Database Designs

Section 3: Advanced Retrieving Data Queries

3.1 Aggregate Operators

Aggregate operators allows us to return more interesting information from our data. Below is an example Sales_Data table containing sales for two ASDA stores.

Sales_Data			
storeLocation	productName	salesDate	revenue
ASDA Coventry	Milk	November 02, 2020	1,233.32
ASDA Birmingham	Bread	November 02, 2020	5,434.74
ASDA Coventry	Coffee	November 02, 2020	3,855.96
ASDA Coventry	Coffee	November 02, 2020	2,280.90
ASDA Birmingham	Coffee	November 02, 2020	2,110.95
ASDA Coventry	Milk	November 01, 2020	4,558.24
ASDA Birmingham	Milk	November 01, 2020	6,849.99
ASDA Birmingham	Bread	November 01, 2020	2,543.57

The insights an analyst or business owner may be interested in retrieving from the above dataset are: Total Revenue, Best Performing Stores, Best Performing Products, Best Product-Store Combination that Sold Best.

We would like to find patterns in the data so that we can drive business decisions that would help the business to perform better. Below are some example Aggregate Operations which can help us retrieve more interesting data insights.

```
SELECT SUM(revenue) totalRevenue
FROM Sales_Data;
```

The **SUM** aggregate operator wraps the column in brackets for the column data we wish to apply the sum function. Whatever result is returned from the aggregate operator is then returned in a column we have decided to name as totalRevenue. By omitting the **WHERE** clause from the select statement means that it will sum over all data within the Sales_Data table.

The **SUM** is a function that operates over an entire column and not just a single cell and the result returned is a single value which can be named. The **SUM** can operate over an entire column or any subset of a column and does not necessarily have to be an entire column.

The reason SUM is called an aggregate function is because it acts on an aggregation of cells and not just a single cell.

```
SELECT AVG(revenue) averageRevenue
FROM Sales_Data;
```

The **AVG** aggregate operator is a function that operates over an entire column or a subset of a column and not just a single cell and returns a single value which can be named. This function averages all the values in the column. The syntax is the same as the **SUM** function above but it returns the average instead.

3.2 Group By Clause

The **GROUP BY** clause allows us to aggregate the returned data by grouping data in a certain logical unit. An example command using the **GROUP BY** clause would look like the below:

```
SELECT storeLocation, SUM(revenue) totalRevenue
FROM Sales_Data
GROUP BY storeLocation;
```

The result from the above query will return a table with two columns, storeLocation and totalRevenue, with a total sum of revenue for each store (i.e. a row per unique store).

The column we specify after the **GROUP BY** clause define what groups we would want to use and subtotal. Using the **SUM** aggregate operator with the **GROUP BY**, would define to what groups would we sum up i.e. what is the logical unit that we would run the **SUM** operation on. The **GROUP BY** can be used with other aggregate operators such as the **AVG** function.

We can visualise the **GROUP BY** as sorting the data before we perform the other operations i.e. before performing the **SUM** operation on the revenue column for every group. This will give a flatten result table demonstrated below:

Sales_Data			
storeLocation	productName	salesDate	revenue
ASDA Birmingham	Bread	November 02, 2020	5,434.74
ASDA Birmingham	Coffee	November 02, 2020	2,110.95
ASDA Birmingham	Milk	November 01, 2020	6,849.99
ASDA Birmingham	Bread	November 01, 2020	2,543.57
ASDA Coventry	Milk	November 02, 2020	1,233.32

ASDA Coventry	Coffee	November 02, 2020	3,855.96
ASDA Coventry	Coffee	November 02, 2020	2,280.90
ASDA Coventry	Milk	November 01, 2020	4,558.24

In the example query we can see that ASDA Birmingham is the best performing store

Sales_Data Query Results	
storeLocation	totalRevenue
ASDA Birmingham	16,939.25
ASDA Coventry	11,928.42

based on total revenue.

The **SELECT** statement allows us to choose which columns we wish to display in our results after we have completed the thinning of the data using the **GROUP BY**. We can confidently remove the productName and salesDate columns from the results table without losing any information. However, we cannot **GROUP BY** any column that is not present in the select statement since the results would not make any sense. Therefore, any column specified in the **GROUP BY** statement must also be present in the **SELECT** statement i.e. it must be part of the final result.

Remember: The **WHERE** clause is generally used when we want to filter the number of rows that meet the **WHERE** clause condition. Therefore, if we want all rows we omit the **WHERE** clause. The **GROUP BY** clause on the other hand is used when we want to group our data in a logical unit. It is therefore possible to use both the **GROUP BY** and **WHERE** clauses at the same time for a more complex query.

The **GROUP BY** clause allows us to group by multiple columns so that we can answer more complex questions that uses a combination of columns. An example query would look like the below:

```
SELECT productName, storeLocation, AVG(revenue) averageRevenue
FROM Sales_Data
GROUP BY productName, storeLocation;
```

The **GROUP BY** clause above groups based on the unique combination of the two columns specified in the statement. In the above example, this provides a unique combinations of: ASDA Birmingham Bread, ASDA Birmingham Coffee, ASDA Birmingham Milk, ASDA Coventry Coffee and ASDA Coventry Milk.
The **AVG** operation will then be performed on each unique **GROUP BY** unit combination to provide the averages of each unit groups.

