Query Evaluation Techniques for large DB

By: David Lee

(modified slides by Daniela Stasa and Rachel Pottinger)

Purpose

- To analyze practical query evaluation techniques including execution of complex query evaluation plans and efficient algorithms in large databases

Discussion: Why not more DBs?

- On the first page, the author states that DBMSs have not been used for two reasons. 1. application development and maintenance is difficult. 2. the data in those areas is SO big, that speed trumps all, and people would rather hand-code. Why do *you* think databases aren't used more? Why don't you use them on *your* data?

Steps

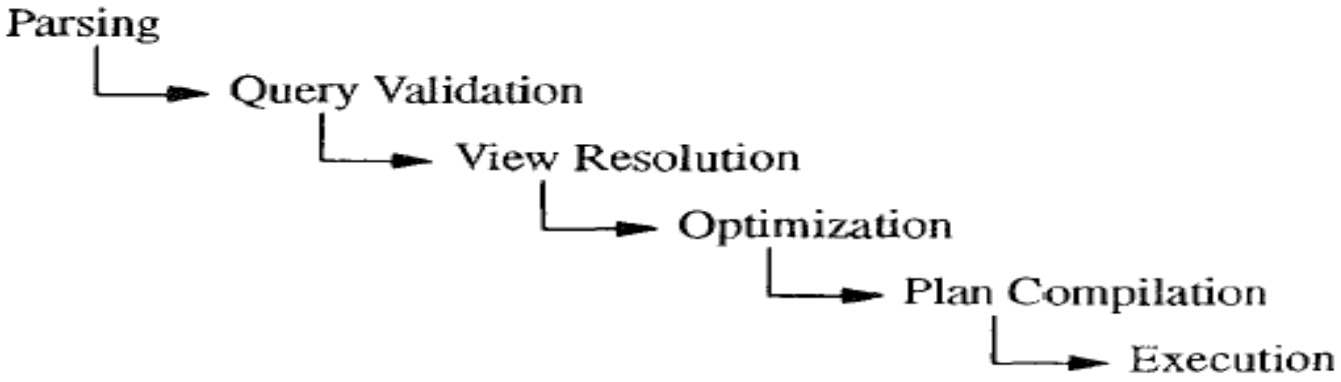


Figure 2. Query processing steps.

- Parses then validates an SQL query to a query tree in logical algebra (ie relational algebra)
- Optimizer translates the query tree in logical algebra to an optimized physical plan (QEP) with minimum cost
- Optimal physical plan is prepared for execution and compiled into machine code
- query execution engine executes the plan

Query execution engine

- What is it?
  - Collection of query execution operators and mechanisms for operator communication and synchronization
  - Query execution engine runs an optimal plan chosen by the query optimizer
  - Pipelining is the parallel execution of different operators in a single query.

Some of the techniques discussed

- Algorithms and their execution costs
- Sorting versus hashing
- Parallelism
- Resource allocation
- Scheduling issues
- Performance-enhancement techniques
- And more …

Some notes

- On the context
  - While many of the techniques were developed for relational database systems most are applicable to any data mode that allows queries over sets and lists.
- Type of queries
  - Discusses only read-only queries but mostly applicable to updates.

Architecture of query execution engines

- Focus on useful mechanisms for processing sets of items ie:
  - Records
  - Tuples
  - Entities
  - Objects

Physical Algebra

- Taken as a whole, the query processing algorithms form an algebra which we call physical algebra of a database system

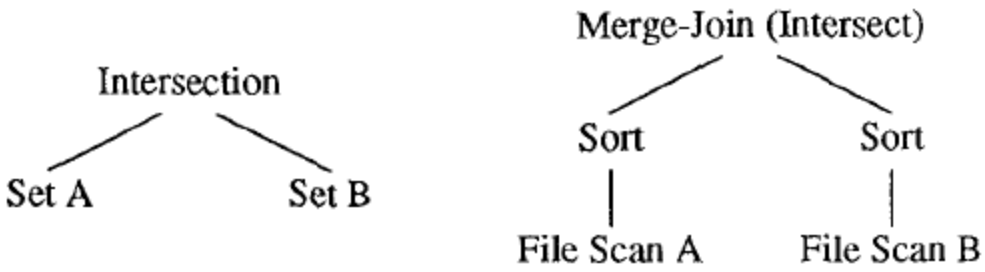


Figure 3. Logical and physical algebra expressions.

