Query Evaluation

Doing what we're told to do.

Hoyt Koepke

Purpose of the Paper

- Survey efficient algorithms and software architectures for database query execution.
  - How database systems handle complex queries.
    - “Complex” means that it requires a number of algorithms to work together.
  - How database systems handle queries into large amounts of data.
    - Large can be arbitrarily large.
    - Practically, it means that some or most of the data retrieved cannot fit into main memory.
- Limited to read-only queries
  - Had to keep the paper to a reasonable length.

Outline

- Structure of the query evaluation engine.
  - Takes logical operators from the query optimizer and translates them into efficient physical operations.
- Discussion Time.
- The internals of the engine.
  - The primary operations (at least those in our reading).
  - Sorting, hashing, disk access, joins.
- More Discussion Time.
- Class over.
- Lunch on your own.

Outline

- Structure of the query evaluation engine.
  - Takes logical operators from the query optimizer and translates them into efficient physical operations.
- Discussion Time.
- The internals of the engine.
  - The primary operations (at least those in our reading).
  - Sorting, hashing, disk access, joins.
- More Discussion Time.
- Class over.
- Lunch on your own.

Structure of a Query Evaluation Engine

Picks up where the Query Optimizer leaves off.

Hoyt Koepke

Input From Query Optimizer

- Query optimizer produces plans of execution (which works with logical operators).
  - Logical operators provide higher level description of the logical flow of execution.
    - E.g. what to join to what, ordering, predicates, etc.
    - Do not provide the exact details of how to implement these operations.
  - These are optimized at a high level given available information.
    - As we've discussed in class, of course.
    - Plus: The search space is bounded in part by the capabilities of the query execution engine.
- The Execution Engine Turns these Logical Operators into Physical Operations.

Physical Operations

- Implementation of the Logical Operators.
  - There's more than one way to skin a set of cats.
- Goal: Find the most efficient way of implementing the lower-level operations.
  - Execution engine closely takes into account the available resources (like available memory).
  - For any given task, it chooses from variety of different algorithms – all ending up with the same result but each better under certain conditions.

Physical Algebra

- Query processing algorithms form an algebra called the physical algebra of a database system.
  - Take zero or more sets or sequences as inputs
  - Output one or more sets or sequences as an output.
  - Why call it a type of algebra?
    - Allows for precise definition and logical equivalence.
    - Because it really is just math…
- The logical algebra used by the query optimizer does not translate exactly into the physical algebra.
  - Several physical operators for one logical operator.
  - Varying representations of the data.

Discussion

Hoyt Koepke

Outline

- Structure of the query evaluation engine.
  - Takes logical operators from the query optimizer and translates them into efficient physical operations.
- Discussion Time.
- The internals of the engine.
  - The primary operations (at least those in our reading).
  - Sorting, hashing, disk access, joins.
- More Discussion Time.
- Class over.
- Lunch on your own.

The Operations

All the fun stuff.
At least for some people.

Hoyt Koepke

The Operations

- Sorting
  - Usually use variants of divide-and-conquer algorithms such as mergesort or quicksort.
- Hashing
  - Used to reduce IO requirements in matching operations.
- Disk Access
  - Physically reading in the data; takes into account cluster size of the disk,
- Joins
  - Techniques include nested-loops, merge-joins, hash-joins, etc.

Iterators

- The algorithms are implemented as iterators.
  - Essentially a way to work with individual elements while needing to use tables and sets.
  - Each iterator calls another iterator to get the input that it needs.
    - In this way, they "schedule" each other.
  - These iterators store use "granules" to store the intermediate results.
    - These are structures that allow iterators to return outputs that are part of the result of an operation done over the entire table,
    - E.g. returning a table in sorted order.
- Handy way to organize the algorithms efficiently.

Sorting

- For sorting large data sets there are two algorithmic variations, both based on divide and conquer:
  - Mergesort, which manages subsets of the data on the disk.
  - Quicksort, which sorts logically in memory then combines the results physically on disk.
  - Each of these is implemented as an iterator.
- The deciding issue is how much of the data can fit into main memory.
- When it doesn't, it's complicated. There's lots of parameters to be adjusted.
  - A point of practical importance is the fan-in (how many sub-runs or elements each run combines).