The above example can help us answer the question of what is the worst performing product in which store.

3.3 Order By Clause

The **ORDER BY** clause allows us to order our results either in Ascending or Descending order. The ordering will depend on the data type i.e. numbers ordered smallest to largest in ascending order and vice versa for descending order while strings are ordered A-Z for ascending order and vice versa for descending. Below is an example of ordering the results of our query using the **ORDER BY** clause:

```
SELECT storeLocation, SUM(revenue) totalRevenue
FROM Sales_Data
GROUP BY storeLocation
ORDER BY storeLocation;
```

```
SELECT storeLocation, SUM(revenue) totalRevenue
FROM Sales_Data
GROUP BY storeLocation
ORDER BY totalRevenue DESC;
```

In the first example the column storeLocation column is forced to order the results in ascending order. In the second example, using the **DESC** keyword forced the results to order the column by descending order.

By default SQL does not order the results that are returned; therefore, using the **ORDER BY** clause there is no longer any ambiguity in the order of the result.

Whatever we specify in the **ORDER BY** clause must be present in the **SELECT** clause which is similar ruling as the **GROUP BY** clause. We can also **ORDER BY** more than one column as seen in the example below:

```
SELECT productName, storeLocation, AVG(revenue) averageRevenue
FROM Sales_Data
GROUP BY productName, storeLocation
ORDER BY storeLocation, productName DESC;
```

In the above example, SQL would order columns in the order they are specified in the query. Therefore, storeLocation will be ordered first in ascending order and then productName will be ordered second in descending order. Each column must be specified the type of ordering required and must be present in the **SELECT** clause.

It is important to note, a SQL query that contains both a **GROUP BY** and a **ORDER BY** clause, the ordering is applied after the grouping i.e. the grouping occurs first and then the ordering is performed on the results. There are many rules like this in SQL which we will see as we continue; for example, the **LIMIT** clause is the only clause that can apply after the **ORDER BY** clause.

3.4 Limit Clause

The **LIMIT** clause allows us to limit the number of results returned from our query results. This is another way of filtering our results once the query has run. Below is an example of using the **LIMIT** clause:

```
SELECT storeLocation, SUM(revenue) totalRevenue
FROM Sales_Data
GROUP BY storeLocation
ORDER BY totalRevenue
LIMIT 1;
```

The above example limits the results returned to the very first row in the results returned. This value can be any number and will limit the number of rows returned from the results. Therefore, if we chose to **LIMIT** up to 3 results and a query returned more than 3 results, the limit will return the first 3 results, whereas, if the query returned less than or equal to 3 results then the first few records up to the limit will be returned in the results.

This will help our query to automatically find the results, without requiring to manually check for the answer, within a single SQL query. For example, the above query allows us to answer the question of which is the worse performing store by revenue using a single query. Adding the **DESC** ordering on totalRevenue will answer which is the best performing store by revenue.

If we want to **LIMIT** the number of results from our SQL query but have an **ORDER BY** clause in the query, the **LIMIT** clause must always come after the **ORDER BY** clause. It is not needed to have the **ORDER BY** clause in order to use the **LIMIT** clause in a SQL query.

3.5 Count and Count Distinct

The **COUNT** operator is an aggregate operator similar to **SUM**, **AVG**, **MIN**, **MAX**, etc. The **COUNT** operator operates on the column(s) defined in the round brackets. The **DISTINCT** keywords is used to return unique values i.e. it tells SQL to ignore duplicates within the counting. An example syntax is provided below:

```
SELECT COUNT(storeLocations) uniqueStoreLocations
FROM Sales_Data

SELECT COUNT(DISTINCT storeLocations) uniqueStoreLocations
FROM Sales_Data
```

The first syntax example returns a count of all rows while the second syntax returns a count of unique values only.

3.6 Introduction to SQL Joins

SQL Provides many ways to combine information from two or more tables in order to query the new combined table information. The way we combine a table is called a “Join” and there are a while variety of joins.

In the previous sections we have seen a Sales_Data table containing a lot of information within a single table. In a real life scenario the table would actually be split into multiple tables. The Sales_Data table would look something like the below tables:

Stores_Data		
storeID	storeLocation	city
1	ASDA Small Heath	Birmingham
2	ASDA Binley	Coventry
3	ASDA Wilmslow	Manchester
4	ASDA Stratford	London
5	ASDA Bebington	Liverpool

Products_Data	
productID	productName
1	Bread
2	Biscuits
3	Coffee
4	Milk
5	Instant Noodles

Sales_Data			
storeID	productID	salesDate	revenue
2	4	November 02, 2020	1,233.32
1	1	November 02, 2020	5,434.74
2	3	November 02, 2020	3,855.96
2	3	November 02, 2020	2,280.90

1	3	November 02, 2020	2,110.95
2	4	November 01, 2020	4,558.24
2	4	November 01, 2020	6,849.99
1	1	November 01, 2020	2,543.57

The Stores_Data table stores data that relate to store information only while the Product_Data table stores data related to product information. The Sales_Data makes reference to both the Stores_Data and Products_Data tables using the IDs.
This in one potential way of designing and splitting the tables. This is to ensure the data about an entity fits into a single table. This allows table data to be more flexible to changes by splitting the information by entities into its own tables.

