ADVANCED SQL JOINS

Thinking in Sets

Scott Peters
https://advancedsqlpuzzles.com

Last Updated: 05/12/2022

The SQL statements and diagrams from this document can be found in following Git repository.
https://github.com/smpetersgithub/AdvancedSQLPuzzles/tree/main/Database%20Writings

The examples are based on Microsoft's SQL Server. The provided SQL statements can be modified easily to fit any dialect of SQL.

I welcome any corrections, new tricks, new techniques, dead links, misspellings, bugs, and especially any new puzzles that would be a great fit for this document.

Please contact me through the contact page on my website.

The most essential concept of understanding SQL is how joins operate. Any SQL statement that combines rows from two or more tables, views, table functions, must make use of a join. The ANSI-standard specifies three types of joins:

- INNER JOIN
- OUTER JOIN
- CROSS JOIN

Even before we begin looking into these types of joins, it is best to understand table operators and how they interact with the different logical query-processing phases. SQL does not process the query in the order in which it is written, as the SELECT statement is processed almost last.

The correct processing order is:

1) FROM
2) WHERE
3) GROUP BY
4) HAVING
5) SELECT
6) ORDER BY

Once a query first enters the FROM statement, there are four types of table operators that can be performed, and each of these operators has a series of subphases.

1) JOIN
2) APPLY
3) PIVOT
4) UNPIVOT

The APPLY operator might be unfamiliar to some people, as this operator is used when you want to join a table to a table-valued function. For more information on the APPLY, PVIOT, and UNPIVOT operators, a quick internet search will fill in the gaps. Our focus will be on the JOIN operator.

The sub-phases involved in the join operator are:

1) Create a cartesian product
2) Process the ON predicate logic

3) Add the outer rows if an OUTER JOIN was specified

And then repeat for each table specified in the FROM clause.

I have also included this diagram in Appendix A to provide a larger view. You can also download this from the provided Git repository.

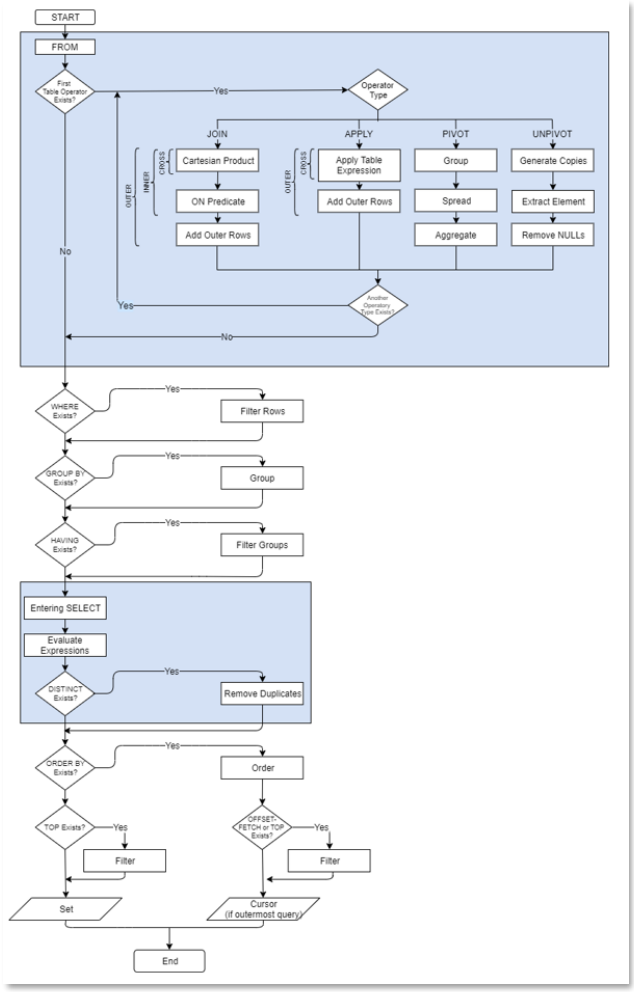

Figure 1

From this diagram we must take note that there is only one true type of table join, the cartesian product. INNER and OUTER JOINS should be considered restricted cartesian products, where the ON predicate specifies the restriction. If you are unfamiliar with a cartesian product, a quick internet search will fill in the gaps.

The below query is an example of a restricted cartesian product, where the restriction is the value in the Customer ID field must match.

```
SELECT *
FROM   Customers emp INNER JOIN
       Orders ord ON emp.CustomerID = ord.CustomerID;
```

If we modify the above query to use an OUTER JOIN, the query performs an additional step by adding the outer rows.

The biggest difference between INNER and OUTER JOINS, is that for INNER joins the join logic acts as a filter where both columns need to equate to each other, thus restricting the output to only the records that have these matching column values. For OUTER JOINS, the join logic acts as a matching criterion, where the record count output is not affected by this join logic as it will display the results from the preserved table. For the OUTER JOINS, additional logic is needed in the WHERE clause to add filtering logic to the result set.

For INNER and OUTER JOINS, these types of joins require a comparison operator to equate rows from the participating tables based on a common field in both the tables. These comparison operators are described as equi-joins or non-equi joins (sometimes referred to as theta joins).

Equi-joins look for equality between the joining columns in the tables, and non-equi joins look for anything other than equality between the joining columns, such as greater than or inequality.

Many of these types of joins are rooted in relational algebra. Introduced by Edgar F. Codd in 1970, relational algebra uses algebraic structures with a well-founded semantics for modeling data and defining queries on it. If you are unfamiliar with relational algebra and Edgar F. Codd, I recommend picking up a college level textbook on the subject.

There are also set operators. The difference between a join and set operation, while both concepts can be used to combine data from multiple tables, a join combines columns from separate tables which are often of different types, whereas set operations combine rows from separate relations, which are of the same type. When defining type, we must think of similarity between datasets. A table of customers and a table of orders are of two different types, as one consists of customer information and the other consists of order information. If we were to compare relations containing only the Customer ID field from the Customers table to only the Customer ID of the Orders table, then we have identical types, Customer ID types.

The different set operators are:

1) UNION
2) UNION ALL
3) INTERSECT
4) MINUS

We will discuss set operators more in a bit.

