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.

4.3 Query-In-A-Query

Subqueries are very useful because they allow us to write SQL queries entirely free of hardcoded values or very large intermediary tables. Subqueries at first can seem very difficult to get use to at the beginning but once you know how to use them they are

extremely powerful. We will now explore how to use a query within a query (also known as subqueries) using the tables below.

Stores_Data		
storeID	storeLocation	city
1	ASDA Wilmslow	Manchester
2	ASDA Stratford	London

Products_Data	
productID	productName
1	Bread
2	Milk
3	Noodles
4	Nutella

Sales_Data			
storeID	productID	salesDate	totalRevenue
1	1	November 20, 2020	7,233.32
1	3	November 20, 2020	3,234.84
1	2	November 20, 2020	5,865.55
1	2	November 20, 2020	6,849.99
2	3	November 20, 2020	2,110.95
2	2	November 20, 2020	4,558.24
2	4	November 20, 2020	2,284.75

If a database existed like the above it is most likely that the business would like to pull a report to see what are the annual revenue is like for various products and make business decisions around this information. The query to extract this report would look like the following:

```
SELECT p.productName, YEAR(date), SUM(totalRevenue)
FROM Sales_Data AS s
INNER JOIN Products_Data AS p
        ON s.productID = r.productID
WHERE (p.productName = 'Bread' or p.productName = 'Milk')
        AND (YEAR(date) = 2020)
GROUP BY p.productName, YEAR(date);
```

In the above syntax we are matching on the productID column and combining the Products_Data table onto the Sales_Data table using an **INNER JOIN**. The **WHERE** clause shows that we are interested in products 'Bread' and 'Milk' which are believed to be the top sellers and we are interested in the current year. Finally, we wan to **GROUP BY** the productName and the year. In conclusion this query will return back using the **SELECT** statement the productName, year and the **SUM** of the totalRevenue for each product within the current year groups.

The above query will only return information for the year 2020 and about 'Bread' and 'Milk' because that is the date and products specified in the **WHERE** clause. This query would get the job done to extract the report from the database for the business. However, there are many issues with the above query which relate to hardcoding the values. Our objective is to re-write the code to avoid any hardcoded values.

First, the query is hardcoded for the year 2020 which means the query would need to be updated every year. To avoid this hardcoding of values we can plug one query into another using subqueries as seen below.

```
SELECT p.productName, YEAR(date), SUM(totalRevenue)
FROM Sales_Data AS s
INNER JOIN Products_Data AS p
        ON s.productID = r.productID
WHERE (p.productName = 'Bread' or p.productName = 'Milk')
        AND (YEAR(date) = (SELECT YEAR(MAX(date)) FROM Sales_Data))
GROUP BY p.productName, YEAR(date);
```

Here we are using one query inside another larger query. The inner query (highlighted in yellow) returns the maximum value of the year from the Sales_Data table. This query will look for the last transaction from the Sales_Data table because this would be the maximum (latest) date and return the year value from the date. Therefore, the inner query returns a single value i.e. the year in which the last transaction occurred.
In the **WHERE** clause of the outer query it simply compares the year of the date that is returned from the inner query. The **WHERE** clause functions as it did before.

Note that the the inner queries are always evaluated first.

The next issue relates to the hardcoded products value within the **WHERE** clause. What if we do not want to hardcode these values. Let's imagine that we instead have a TopSellers_Data table which has the productID and productName columns. This table holds the top products we are interested in at any point in time.

TopSellers_Data	
productID	productName
1	Bread
2	Milk

Instead of hardcoding the products in our query we instead want to retrieve the products from the TopSellers_Data table. Again we can do this using a subquery as seen below:

```
SELECT p.productName, YEAR(date), SUM(totalRevenue)
FROM Sales_Data AS s
INNER JOIN Products_Data AS p
        ON s.productID = r.productID
WHERE (p.productName IN (SELECT p.productName FROM TopSellers_Data)
        AND (YEAR(date) = (SELECT YEAR(MAX(date)) FROM Sales_Data))
GROUP BY p.productName, YEAR(date);
```

Once again we have plugged one query into another (the inner query is highlighted in yellow). The inner **SELECT** statement simply returns all the products from the TopSellers_Data table. This TopSellers_Data table can be modified/updated and the query will always return the latest top products. The inner query returns a range of values which the outer query can then use these values using the **IN** keyword for its **WHERE** clause check for the current top products.