In Section 2.5 and 2.6 we briefly saw examples of joining tables using a match on the id columns for example:

```
SELECT s.studentID, subjectName, currentGrade
FROM STUDENTS AS s, SUBJECTS AS c
WHERE (gender = 'F')
        AND s.studentID = c.studentID;
```

In the example above we had to match the id columns from the students table and the subjects table in order to return a results set from both tables where the criteria match.

SQL Joins indicate connections between two or more tables and is a very useful way to connect tables and query them as a unit. There are many different ways to join tables:

- **Cross** Join (a.k.a cartesian join)
- **Inner** Join
- **Outer** Join
- **Natural** Join

The **Outer** Join can be split into sub-categories of:

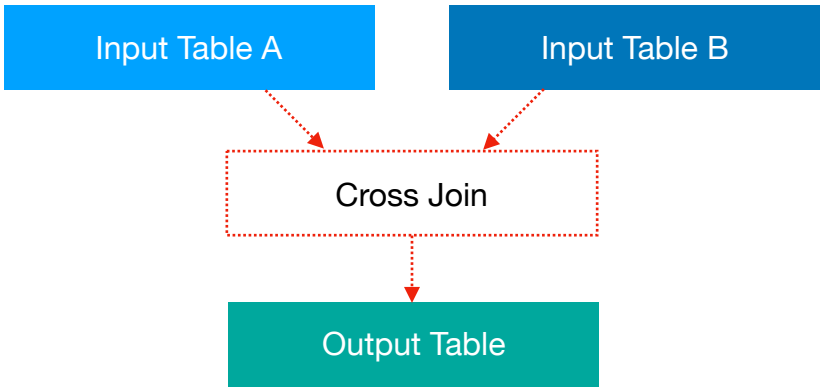
- **Left** Outer Join
- **Right** Outer Join
- **Full** Outer Join

These are all the different ways in which we can join tables and we will explore each joins and how they operate in the next couple of sections.

3.7 Cross Joins (Cartesian Joins)

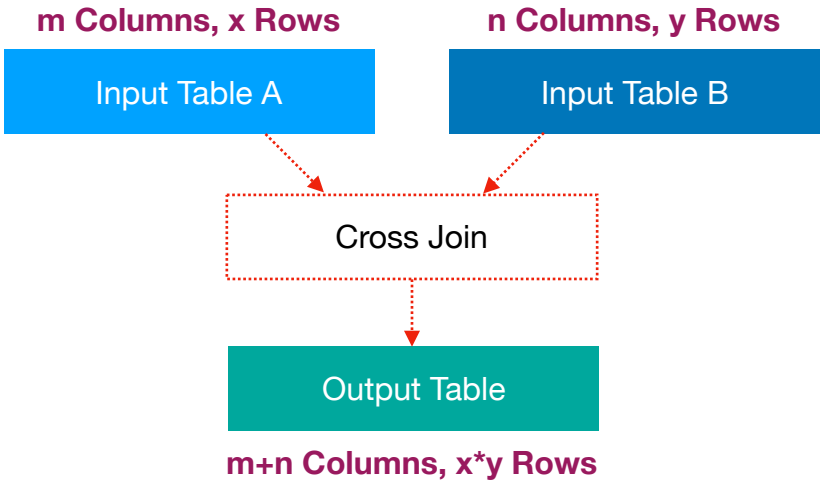
The Cross Join (Cartesian Join) is the most intuitive and easiest to understand join but is also a join that is not as widely/typically used. When joining tables we must always consider the input tables as well as the output tables. What do we mean by this?

Joins can be applied to two or more tables but for simplicity we will focus on joining two tables only.



The result of the join is also a table which we would call as the output table. The output table is not a real table that exists in the database. The input tables might exists in the database but the output table is a virtual table created by the join.

If Input Table A has m columns and x rows while the Input Table B has n columns and y rows. The output table from a cross join would have M+N columns and x*y rows.



The Cross Join will create an output table with 1 row for every combination of the input rows i.e. every row in Input Table A combines with every row in Input Table B. Therefore, for every combination of rows there is one row in the output table. Below is an example syntax using the stores and products table:

```
SELECT storeLocation, productName
FROM Stores_Data
CROSS JOIN Products_Data;
```

This query will return from the two tables a virtual output table results of the below. We can use this output table to query as though the table actually existed as a table in the database.

