Query Evaluation Techniques for large DB

Part 1

Fact:
- While data base management systems are standard tools in business data processing they are slowly being introduced to all the other emerging data base application areas
- Why?

Problems
- The restrictive nature of data definition and manipulation language makes can make application development and maintenance cumbersome
- Data volume can be so large that the performance is more important that higher level of abstraction or programmer productivity achieved by DBMS

Something to think about
- Why do you think databases aren't used more?
- How do database systems' needs for dealing with large amounts of data differ from other applications?
- How big do you think data has to be before it's large enough to bother with all of these concerns?

Purpose
- To survey efficient algorithms and software architectures of *query execution engines* for executing *complex queries* over *large databases*

Vocabulary
- "Complex" query is *one that requires a number of query processing algorithms to work together*
- "Large" database uses files with sizes from several megabytes to many terabytes
Query execution engine
- What is it?
  - It is a collection of query execution operators and mechanisms for operator communication and synchronization
  - The facilities of the query execution engine define the space of possible plans that can be chosen by the query optimizer.

Steps
- Translates the logical query from SQL into a query tree in logical algebra.
- The query tree in logical algebra is translated into a physical plan.
- The optimizer then expands the search space and finds the best plan.
- The optimal physical plan is copied out of optimizer's memory structure and sent to the query execution engine.
- The query execution engine executes the plan using the relations in the stored database as input, and produces an output.

Some of the techniques discussed
- Discusses a large variety of query execution techniques that must be considered when designing and implementing the query execution module of a new database management system
  - 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 have been developed in the context of the relational database systems most of them are applicable to any data model, provided that the data model permits queries over bulk data types such as sets and lists.
- Type of queries
  - Discusses only read-only queries but most of the techniques applicable to update requests.

Architecture of query execution engines
- Focus on useful mechanisms for processing sets of items
  - Records
  - Tuples
  - Entities
  - Objects
Query processing algorithms

- Think of them as algebra operators consuming 0 or more inputs and producing one (some time more) outputs.
- Query execution engine consist of a collection of operators and mechanisms to execute complex expressions using multiple operator, including occurrences of the same operator.

Physical Algebra

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

Physical vs. Logical Algebra

- Equivalent but different
- Logical algebra is related to data model and defines what queries can be expressed in data model
- Physical algebra is 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
- Must map from logical to physical
  - (not trivial) why?
    - It involves algorithm choices
    - Logical and physical operators not directly mapped

Physical vs. Logical Algebra

- Example of not directly mapping
  - Some operators in physical algebra may implement multiple logical operators
  - Some physical operators implement only part of a logical one
  - Some physical operators doesn’t exist in the logical algebra
  - Some properties that hold for logical operators do not hold for their physical counterparts

Issues

- Synchronization and data transfer between operators main issue
  - In a query with 2 joins how are the results of the first join passed to the second one?
  - Create & read temporary file
  - One process for each join, use inter process communication mechanisms to transfer data between operators
  - Writing rules based translation programs
consider a bit less on differences
Daniela, 1/30/2006
Another alternative

- Implement operators in such a way that they schedule each other within a single process in operating system.
  - Similar to procedure call
  - Cheaper than inter-process communication

Iterators

- Two important features of operators
  - Can be combined into arbitrarily complex evaluation plans
  - Any number of operators can schedule and execute each other in a single process without assistant from underlying OS

Implementation issues

- Prepare an operator for producing data
  - Open
- Produce an item
  - next
- Perform final housekeeping
  - close

Observations

- The entire query plan is executed within a single process
- Operators produce an item at a time on request
- Items never wait in a temporary file or buffer (pipelining)
- Efficient in time-space-product memory cost
- Iterators can schedule any type of trees including bushy trees
- No operator is affected by the complexity of the whole plan

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 (see Sukesh)

Design Issues

- Sorting should be implemented as an iterator
  - In order to ensure that the sort module interfaces well with the other operators, (e.g., file scan or merge-join).
  - The 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 the sort operator can be inserted into a query plan at any place or at several places.
why are they comparable?

Daniela, 1/30/2006
More on Sorting

- For sorting large data sets there are two distinct sub-algorithms:
  - One for sorting within main memory
  - One for managing subsets of the data set on the disk.
- For practical reasons, e.g., ensuring that a run fits into main memory, the disk management algorithm typically uses physical dividing and logical combining (merging).
- A point of practical importance is the fan-in or degree of merging, but this is a parameter rather than a defining algorithm property.

Quick Sort vs. Replacement Selection

- Run files in RS are typically larger than memory, as opposed 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/O pattern and the disadvantage of more complex memory management.

Consideration to improve merge efficiency

- First, scans are faster if read-ahead and write-behind are used.
- The use of large cluster sizes for the run files (if permitted by the OS) very beneficial
- The number of runs $W$, is typically not a power of $F$; therefore, some merges proceed with fewer than $F$ inputs.

