Lecture #12: Joins Algorithms

15-445/645 Database Systems (Fall 2026)
https://15445.courses.cs.cmu.edu/fall2026/
Carnegie Mellon University
Jignesh Patel

1 Introduction

The goal of a good database design is to minimize unnecessary repetition of information. This is why tables are often composed based on *normalization theory*. Joins are then needed to reconstruct the original tables.

We focus on **inner equijoin** algorithms for combining two tables. An *equijoin* algorithm joins tables where keys are equal. These algorithms can be tweaked to support other joins. For binary joins, we often prefer the left table (the "outer table") to be the smaller one of the two. Multi-way joins exist primarily in research literature (e.g., worst-case optimal joins).

2 Join Operators

In a query plan, the operators are arranged in a directed tree where the data flows from the leaves towards the root. The output of the root node is the result of the query. The following subsections will discuss the decisions for designing any join operator.

Operator Output

For a tuple $r \in R$ and a tuple $s \in S$ that match on join attributes, the join operator concatenates r and s together into a new output tuple.

In reality, the content of output tuples generated by a join operator varies. It depends on the DBMS's query processing model, storage model, and the query itself. There are multiple approaches to designing the join operator's output.

- **Data approach:** The join creates new output tuples containing copies of the values in the joined tables. These tuples are put into an intermediate result table for the operator. The advantage of this approach is that future query plan operators never need to return to the base tables to get data. This is more useful in the case of row stores since all the tuple data will be brought into memory anyway. The disadvantage is that this requires more memory to materialize the entire tuple. This is called *early materialization*.
  The DBMS can also do additional computation and omit attributes that will not be needed later in the query to optimize this approach further.
- **Record id approach:** The DBMS only copies the join keys and the matching tuple record ids. This approach is ideal for column stores since the join can avoid bringing additional columns into memory to copy them. This is called *late materialization*.

In the real world, DBMS often mix and match between the two approaches, wherein based on the number of tuples, columns, and further operators, a combination of the two strategies can be taken.

Cost Analysis

The cost metric used here to analyze different join algorithms will be the number of disk I/Os used to compute the join. For now, compute and network costs will be ignored, as I/O costs dominate in disk-based DBMSs.

Note that I/Os needed to output the join result are not considered in this analysis. This is because the output cost depends on the data and will be the same for all join algorithms.

Given a query that joins table R with table S, assume the DBMS knows the following about those tables:
- M pages in table R (Outer Table), m tuples total
- N pages in table S (Inner Table), n tuples total

Also, note that $R \bowtie S$ (the natural join of tables R and S) is the most common operation and must be carefully optimized. One inefficient algorithm is to compute $R \times S$ (the cross product of tables R and S) and select relevant tuples. However, the cross-product is huge and results in a very inefficient approach.

In general, there will be many algorithms and optimizations which can reduce join costs in some cases, but no single algorithm works well in every scenario.

3 Nested Loop Join

At a high level, this join algorithm contains two nested for loops that iterate over the tuples in both tables and perform a pairwise comparison. If the tuples match the join predicate, then it outputs them. The table in the outer for loop is called the *outer table*, while the table in the inner for loop is called the *inner table*.

The DBMS will always want to use the "smaller" table as the outer table. Smaller generally is based on the number of pages but can sometimes mean tuples as well. We will buffer as much of the outer table in memory as possible, and use indexes where available.

Naïve Nested Loop Join

Compare each tuple in the outer table with each tuple in the inner table. This is the worst-case scenario where the DBMS must scan the entirety of the inner table for each tuple in the outer table without any caching or access locality.
Cost: $M + (m \times N)$

Block Nested Loop Join

For each block in the outer table, fetch each block from the inner table and compare all the tuples in those two blocks. Based on the # of pages and not the # of tuples, the smaller table should be the outer table. This algorithm performs fewer disk accesses because the DBMS scans the inner table for every outer table block instead of for every tuple.
Cost: $M + (M \times N)$

Buffer Pool Usage: If the DBMS has B buffer pool frames available to compute the join, it can use $B - 2$ buffers to scan the outer table. It will use one buffer to scan the inner table and one buffer to store the join's output.
Cost: $M + \left(\left\lceil \frac{M}{B-2} \right\rceil \times N \right)$

Index Nested Loop Join

The previous nested loop based algorithms perform poorly because the DBMS has to do a sequential scan to check for a match in the inner table. However, if the database already has an index for one of the tables on the join key, it can use that to speed up the comparison. The DBMS can use an existing index or build a temporary one for the join operation.

The inner table will be the one that requires an index. Assume the cost of each index probe is some constant value C per tuple and the DBMS does a sequential scan on the outer table.
Cost: $M + (m \times C)$

4 Sort-Merge Join

At a high level, a sort-merge join sorts the two tables on their join key(s). The DBMS can use the external mergesort algorithm for this. It then steps through each table with cursors and emits matches (like in Mergesort).

This algorithm is useful if one or both tables are already sorted on join attribute(s) (like with a clustered index) or if the output needs to be sorted on the join key anyway.