Stores_Data		
storeID	storeLocation	city
1	ASDA Small Heath	Birmingham
2	ASDA Binley	Coventry

Products_Data	
productID	productName
1	Bread
2	Biscuits
3	Coffee

The columns between the two tables are added together i.e. m + n columns.
The rows are all the possible combinations between the two table rows i.e. x*y rows.

CROSS_JOIN_OUTPUT_TABLE				
storeID	storeLocation	city	productID	productName
1	ASDA Small Heath	Birmingham	1	Bread
1	ASDA Small Heath	Birmingham	2	Biscuits
1	ASDA Small Heath	Birmingham	3	Coffee
2	ASDA Binley	Coventry	1	Bread
2	ASDA Binley	Coventry	2	Biscuits
2	ASDA Binley	Coventry	3	Coffee

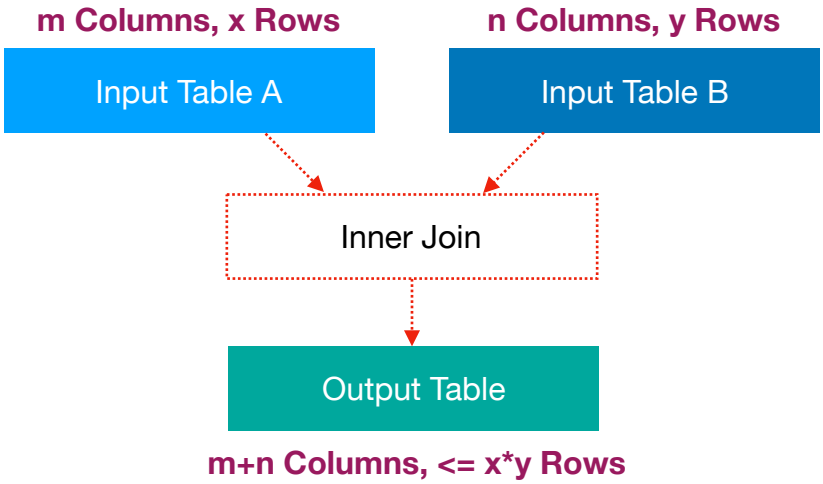
The **SELECT** statement within the query will return the storeLocation and the productName columns from this virtual output table as the final result table for the query as seen below:

RESULTS_TABLE	
storeLocation	productName
ASDA Small Heath	Bread
ASDA Small Heath	Biscuits
ASDA Small Heath	Coffee
ASDA Binley	Bread
ASDA Binley	Biscuits
ASDA Binley	Coffee

Cross Joins are very resource-intensive because it puts a tremendous strain on the database when performing a cross join. The database will require a lot of processing power and time to create the virtual output table.
For example, if we had an Input Table A with 1000 rows and an Input Table B with 1000 rows, this will create an Output Table of 1million rows.
Therefore, it is not typically used Cross Joins unless it is certain that the output tables are very small e.g. less than a 100 rows. Doing this type of join can put strain on the database and potentially bring down the system as a result.

3.8 Inner Joins

The Inner Join also has input tables and produces a virtual output table. The output table has all columns of both tables but the total rows is always less than or equal to the number of row combinations. We can see this represented in the diagram below.

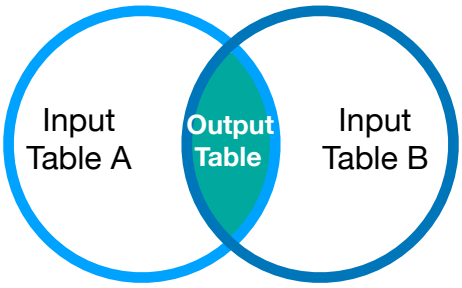


The Inner Join is logically equivalent to doing a Cross (Cartesian) Join and then applying a filter condition on the results of the Cross (Cartesian) Join. This is a logical equivalence and not what actually occurs under the hood when performing a Inner Join. This is the reason for why the output table of an Inner Join always has less than or equal to the number of rows in an output table of a Cross Join.

The filter condition almost always involves linking two columns of the two joined tables. If the condition tests for equality the join is called an Equi-Join (i.e. the value of a column in Table A is the same/equal to the value in the column of Table B).

When using an Inner Join, never actually write an Inner Join as a Cross Join with a filter. This is purely to logically visualise what an inner join is doing, furthermore, Cross Joins are very resource intensive. When using Inner Joins the DBMS performs some internal optimisation under the hood to not receive all the necessary columns which is why it is much faster than performing a Cross Join with a filter.

Below is the more commonly used diagram to represent an Inner Join. Where the inner parts of the two input tables columns match on the condition, this forms the data rows for the virtual output table of the Inner Join.



Below is an example Inner Join SQL query syntax joining two tables:

```
SELECT storeLocation, salesDate, revenue
FROM Stores_Data AS s
        INNER JOIN Sales_Data AS rev
        ON s.storeID = rev.storeID AND s.storeLocation = 'ASDA Binley';
```

The **SELECT** clause selects some columns from the output table while the **FROM** clause has a whole bunch of things going on to perform the Join to create the output table. First we specify in the **FROM** clause the first input table to involve in the Inner Join. The **INNER JOIN** keyword is used followed by the name of the second input table. This will indicate to the DBMS to perform an inner join using the two input tables specified in the SQL query. Finally, the keyword **ON** is used to specify the filter condition. The **INNER JOIN** keyword is always used with the **ON** keyword.

