Advanced SQL Joins

Scott Peters

www.advancedsqlpuzzles.com

Last Updated 05/01/2021

“Can you name all the different types of joins?”

Sounds like a simple question. Most developers will answer with the standard inner and outer join. Some developers will remember to include the full outer join (and hopefully give a real-world example of when to use it).

But there are many more ways to answer this question. Besides inner and outer joins; there are self joins, cross joins, equi joins, non equi joins, theta joins, natural joins, semi joins, and anti joins.

But even before we begin looking into these types of joins, it is best to understand the JOIN OPERATORS and how they interact with the 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) SELECT 5) ORDER BY. To best understand the flow of an SQL query, please refer to the following diagram on the next page that I have copied from the Microsoft book “T-SQL Querying”.

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

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

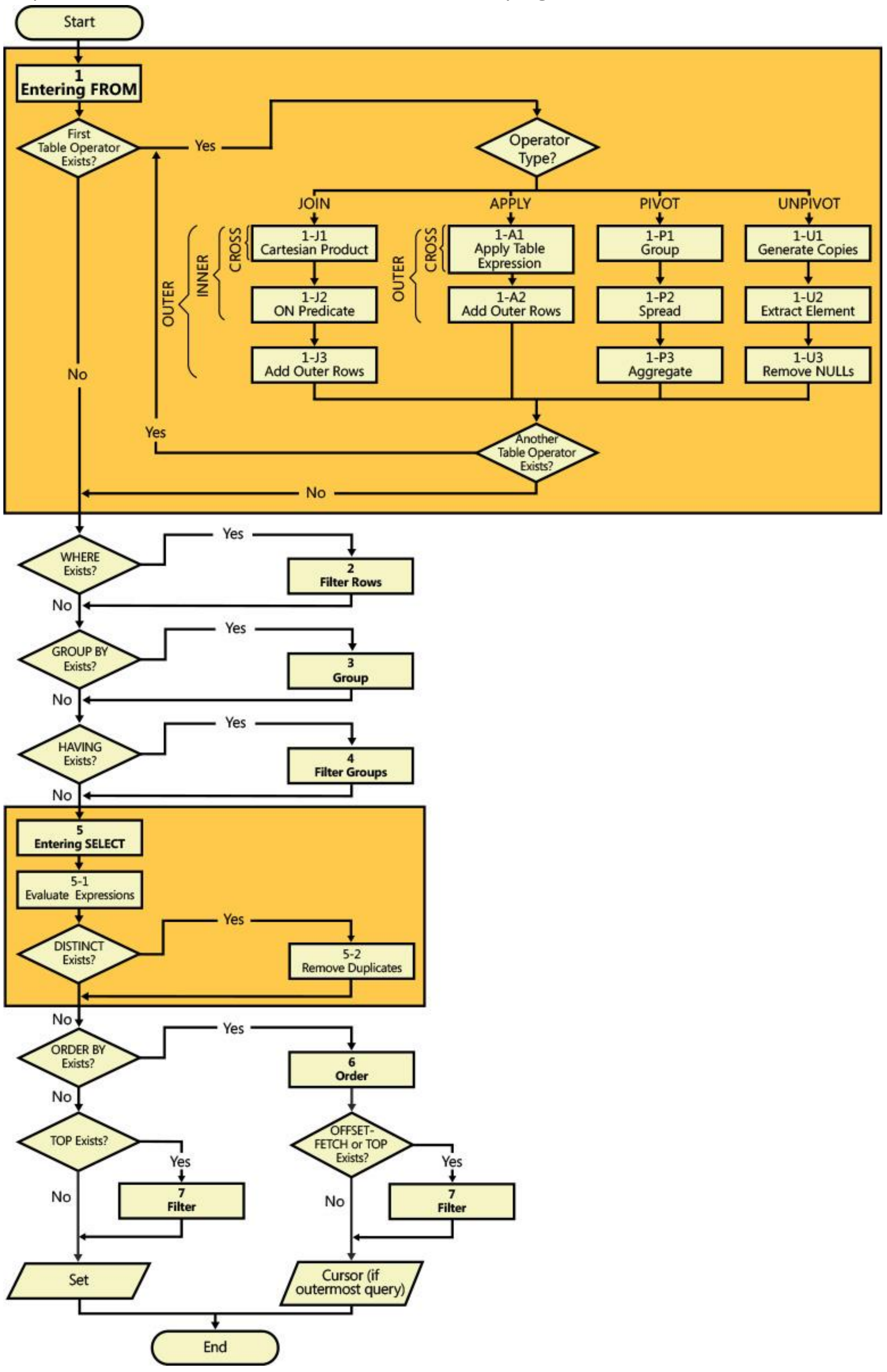
The APPLY operator might be unfamiliar to some people, as this operator is used when you want to join a table to a table-valued function. For more information on the APPLY, PVIOT, and UNPIVOT operators, see the SQL Server documentation. Our focus will be on the JOIN operator.