The final issue with the original query is to do with the size of our table. Let us imagine as time passes the business is operating extremely well. This means more sales will be recorded in the Sales_Data table and the possibility that the Sales_Data table eventually becomes enormous. We now have an issue that the table is too large to perform the **INNER JOIN** on the Sales_Data table which could take too long to generate the report. What we wan to do is reduce the number of rows that we select from the Sales_Data table. We know that at any point in time we are only interested in a few products which are present in the TopSellers_Data table. So How can we reduce the number of rows in the Sales_Data table that should be part of the query? Once again we would use subqueries as seen below:

```
SELECT p.productName, YEAR(date), SUM(totalRevenue)
FROM (SELECT * FROM Sales_Data WHERE productID IN (SELECT productID
        FROM TopSellers_Data)) AS s
INNER JOIN Products_Data AS p
        ON s.productID = r.productID
WHERE (p.productName IN (SELECT p.productName FROM TopSellers_Data)
        AND (YEAR(date) = (SELECT YEAR(MAX(date)) FROM Sales_Data))
GROUP BY p.productName, YEAR(date);
```

Instead of only using the Sales_Data table in the **INNER JOIN**, we can use a **SELECT** statement which returns a table to be one of the tables in the **INNER JOIN**. Remember the result from a **SELECT** statement is also a virtual table which can therefore be used as part of any of the joins i.e. wherever we use a table, we can use a query. Once again we have used one query plugged into another query to improve the performance of our query. We are now using a subset of the Sales_Data table which helps improve the performance of our query. The outer query uses the subset table (which is also a table) in the **FROM** clause to do the **INNER JOIN**.

We now have a final subquery that looks like the below:

```
SELECT p.productName, YEAR(date), SUM(totalRevenue)
FROM (SELECT * FROM Sales_Data WHERE productID IN (SELECT productID
        FROM TopSellers_Data)) AS s
INNER JOIN Products_Data AS p
            ON s.productID = r.productID
WHERE (p.productName IN (SELECT p.productName FROM TopSellers_Data)
           AND (YEAR(date) = (SELECT YEAR(MAX(date)) FROM Sales_Data))
GROUP BY p.productName, YEAR(date);
```

We can see that this query is entirely free from any hardcoded values or very large intermediate tables all thanks to the use of subqueries rather than the tables itself.

To conclude, subqueries allows us to use non-hardcoded values by returning values from subqueries as well as optimising the performance of our queries by using subset tables created by subqueries. As we can see subqueries can be difficult to grasp at the beginning but once we understand how to harness them they become very powerful and useful tools when querying databases.

4.4 Inserting via Subqueries

We have been exploring the different ways of how we can use subqueries as plug-and-play. We know that queries are perfect substitutes for tables in SQL. In this section we will explore how we can populate a table using subqueries.

We could use a subquery to populate a table via a **INSERT**. The **INSERT** statement is the mechanism we use to get data into a table. We have already explored the **CREATE TABLE** and **INSERT** statements in Section 2.8 and 2.10. The **CREATE TABLE** is used to create a table in a database while the **INSERT** statement is used to insert new tuples (row data) into a table.

Using the **INSERT** statement requires us to use the statement many times to insert many data into a table (we also explored bulk loading data to speed up inserted data into a table using an external data source).

It is also possible that the data we want to insert into a table already exists in another table in some other form. This is where subqueries are useful as a plug-and-play solution. Subqueries are far more convenient to populate a table from existing tables in the database.

There are two methods we can do this:

1. Create the table as usual and then insert the data using a subquery, or
2. Create the table and populate it directly using a subquery in one go

Both methods are demonstrated below starting with the first method then followed by the second method:

Method 1:

```
CREATE TABLE EmailAddresses (
        Email VARCHAR(30) NOT NULL,
        Category VATCHAR(10) NOT NULL
);

INSERT INTO EmailAddresses
(SELECT DISTINCT Email, 'Student' AS Category FROM Students)
UNION
(SELECT DISTINCT Email, 'Faculty' AS Category FROM Faculty);
```

In this first example, we create a new table called EmailAddresses (using the normal process) which has two columns of Email and Category. This is then followed by a second command that **INSERT INTO** the EmailAddresses table. However, this is not the normal **INSERT INTO** command we are use to, instead what follows the statement is a query which is unusual.
The query that follows the **INSERT INTO** statement is a **UNION** of two queries . Note that this does not necessarily need to always be a **UNION** but rather it needs to be any query where the result will be inserted into the table. There is one requirement. The query needs to match the table in the number of and type of columns.

