Lecture #11: Joins Algorithms

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Joins

The goal of a good database design is to minimize the amount of information repetition. This is why we compose tables based on normalization theory. Joins are therefore needed to reconstruct original tables.

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, contents of output tuples generated by a join operator varies. It depends on the DBMS's processing model, storage model, and the query itself:

- **Data:** Copy the values for the attributes in the outer and inner tables into tuples put into an intermediate result table just for that operator. The advantage of this approach is that future operators in the query plan never need to go back to the base tables to get more data. The disadvantage is that this requires more memory to materialize the entire tuple.
- **Record Ids:** The DBMS only copies the join keys along with the record ids of the matching tuples. This approach is ideal for column stores because the DBMS does not copy data that is not needed for the query. This is called *late materialization*.

Cost Analysis

The cost metric that we are going to use to analyze the different join algorithms will be the number of disk I/Os used to compute the join. This includes I/Os incurred by reading data from disk as well as writing intermediate data out to disk.

Variables used in this lecture:
- M pages in table R, m tuples total
- N pages in table S, n tuples total

2 Nested Loop Join

At a high-level, this type of join algorithm is comprised of two nested `for` loops that iterate over the tuples in both tables and compares each unique of them. If the tuples match the join predicate, then output 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 can be in terms of the number of tuples or number of pages. The DBMS will also want to buffer as much of the outer table in memory as possible. If possible, leverage an index to find matches in inner table.

Simple Nested Loop Join

For each tuple in the outer table, compare it with each tuple in the inner table. This is the worst case scenario where you assume that there is one disk I/O to read each tuple (i.e., there is no 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. This algorithm performs fewer disk access because we scan the inner table for every outer table block instead of for every tuple.
Cost: $M + (M \times N)$

If the DBMS has B buffers available to compute the join, then it can use $B - 2$ buffers to scan the outer table. It will use one buffer to hold a block from the inner table and one buffer to store the output of the join.
Cost: $M + \left(\left\lceil \frac{M}{B-2} \right\rceil \times N\right)$

Index Nested Loop Join

The previous nested loop join algorithms perform poorly because the DBMS has to do a sequential scan to check for a match in the inner table. But if the database already has an index for one of the tables on the join key, then it can use that to speed up the comparison. The outer table will be the one <u>without</u> an index. The inner table will be the one with the index.

Assume the cost of each index probe is some constant value C per tuple.
Cost: $M + (m \times C)$

3 Sort-Merge Join

The high-level is to sort the two tables on their join key. Then perform a sequential scan on the sorted tables to compute the join. This algorithm is useful if one or both tables are already sorted on join attribute(s).

The worst case scenario for this algorithm is if the join attribute for all the tuples in both tables contain the same value. This is very unlikely to happen in real databases.

- **Phase #1 – Sort:** First sort both input tables on the join attribute.
- **Phase #2 – Merge:** Scan the two sorted tables in parallel, and emit matching tuples.

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

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

Total Cost: Sort + Merge

4 Hash Join

The high-level idea of the hash join algorithm is to use a hash table to split up 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 join can only be used for equi-joins on the complete join key.

If 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 in bucket

s_i. Thus, R tuples in bucket r_i need only to be compared with S tuples in bucket s_i.

Basic Hash Join

- **Phase #1 – Build:** 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.
- **Phase #2 – Probe:** Scan the inner relation and use the hash function h_1 on each tuple to jump to a location in the hash table and find a matching tuple. Since there may be collisions in the hash table, the DBMS will need to examine the original values of the join attribute(s) to determine whether tuples are truly matching.

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, then the join has to use a dynamic hash table or allow for overflow pages.

Grace Hash Join / Hybrid Hash Join

When the tables do not fit on main memory, you do not want the buffer pool manager constantly swapping tables in and out. The Grace Hash Join is an extension of the basic hash join that is also hashes the inner table into partitions that are written out to disk. The name "Grace" comes from GRACE database machine developed during the 1980s in Japan.

- **Phase #1 – Build:** 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, then use *recursive partitioning* with a second hash function h_2 (where $h_1 \neq h_2$) to further divide the bucket.
- **Phase #2 – Probe:** For each bucket level, retrieve the corresponding pages for both outer and inner tables. Then perform a nested loop join on the tuples in those two pages. The pages will fit in memory, so this join operation will be fast.

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