Here 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. The sub-phases involved in the join operator are:

1) Create a cartesian product
2) Then 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.

Copied from the Microsoft book, “T-SQL Querying”



The below query is a restricted cartesian product, where the restriction is the CustomerID must be equal in both tables.

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

If we modify the query to use an outer join, then query performs an additional step by adding the outer rows.

Now that we have that out of the way, let's have some fun and learn a few new tricks.

Self Joins

A self join is a join in which a table is joined with itself (a unary relationship). Often the table will have a foreign key column which references its own primary key, as the below example demonstrates.

Example 1

List all employees and the name of their manager.

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

```
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;
```

Returns the following result set

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 also worth mentioning here 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 president.

```
WITH cte_Recursion AS
(
SELECT EmployeeID, [Name], Title, ManagerID, 0 AS Depth
FROM   Employee
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
       Employee 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
       Employee b ON a.ManagerID = b.EmployeeID
ORDER BY 1;
```

Returns the following result set

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 then one customer along with the customer details.

Customer ID	**Name**	**City**
1	Smith	Milwaukee
2	Harshaw	Detroit
3	Brown	Dallas
4	Williams	Detroit

```
SELECT a.CustomerID, a.[Name], a.City
FROM    Customer a INNER JOIN
        Customer b ON a.City = b.City AND a.[Name] <> b.[Name];
```

Returns the following result set

CustomerID	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, as this becomes a bit more obvious to the query's intent.

```
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;
```

Left and Right Outer Joins

One nuance that I want to point out with outer joins is how to properly apply predicate logic. Here we will apply the predicate logic in the WHERE and the FROM clauses and note their differences. Let's work through an example.

List all the customers regardless if they have a favorite color. If their favorite color is blue, return blue, else return NULL.

Customer

Customer ID	Name
1	Joe
2	Sally
3	Victor

Favorite Color

Customer ID	Favorite Color
1	Blue
2	Green

Here is one correct way to solve the above challenge that satisfies the requirements. Many developers do not realize they can put the predicate logic in the FROM clause of the SQL statement and retain the outer join.

```
SELECT cust.CustomerID, cust.[Name], fav.FavoriteColor
FROM    Customer cust LEFT OUTER JOIN
        FavoriteColor fav ON cust.CustomerID = fav.CustomerID
                            AND fav.FavoriteColor = 'Blue';
```

Returns the following result set

Customer ID	Name	Favorite Color
1	Joe	Blue
2	Sally	
3	Victor	

If we place the predicate logic in the WHERE clause of the SQL statement, then the outer join becomes an inner join, and the statement becomes incorrect.

```
SELECT cust.CustomerID, cust.[Name], fav.FavoriteColor
FROM    Customer cust LEFT OUTER JOIN
        FavoriteColor fav ON cust.CustomerID = fav.CustomerID
WHERE   fav.FavoriteColor = 'Blue';
```

Returns the following incorrect result set.

