Joins

Kaitlyn Keil and Paige Pfenninger

May 11, 2019

1 What Are Joins

In a very general sense, joins are operations in relational databases that combine data tables based on some condition, the 'join predicate.' The join predicate is a special type of comparison predicate that compares the values in the specified row and column and returns either true or false. The row is only included in the new table if the join predicate returns true. The value it looks at is often referred to as the 'join attribute.' Effectively and naively, it is the same as taking a product and then selecting for certain conditions; later in this paper, we will discuss ways joins have been made more efficient. In relational languages such as SQL, the result of this operation is a temporary table that is the combination of tuples from the former tables. This is helpful for queries that are matching primary keys with foreign keys, where a foreign key may be the primary key to another entry in the database. Joins could also be used for finding all valid combinations of something. For example, if you have a table of people looking to buy a house with a maximum price point and another table of houses for sale with a price, you could use a join to determine all possible houses for all possible people.

Because of the range of queries they can support, there are many types of joins, each of which have slightly different behaviors. The following sections will explore the types in more depth. These types are inner, outer (can be left, right, or full), cross, and self.

1.1 Inner Join

An inner join creates a new table by combining column values using the join predicate from two tables and only including the combined tuples that satisfy the predicate, as shown in Figure 1.

The key different between an inner join and outer joins is that inner joins will never return a row if NULL is the entry for the join predicate. This is because NULL values do not match with anything and therefore will never satisfy the join predicate.

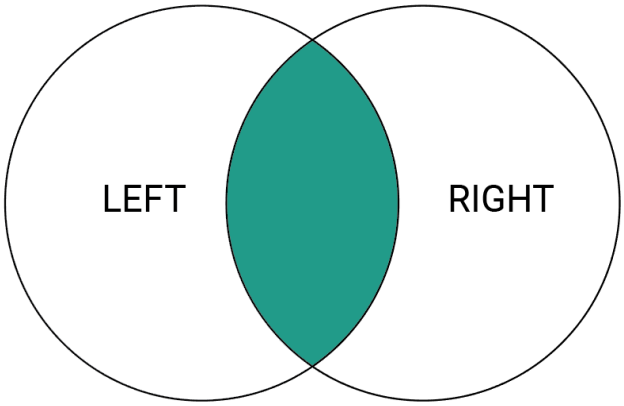

Figure 1: Venn Diagram of an inner join between a left table and a right table

```
SELECT *
FROM table1 INNER JOIN table2
ON table1.column_name > table2.column_name;
```

1.1.1 Equi Join

An equi join is a specific type of inner join that requires the use of = in the join predicate. Inner joins can use several different types of operators, but equi joins specify that = must be used. Equi joins are the most common type of inner joins, and would be used to match primary and foreign keys.

```
SELECT *
FROM table1 INNER JOIN table2
ON table1.column_name = table2.column_name;
```

A further subcategory of equi joins are natural joins, where matches are automatically found based on a shared column name in two tables. For example, if both tables have *dept_id* as a column header, the user can simply call a natural join on the two tables without specifying which column to look at. The result automatically returns the inner join on *dept_id*.

1.2 Cross Join

If no WHERE clause is used, a cross join is simply a Cartesian product of two tables. If a WHERE clause is used, the cross join functions exactly the same as an inner join. Cross joins are used when you want to create a combination of every row from two tables.

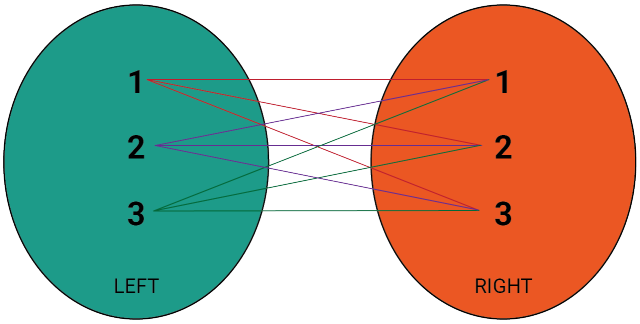

Figure 2: Venn Diagram of a cross join between a left table and a right table.

```
SELECT *
FROM table1
CROSS JOIN table2;
```

1.3 Outer Join

Unlike inner joins, outer joins will return certain rows even if the join predicate is not satisfied, populating these tuples with NULL values where no other tuple exists. The circumstances where NULL values are returned depend on the type of outer join being used.