Lastly, there are also join algorithms. SQL is a declarative language; we tell the database what to do, not how to do it. When an SQL statement is executed, a query optimizer will try and retrieve the data in the most efficient way possible. During this query processing, a join algorithm is chosen based upon the available indexes, statistics, and size of the tables (or views, table valued function, etc.…).

Three fundamental algorithms exist for performing a join operation:
1) Nested loop join
2) Sort-merge
3) Hash join

The **nested loop join** is a logical structure in which one loop resides inside another loop. This join typically occurs when one table is smaller than the other table by a considerable size. The larger this difference between the number of rows in the inputs, the more benefit this operator will provide. The inner loop executes for each outer row and searches for matching rows in the inner input table.

The **merge join** requires both the inputs to be sorted on keys and requires a minimum of one equality expression between the matching columns. The algorithm reads a row for each input and compares them using a join key. When a match is detected, both rows in the sets are returned. If a match is not detected, the row with the smaller value is discarded. Because there is a sort key, the algorithm can efficiently transverse the input tables.

A **hash join** is used when large tables are joined, often where an index is not available. The hash join builds a hash table in memory and then scans for matches. It is the least efficient of the joins.

Nested loops are the only algorithm that supports non-equi joins. Merge and hast joins require an equi-join predicate in the queries logic.

JOIN DIAGRAMS

Traditionally, Venn diagrams are used to show a visual explanation of SQL joins. Joins are relatively easy to understand and using Venn diagrams works for pedagogical reasons. But using Venn diagrams to describe joins is using a wrong comparison, as Venn diagrams are intended to show set operations and not join operations. A good example of the limitation in Venn diagrams is that it is not able to show the CROSS JOIN properly.

Although not as user-friendly as Venn diagrams, join diagrams are a more proper way of demonstrating joins. Here we have a simple join diagram showing the CROSS, INNER, LEFT, and FULL JOIN. I have purposely left out the RIGHT JOIN, as it is simply a reversal of the LEFT JOIN.

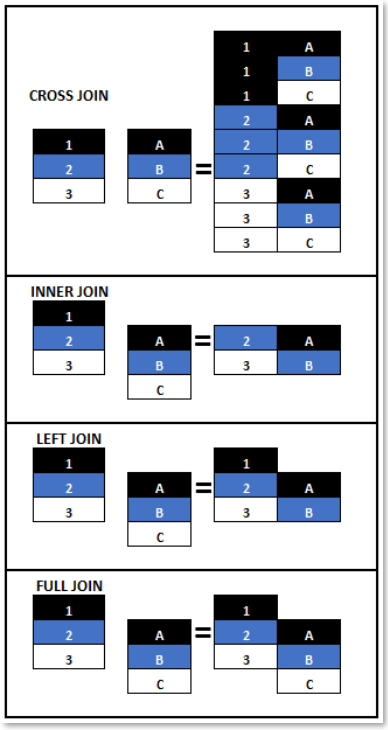

Figure 2

Now that we understand the different joins and their processing order, let's move into some specific examples.

In this demonstration, I quickly move through the different types of joins and give an overview of each. I hope this document works as a quick reminder in the nuances of join theory, and I try to keep things on the advanced level by adding NULL markers to the example tables.

Given the following tables, determine the output of each SQL statement.

For the following examples you will need to understand the behavior of NULL markers. I've provided some rather simple tables with minimal records. In some statements, the example data may not be sufficient, and in these cases feel free to add your own data and experiment.

Here is the example data will be using. As we will discover in the following statements, NULL markers are neither equal nor not equal to each other, they are unknown.

The DDL statements to create these tables can be found in the GitHub repository located here.

Table A

ID	Fruit	Quantity
1	Apple	17
2	Peach	20
3	Mango	11
4	<NULL>	5

Table B

ID	Fruit	Quantity
1	Apple	17
2	Peach	25
3	Kiwi	20
4	<NULL>	<NULL>

INNER JOINS

The INNER JOIN selects records from two tables given a join condition. This type of join requires a comparison operator to combine rows from the participating tables based on a common field(s) in both the tables. This comparison operator acts as a filter.

We can use both equi-join and non-equi join operators between the joining fields, as we will demonstrate below.

To start, here is the most common join you will use, an INNER JOIN between two tables.

A join between two tables is called a binary relationship. In this example, we use an equi-join as we are looking for equality between the two fields. Note the output does not contain any NULL markers, as NULL markers are neither equal to nor not equal to each other, they are unknown. The below statement represents the SQL-89 syntax.

```
--1.1
SELECT a.Fruit,
       b.Fruit
FROM   ##TableA a INNER JOIN
       ##TableB b ON a.Fruit = b.Fruit;
```

Fruit	Fruit
Apple	Apple
Peach	Peach

We can also specific the matching criteria in the WHERE clause with the SQL-89 syntax.

```
--1.2
SELECT a.Fruit,
       b.Fruit
FROM   ##TableA a,
       ##TableB b
WHERE  a.Fruit = b.Fruit;
```

Fruit	Fruit
Apple	Apple
Peach	Peach

Remembering that all types of joins are restricted cartesian products, the following query produces the same results as 1.1 and 1.2.

```
--1.3
SELECT a.Fruit,
       b.Fruit
FROM   ##TableA a CROSS JOIN
       ##TableB b
WHERE  a.Fruit = b.Fruit;
```

Fruit	Fruit
Apple	Apple
Peach	Peach

This next statement incorporates an INNER JOIN with a non-equi join, which looks for inequality between two fields. Normally you will combine a non-equi join with an equi-join when creating predicate logic (although not always the case). This statement uses both an equi-join and a non-equi join to find fruits which exist in both tables but have different quantities.

```
--1.4
SELECT a.Fruit,
       b.Fruit
FROM   ##TableA a INNER JOIN
       ##TableB b ON a.Fruit = b.Fruit AND a.Quantity <> b.Quantity;
```

Fruit	Fruit
Peach	Peach

This query uses both an equi-join and a non-equi join, and functions similar to a CROSS JOIN, but because we have NULL markers in the table, they are eradicated as NULL markers are neither equal nor not equal to each other, they are unknown.

```
--1.5
SELECT a.Fruit,
       b.Fruit
FROM   ##TableA a INNER JOIN
       ##TableB b ON a.Fruit <> b.Fruit OR a.Fruit = b.Fruit;
```