CustomerID	Name	FavoriteColor
1	Joe	Blue

Full Outer Joins

In SQL, the FULL OUTER JOIN combines the results of both left and right outer joins and returns all (matched or unmatched) rows from the tables on both sides of the join clause.

Often developers have a hard time coming up with real world examples of when to use a FULL OUTER JOIN. Sure, we know to use it when we want matched and unmatched rows from both tables, but describing a real world scenario using a FULL OUTER JOIN eludes some developers.

Let's look at a few real-world examples.

Example 1

You are tasked with providing a list of dance partners from the following table.
Provide an SQL statement that matches each Student ID with an individual of the opposite gender.

Note there is a mismatch in the number of students, as one female student will be left without a dance partner. Please include this individual in your list as well.

Student ID	Gender
1001	M
2002	M
3003	M
4004	M
5005	M
6006	F
7007	F
8008	F
9009	F

```
WITH cte_Males AS
(
SELECT  ROW_NUMBER () OVER (ORDER BY StudentID) AS RowNumber,
        StudentID,
        Gender
FROM    #DancePartners
WHERE   Gender = 'M'
),
cte_Females AS
(
SELECT  ROW_NUMBER () OVER (ORDER BY StudentID) AS RowNumber,
        StudentID,
        Gender
FROM    #DancePartners
WHERE   Gender = 'F'
)
SELECT  a.StudentID, a.Gender, b.StudentID, b.Gender
FROM    cte_Males a FULL OUTER JOIN
        cte_Females b ON a.RowNumber = B.RowNumber;
```

Returns the following result set

Student ID	Gender	Student ID	Gender
1001	M	6006	F
2002	M	7007	F
3003	M	8008	F
4004	M	9009	F
5005	M	NULL	NULL

Example 2

You are tasked with auditing the inventory of two grocery baskets.

Provide an SQL statement that describes which items are in both baskets, and which items exist solely in their respective basket.

Grocery Basket	Item
1	Milk
1	Sugar
1	Eggs
1	Flour
2	Milk
2	Sugar
2	Butter
2	Bread

I will refrain from providing the SQL code, as it's essentially the same as the first example.

Natural Joins

A natural join is a type of join that creates an implicit join based on the common columns it the two tables being joined.

Here are some common usages of the natural join syntax from the Oracle documentation.

```
SELECT *
FROM   Countries NATURAL JOIN
       Cities;

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

SELECT *
FROM   Countries NATURAL LEFT JOIN
       Cities
```

Cross Joins

A cross join produces a result set of all possibilities. It is the product of the number of rows in the first table multiplied by the number of rows in the second table.

A common usage for the cross join is to create a full set of elements in which you want to report on. Let's look at the following table where each distributor has a sale, but not necessarily a sale in each month in the first quarter.   From this table, we want to build a reporting table for each distributor for each month, where a $0 record will be created for the months a distributor did not have a sale.

Here is our input table.

Distributor	Month	Sales
Ace	Jan-2019	$114.35
Ace	Feb-2019	$185.36
Direct Parts	Jan-2019	$5,263.36
Direct Parts	Feb-2019	$856.35
Direct Parts	Mar-2019	$56,465.23

```
WITH
cte_DistinctDistributor as
(
SELECT  DISTINCT Distributor
FROM    Sales
),
cte_DistinctMonth as
(
SELECT  DISTINCT ReportMonth
FROM    Sales
)
SELECT  a.Distributor,
        b.ReportMonth,
        ISNULL(c.Sales,0) AS Sales
FROM    cte_DistinctDistributor a CROSS JOIN
        cte_DistinctMonth b LEFT OUTER JOIN
        #Sales c on a.Distributor = c.Distributor AND
       b.ReportMonth = c.ReportMonth;
```

The above query produces the below result set with the $0 record for Ace.