In the above the query has to select two columns because the EmailAddresses table has two columns of Email and Category and the type/order of the columns returned from the query must both be the same type/order as specified in the EmailAddresses columns i.e. VARCHAR for both.

Method 2:

```
CREATE TABLE EmailAddresses (
        Email VARCHAR(30) NOT NULL,
        Category VATCHAR(10) NOT NULL
)
AS
(SELECT DISTINCT Email, 'Student' AS Category FROM Students)
UNION
(SELECT DISTINCT Email, 'Faculty' AS Category FROM Faculty);
```

In the second example, the first part before the **AS** keyword is the **CREATE TABLE**'s table definition. The second part after the **AS** keyword is the query. Both of these parts are combined together to form one SQL Statement using **AS** keyword.
The query is exactly the same as the first method example and the same rules applies for the second method i.e. the same number of columns and column types have to match the table specified in the **CREATE TABLE** portion of the statement.
The **AS** keyword is the linking portion of the statement which links the **CREATE TABLE** with the query.
There is nothing special with the second method other than combing two separate statements into one statement.

We can refactor the above statement so that it is even shorter by eliminating some part of statement which we considered as not important:

```
CREATE TABLE EmailAddresses
AS
(SELECT DISTINCT Email, 'Student' AS Category FROM Students)
UNION
(SELECT DISTINCT Email, 'Faculty' AS Category FROM Faculty);
```

The above has removed the table column definitions entirely because we can figure out what columns the EmailAddresses table should have on the basis of **SELECT** query. However, skipping this step is not recommended because any constraints and keys we want to specify in the EmailAddresses table will be missing.

Note that we can also use subqueries in **UPDATE** and **DELETE** statements in a similar fashion to the above.

This concludes the three different ways we can use subqueries in a plug-and-play manner.

4.5 NOT NULL Constraints

Constraints are conditions that data in a database must satisfy. The database administrators/architects specifies the constraints/conditions when creating a database table and the DBMS enforces the constraints. This functionality is a huge part of the appeal of DBMS over say a File System for data storage.

Transactions are logical units of work and the database guarantees that each transaction satisfies the four properties known as the ACID properties. ACID is a famous acronym for Atomicity, Consistency, Isolation and Durability. The underpinning of the ACID properties is the Consistency i.e. it is the most important property because it encapsulates a lot that occurs with the database. The Consistency in a database is explicitly measured in terms of constraints. As long as the constraints in a database are all satisfied the database is considered consistent. Constraints collectively are sometime called integrity constraints because the explicit purpose of the constraint is to make sure the integrity of the data in the database are maintained and not compromised.

There are four common types of constraints that we should understand:

- NOT NULL Constraints
- Primary Key Constraints
- Foreign Key Constraints
- Check Constraints

Important Note: Constraints are usually added when the table is created but can also be added after a table has been created using the **ALTER TABLE** command. If we try to add a constraint to a table and the pre-existing data in the table does not satisfy the constraint, the DBMS will fail to alter the table and the SQL interpreter will provide an error message telling us that the data is not in a state of integrity to begin with. We can go ahead and fix up the data and then re-run **ALTER TABLE** command.

NOT NULL constraints (and default values constraints) help define whether NULL values are allowed/permissible in a particular column of a table.

Remember that NULL implies that a value does not exist. This is a special value and is not the same as a blank string or zero value which are values that do exist. NULL is also neither TRUE or FALSE (boolean values) and is simply NULL (absence of existence)

Any column can contain a value of NULL provided the table has been defined in a way that allows NULL values in that column.

Default Value are a closely related concept to NOT NULL constraints. They are a way of specifying what value a particular column should take when data value for that column is not present. We use the **DEFAULT** keyword followed by a value to specify the default value for that column when setting up the table. Therefore, we can specify a table with a NOT NULL constraint but also provide a default value. If an insert statement skips the value of the column, the default value will be assigned instead of a NULL. This will ensure the NOT NULL constraint is being fulfilled while at the same time not restricting users in situations where a value is not provided.

4.6 Primary Key Constraints

There are two types of Key Constraints: Primary Key and Foreign Key.

A key is a set of columns whose values are unique for each row in a table. A Primary key is a special type of key/candidate key. A key/candidate key is a set of columns whose values are unique in each row of a table. A table can have any number of keys/candidate keys of which only one of the key is marked as being special.