Fruit	Fruit
Apple	Apple
Peach	Apple
Mango	Apple
Apple	Peach
Peach	Peach
Mango	Peach
Apple	Kiwi
Peach	Kiwi
Mango	Kiwi

A CROSS JOIN creates all possible permutations of the two tables and will include the NULL markers.

```
--1.6
SELECT a.Fruit,
       b.Fruit
FROM   ##TableA a CROSS JOIN
       ##TableB b;
```

Fruit	Fruit
Apple	Apple
Peach	Apple
Mango	Apple
<NULL>	Apple
Apple	Peach
Peach	Peach
Mango	Peach
<NULL>	Peach
Apple	Kiwi
Peach	Kiwi
Mango	Kiwi
<NULL>	Kiwi
Apple	<NULL>
Peach	<NULL>

Mango	<NULL>
<NULL>	<NULL>

Here are some other examples of INNER JOINS using non-equi joins.

```
--1.7
SELECT *
FROM   ##TableA a INNER JOIN
       ##TableB b ON a.Quantity >= b.Quantity;
```

ID	Fruit	Quantity	ID	Fruit	Quantity
1	Apple	17	1	Apple	17
2	Peach	20	1	Apple	17
2	Peach	20	3	Kiwi	20

```
--1.8
SELECT *
FROM   ##TableA a INNER JOIN
       ##TableB b ON a.Fruit = b.Fruit AND a.Quantity BETWEEN 10 AND 20;
```

ID	Fruit	Quantity	ID	Fruit	Quantity
1	Apple	17	1	Apple	17
2	Peach	20	2	Peach	25

Functions can be used in the join condition as well. Assigning the empty string to a NULL value via the ISNULL function causes the NULLs to now equate to each other. Note, the empty string does not have an ASCII value assigned to it. As demonstrated below, the CHAR function will return an empty set.

```
--1.9
SELECT a.Fruit,
       b.Fruit
FROM   ##TableA a INNER JOIN
       ##TableB b ON ISNULL(a.Fruit,'') = ISNULL(b.Fruit,'');
```

Fruit	Fruit
Apple	Apple
Peach	Peach
<NULL>	<NULL>

OUTER JOINS

OUTER JOINS consist of LEFT, RIGHT, and FULL OUTER JOIN.

LEFT OUTER JOIN and RIGHT OUTER JOIN function the same; the LEFT OUTER JOIN returns all records from the left table, and the RIGHT OUTER JOIN returns all records from the right table.

It is best practice to use the LEFT OUTER JOIN over the RIGHT OUTER JOIN, as we naturally read from left to right. I will only demonstrate the LEFT OUTER JOIN for this reason.

LEFT OUTER JOIN

The most widely used case for the LEFT OUTER JOIN is when we want all values in Table A, regardless of their presence in Table B. A join condition in an OUTER JOIN acts as a matching criterion and not as a filtering mechanism.

```
--2.1
SELECT a.Fruit,
       b.Fruit
FROM   ##TableA a LEFT OUTER JOIN
       ##TableB b ON a.Fruit = b.Fruit;
```

Fruit	Fruit
Apple	Apple
Peach	Peach
Mango	<NULL>
<NULL>	<NULL>

A LEFT OUTER JOIN is one of several methods to determine records in Table A that do not exist in Table B. The following statement returns all values in Table A that do not exist in Table B. There are other (and possibly better) ways of writing this query which we will cover later.

```
--2.2
SELECT a.Fruit
FROM   ##TableA a LEFT OUTER JOIN
       ##TableB b ON a.Fruit = b.Fruit
WHERE  b.Fruit IS NULL;
```

Fruit
Mango
<NULL>

Predicate logic placed in the ON clause behaves differently than predicate logic in the WHERE clause. Notice the difference in output between the following queries.

```
--2.3
SELECT a.Fruit,
       b.Fruit
FROM   ##TableA a LEFT OUTER JOIN
       ##TableB b ON a.Fruit = b.Fruit AND b.Fruit = 'Apple';
```

Fruit	Fruit
Apple	Apple
Peach	<NULL>
Mango	<NULL>
<NULL>	<NULL>

Compare the following output to the previous query where the predicate logic was placed in the ON clause. Placing a predicate on the outer joined table in the WHERE clause causes this to function as an INNER JOIN.

Commented [SP1]: Appears I am missing a query here

```
--2.4
SELECT a.Fruit,
       b.Fruit
FROM   ##TableA a LEFT OUTER JOIN
       ##TableB b ON a.Fruit = b.Fruit
WHERE  b.Fruit = 'Apple';
```

Fruit	Fruit
Apple	Apple

You can also have nested SELECT statements which act as OUTER JOINS, as demonstrated below. The result of this query is the same as 2.1.

```
--2.5
SELECT  a.Fruit,
        (SELECT b.Fruit FROM ##TableB b WHERE a.Fruit = b.Fruit) AS Fruit
FROM    ##TableA a;
```

FULL OUTER JOIN

The FULL OUTER JOIN is an often-underutilized join as it has a more specific use case than the other joins. This join is best used to compare two similar tables as shown below. Remember to use this type of join when you want to compare two shopping baskets.

The following shows the contents of fruits in both Table A and Table B.

```
--3.1
SELECT a.Fruit,
       b.Fruit
FROM   ##TableA a FULL OUTER JOIN
       ##TableB b ON a.Fruit = b.Fruit;
```

Fruit	Fruit
Apple	Apple
Peach	Peach
Mango	<NULL>
<NULL>	<NULL>
<NULL>	Kiwi
<NULL>	<NULL>

This FULL OUTER JOIN will list the items that do not exist in either of the two tables. The results are Mango and Kiwi, along with two NULL records, as NULL markers are not equal.

```
--3.2
SELECT a.Fruit,
       b.Fruit
FROM   ##TableA a FULL OUTER JOIN
       ##TableB b ON a.Fruit = b.Fruit
WHERE  a.Fruit IS NULL OR b.Fruit IS NULL;
```

Fruit	Fruit
Mango	<NULL>
<NULL>	<NULL>
<NULL>	Kiwi
<NULL>	<NULL>

NATURAL JOIN

Some database systems support the NATURAL JOIN construct. It is simplified way of specifying a join where two tables have commonly named column names. SQL Server currently does not support natural joins.