Hashing

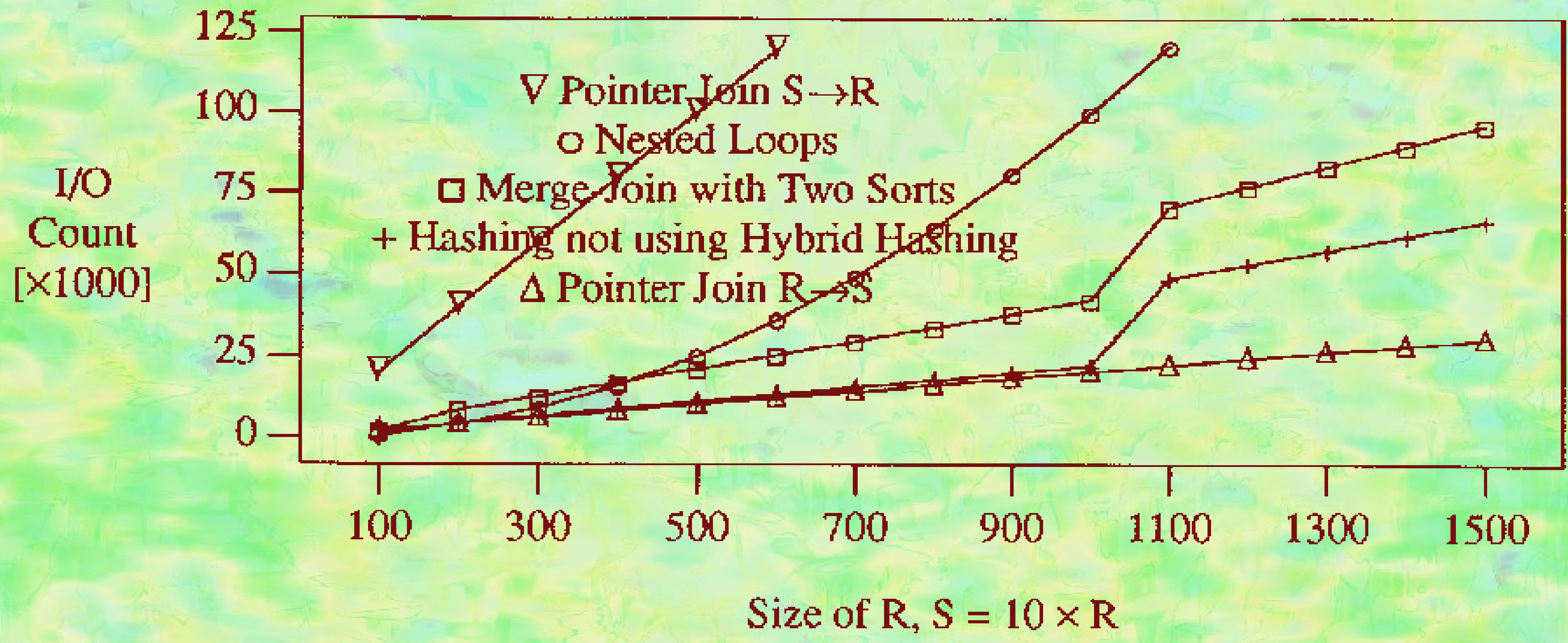
- Idea: Create a low-memory hash of the data, then work with that instead of the actual data.
  - Why? It's much more efficient at utilizing memory.
  - Sufficient for all matching operations (as long as the hash function is non-intersecting).
- Still may have to deal with memory management.
  - Solution: Hybrid-hashing, which pages sections of the hash data to disk when the memory is full.
    - Still utilizes all available memory.
- Hash Key: For some operations, a unique key on the data will work if it's involved in the matching.

Disk Access

- Most of the issues here are inherent to all disk management issues, so I won't go into detail.
- Main database-specific issue is the use of indexes.
  - Not only does this provide excellent lookup speed, it also reduces greatly the amount and spread of disk accessing.
  - Forms of indexes:
    - The most popular form (at time of writing) was a B-tree or variations of it (such as B+ or B*).
    - One database system (Postgres) uses R-trees for multidimensional indexing.
- Databases can also use a custom IO buffer to localize IO operations on the disk.

Joining

- Essentially three methods: Nested-loop Joins, Merge-based, & hash-based join algorithms.
- Performance of each varies depending on the number of inputs.



Types of Join

- Inner Join on Tables A and B
  - Returns a table with attributes from both A and B, where the values of an attribute of A equal those in one of B.
- Left Outer Join on Tables A and B
  - Same as Inner Join, except that all entries of A are included in the resulting table and the corresponding attributes of B are null if no match exists.
- Right Outer Join on Tables A and B
  - Same as left outer join, except that all the entries of B are included and the unmatched entries of A are excluded.
- Left Semi-Join on Tables A and B
  - Same as Inner Join except that only the attributes of A are included in the returned table.

Executing a Join: Nested Loop

- Nested Loop Join:
  - Simple – for each new tuple in the outer table, scan all the inner tables looking for a match.
  - Works fine for small queries.
  - Quickly becomes inefficient ($O(n^2)$). Reason is that the tuples included at each stage are scanned again by all subsequent stages.
  - But it's not as clear cut as this – IO and memory use can be as much a performance measure as algorithmic speed.
    - This method requires relatively little memory use.
  - Some techniques speed this up.
    - Block Nested Loops scan the inner table once per block of tuples in the outer loop by using small mappings.

Executing a Join: Merge Join

- Faster than Nested Loop - O(n log n)
- Similar idea to merge-sort.
  - Requires that both input tables be sorted on the joining attribute.
  - The conditions of the join are implemented as the sorted tables are merged into one table.
- Advantages:
  - Much Faster.
- Disadvantages:
  - Requires tables to be sorted.
  - Much higher demands on memory.
  - Complicated to implement.

Executing a Join: Hash-Join

- Tries to use memory efficiently.
- Forms an in-memory hash table of the “build” input and then probes the existence of items in the “probe” input using the item's hash.
- Optimal performance means balancing the size of the build input and probe input.
  - Can be very efficient if the build input fits into memory.
  - Uses recursive partitioning to break the problem down into smaller parts in order ensure the build input fits in memory.
- Along with nested-loop and merge-join, form the join operation toolkit in most modern query engines.

More Discussion

Hoyt Koepke