In the example above there are two filter conditions separated by the **AND** keyword. The first condition is an Equi-Join since it is linking the two columns from both input tables via an equality check. The second condition checks for the storeLocation from the Stores_Data is equal to ASDA Binley. This query will provide the following result:

Stores_Data		
storeID	storeLocation	city
1	ASDA Small Heath	Birmingham
2	ASDA Binley	Coventry

Sales_Data			
storeID	productID	salesDate	revenue
2	4	November 02, 2020	1,233.32
1	1	November 02, 2020	5,434.74
2	3	November 02, 2020	3,855.96
2	3	November 02, 2020	2,280.90

The columns between the two tables are added together i.e. m + n columns.
The rows are all the possible combinations between the two table rows i.e. x*y rows.
(Remember that the DBMS does not actually perform a Cross Join under the hood, but this allows us to logically visualise what an Inner Join does to grasp the concept).

CROSS_JOIN_OUTPUT_TABLE						
storeID	storeLocation	city	storeID	productID	salesDate	revenue
1	ASDA Small Heath	Birmingham	2	4	November 02, 2020	1,233.32
1	ASDA Small Heath	Birmingham	…	…	…	…
2	ASDA Binley	Coventry	2	4	November 02, 2020	1,233.32
2	ASDA Binley	Coventry	…	…	…	…

The rows are then filtered based on the condition i.e. <= x*y rows.

INNER_JOIN_OUTPUT_TABLE						
storeID	storeLocation	city	storeID	productID	salesDate	revenue
~~1~~	~~ASDA Small Heath~~	~~Birmingham~~	~~2~~	~~4~~	~~November 02, 2020~~	~~1,233.32~~
~~1~~	~~ASDA Small Heath~~	~~Birmingham~~	~~…~~	~~…~~	~~…~~	~~…~~
2	ASDA Binley	Coventry	2	4	November 02, 2020	1,233.32
~~2~~	~~ASDA Binley~~	~~Coventry~~	~~1~~	~~1~~	~~November 02, 2020~~	~~5,434.74~~
2	ASDA Binley	Coventry	2	3	November 02, 2020	3,855.96
2	ASDA Binley	Coventry	2	3	November 02, 2020	2,280.90

The **SELECT** statement within the query will return the storeLocation, salesDate and revenue columns from this virtual Inner Join output table as the final result table as seen below:

RESULTS_TABLE		
storeLocation	salesDate	revenue
ASDA Binley	November 02, 2020	1,233.32
ASDA Binley	November 02, 2020	3,855.96
ASDA Binley	November 02, 2020	2,280.90
ASDA Binley	…	…

3.9 Outer Joins (Left Outer Join)

To recap, a Join is a way of connecting two or more tables together so that we can query the data from separate tables as one single unit. The Outer Join has three types: Left Outer Join, Right Outer Join and the Full Outer Join.

We will explore each type of Outer Join separately to understand how each joins operate using two new table of Movies and Reviews for demonstration.

Movies_Data		
movieID	movieName	releaseYear
1	The Godfather	1972
2	The Departed	2006
3	Interstellar	2014
4	Parasite	2019
5	Tenet	2020

Reviews_Data		
movieID	userReview	ratingStars
1	Amazing	5
1	Ok, not great	3
3	Thought-Provoking	4
1	Classic	5
2	Genre-Defining	4
1	Overrated	1
4	Cinematic Masterpiece	5

The Left Outer Join syntax looks very similar to the Inner Join but behaves slightly differently. Below is an example syntax for a Left Outer Join:

```
SELECT m.movieID, r.userReview
FROM Movies_Data AS m
        LEFT OUTER JOIN Reviews_Data AS r
        ON m.movieID = r.movieID;
```

The two tables being connected are the Movies_Date (as the left table) and the Reviews_Data table (as the right table). The **LEFT OUTER JOIN** keyword indicates the type of join being performed while the **ON** keyword defines the filter criteria to apply to create the virtual Left Outer Join output table. The **SELECT** statements is used on the virtual output table to return the selected columns.

The **LEFT OUTER JOIN** takes all the rows from the left table and then finds/match every rows from the right table based on the condition criteria specified for the join. This operates exactly the same as the inner join.

LEFT_OUTER_JOIN_OUTPUT_TABLE					
movieID	movieName	releaseYear	movieID	userReview	ratingStars
1	The Godfather	1972	1	Amazing	5
1	The Godfather	1972	1	Ok, not great	3
1	The Godfather	1972	1	Classic	5
1	The Godfather	1972	1	Overrated	1

However, the difference between the **LEFT OUTER JOIN** and the **INNER JOIN** is apparent when no rows from the right table matches no rows from the left table based on the criteria set.