Here are some examples of the NATURAL JOIN syntax from the Oracle documentation. The benefit of natural joins is that the output does not contain duplicate fields as the commonly named columns between the tables will be condensed into one field.

```
--4.1
SELECT *
FROM   Countries NATURAL JOIN
       Cities;

--4.2
SELECT *
FROM   Countries NATURAL JOIN Cities
USING  (Country, Country_ISO_Code);

--4.3
SELECT *
FROM   Countries NATURAL LEFT JOIN
       Cities;
```

CROSS JOINS

A CROSS JOIN creates all permutations (i.e., a cartesian product) of the two joining tables. It will produce a result set which is the number of rows in the first table multiplied by the number of rows in the second table.

Here is a simplest form of the CROSS JOIN.

```
--5.1
SELECT a.Fruit,
       b.Fruit
FROM   ##TableA a CROSS JOIN
       ##TableB b;
```

Fruit	Fruit
Apple	Apple
Peach	Apple
Mango	Apple
<NULL>	Apple
Apple	Peach
Peach	Peach
Mango	Peach
<NULL>	Peach
Apple	Kiwi
Peach	Kiwi
Mango	Kiwi
<NULL>	Kiwi
Apple	<NULL>

Peach	<NULL>
Mango	<NULL>
<NULL>	<NULL>

Placing the join condition in the WHERE clause of a CROSS JOIN creates an INNER JOIN. Functions such as a CASE statement can be incorporated into the predicate logic as well.

```
--5.2
SELECT a.Fruit,
       b.Fruit
FROM   ##TableA a CROSS JOIN
       ##TableB b
WHERE  (CASE a.Fruit WHEN 'Orange' THEN 'Tangerine' ELSE a.Fruit END) = b.Fruit;
```

F+ruit	Fruit
Apple	Apple
Peach	Peach

You can also add complicated join logic like the following.

```
--5.3
SELECT a.Fruit,
       b.Fruit
FROM   ##TableA a,
       ##TableB b
WHERE  (CASE WHEN a.Quantity > 15 THEN a.Quantity ELSE a.ID END) = b.Quantity;
```

Fruit	Fruit
Apple	Apple
Peach	Kiwi

SEMI AND ANTI JOINS

An **anti-join** uses the NOT IN or NOT EXISTS operators. The **semi join** uses the IN or EXISTS operators.

The benefits of semi and anti-joins are they remove the risk of returning duplicate rows. The result set can only contain the columns from the outer semi-joined table. They are also used for readability, as no fields in the joined table can become part of the queries projection (i.e., the SELECT statement).

Because NULLS are a three valued system (true, false, or unknown), this statement returns an empty dataset as it cannot properly determine if the values in Table A are not in Table B.

```
--6.1
SELECT Fruit
FROM   ##TableA
WHERE  Fruit NOT IN (SELECT Fruit FROM ##TableB);
```

Fruit

<Empty Data Set>

Adding an ISNULL function to the inner query is one way to alleviate the issue of NULL markers.

```
--6.2
SELECT Fruit
FROM   ##TableA
WHERE  Fruit NOT IN (SELECT ISNULL(Fruit,'') FROM ##TableB);
```

Fruit
Mango

You can also place predicate logic in the WHERE clause to eradicate the NULL markers.

```
--6.3
SELECT Fruit
FROM   ##TableA
WHERE  Fruit NOT IN (SELECT Fruit FROM ##TableB WHERE Fruit IS NOT NULL);
```

Fruit
Mango

The opposite of anti-joins are semi-joins. Using the IN operator, this query will return results and we do not need any special logic to handle NULL markers as we can determine which values are in Table B.

```
--6.4
SELECT Fruit
FROM   ##TableA
WHERE  Fruit IN (SELECT Fruit FROM ##TableB);
```

Fruit
Apple
Peach

The IN and NOT IN operators can also take a hard coded list of values as its input. For this example, we use the IN operator. Even though we include a NULL marker in the inner query, the results do not include a NULL marker.

```
--6.5
SELECT Fruit
FROM   ##TableA
WHERE  Fruit IN ('Apple','Kiwi',NULL);
```

Fruit
Apple

EXISTS is much the same as IN. But with EXISTS you must specify a query and you can specify multiple join conditions.

```
--6.6
SELECT Fruit
FROM   ##TableA a
WHERE  EXISTS (SELECT 1 FROM ##TableB b WHERE a.Fruit = b.Fruit AND
                                              a.Quantity = b.Quantity);
```

Fruit
Apple

Here is the usage of the NOT EXISTS.

```
--6.6
SELECT Fruit
FROM   ##TableA a
WHERE  NOT EXISTS (SELECT 1 FROM ##TableB b WHERE a.Fruit = b.Fruit AND
                                              a.Quantity = b.Quantity);
```

Fruit
Peach
Mango
<NULL>

BEHAVIOR OF NULLS

Up to this point we have used temporary tables to arrive at our conclusions concerning NULL markers. However, we can simply forgo the tables and simply use predicate logic to determine how NULL markers react to the different SQL constructs.

The following SQL statements equate to UNKNOWN and result an empty set.

```
--7.1
SELECT 1 WHERE NULL = NULL;
SELECT 1 WHERE 1 = NULL;
SELECT 1 WHERE NULL <> NULL;
SELECT 1 WHERE NULL = NULL;
SELECT 1 WHERE 1 = NULL;
SELECT 1 WHERE NULL <> NULL;
SELECT 1 WHERE 1 <> NULL;
SELECT 1 WHERE NULL > NULL;
SELECT 1 WHERE 1 > NULL;
```

The following SQL statements equate to TRUE.

```
--7.2
SELECT 1 WHERE NULL IS NULL;
SELECT 1 WHERE 1 IS NOT NULL;
SELECT 1 WHERE 1 = 1;
```

The following SQL statements equate to FALSE.

```
--7.3
SELECT 1 WHERE 1 IS NULL;
SELECT 1 WHERE NULL IS NOT NULL;
SELECT 1 WHERE 1 <> 1;
```

VALUES KEYWORD

Specific to SQL Server, we can also use the VALUES constructor to easily create data. It's not often used in this manner, but it is worth noting the use.