Physical vs. Logical Algebra

- Equivalent but different
- Logical algebra: related to data model and defines what queries can be expressed in data model (ie: relational algebra)
- Physical algebra: system specific
  - Different systems may implement the same data model and the same logical algebra but may use different physical algebras

Physical vs. Logical Algebra

- Specific algorithms and therefore cost functions are associated only with physical operators not logical algebra operators
- Mapping logical to physical non–trivial:
  - Logical and physical operators not directly mapped
    - Sort algorithms not represented in logical algebra
  - Logical algebra joins are intersect and union whereas physical algebra operators are nested loop or hash join
  - etc

Sorting & Hashing

- The purpose of many query-processing algorithms is to perform some kind of matching,
  - i.e., bringing items that are “alike” together and performing some operation on them.
- There are two basic approaches used for this purpose:
  - sorting
  - and hashing.
- These are the basis for many join algorithms

Sorting

- All sorting in databases uses some kind of merge joining
  - i.e. sort a small set and keep merging it into larger and larger sets until there are no more sets left
- If a set can fit into main memory, quicksort() is used

Design Issues

- Sorting should be implemented as an **iterator**
  - In order to ensure that **sort module** interfaces well with the other operators, (e.g., file scan or merge-join).
- Input to the sort module must be an iterator, and sort uses open, next, and close procedures to request its input
  - therefore, sort input can come from a scan or a complex query plan, and sort operator can be inserted into a query plan at any place or at several places.

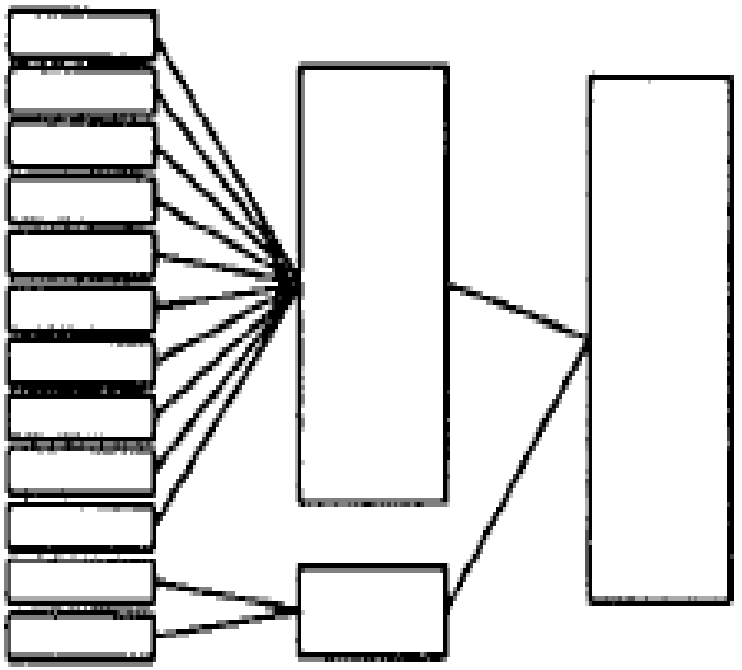
More on Sorting

- For sorting large data sets there are two distinct sub-algorithm :
  - One for sorting within main memory
  - One for managing subsets of the data set on the disk.

- QS and MS use divide and conquer.
  - MS divides physically, then merges
  - QS divides on logical keys, then combines

Level 0 run

- There are two alternative methods for creating initial runs
  - In-memory sort algorithm (usually quick sort)
  - Replacement Selection (aka heapsort)



Quick Sort vs. Replacement Selection (aka HeapSort)

- Run files in RS are typically larger than memory ,as oppose to QS where they are the size of the memory
- Qs results in burst of reads and writes for entire memory loads from the input file to initial run files while RS alternates between individual read and write
- In RS memory management is more complex
- The advantage of having fewer runs must be balanced with the different I/0 pattern and the disadvantage of more complex memory management.