1.3.1 Left Outer Join

In left outer joins, all tuples in the first table specified will have at least one row in the resulting temporary table. If no tuple in the second table specified satisfies the join predicate, the resulting row will contain the values from the left table and NULL values in the remaining spaces. If there are multiple tuples that satisfy the join predicate, each match will have its own row, and no row with NULL values will exist.

```
SELECT *
FROM table1 LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

1.3.2 Right Outer Join

Right outer joins are the same as left outer joins, but returning at least one row for each entry in the right table, as shown in Figure 4.

```
SELECT *
FROM table1 RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

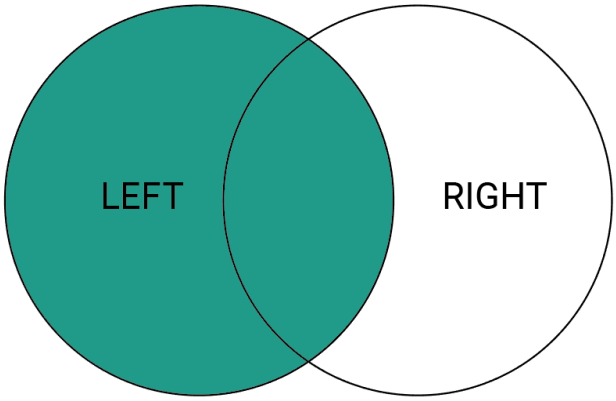

Figure 3: Venn Diagram of a left outer join between a left table and a right table.

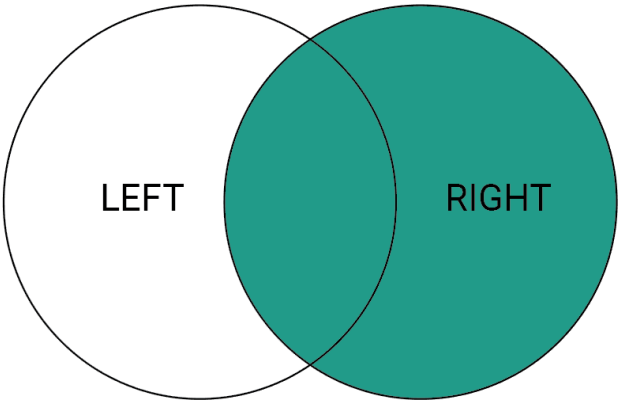

Figure 4: Venn Diagram of a right outer join between a left table and a right table.

1.3.3 Full Outer Join

Full outer joins behave the same way that left and right outer joins do, except that all tuples from both tables will return a row, regardless of if they have a tuple in the other table that satisfies the join predicate.

```
SELECT *
FROM table1 OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

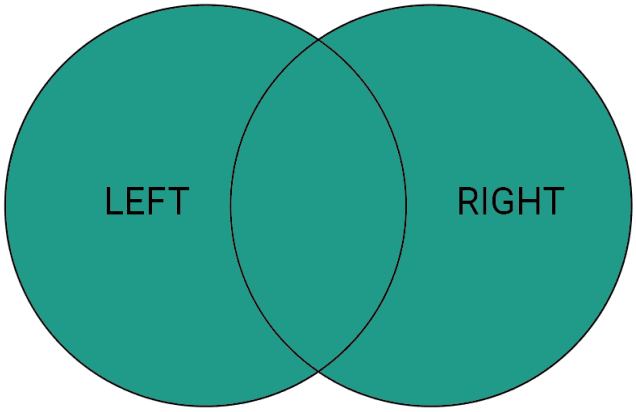

Figure 5: Venn Diagram of a full outer join between a left table and a right table.

1.4 Self Join

Self joins are a special case of join that join a table with itself. This is most commonly used in the case of foreign keys that reference other entries in the same table; to use an example from class, a table $EMPLOYEES$ which has the columns *emp_id* and *sup_id*, where *sup_id* refers to another employee's *emp_id*, could use a self join such as