The VALUES constructor specifies a set of row value expressions to be constructed into a table and allows multiple rows of data to be specified in a single DML statement. Normally we use the VALUES constructor to specify the data to insert into a table, as we initially did with our test data, but it can also be used as a derived table in an SQL statement.

Here is a basic example of using the VALUES constructor.

```
--8.1
SELECT a, b
FROM   (VALUES (1, 2), (3, 4), (5, 6), (7, 8), (9, 10)) AS MyTable(a, b);
```

a	b

1	2
3	4
5	6
7	8
9	10

Here is a more elaborate example where the VALUES constructor specifies the values to return.

```
--8.2
SELECT a.Fruit
FROM   ##TableA a INNER JOIN
       (VALUES ('Kiwi'), ('Apple')) AS b(Fruit) ON a.Fruit = b.Fruit;
```

Fruit
Apple

You can also place functions into the VALUES constructor.

```
--8.3
SELECT a.Fruit,
       b.ID
FROM   ##TableA a CROSS JOIN
       (VALUES (NEWID())) AS b(ID);
```

Fruit	New_ID
Apple	72A95D25-1224-405E-8879-295C2DD0891A
Peach	3767AF47-17B1-4D92-9FBA-632E07A7D10D
Mango	60B37D3D-2FB4-4F24-B486-2CBCD2C09A6B
<NULL>	6466B482-C607-497D-A355-0A9A97E61711

SET OPERATORS

Now let's move our attention away from join theory and focus on set operators. The difference between a join and set operation, is that although both concepts are used to combine data from multiple tables, a join combines columns from separate relations, whereas set operations combine rows.

Earlier we showed that Venn diagrams are good for set theory, but not optimal for join theory. Venn diagrams are for showing differences between sets, which in traditional set theory there are 5 basic operations, which are depicted in the following figure.

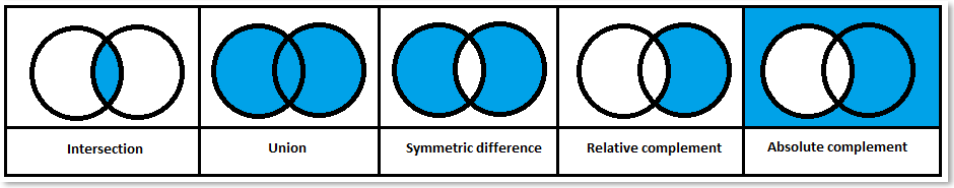

Figure 3

SQL only supports the intersection, union, and relative complement in set theory (with the constructs INTERSECT, UNION, and MINUS/DIFFERENCE). SQL does not have constructs for the symmetric difference or the absolute complement. Also, none of these properly show a LEFT JOIN or a RIGHT JOIN, which is another example of how Venn diagrams should be used for set and not join theory.

The SQL standard for SET OPERATORS does not use the term *EQUAL TO* or *NOT EQUAL TO* when describing their behavior. Instead, it uses the terminology of *IS [NOT] DISTINCT FROM* and the following expressions are TRUE when using SET OPERATORS.

- NULL is not distinct from NULL
- NULL is distinct from "Apples".

The phrase "Null is not distinct from NULL" may seem difficult to understand at first, but this simply means that NULLS are treated as equalities in the context of SET OPERATORS. In the following examples, we see the SET OPERATORS treat the NULL markers differently than the JOIN syntax.

Here are some examples of set operators in SQL and how they interact with the NULL markers in our example tables.

UNION will return all rows without duplication. Here the UNION interprets the NULLS as similar and combines the record into one. The UNION operator demonstrates that NULL is not distinct from NULL.

```
--9.1
SELECT Fruit FROM ##TableA
UNION
SELECT Fruit FROM ##TableB;
```

Fruit
<NULL>
Apple
Kiwi
Mango
Peach

The UNION ALL operator returns all values including each NULL marker.

```
--9.2
SELECT Fruit FROM ##TableA
UNION ALL
SELECT Fruit FROM ##TableB;
```

Fruit
Apple
Peach
Mango
<NULL>
Apple
Peach
Kiwi
<NULL>

INTERSECT will return matching rows and the NULL marker is included in the output.

```
--9.3
SELECT Fruit FROM ##TableA
INTERSECT
SELECT Fruit FROM ##TableB;
```

Fruit
<NULL>
Apple
Peach

EXCEPT will return the records in the table in the first statement that do not exist in the second statement below it. Some SQL languages use MINUS or DIFFERENCE instead of EXCEPT.

```
--9.4
SELECT Fruit FROM ##TableA
EXCEPT
SELECT Fruit FROM ##TableB;
```

Fruit
Mango

SELF JOINS

Now that we have discussed both joins and set operations (and how they interact with NULL markers) with our example tables, let's move into more specific examples. We will start with self joins, which are joins where the table is joined to itself. Self joins are also called unary relationships.

Example 1

The most common example used to demonstrate a self-join is the Manager and Employee hierarchical relationship. In this relationship, the table contains both a foreign key (Manager ID), which relates to its primary key (Employee ID).

Here is an example of such a hierarchy.

Employee ID	Name	Title	Manager ID
1	Ramirez	President	<NULL>
2	Baers	Vice President	1
3	Santana	Vice President	1
4	Perkins	Director	2
5	Mercer	Director	3

We can use the following self-join to determine each employee's manager.

```
--10.1
SELECT a.EmployeeID,
       a.Name,
       a.Title ,
       a.ManagerID,
       b.Name AS ManagerName,
       b.Title
FROM   #Employees a INNER JOIN
       #Employees b ON a.ManagerID = b.EmployeeID;
```

Employee ID	Name	Title	Manager ID	Manager Name	Title
2	Baers	Vice President	1	Ramirez	President
3	Santana	Vice President	1	Ramirez	President
4	Perkins	Director	2	Baers	Vice President
5	Mercer	Director	3	Santana	Vice President

It is worth mentioning that the above problem lends itself to using a self-referencing common table expression to determine the level of depth each employee has from the highest tier.