There should always be a Primary Key in every table created and if it is missing there should be a very valid reason - typically because there is a Foreign Key constraint. If a table has neither a Primary Key nor a references to another Primary Key in another table then this might be a sign that the table design is flawed.

The DBMS will often construct an index on the primary key even without being explicitly told to do so. The database is predicting ahead of time that if we mark a column or a set of columns as a primary key then we are going to be doing a lot of queries/lookup on the primary key and therefore it makes sense for the database to be ahead of our requests and make it easy to perform these queries. Smart database optimisations such as these which are based on heuristic signals that database designers and optimiser have picked up over the years of preparing database engines are an important reason for why database systems are getting so much better steadily over time.

This distinction between a Primary Key and a general candidate key is something that is often asked in interviews so it is important to remember this constraint.

A Primary Key could include either a single column or multiple columns. A multiple columns key is also known as a Composite Primary Key and the DBMS will check that the combination of values from the selected columns are unique.

Primary Key columns can never contain NULL values. This is a common sense rule to keep in mind because it does not make sense for a Primary Key column to not exist and NULL means a value does not exist.

Primary Keys are a type of constraint because it specifies a condition that the data in the table must satisfy. This is an Integrity constraint and by far the most important type of Integrity constraint. The DBMS will throw an error if this constraint is violated for example inserting two rows with the same values for the Primary Key column(s). This is how DBMS can maintain the ACID properties i.e. Consistency and Integrity of the data.

We specify the Primary Key when creating a new table. This can be specified at the very end of the **CREATE TABLE** command (i.e. after the table columns have been defined) or inline if it is a single column Primary Key (i.e. add **PRIMARY KEY** after the column which is the key attribute) as seen below:

```
CREATE TABLE Students (
        studentID INT NOT NULL AUTO_INCREMENT
        firstName VARCHAR(30) NOT NULL
        PRIMARY KEY(StudentId)
);

CREATE TABLE Students (
        studentID INT NOT NULL AUTO_INCREMENT PRIMARY KEY
        firstName VARCHAR(30) NOT NULL
);
```

We cannot use the inline method to define multi-column Composite Primary Keys which requires the former explicit method out-of-line definition. We can add Primary Key constraints after a table has been created and pre-existing data exists in the table. We would do this using the **ALTER TABLE ADD INDEX** command. This command will go through all the pre-existing data in the table to ensure that it does not violate this Primary Key constraint to begin with before applying the alteration to the table.

To conclude on Primary Key Constraints, any columns or set of columns could be declared using the **UNIQUE** keyword and the syntax is exactly like that of a **PRIMARY KEY**. A table can have only one **PRIMARY KEY** but any number of **UNIQUE** constraints; however, the **PRIMARY KEY** constraint implies uniqueness but the reverse is not true for **UNIQUE** constraints.

```
CREATE TABLE Students (
        studentID INT NOT NULL AUTO_INCREMENT PRIMARY KEY
        firstName VARCHAR(30) NOT NULL
        emailAddress VARCHAR(30) NOT NULL UNIQUE
);
```

4.7 Foreign Key Constraints

The Foreign Key Constraint is used to define a column with a table that is also a key in another table. Such constraints are called Foreign Key Constraints or Referential Integrity Constraints. It is called a Referential Integrity Constraint because one table refers to another table to check its integrity is maintained and cannot be purely determined internally within the table alone i.e. it is only possible to determine the integrity via the reference to the external table.

```
CREATE TABLE Campus_Housing (
        studentID INT NOT NULL,
        dormitoryName VARCHAR(50),
        apartmentNumber INT,
        CONSTRAINT fk_student_studentid,
        FOREIGN KEY(studentID),
        REFERENCES Students(studentID)
);
```

In the example syntax above, the **CONSTRAINT** keyword is used to setup the referential integrity of which we can provide a name for this Foreign Key Constraint e.g. fk_student_studentid. The **FOREIGN KEY**(studentID) refers to the studentID columns from the internal Campus_Housing table while the **REFERENCES** Students(studentID) refers to the studentID from the external Students table. Again we can add a foreign key constraint at a later time after the table has been created and has pre-existing data but the table must satisfy the constraint before it is applied to the table.

We should always make sure to add our constraints during the creation of the table and that our tables have either a Primary Key or Foreign Key constraint. Foreign Keys are usually added when one table relies on another table to make sense of the data (but not vice versa). In the example above the Campus_Housing table relies on the Students table to make sense of the data held in that table while the Students table does not rely on the Campus_Housing table and can make sense on its own.