The worst-case scenario for this algorithm is if the join attribute for all the tuples in both tables contains the same value, which is very unlikely to happen in real databases. In this case, the cost of merging would be $M \cdot N$ since, for each outer page, we will have to match the entire inner table. Most of the time, though, the keys are mostly unique, so the merge cost can be assumed to be approximately $M + N$.

Assume that the DBMS has B buffers to use for the algorithm:

- Sort cost for table R: $2M \times \left(1 + \lceil log_{B-1} \lceil \frac{M}{B} \rceil \rceil\right)$
- Sort cost for table S: $2N \times \left(1 + \lceil log_{B-1} \lceil \frac{N}{B} \rceil \rceil\right)$
- Merge cost: $M + N$
- Worst-Case Merge Cost: $M \times N$

Total Cost: Sort + Merge

5 Hash Join

Hash join algorithms use a hash table to split the tuples into smaller chunks based on their join attribute(s). This reduces the number of comparisons that the DBMS needs to perform per tuple to compute the join. Hash joins can only be used for equi-joins on the complete join key.

If a tuple $r \in R$ and a tuple $s \in S$ satisfy the join condition, then they have the same value for the join attributes. If that value is hashed to some value i, the R tuple has to be in bucket r_i, and the S tuple has to be in bucket s_i. Thus, the R tuples in bucket r_i need only to be compared with the S tuples in bucket s_i.

Basic Hash Join

- **Phase #1 – Build:** First, scan the outer relation and populate a hash table using the hash function h_1 on the join attributes. The key in the hash table is the join attributes. The value depends on the implementation, and can be a full tuple or a record ID (early vs. late materialization).
- **Phase #2 – Probe:** Scan the inner relation and use the hash function h_1 on each tuple's join attributes to jump to the corresponding location in the hash table and find matching tuples. Since there may be collisions in the hash table, the DBMS must examine the original values of the join attribute(s) to determine whether tuples satisfy the join condition. The probe table can be of any

size, with only the build side needing to fit in memory.

One optimization for the probe phase is the usage of a Bloom Filter, which is created as the DBMS builds the hash table on the outer table in the first phase. This is a probabilistic data structure that can fit in CPU caches and answer the question "*is key x in the hash table?*" with either *definitely no* or *probably yes*. It reduces the amount of disk I/O by preventing disk reads that will not result in a match. Such techniques of providing extra metadata are called **sideways information passing**.

If the DBMS knows the size of the outer table, the join can use a static hash table. If it does not know the size, the join must use a dynamic hash table or allow for overflow pages.

Grace Hash Join / Partitioned Hash Join

When the tables do not fit in the main memory, the DBMS has to swap tables in and out at random, leading to poor performance and not leveraging the buffer pool. The Grace Hash Join is an extension of the basic hash join that also hashes the inner table into partitions that are written out to disk.

- **Phase #1 – Build:** First, scan both the outer and inner tables and populate a hash table using the hash function h_1 on the join attributes. The hash table's buckets are written out to disk as needed. If a single bucket does not fit in memory, the DBMS can use *recursive partitioning* with different hash function h_2 (where $h_1 \neq h_2$) to further divide the bucket. This can continue recursively until the buckets fit into memory.
- **Phase #2 – Probe:** For each bucket level, retrieve the corresponding outer and inner table pages. Then, perform a hash join on their contents.

Partitioning phase cost: $2 \times (M + N)$
Probe phase cost: $M + N$
Total Cost: $3 \times (M + N)$

One edge case is that if a single join key has too many matching records that do not fit in memory, we can use a **block nested loop join** just for that key, which avoids random I/O in exchange for sequential I/O.

Hybrid hash join optimization: adapts between basic hash join and Grace hash join; if the keys are skewed, keep the hot partition in memory and immediately perform the comparison instead of spilling it to disk. Difficult to implement correctly and rarely done in practice.

6 Conclusion

Joins are an essential part of relational databases, and it's critical to have efficient algorithms for them.

Algorithm	I/O Cost	Example
Naïve Nested Loop Join	$M + (m \cdot N)$	1.4 hours
Block Nested Loop Join	$M + \left(\left\lceil \frac{M}{B-2} \right\rceil \cdot N\right)$	0.65 seconds
Index Nested Loop Join	$M + (m \cdot C)$	Varies
Sort-Merge Join	$M + N +$ (sort cost)	0.75 seconds
Hash Join	$3 \cdot (M + N)$	0.45 seconds

Figure 1: The table above assumes the following: $M = 1000, m = 100000, N = 500, n = 40000, B = 100$ and 0.1 ms per I/O. Sort cost is $R + S = 4000 + 2000$ IOs, where $R = 2 \cdot M \cdot \left(1 + \lceil \log_{B-1} \lceil M/B \rceil \rceil\right) = 2000 \cdot (1 + \lceil \log_{99} \lceil 1000/100 \rceil \rceil) = 4000$ and $S = 2 \cdot N \cdot \left(1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil\right) = 1000 \cdot (1 + \lceil \log_{99} \lceil 500/100 \rceil \rceil) = 2000$.

Hash joins are almost always better than sort-based join algorithms, but there are cases in which sort-based joins would be preferred. This includes queries on non-uniform data, when the data is already sorted on the join key, or when the result needs to be sorted. Good DBMSs will use either or both.