When no rows matches between the two tables, the **INNER JOIN** returns no data. The **LEFT OUTER JOIN**, however, returns only the data from the left table and no data from the right table, hence the name of Outer Join.

What does it mean by “no match exists”? We already know that “does not exist” equates to NULL and a NULL value indicates the absence of information and is not a specific value that is assigned within the table. Therefore, the **LEFT OUTER JOIN** will apply NULL values to the right table and will assume that such a row exists in the right table in order to perform a Outer Join. This can be seen demonstrated with movieID 5 which has no review information:

LEFT_OUTER_JOIN_OUTPUT_TABLE					
movieID	movieName	releaseYear	movieID	userReview	ratingStars
5	Tenet	2020	NULL	NULL	NULL

Therefore, the above example will create a output table containing all columns from both tables, all rows from the left table and matching rows from the right table. Where there are no matching rows from the right table it will assume NULL values to indicate the absence of information from the right table.

The **LEFT OUTER JOIN** virtual output table for a would look like the table below:

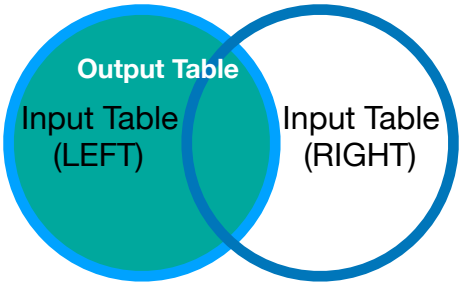
LEFT_OUTER_JOIN_OUTPUT_TABLE					
movieID	movieName	releaseYear	movieID	userReview	ratingStars
1	The Godfather	1972	1	Amazing	5
1	The Godfather	1972	1	Ok, not great	3
1	The Godfather	1972	1	Classic	5
1	The Godfather	1972	1	Overrated	1
2	The Departed	2006	2	Genre-Defining	4
3	Interstellar	2014	3	Thought-Provoking	4
4	Parasite	2019	4	Cinematic Masterpiece	5
5	Tenet	2020	NULL	NULL	NULL

The row for the movie Tenet with the review table values of NULL will never be present if the syntax was an **INNER JOIN**. This is the usefulness of a Outer Join whereby it will always return the outer table rows.

The **SELECT** statement within the query will return the movieID and userReview columns from this virtual Left Outer Join output table as the final result table as seen below:

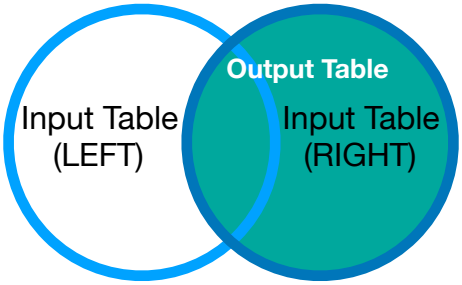
RESULTS_TABLE	
movieID	userReview
1	Amazing
1	Ok, not great
1	Classic
1	Overrated
2	Genre-Defining
3	Thought-Provoking
4	Cinematic Masterpiece
5	NULL

To conclude, the **LEFT OUTER JOIN** virtual output table is commonly represented using the diagram below. The virtual output table contains all the rows from the table left table and all the matching rows from the right table. Where there are no matching rows in the right table, the output table will use NULL values.



3.10 Outer Joins (Right Outer Join)

The Right Outer Join operates exactly the same as the Left Outer Join except that it works in the opposite i.e. a mirror image of the Left Outer Join. The virtual Right Outer Join output table includes all rows from the right table and matching rows from the left. Where there are no matching rows from the left table, the output table will use NULL values.




```
SELECT m.movieID, r.userReview
FROM Movies_Data AS m
        RIGHT OUTER JOIN Reviews_Data AS r
        ON m.movieID = r.movieID;
```

The example syntax above will provide the following output tables:

RIGHT_OUTER_JOIN_OUTPUT_TABLE					
movieID	movieName	releaseYear	movieID	userReview	ratingStars
1	The Godfather	1972	1	Amazing	5
1	The Godfather	1972	1	Ok, not great	3

3	Interstellar	2014	3	Thought-Provoking	4
1	The Godfather	1972	1	Classic	5
2	The Departed	2006	2	Genre-Defining	4
1	The Godfather	1972	1	Overrated	1
4	Parasite	2019	4	Cinematic Masterpiece	5

RESULTS_TABLE	
movieID	userReview
1	Amazing
1	Ok, not great
3	Thought-Provoking
1	Classic
2	Genre-Defining
1	Overrated
4	Cinematic Masterpiece

Notice how the movie Tenet is not present within the results table. This is because the **RIGHT OUTER JOIN** ensures all rows from the right table is present in the virtual Right Outer Join output table but does not require every rows from the left table to be present.

Lets imagine a scenario where the Movies_Data table (left table) no longer has Interstellar in its' table but the Reviews_Data has a review for this movieID.