Merging parameters

- The maximal merge fan-in $F = \lceil M/C - 1 \rceil$.
  - $M$- size of the memory
  - $C$- size of a cluster, the unit of I/O
- The number of merge level $L = \lceil \log_f(W) \rceil$
  - $W$-number of level-0 run files

Consideration to improve merge efficiency

- Since some operations require multiple sorted inputs, for example merge-join and sort output can be passed directly from the final merge into the next operation memory must be divided among multiple final merges. Thus, the final fan-in $f$ and the "normal" fan-in $F$ should be specified separately in an actual sort implementation.
- Example: Two merge-join inputs are 1 MB and 9 MB, and if 20 clusters are available for inputs into the two final merges, then 2 clusters should be allocated for the first and 18 clusters for the second input (1/9=2/18).
D3  delete?
Daniela, 1/30/2006

D4  delete?
Daniela, 1/30/2006

D7  pipelining
Daniela, 1/30/2006
Hashing

- An alternative to sorting
- Hashing should be considered because the expected complexity of set algorithms based on hashing 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.

Problems

- When hash table is larger than memory, hash table overflow occurs and must be dealt with.
- There are basically two methods:
  - avoidance
  - resolution.

Solutions

- In either case, the strategy is:
  - the input is divided into multiple partition files such that partitions can be processed independently from one another,
  - the concatenation of the results of all partitions is the result of the entire operation.

Avoidance

- The input set is partitioned into $F$ partition files
  - $F = \lceil \frac{M}{C} - 1 \rceil$
- If it turns out that fewer partitions than have been created would have been sufficient to obtain partition files that will fit into memory, bucket tuning (collapsing multiple small buckets into larger ones) and dynamic destaging (determining which buckets should stay in memory) can improve the performance of hash-based operations

Resolution

- start with the assumption that overflow will not occur, but resort to basically the same set of mechanisms as hash table overflow avoidance once it does occur. No real system uses this naive hash table overflow resolution because so-called hybrid hashing is as efficient but more flexible.

Hybrid

- Hybrid hash algorithms start out with the (optimistic) premise that no overflow will occur;
  - if it does, however, they partition the input into multiple partitions of which only one is written immediately to temporary files on disk. The other $F - 1$ partitions remain in memory.
  - If another overflow occurs, another partition is written to disk. If necessary, all $F$ partitions are written to disk.
- Thus, hybrid hash algorithms use all available memory for in-memory processing, but at the same time are able to process large input files by overflow resolution,
Hash overflow

Example
- Input \( R = 240 \) pages
- Memory \( M = 80 \) pages
- Cluster \( C = 8 \) pages
- \( F = \frac{80}{8} - 1 = 9 \)
- Number of partitions files
  - \( K = \frac{240 - (80 - 8)}{80 - 8} = 3 \)
  - \( K \times C = 24 \) pages will be used as output buffers

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.

Some Index Structures:
- May be used:
  - as primary structures to store actual data
  - As redundant structures that do not contain data but pointers to the actual data items in a separate data file
- Clustered & Un-clustered
  - The order or organization of index entries determines the order of items in the data file.
- Sparse & Dense
  - Sparse: indices do not contain an entry for each data item in the primary file, but only one entry for each page of the primary file;
  - Dense: there are the same number of entries in the index as there are items in the primary file.
  - Non-clustering indices must always be dense.

Separating index scan and record lookup.
- Advantages to separating index scan and record lookup:
  - First, it is possible to scan an index only without ever retrieving records from the underlying data file.
  - Second, even if none of the existing indexes is sufficient by itself, multiple indices may be "joined" on equal RIDs to obtain all attributes required for a query
  - Third, if two or more indices apply to individual clauses of a query, it may be more effective to take the union or intersection of RID lists obtained from two index scans than using only one index
  - Fourth, joining two tables can be accomplished by joining the indices on the two join attributes followed by record retrievals in the two underlying data sets
  - Fifth, for non-clustering indices, sets of RIDs can be sorted by physical location, and the records can be retrieved very efficiently, reducing substantially the number of disk seeks and their seek distances.

Buffer Management
- Goal: To reduce I/O cost by cashing data in an I/O buffer.
- Design Issues
  - Recovery
  - Replacement policy
  - Performance effect of buffer allocation
  - The interactions of index retrieval and buffer management
- Implementation Issues
  - Interface provided: fixing–unfixing
  - Intermediate results kept in a separate buffer
Slide 38

D9  delete?
Daniela, 1/30/2006

Slide 41

D11  consider skipping
Daniela, 1/30/2006
Summary

- So far we discussed some of the architectures, hash and Sorting and disc access
- Sukesh will take from here

Let's gather

- There are many issues that could be covered by either the OS or the database. Break into groups and discuss some of these issues. For each issue, what are the pros and the cons of handling it in the database?

