_TWO			
String	Number	Date	Boolean
🟥	🟥	🟥	🟥
🟧	🟧	🟧	🟧
🟨	🟨	🟨	🟨
🟧	🟧	🟧	🟧

The **UNION** set operator creates a new query table which combines the two queries together but creates one copy of the common row i.e. removes duplicate rows.

UNION_QUERY			
String	Number	Date	Boolean
🟦	🟦	🟦	🟦
🟦	🟦	🟦	🟦
🟦	🟦	🟦	🟦
🟥	🟥	🟥	🟥
🟧	🟧	🟧	🟧
🟨	🟨	🟨	🟨
🟧	🟧	🟧	🟧

The **UNION ALL** set operator creates a new query table which combines the two queries together containing all duplicate rows.

UNION_ALL_QUERY			
String	Number	Date	Boolean
🟦	🟦	🟦	🟦
🟨	🟨	🟨	🟨
🟦	🟦	🟦	🟦
🟦	🟦	🟦	🟦
🟥	🟥	🟥	🟥
🟧	🟧	🟧	🟧
🟨	🟨	🟨	🟨
🟧	🟧	🟧	🟧

The **INTERSECT** set operator finds the commonality between the two sets that it is intersecting i.e it returns the common row between the two query tables.

INTERSECT_QUERY			
String	Number	Date	Boolean
🟨	🟨	🟨	🟨

Finally, the **EXCEPT** set operator creates a new query table which removes everything in the first table (i.e. the left table) that is present in the second table (i.e. the right table). This returns the difference between the two sets of input queries.

EXCEPT_QUERY			
String	Number	Date	Boolean

EXCEPT_QUERY			
String	Number	Date	Boolean

Important Note: The Except query can return either of the above results depending on which table was made as the left table in the subquery. The left table is always the first table mentioned in the query within the **FROM** clause.

The **UNION**, **UNION ALL**, **INTERSECT** and **EXCEPT** are all a kind of Set operations. We know that a query is a command that returns a table and not really a Set. The Entity Relationship theory tells us that a table is a collection of tuples (a tuple relates to one row within the table). A Set cannot contain duplicates but a table can.

By default the **UNION** will eliminate duplicates but SQL has the Union All operator to keep all the duplicate tuples if we seek that operation behaviour. There is another rule which applies to the **UNION** Set operator which is where individual queries that participate in a **UNION** operator cannot have the **ORDER BY** clause. This is because the elements of a Set are not ordered. Below is an example demonstrating this:

Pet_Owners	
AptNumber	Name
123	John
345	Tim
349	Vandana
567	Bilal

Flat_Owners	
FlatNumber	Name
234	Mary
567	Bilal
879	Jane
903	Ellen

In this example we have two tables of Pet_Owners and Flat_Owners both have the same columns i.e. they have in common the same number, order and type of columns. The names of the columns is irrelevant. In the above example, both tables have two columns of integer and string in that specific order. Below is the syntax to perform the **UNION** Set operator on the two tables.

```
(SELECT AptNumber AS FlatNumber, Name FROM Pet_Owners)
UNION
(SELECT FlatNumber, Name FROM Flat_Owners);
```

It is OK for the column names in the physical tables to be different. This is because in a query we can easily alias one column name to another - notice the AptNumber **AS** FlatNumber which simply glosses/hides the fact that the original column name was something different. This is the reason for why the actual name from the physical table is irrelevant. The final subquery table will return a results table with two columns called FlatNumber and Name as seen below:

UNION_RESULTS_TABLE	
FlatNumber	Name
123	John
345	Tim
349	Vandana
234	Mary
567	Bilal
879	Jane
903	Ellen

As previously mentioned, individual queries that have been ordered by the **ORDER BY** clause cannot be used with the UNION set operator. Therefore, the below example syntax is invalid and will not work returning an error:

```
(SELECT AptNumber AS FlatNumber, Name FROM Pet_Owners ORDER BY Name)
UNION
(SELECT FlatNumber, Name FROM Flat_Owners ORDER BY Name);
```

However, we can write the syntax so that the results of the **UNION** is ordered by the column Name. In this case the **ORDER BY** clause will work on ordering the **UNION** results table:

```
(
  (SELECT AptNumber AS FlatNumber, Name FROM Pet_Owners)
  UNION
  (SELECT FlatNumber, Name FROM Flat_Owners)
) ORDER BY Name;
```

Pay particular attention to the curly brackets which wraps around the whole **UNION** set operator subquery. The **ORDER BY** clause is applied to the results of the **UNION** because it is falls outside the wrapping curly brackets. Therefore, it is only the individually queries participating in the **UNION** that cannot have the **ORDER BY** clause and the reason for why the second syntax works without throwing any errors.