In the below example, the common table expression cte_Recursion references itself in the UNION ALL statement.

```
--10.2
WITH cte_Recursion AS
(
SELECT EmployeeID,
       Name,
       Title,
       ManagerID,
       0 AS Depth
FROM   #Employees
WHERE  ManagerID IS NULL
UNION ALL
SELECT b.EmployeeID,
       b.Name,
       b.Title,
       b.ManagerID,
       a.Depth + 1 AS Depth
FROM   cte_Recursion a INNER JOIN
       #Employees b on a.EmployeeID = b.ManagerID
)
SELECT a.EmployeeID,
       a.Name,
       a.Title,
       a.ManagerID,
       b.Name AS ManagerName,
       b.Title AS ManagerTitle,
       a.Depth
FROM   cte_Recursion a LEFT JOIN
       #Employees b ON a.ManagerID = b.EmployeeID
ORDER BY 1;
```

The above query returns the following dataset.

Employee ID	Name	Title	Manager ID	Manager Name	Manager Title	Depth
1	Ramirez	President	<NULL>	<NULL>	<NULL>	0
2	Baers	Vice President	1	Ramirez	President	1
3	Santana	Vice President	1	Ramirez	President	1
4	Perkins	Director	2	Baers	Vice President	2
5	Mercer	Director	3	Santana	Vice President	2

Example 2

Here is another example problem that can be solved with a self-join. Unlike the above problem, this table does not have a foreign key that references its primary key.

List all cities that have more than one customer along with the customer details.

ID	Name	City
1	Smith	Milwaukee

2	Harshaw	Detroit
3	Brown	Dallas
4	Williams	Detroit

The syntax to solve this puzzle with a self-join is shown below.

```
--10.3
SELECT a.ID,
       a.Name,
       a.City
FROM    #Customer a INNER JOIN
        #Customer b ON a.City = b.City AND a.Name <> b.Name;
```

The above query uses a self-join and returns the following result set.

ID	Name	City
2	Harshaw	Detroit
4	Williams	Detroit

Because the Customer table does not have a foreign key relationship, the above query could (and most probably should) be written using the following syntax. This statement is slightly more verbose, but the intent of the statement becomes a bit more obvious.

```
--10.4
WITH cte_CountCity AS
(
SELECT City
FROM   #Customer
GROUP BY City
HAVING COUNT(City) > 1
)
SELECT b.*
FROM   cte_CountCity a INNER JOIN
       #Customer b on a.City = b.City;
```

Example 3

Often, if you need to use a self-join there are options you can use (such as window functions) to avoid the use of a self-join. Let's look at an example.

Given the below dataset consisting of the weight of various animals, create a cumulative total column summing the current row plus all previous rows.

ID	Animal	Weight
1	Elephant	13000

2	Rhinoceros	8000
3	Hippopotamus	3000
4	Giraffe	2000
5	Water Buffalo	2000

This can be accomplished via the following self-join. Notice the use of a CROSS JOIN and a non-equi join.

```
--10.5
SELECT a.ID,
       a.Animal,
       SUM(b.Weight) AS Cummulative_Weight
FROM   #Animals a CROSS JOIN
       #Animals b
WHERE  a.ID >= b. ID
GROUP BY a.ID, a.Animal;
```

ID	Animal	Cummulative Weight
1	Elephant	13000
4	Giraffe	26000
3	Hippopotamus	24000
2	Rhinoceros	21000
5	Water Buffalo	28000

However, a better way to write this statement is by using a windowing function.

```
--10.6
SELECT ID,
       Animal,
       SUM(Weight)
              OVER (ORDER BY ID ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
              AS Cummulative_Weight
FROM   #Animals;
```

ID	Animal	Cummulative Weight
1	Elephant	13000
4	Giraffe	26000
3	Hippopotamus	24000
2	Rhinoceros	21000
5	Water Buffalo	28000

I recommend going through the full feature list of windowing techniques. Functions such as MIN, MAX, COUNT, AVG, SUM, RANK, DENSE_RANK, ROW_NUMBER, NTILE, CUME_DIST, LAG, and LEAD all have windowing capabilities.

POSITIVE BALANCES

Earlier we used two tables containing fruits and their quantities to show the various ways we can write SQL joins. The fruit table example works for pedagogical reasons but isn't the best real-world scenario.

A good puzzle to showcase the multitude of ways a problem can be solved with SQL is the following:

Given the below table, write as many different SQL statements as possible that returns all accounts who never had a positive balance. The result set should only include account 45643, as this is the only account who never had a positive balance.

ID	Balance
45643	-123
45643	-343
45643	-34
18797	-45
18797	-56
18797	67
18797	78
87765	12
87765	567
87765	343

There are 5 different constructs we can use to create the desired output.

1) HAVING
2) OUTER JOIN
3) EXCEPT
4) NOT IN
5) NOT EXISTS

Going through each of the possible solutions, we arrive at the following. Each of the SQL statements will output account 45643.

EXCEPT Set Operator

```
--11.1
SELECT DISTINCT ID FROM #AccountBalances WHERE Balance < 0
EXCEPT
SELECT DISTINCT ID FROM #AccountBalances WHERE Balance > 0;
```

ID
45643

HAVING with MAX

```
--11.2
SELECT ID
FROM   #AccountBalances
GROUP BY ID
HAVING MAX(Balance) < 0;
```

ID
45643

NOT IN

```
--11.3
SELECT DISTINCT
       ID
FROM   #AccountBalances
WHERE  ID NOT IN (SELECT ID FROM #AccountBalances WHERE Balance > 0);
```

ID
45643

NOT EXISTS

```
--11.4
SELECT DISTINCT ID
FROM   #AccountBalances a
WHERE  NOT EXISTS (SELECT 1 FROM #AccountBalances b WHERE Balance > 0 AND
                                                          a.ID = b.ID);
```

ID
45643

LEFT OUTER JOIN

```
--11.5
SELECT DISTINCT
       a.ID
FROM   #AccountBalances a LEFT OUTER JOIN
       #AccountBalances b ON a.ID = b.ID AND b.Balance > 0
WHERE  b.ID IS NULL;
```

ID
45643

As you can see, we have a multitude of ways to write this query. In the real world, I would test each of these different syntaxes to see which statement gives the most optimized execution plan. Performing such analysis here is out of scope, and I will reserve this for a later time.

THINKING IN SETS

Ultimately SQL is an extension of your ability to solve set based problems. Now that we have worked through examples of the different joins, let's work through an example problem that demonstrates how to think in sets.

The following puzzle is more suited for solving with a graph-based database system rather than a relational database system, but ultimately it gives the best example. The solution to this problem requires multiple steps and some unique ways of joining the tables to arrive at the desired output.