```
SELECT *
FROM table1 AS A JOIN table1 AS B
ON A.emp_id = B.sup_id;
```

in order to match all employees with their supervisor. This operation typically requires renaming of columns, but allows for joining a table with itself without creating a duplicate table.

1.5 Why Do You Need Joins

All of the join operations can be achieved with projecting and selecting. However, using the join syntax is generally more readable, handles NULL values inherently, and allows for specifically optimized operations. Also, for some advanced queries, joins allow the user to specify the order in which to scan relations and filter these relations before joining, which can increase efficiency.

1.6 What Are Common Join Issues

Without being optimized in some way, joins can be very slow, particularly for large datasets. Primary keys can also be messy in the resulting temporary table, depending on how they are constructed; NULL values in particular can make this difficult

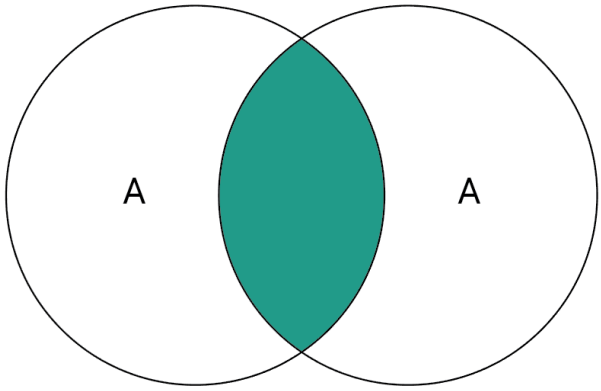

Figure 6: Venn Diagram of a self join between a table and itself.

to handle. Finally, as specified before, they continue to essentially be product and select, which means they are often just an extra thing to learn.

2 How Are Joins Implemented

We have mentioned optimization a few times during this paper. Now we will discuss a few ways that joins might be implemented, as well as comparing some of their relative performance metrics.

2.1 Nested Loop

The first naive approach known as the nested loop. In this implementation, for each tuple in table A, it loops through every tuple in table B to find any that satisfy the join predicate. Each tuple that does is added to the temporary table, returned at the end. This works well for comparisons that aren't equality, and therefore might have multiple valid entries; however, as an $O(n * m)$ operation for two tables of size n and m respectively, it is not generally efficient.

2.2 Hash

The hash implementation only works for equality comparisons, but operates more quickly than the nested loop. In the hash implementation, the tuples of table A are used to create a temporary hash table, where the join attribute is the key. The tuples of table B are then iterated through, probing the hash table for the join attribute. If a result is found, it is added to results.

This is an implementation that trades speed for memory, as it is an $O(n)$ operation (where n is the size of the larger table). It is often optimized so that the

smaller table is used to construct the hash table, though this is not always a viable solution, depending on the specific structure of the query and tables.

2.3 Symmetric Hash

The symmetric hash is an enhancement of the hash join algorithm that is specifically designed for data streams. In this implementation, each stream creates a hash table. Each of these new entries probes the other table for a match; if found, it combines the tuples and adds them to the result. This allows for a more dynamic set up.

2.4 Block Nested Loop

The block nested loop is a move efficient version of the nested loop join. In a block nested loop join, entries from the first table are loaded into a buffer until the buffer is full. Then each element from the second table is compared with the elements in the buffer. Any matching elements are sent to the result stream. After every element of the second table has been compared to the buffer, the buffer is emptied and then refilled with more data from the first table. This process continues until all of the data from the first table has been put into the buffer. This is faster than the nested loop join because makes good utilization of the buffer by continually loading it with as much data from the first table as possible. Each element in the second table is only scanned based on the number of refills of the buffer rather than the number of elements in the first table like in nested loop joins. It is generally only used for outer joins and semi-joins.

2.5 Merge

The merge implementation is faster than nested loops and less memory-intensive than hashing. For this implementation, both tables are sorted on the join attribute. They are then both iterated through with separate pointers, with the pointers advancing whenever the values for one of them is no longer relevant.

This implementation is best for comparing greater than/less than values or equality. It has a time complexity of $O(log(n))$, where n is the size of the larger table. Most of this time complexity comes from the sort itself.

3 Implementation and Results

We implemented versions of the nested loop join and the hash join, based off of the relation homework assignments throughout the course. Our implementation can be found at this GitHub repository.

In this repository, we run a time comparison between the two implementations, based only on equi-joins as the implementations are otherwise identical. From this, we got the following results:

Join	Nested Loop	Hash Join
Inner	1.39	1.00
Left Outer	1.63	1.27
Right Outer	1.47	1.08
Full Outer	1.65	1.34

This lets us see that the hash implementation is, in fact, faster. However, this only is true for equality joins, and for those, would also use more memory, if the databases were large enough for this to be an issue.

4 Next Steps

We would extend this project by implementing merge joins for a third comparison, loading a larger database, and including a memory analyzer for each implementation, so that we could have more detailed descriptions on the benefits and drawbacks of each. From our research, merge sort should be between the two with performance.

References

[1] Assorted. *Join (SQL)*. 2019. URL: `https://en.wikipedia.org/wiki/Join_(SQL)`.

[2] Periscope Data. *How Joins Work*. 2017. URL: `https://www.periscopedata.com/blog/how-joins-work`.

[3] SQL Join. *SQL Joins Explained*. URL: `http://www.sql-join.com/`.

[4] MariaDB. *Block-Based Join Algorithms*. 2019. URL: `https://mariadb.com/kb/en/library/block-based-join-algorithms/`.