Movies_Data		
movieID	movieName	releaseYear
1	The Godfather	1972
2	The Departed	2006
4	Parasite	2019
5	Tenet	2020

The **RIGHT OUTER JOIN** output table would assume NULL values in the left table where no match exists between the right and left tables as seen below.

RIGHT_OUTER_JOIN_OUTPUT_TABLE					
movieID	movieName	releaseYear	movieID	userReview	ratingStars
1	The Godfather	1972	1	Amazing	5
1	The Godfather	1972	1	Ok, not great	3
NULL	NULL	NULL	3	Thought-Provoking	4
1	The Godfather	1972	1	Classic	5
2	The Departed	2006	2	Genre-Defining	4
1	The Godfather	1972	1	Overrated	1
4	Parasite	2019	4	Cinematic Masterpiece	5

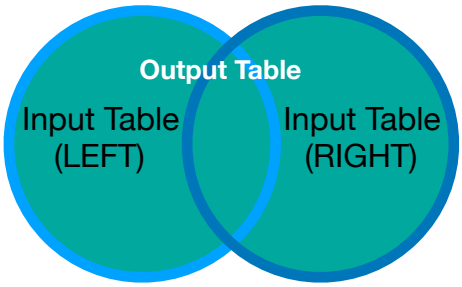
RESULTS_TABLE	
movieID	userReview
1	Amazing
1	Ok, not great
NULL	Thought-Provoking
1	Classic
2	Genre-Defining
1	Overrated
4	Cinematic Masterpiece

Important Note: typically if a movie does not exists it should not have any reviews for it as a system/database design. The above is to hypothetically demonstrate what would happen in a **RIGHT OUTER JOIN**, in such a case.

3.11 Outer Joins (Full Outer Join)

The Full Outer Join is a combination of the Left Outer Join and the Right Outer Join i.e. rows from both the left and right tables will be present in the output table with NULLs if needed.

This is typically represented using the below diagram:



To demonstrate the Full Outer Join, will use the scenario where the Movies_Data table no longer has Interstellar in its' table but the Reviews_Data has a review for this movieID. The syntax can be seen below:

```
SELECT m.movieID, r.userReview
FROM Movies_Data AS m
        FULL OUTER JOIN Reviews_Data AS r
        ON m.movieID = r.movieID;
```

The syntax is very similar to what we have seen before but using the **FULL OUTER JOIN** keyword instead. We can see the **FULL OUTER JOIN** returns all rows from both tables and adds NULL values where either table have no match.

FULL_OUTER_JOIN_OUTPUT_TABLE					
movieID	movieName	releaseYear	movieID	userReview	ratingStars
1	The Godfather	1972	1	Amazing	5
1	The Godfather	1972	1	Ok, not great	3
1	The Godfather	1972	1	Classic	5
1	The Godfather	1972	1	Overrated	1
2	The Departed	2006	2	Genre-Defining	4
NULL	NULL	NULL	3	Thought-Provoking	4
4	Parasite	2019	4	Cinematic Masterpiece	5
5	Tenet	2020	NULL	NULL	NULL

RESULTS_TABLE	
movieID	userReview
1	Amazing
1	Ok, not great
NULL	Thought-Provoking
1	Classic
2	Genre-Defining
1	Overrated
4	Cinematic Masterpiece
5	NULL

This is how the **FULL OUTER JOIN** behaves like both the **LEFT OUTER JOIN** as well as the **RIGHT OUTER JOIN**.

3.12 Natural Joins

A natural join is simply an Inner or Outer Join where two additional conditions are satisfied:

1. The two joined tables have a column with the same name
2. The join condition is an equality check on those two columns

Therefore, a Natural Join is a syntactical sugar i.e. shorthand for the Inner or Outer Join. It does not add any new functionality which we have not learned from the joins we have explored in the previous sections.

We can take the previous Left Outer Join and Right Outer Join examples and re-express them as a Natural Join.

Both the Movies_Data and Reviews_Data tables have a common column of movieID and when we perform a join we do an equality check for m.movieID = r.movieID. Therefore we can re-express the syntax using the Natural Join shorthand.

Left Outer Join

```
SELECT m.movieID, r.userReview
FROM Movies_Data AS m
        LEFT OUTER JOIN Reviews_Data AS r
        ON m.movieID = r.movieID;
```

Re-Expressed as a Natural Join

```
SELECT m.movieID, r.userReview
FROM Movies_Data AS m
        LEFT OUTER JOIN Reviews_Data AS r
```

Right Outer Join

```
SELECT m.movieID, r.userReview
FROM Movies_Data AS m
        RIGHT OUTER JOIN Reviews_Data AS r
        ON m.movieID = r.movieID;
```

Re-Expressed as a Natural Join

```
SELECT m.movieID, r.userReview
FROM Movies_Data AS m
        RIGHT OUTER JOIN Reviews_Data AS r
```

We use the **NATURAL LEFT OUTER JOIN** or **NATURAL RIGHT OUTER JOIN** keywords and remove the equality check entirely from the syntax because the equality check is implicitly implied. This saves us the effort of specifying the match condition.