Mutual Friends Puzzle

Given the following connections, determine the number of mutual connections between the friends.

Friend 1	Friend 2
Jason	Mary
Mike	Mary
Mike	Jason
Susan	Jason
John	Mary
Susan	Mary

Here is the desired output.

Friend 1	Friend 2	Mutual Friends
Jason	Mary	2
John	Mary	0
Jason	Mike	1

Mary	Mike	1
Jason	Susan	1
Mary	Susan	1

- Jason and Mary have 2 mutual friends: Mike and Susan.
- John and Mary have 0 mutual friends.
- Jason and Mike have 1 mutual friend: Mary
- And so forth……..

Now, let's break this down into the needed sets to solve this puzzle.

Step 1

First, create a list of reciprocals where each friendship exists twice.

For example, the friendship between Jason and Mary will exist as Jason and Mary and its reciprocal, Mary and Jason.

We now have 12 records from our original 6 records.

Friend 1	Friend 2
Jason	Mary
Jason	Mike
Jason	Susan
John	Mary
Mary	Jason
Mary	John
Mary	Mike
Mary	Susan
Mike	Jason
Mike	Mary
Susan	Jason
Susan	Mary

Step 2

Here is where it gets a bit tricky....

Second, we are going to build off our first dataset and construct a set of individuals that we check to determine if they are a friend.

This is best described by example.

Jason and Mary are friends.

Jason is also friends with Mike and Susan.
Mary is also friends with John, Mike, and Susan.

For the friendship reciprocal of Jason and Mary, we need to check Jason for friendships with John, Mike, and Susan (Mary's friends).

For the friendship reciprocal of Mary and Jason, we need to check Mary for friendships with Mike and Susan (Jason's friends).

Friend 1	Friend 2	Mutual Friend
Jason	Mary	John
Jason	Mary	Mike
Jason	Mary	Susan
Jason	Mike	Mary
Jason	Susan	Mary
John	Mary	Jason
John	Mary	Mike
John	Mary	Susan
Mary	Jason	Mike
Mary	Jason	Susan
Mary	Mike	Jason
Mary	Susan	Jason
Mike	Jason	Mary
Mike	Jason	Susan
Mike	Mary	Jason
Mike	Mary	John
Mike	Mary	Susan
Susan	Jason	Mary
Susan	Jason	Mike
Susan	Mary	Jason
Susan	Mary	John
Susan	Mary	Mike

Step 3

Next, given the above dataset, we create reciprocals of the Friend1 and Friend2 columns and place the values in alphabetical order.

For example, Jason and Mary will remain as Jason and Mary, but the reciprocal of Mary and Jason becomes Jason and Mary. This will give us duplicate rows for us to determine who the mutual friends are.

Friend 1	Friend 2	Mutual Friend Check

Jason	Mike	Mary
Jason	Mike	Mary
Jason	Mike	Susan
Jason	Susan	Mary
Jason	Susan	Mary
Jason	Susan	Mike
John	Mary	Jason
John	Mary	Mike
John	Mary	Susan
Mary	Mike	Jason
Mary	Mike	Jason
Mary	Mike	John
Mary	Mike	Susan
Mary	Susan	Jason
Mary	Susan	Jason
Mary	Susan	John
Mary	Susan	Mike

Step 4

Next, we are going to group and count the above dataset.

Friend 1	Friend 2	Mutual Friend Check	Grouping Count
Jason	Mary	John	1
Jason	Mary	Mike	2
Jason	Mary	Susan	2
Jason	Mike	Mary	2
Jason	Mike	Susan	1
Jason	Susan	Mary	2
Jason	Susan	Mike	1
John	Mary	Jason	1
John	Mary	Mike	1
John	Mary	Susan	1
Mary	Mike	Jason	2
Mary	Mike	John	1
Mary	Mike	Susan	1
Mary	Susan	Jason	2
Mary	Susan	John	1
Mary	Susan	Mike	1

Step 5

Next, we review the Grouping Count column in the previous dataset. If the Grouping Count is 2, then the Mutual Friend Check column is indeed a mutual friend.

Friend 1	Friend 2	Mutual Friend Check	Friend Count
Jason	Mary	John	0
Jason	Mary	Mike	1
Jason	Mary	Susan	1
Jason	Mike	Mary	1
Jason	Mike	Susan	0
Jason	Susan	Mary	1
Jason	Susan	Mike	0
John	Mary	Jason	0
John	Mary	Mike	0
John	Mary	Susan	0
Mary	Mike	Jason	1
Mary	Mike	John	0
Mary	Mike	Susan	0
Mary	Susan	Jason	1
Mary	Susan	John	0
Mary	Susan	Mike	0

Step 6

And then finally sum the results.

Friend 1	Friend 2	Total Mutual Friends
Jason	Mike	1
Jason	Susan	1
John	Mary	0
Mary	Mike	1
Mary	Susan	1

Breaking down each of the above steps into the needed SQL statements, we get the following.

Step 1

First, we are going to create a set of duplicates using the UNION statement, but we create a list of all pairings where each friendship exists twice. Notice in the second UNION we reverse the order, and Friend 2 is before Friend 1 in the SELECT statement.

```
--Step 1
--12.1
WITH cte_Reciprocal_Friends AS
(
SELECT Friend1,
       Friend2
FROM   #Friends
UNION
SELECT Friend2,
       Friend1
FROM   #Friends
)
SELECT *
INTO   #Distinct_Friends_Full_1
FROM   cte_Reciprocal_Friends;
```

Step 2

Step 2 is the tricky part where we need to create a validation list for each set of friends. We use an INNER JOIN with a non-equi join to arrive at our results.

```
--Step 2
--12.2
SELECT a.*,
       b.Friend2 AS Mutual_Friend_Check
INTO   #Mutual_Friend_Check_2
FROM   #Distinct_Friends_Full_1 a INNER JOIN
       #Distinct_Friends_Full_1 b ON a.Friend2 = b.Friend1
WHERE  a.Friend1 <> b.Friend2;
```