Distributor	Month	Sales
Ace	Jan-2019	$114.35
Ace	Feb-2019	$185.36
Ace	Mar-2019	$0
Direct Parts	Jan-2019	$5,263.36
Direct Parts	Feb-2019	$856.35
Direct Parts	Mar-2019	$56,465.23

Relational Algebra

Relational algebra was first created by Edgar F. Codd and is the mathematical backbone for modeling the data stored in relational databases and providing the foundation for the SQL language.

I highly recommend SQL developers pick up a college level textbook on database systems and familiarize themselves with the basics of relational algebra.

Here we will look at some of the join verbiage you will find in a relational algebra textbook and give examples of their usage. It is important to note that different relational algebra textbooks will use different verbiage to describe the various joins. Sometimes equi joins are categorized as natural joins, and theta joins and equi joins are used interchangeably.

Equi Joins

An equi join is simply a join condition containing an equality operator. An equi join can be both an inner join or an outer join.

```
SELECT *
FROM   Table1 t1 INNER JOIN
       Table2 t2 ON t1.ColumnID = t2.ColumnID

SELECT *
FROM   Table1 t1 LEFT OUTER JOIN
       Table2 t2 ON t1.ColumnID = t2.ColumnID
```

If you are reading any college course text books on the subject, equi join may not appear, but instead will discussed as a natural join, which we have covered previously.

Theta Joins / Non Equi Joins

Theta joins, or non-equi joins, depending on who you want to ask, is any type of join that is not an equi join. The operators in use here are (<= , >= , < , > , <>).

Theta Joins are used when joining to a Type 2 SCD table, such as below. In the following syntax I use the BETWEEN statement, which equates to the operators >= and <=.

```
SELECT *
FROM  Account a INNER JOIN
      Transaction t
WHERE t.TransactionDate BETWEEN a.RowBeginDate AND a.RowEndDate
```

When using the inequality operator (<>), the query must also include an equality operator (=) to make the SQL statement relevant. Here we look at an example from earlier and note the [City] field looks for equality and the [Name] field looks for inequality.

```
SELECT a.CustomerID, a.Name, a.City
FROM    Customer a INNER JOIN
        Customer b ON a.City = b.City AND a.Name <> b.Name;
```

Semi Joins / Anti Joins

A semi join between two tables returns rows that match an EXISTS or an IN subquery, without the possibility of duplicating rows. The subquery used in the semi join cannot be used in the projection of the query, making semi joins easily readable.

Here is an example of a correlated subquery that looks for all departments that have an employee whose salary is greater than $100,000.

```
SELECT *
FROM   Departments d
WHERE  EXISTS (SELECT * FROM Employees e WHERE d.DeptID = e.DeptID AND
                                               e.Salary >= 100000);
```

It can also be written with the IN statement (and is much preferred).

```
SELECT   *
FROM     Departments d
WHERE    DeptID IN (SELECT DeptID FROM Employees WHERE Salary >= 100000);
```

The major benefit of semi joins is that it removes the possibility of duplicates. In this example we have two tables populated with the following identical data.

Table 1

Column A
Alpha
Alpha

Table 2

Column A
Alpha
Alpha

If we want to list all the records in Table 1 that have a corresponding value in Table 2, are only option would be to use a semi join.

```
SELECT   *
FROM     Table1
WHERE    ColumnA IN (SELECT ColumnA FROM Table2);
```

Anti joins are the same as semi joins, but they look for exclusion rather than inclusion using the NOT EXISTS or the NOT IN operators. Here are the same queries above, but as anti joins.

```
SELECT *
FROM   Departments d
WHERE  NOT EXISTS (SELECT * FROM Employees e WHERE d.DeptID = e.DeptID AND
                                                   e.Salary >= 100000);

SELECT   *
FROM     Departments d
WHERE    DeptID NOT IN (SELECT DeptID FROM Employees WHERE Salary >= 100000);
```

Conclusion

I hope you learned a few new things about SQL's join operations and can use this document as a guide for your future development work.

If you find any errors, think I'm omitting anything, or just want to say hello, please email me at scottpeters1188@outlook.com