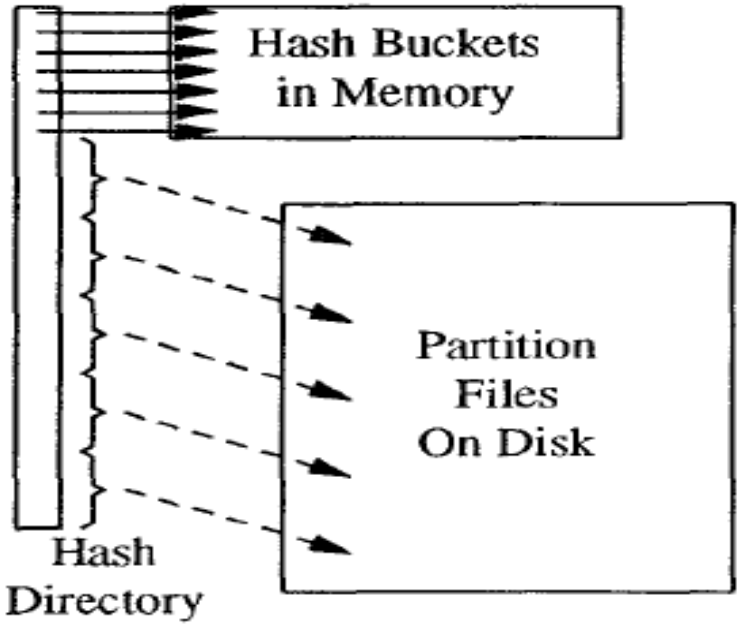
Hashing

- Alternative to sorting
- Expected complexity of hashing algorithms is O(N) rather than O(N log N) as for sorting.
- Hash-based query processing algorithms use an in-memory hash table of database objects to perform their matching task.

Hashing Overflow

- When hash table is larger than memory, hash table overflow occurs and must be dealt with. *Avoidance or Resolution*
- Input divided into multiple partition files such that partitions can be processed independently from one another,
- Concatenation of results of all partitions is the result of the entire operation.

Hash overflow



Associative Access Using Indices

- Goal:
  - To reduce the number of accesses to secondary storage
- How?
  - By employing associative search techniques in the form of indices
  - Indices map key or attribute values to locator information with which database objects can be retrieved. (use of B trees)
  - There are clustered (sparse or dense) and non clustered (must be dense)

Buffer Management

- Goal: reduce I/O cost by cashing data in an I/O buffer.
- Issues
  - Recovery
  - Replacement policy
  - performance effect of buffer allocation
  - Interactions of index retrieval and buffer management
- Implementation
  - Interface provided : fixing (fixed page not subject to replacement) and unfixing
- More on Wednesday

Discussion: DB vs OS

- Many of the topics handled in DBs are also handled in OSs. Sometimes people (e.g., Microsoft) have tried combining the two. Do you think this is a good idea? Why or why not?

BINARY MATCHING OPERATIONS

- Relational join most prominent binary matching operation (others: intersection, union, etc)
- Set operations such as intersection and difference needed for any data model
- Most commercial db systems as of 1993 used only nested loops and merge-join. As per research done for SystemR, these two were supposed to be most efficient.
- SystemR researchers did not consider Hash join algorithms, which are today considered even better in performance.

NESTED-LOOPS JOIN ALGORITHMS: simple elegance

- For each item in one input, scan entire other input to find matches.
- Performance is really poor, because inner input is scanned often.
- Tricks to improve performance include:
  - Use K pages of outer relation and Mem – K pages of inner relation
  - create an index on the join attribute
  - Inner input can be scanned once for each ‘page’ of outer input.

MERGE-JOIN ALGORITHMS

- First sort relations by join attribute
  - So linear scans will encounter join attribute sets at the same time
- Uses QS to sort or can use interesting orderings (if already exists)

MERGE-JOIN VARIANTS

- Heap-Filter merge-join
  - Combination of nested loop join and merge join. # of scans is about 50% of block nested loops
- Hybrid join (used by IBM for DB2), uses elements from index nested-loop joins and merge join, and techniques joining sorted lists on index leaf entries.

HASH JOIN ALGORITHMS

- based on in-memory hash table on the smaller relation (‘build’ input), then scan the larger relation to find matching rows by probing in the hash table (‘probe’
- Very effective if build input fits into memory, regardless of size of probe input.
- overflow avoidance or resolution methods needed for inputs that are larger than memory.
- both inputs partitioned using same partitioning function. Final join result formed by concatenating join results of pairs of partitioning files.
- Recursive partitioning may be used for both inputs
- More effective when the two input sizes are very different (smaller being the build input).

Duality of sort and hash-based algorithms

- Equivalent but uses dividing and merging in different ways
  - Sort
    - Divides data by physical step (mem) and combines via logical step (merging)
  - Hash
    - Divides by logical rule (hash) and combines by physical step (concatenating subsets)
  - Can be seen by observing the disc arm I/O operations for merging and partitioning