Step 3

For step 3 we create a list of reciprocals and place the Friend 1 and Friend 2 columns in alphabetical order. The result of this statement will create duplicates.

```
--Step 3
--12.3
SELECT (CASE WHEN Friend1 < Friend2 THEN Friend1 ELSE Friend2 END) AS Friend1,
       (CASE WHEN Friend1 < Friend2 THEN Friend2 ELSE Friend1 END) AS Friend2,
       Mutual_Friend_Check
INTO   #Reciprocal_Mutual_Friend_Check_3
FROM   #Mutual_Friend_Check_2;
```

Step 4

Step 4 simply does a COUNT and GROUP BY.

```
--Step 4
--12.4
SELECT Friend1,
       Friend2,
       Mutual_Friend_Check,
       COUNT(*) AS Grouping_Count
INTO   #Reciprocal_Mutual_Friend_Check_Count_4
FROM   #Reciprocal_Mutual_Friend_Check_3
GROUP BY Friend1, Friend2, Mutual_Friend_Check;
```

Step 5

For step 5, we create a CASE statement to check if the column Grouping Count contains a 2 (a duplicate from the previous step 3). If the value is 2, then the field Mutual Friend Check is a mutual friend.

```
--Step 5
--12.5
SELECT Friend1,
       Friend2,
       Mutual_Friend_Check,
       (CASE Grouping_Count WHEN 1 THEN 0 WHEN 2 THEN 1 END) AS Friend_Count
INTO   #Reciprocal_Mutual_Friend_Check_Count_Modified_5
FROM   #Reciprocal_Mutual_Friend_Check_Count_4;
```

Step 6

And the last step is to simply sum the column Friend Count to arrive at our answer.

```
--Step 6
--12.6
SELECT Friend1,
       Friend2,
       SUM(Friend_Count) AS Total_Mutual_Friends
FROM   #Reciprocal_Mutual_Friend_Check_Count_Modified_5
GROUP BY Friend1, Friend2;
```

CONCLUSION

Within SQL, the word join can mean many things. There are the traditional joins of INNER, OUTER, and FULL. But the word join can also be used to describe equi-joins, non-equi joins, natural joins, self-joins, or the different join algorithms: nested loop joins, sort-merge, and hash joins.

SQL is a combination of both join and set theory and understanding the various nuances of these two is instrumental in understanding SQL. In join theory, all joins are restricted cartesian products. When a join table operation is initiated, a cartesian product is created, which is then restricted by the ON

condition. If an OUTER JOIN is specified, then an additional step of adding the outer rows is needed. The main difference between an INNER and an OUTER JOIN, is that an INNER JOIN acts as a filter, whereas OUTER JOINS acts as a matching criterion.

We also discussed the different join operators and the processing order of SQL, which is different than its written order. I've included the diagram in Appendix A, and this should give insight into the internal workings of SQL. Although the SELECT clause is the first statement in an SQL statement, the FROM clause is executed first, and then the WHERE clause.

Set operations are also discussed and how Venn diagrams are the better construct to show set operations and not join operations. For join operations, I recommend using join diagrams to show example tables and the results. When performing set operations, the datasets need to be of the same type.

We also worked through example real world problems where self joins are needed to create the desired output. SQL is the language of databases, and there are multiple syntax options for solving set-based problems. The Positive Balances example problem is a perfect example in how many ways we can construct an SQL statement that arrives at the desired output. Ultimately, it is up to the developer to choose the best statement.

I hope this document has helped you understand the behavior of SQL joins. If you have any questions, bug fixes, or want to say hello, please contact me through my website at https://advancedsqlpuzzles.com.

The End.

Appendix A

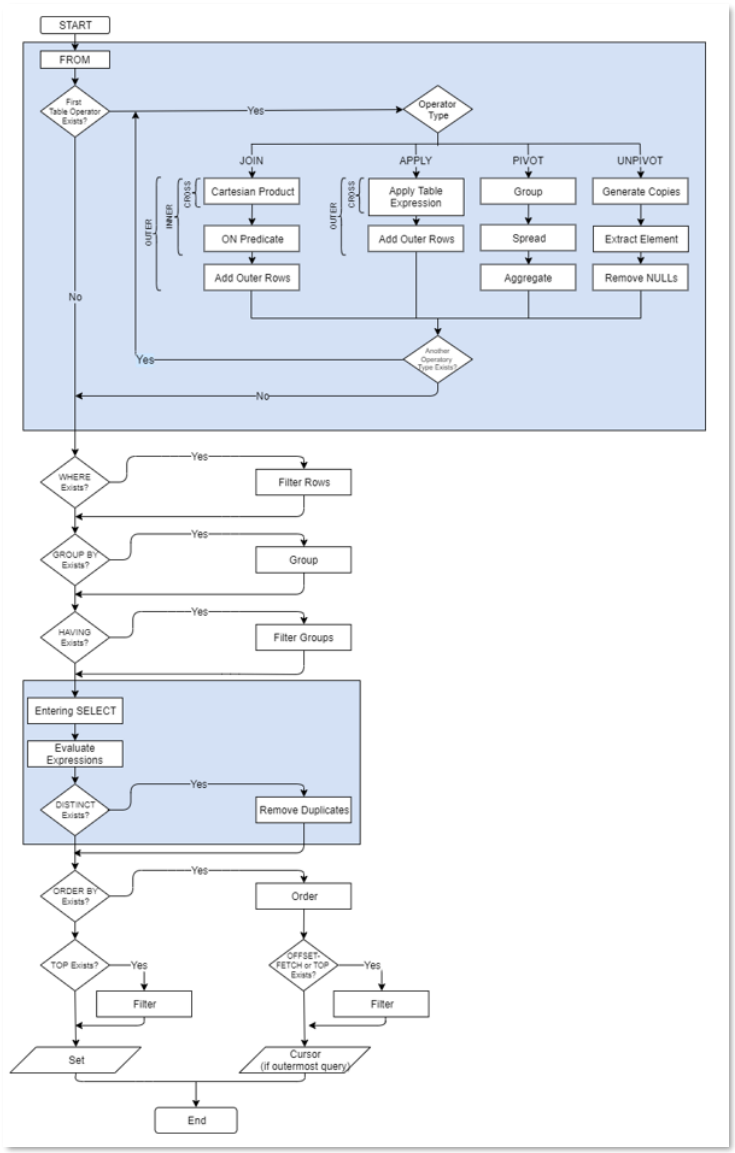

Figure 1