3.13 Self Joins

Any joins (whether it is a Inner Join, Outer Join or Natural Join) where the same table is used as both the left input table and the right input table i.e. a table joining on itself is called a Self Join. This is completely valid and allowed in SQL.

We have now explored every type of Joins available in SQL and fully understand how each joins operates and what data is returned as the virtual output tables i.e. the logical unit to query on.

3.14 Having Clause

The **HAVING** operator tests whether sub-groups satisfy a given condition. Any sub-groups that does not satisfy the condition are eliminated from the output query result. The **HAVING** operator is similar to the **WHERE** clause i.e. both operators are followed by a

condition but the difference between the two is that the **WHERE** clause operates on individual rows while the **HAVING** clause operates on sub-groups created by a **GROUP BY** clause or an aggregate operator.

The **HAVING** operator must always be used along with aggregate operators (it does not necessarily need to be with the **GROUP BY** clause). Aggregate operators such as **SUM**, **MIN**, **MAX**, **AVG**, **COUNT**, etc. create sub-groups of which the **HAVING** operator can work on. In many cases the **HAVING** operator is usually seen accompanied by the **GROUP BY** clause.

Remember: The only time a **HAVING** operator can be used without a **GROUP BY** clause is when the aggregate operator acts for the entire table.

We will use the Movies_Data and Reviews_Data to demonstrate using the **HAVING** operator to answer a question on the dataset.

Movies_Data

movieID	movieName	releaseYear
1	The Godfather	1972
2	The Departed	2006
3	Interstellar	2014
4	Parasite	2019
5	Tenet	2020

Reviews_Data

movieID	revieID	userReview	ratingStars
1	1	Amazing	5
1	2	Ok, not great	3
3	3	Thought-Provoking	4
1	4	Classic	5
2	5	Genre-Defining	4
1	6	Overrated	1
4	7	Cinematic Masterpiece	5

The below syntax answers the following question: What is the average ratings of all movies having at least 2 or more reviews?

We will break down the syntax step by step to fully understand what each statement within the query is actually doing with the dataset to return the final results table.

```
SELECT m.movieName, AVG(r.ratingStars)
FROM Movies_Data AS m
        INNER JOIN Reviews_Data AS r
        ON m.movieID = r.movieID
GROUP BY movieName
HAVING COUNT(reviewID) >= 2;
```

Firstly, the **INNER JOIN** creates a virtual output table by performing a cross join on the Movies_Data and Reviews_Data tables and then filtering on the condition.
The **GROUP BY** operator then creates an aggregate row for each movieName on the virtual output table i.e. one group per unique movie.

INNER_JOIN_OUTPUT_TABLE						
movieID	movieName	releaseYear	movieID	revieID	userReview	ratingStars
1	The Godfather	1972	1	1	Amazing	5
1	The Godfather	1972	1	2	Ok, not great	3
1	The Godfather	1972	1	4	Classic	5
1	The Godfather	1972	1	6	Overrated	1
2	The Departed	2006	2	5	Genre-Defining	4
3	Interstellar	2014	3	3	Thought-Provoking	4
4	Parasite	2019	4	7	Cinematic Masterpiece	5

Once we have these sub-groups, the **HAVING** operator can now start working on its' condition. The **HAVING** operator takes in a filter condition just like the **WHERE** clause. In this example the condition is **COUNT(**reviewID**)** >= 2.
The **HAVING** operator will check each sub-group against the filter criteria and eliminate all sub-groups that do not satisfy the condition.

The **COUNT** operator will perform an aggregate on the reviewID column to count the number of reviews within each sub-group. This count is then used to check whether it is greater than or equal to 2. The **HAVING** operator will use this for filtering the results.

As we can see in the Inner Join virtual output table above, The Godfather has a review count of 4 reviews and will remain in the final results table. Interstellar, Parasite and The Departed all have a review count of 1 review. They all fail the **HAVING** clause's filter

condition and will therefore be eliminated from the results table. This leaves us with the Inner Join virtual output table as seen below:

INNER_JOIN_OUTPUT_TABLE	
movieName	ratingStars
The Godfather	5
The Godfather	3
The Godfather	5
The Godfather	1

Finally, the **SELECT** statement is used to return the columns from the virtual output table. The movieName column is returned along with the ratingStars column which has an aggregate operator used on this column to return the average rating. The **AVG** operator is used to calculate the average rating for The Godfather which is 3.5 (5 + 3 + 5 + 1 = 14 / 4 = 3.5). The example SQL query returns the final results table as seen below:

RESULTS_TABLE	
movieName	ratingStars
The Godfather	3.5

Important Note: We cannot use both the **WHERE** and **HAVING** operators within the same query. Therefore, if we have a **HAVING** operator within the query we must not use the **WHERE** clause and vice versa. This is because both use a filter condition to filter out the final results table. The **WHERE** clause should be used when applying the filtering condition on each individual rows while the **HAVING** clause should be used when applying the filtering condition on each aggregated sub-groups.

END