Query Evaluation for Large Databases

--- Goetz Graefe

Summary

The main problems addressed in this paper are

- The effective and efficient algorithms required to access and manipulate large data sets to provide acceptable performance.
- A large variety of query execution techniques must be considered when designing and implementing the query execution module of a new database management system.
- He discussed actual data manipulation methods with algorithms for aggregation and duplicate removal with binary matching operations such as join and intersection.
- He reviewed many dualities between sorting and hashing and points out their differences that have an impact on the performance of algorithms based on either one of these approaches.

- It is Important to survey efficient algorithm and software architectures for query execution engines to execute complex queries over large system.

- All Query execution techniques like:
  1) Algorithms and their execution costs.
  2) Sorting versus hashing.
  3) Parallelism, resource allocation and scheduling issues in complex queries.
  4) Special operations for emerging database application domains must be considered to increase the performance of large database system.

The Main Ideas discussed are

- **One-to-One match Operations**: A group of operators that combine information from two inputs and condensing them using aggregation to provide more reliable shared storage. Ex. – Joins.
- Binary Matching problems occur for object-oriented databases, which can be overcomed by conventional join strategies.

- He discussed 4 types of Algorithms for binary matching.
  1. **Nested-Loop Join Algorithm**: Simplest and direct algorithm. For each item in one input, scan the entire other input and find matches. It can compute Cartesian product and join with arbitrary two-relation comparison predicate. Doesn't work well for large inputs because the inner input is scanned very often. The improvement options are the scan of inner input can be terminated after the first match for an item in other input, instead of scanning the inner input once for each item from other input, the inner input can be scanned once for each page of the other input (Block nested loops join) and filling all memory except K pages with pages of other input. Index Nested Loops join creates a temporary or permanent index on inner input's join attribute to replace file scans by index lookups. They can work fast if one of the inputs is small and other indexed input is large that the number of index is smaller than number of pages in larger input. We can even use two ordered indices like B-trees to switch roles of inner and outer join after each index look up.
  2. **Merge-Join Algorithm**: Needs both inputs to be sorted on join attribute. Value packets are items with equal join attribute values, which are used to reduce I/O operations. Interesting Orderings are used where it is not required to sort intermediate results for later merge-joins. Heap-Filter merge-join is combination of nested-loops join and merge-joins. Hybrid Join combines elements from index nested loops join, merge join and sorted lists of index leaf entries.
  3. **Hash Join Algorithm**: Based on building an in-memory hash tables on one smaller input and then probing this hash table to use items from the other input. They require overflow avoidance for larger build inputs. The build and probe inputs are partitioned and the final join can be obtained by concatenating the join results of pairs of partitioning files. Recursive partitioning is required if a build partition file is larger than memory. The I/O cost for Hash join algorithm can be determined by levels without hash table and the fraction of input remaining in the memory in deepest recursion level.
  4. **Pointer-Based Joins**: Pointers are links between data items which represent a limited form of pre computed results similar to indices and join indices. The nested –loop join based on pointer-based join scans through one input and retrieves the appropriate other input tuple for each tuple in input. This doesn't work well with large data sets. The Merge-join based on pointer join sorts the input on pointer address value and then retrieves all other input values in one page over the disk. The Hybrid

join based on pointer join partitions only input on pointer values and ensures that the input tuples with other input pointers to same page are brought together, and then pages and tuples of other input. This works faster than other two pointer based joins. Difficult to maintain and outperform the standard value based join algorithms.

- **Performance Comparison**: The nested-loop join is not suitable for medium and large size relations because the cost is proportional to the size of Cartesian product of two inputs. The Merge-join and Hash join algorithms are logarithmic and Hash join is faster than merge based hash join because the merge levels are determined individually for each file. Pointer Joins can be efficient or inefficient.

- He discussed the similarities and dualities of Sort and Hash based algorithms for better understanding and of two approaches and trade offs.

- Both approaches permit in-memory versions for small data sets and disk based versions for larger data sets. But the sorting algorithm uses quick sort and hashing algorithm uses classical hashing. The Divide & Conquer paradigm is similar for both algorithms. In Hash based, large inputs are partitioned where as in sort based they use single-level merge.

- He mentioned the dualities between both algorithms based on different other aspects like I/O patterns, Large Inputs, Optimizations, Resource Sharing, Interested Orderings, Grouping & Aggregation etc.

- The choice between both algorithms is not determined by input sizes or memory size; But It should be governed by sizes of two inputs into binary operators relative to each other. The danger of performance impairments due to skewed data and hash value distributions.

Question: Depending on the input size and cost of implementation, we choose any of these algorithms. So, do we have all these algorithms implemented in present day DBMS or we have a different DBMS implementing Different algorithms?

Submitted By	**Submitted To**
Lakshman Madhav Kollipara	Arash Termehchy
SID- 932655504	CS540- Database Management Systems