Binary Matching Operations

- Relational join most prominent binary matching operation. Others include left and right semi-joins, left and right anti-semi-joins, symmetric anti-join, intersection, union, left and right differences, and symmetric or anti-difference.
- Set operations such as intersection and difference will be used for any data model, be it an OO implementation, or a RDBMS.
- Most commercial db systems today (June of '93) use 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

- It is the most direct algorithm for binary matching. For each item in one input, scan the entire other input to find matches. Its main advantage is its simplicity.
- Performance is really poor, because the inner input is scanned often.
- To improve performance:
  - for one-to-one match operations, where a single match carries all necessary information, a scan of the inner input can be stopped after the first match.
  - inner input can be scanned once for each page of the outer input.
  - scan inner input alternating back and forward, thus reusing the last page of the previous scan.
  - larger input should be the outer one.
  - if possible, use an index on the attribute to be matched in the inner input.
  - hash indices are the fastest for exact match queries.

Merge-Join Algorithms

- It requires that both inputs are sorted on the join attribute (similar to merge process used in sorting).
- Some systems include the notion of 'value packet', meaning all items with equal join attribute values. An iterator's next call returns a value packet.
- Interesting Orderings in the SystemR query optimizer:
  - Since set operations can be evaluated using any sort order, as long as the same sort order is present in both inputs, the effect of interesting orderings can be seen in a one-to-one match operators based on merge-join.
  - 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 one input (smaller one, called 'build input'), and probing this table using items from the other input (called 'probe input').
- works very fast, if the build input fits into memory, regardless of the size of the probe input.
- overflow avoidance methods needed for larger build inputs.
- both inputs are partitioned using same partitioning function. The final join result can be formed by concatenating the join results of pairs of partitioning files.
- Recursive partitioning may be used for both inputs, if very large in size.
- More effective when the two input sizes are very different (smaller being the build input).

DUALITY OF SORT AND HASH BASED JOIN ALGORITHMS - TRADEOFFS

- Both cut the data if it does not fit into memory. Sort-join uses a 'physical' rule to split large data inputs, and later a 'logical' rule to combine them, whereas Hash algorithms use a 'logical' rule to split and 'physical' to recompose.
- 'fan-in' in Sort based algorithms, and 'fan-out' in Hash based, both depend on memory size limitation.
- Merging can be done at multiple levels, and partitioning can be recursive. In merging, each level grows by factor of fan-in, and in partitioning the factor of fan-out. Memory not used optimally in the naïve versions of the algorithms.
- When data fits in memory, disk I/O can be avoided: lead to bucket tuning, and histogram-driven recursive hybrid Hash algorithms.
- For Hash based, only one input resides in memory (advantage over Sort-based).
- Partitioning Skew for Hashing, and optimized merging for Sort-based algorithms needed.

CONCLUSION:

The choice of Hash based or Sort based should be based on relative sizes of inputs and the danger of performance loss due to skewed data or hash value distribution.

- Does the number of generally used joins seem large or small to you? Why?
- Are you surprised by any of the joins that are used?

DUALITY OF SORT AND HASH BASED JOIN ALGORITHMS - TRADEOFFS

- consider interesting orderings (from SystemR): e.g., multiple merge-joins on the same attribute can be performed without sorting intermediate results (Union and intersection of sets can use this).
- Hash based algorithms produce output in unsorted format… during partitioning the equality of attributes compared needs to be used instead, during partitioning.

EVALUATION OF COMPLEX QUERY PLANS

- Extensive pipelining considered, e.g. hybrid hash joins, bushy trees etc.
  - Concerns for binary matching iterators to be used in bush plans:
  - when one subplan stops for disk I/O, another one gets going.
  - binary match operations, especially hybrid hash joins, should be paranthesized. This will permit the optimizer to force a stop point.
  - binary operator implementation should include a switch that controls which subplan is initiated first.
  - if multiple operators are active concurrently, memory division should be proportionable.
  - improvements in disk I/O performance should be considered (as it evolves).
  - multiprocessor scheduling to be explored (as it evolves).
SUMMARY

- We have surveyed execution algorithms
- Interest growing in extensible and OO systems database
- New systems will use efficient and parallel algorithms and perform better than present day RDBMS systems do
- Precomputation and compression will provide performance benefits with large data volumes
- Concept of 'fixed number of parameterized operators' will help meet new challenges
- Automatic optimization and parallelization needed to enhance performance

To wrap it up

- Do you think that in the time since then the issues would have gotten better, because memories have gotten larger, or worse because there is a bigger gap between the time it takes to access memory and the time to access things on disk?