Overview of Database Operation Implementation

Deb VanderMeer

January 24, 2000

1 Moving from a Logical Algebra to a Physical Algebra

A logical algebra (e.g., relational algebra) describes a query mathematically. A physical algebra describes what operations must be performed to implement the query in a database system.

The typical operations needed are:

1. Sorting
2. Hashing
3. Select
4. Join
5. Universal Quantification
6. Aggregation

2 Sorting Methods

External Sorting methods are required for the Sort-Merge join algorithm and for sorting output, e.g., for ORDER BY. These methods are suitable for sorting database files that are larger than available memory.

2.1 Sort-Merge strategy

1. **Sorting Phase**: Sort chunks of the size of available memory using quicksort.
2. **Merging Phase**: Merge sorted buckets pairwise, then merge the resulting (sorted) buckets pairwise until all buckets are merged.

A major problem with sorting – it requires that intermediate results be materialized.

3 Hashing

Hashing is often used for matching, as in join processing. Hashing typically assumes that the hash function results in buckets of roughly equal size. The biggest problem with hashing is overflow – this happens when a hash bucket becomes larger than available memory. There are two methods of dealing with overflow:

1. Avoidance – Choose a fanout (i.e., number of buckets) based on the size of available memory and block size (cluster size in the paper) that will very likely avoid overflow. After hashing, tune buckets by combining those that will fit together in memory.

2. Resolution – When overflow occurs, partition the bucket into two buckets, and write one to disk. If the bucket in memory overflows again, repeat.

4 Selection

Selection requires finding the items that meet the specifications in the query. This can alternatively be thought of as a filter, in which records not meeting the criteria are filtered out.

Search methods for simple selection (i.e., one condition), point queries:

- **Linear Search**: Retrieve every record in the file and test whether value(s) meet selection condition.
- **Binary Search**: Works if the table is sorted by the selection attribute. Rarely used – if the table is sorted by the selection attribute, there is generally an index.
- **Primary Index**: Simply use the index.

Search methods for simple selection (i.e., one condition), range queries:

- **Primary Index**: Find the record satisfying the corresponding equality condition, and return records before it (<) or after it (for >).
- **Clustering Index**: Works for equality queries – simply return all records pointed to by the index.
- **Secondary Index**: i.e., B+ tree. Works for equality and inequality queries.

Search methods for conjunctive selections (i.e., conditions separated by AND):

- **Using a Single Index**: Return records using one of the simple selections methods (other than Linear Search), and then check whether each record returned meets the other conditions.
- **Using a Conjunctive Index**: Requires that all attributes in the selection be indexed by a composite index.
- **Using Intersection of RIDs**: If the attributes involved are indexed, and the indexes use record pointers rather than block pointers, we can find the intersection of the RIDs and access the records directly.

For disjunctive selections (i.e., separated by OR), little optimization is possible – typically, a linear scan of the entire file is needed.

5 Join Methods

- **Nested Loop**: Fill all available memory (saving two blocks for the other table and output) with records from one table. Bring in one block at a time of the other table for comparison. For lowest cost, use the smaller relation as the one that is loaded over and over (the one that occupies a single block at a time). nb – Sham's book says to use the smaller file as the outer loop file, and Graefe says to use the larger input file. guess "inner" and "outer" sorta depend on your point of view....
- **Merge**: Requires that both input files be sorted on the join attribute. Once sorted, merging requires a linear scan of each file.
- **Hash**: Build a hash table on the smaller input (called the build input), and probe it using items from the larger file. If the hash table, need to perform recursive partitioning on both files. See page 108 of Graefe.
- **Pointer-Based**: Maintain pointers between matching records in tables. Makes joins faster, but requires maintenance.

See page 110 of Graefe for join method comparison chart.

6 Universal Quantification

The notorious "all" thing. Typically implemented efficiently with a hash-based approach rather than a sort-based approach.

7 Aggregation

Two types:

1. Scalar Aggregates – produces a single value (sum, max, min, count). We can typically exploit indexes for some (max, min, count), and use a single pass over the data for others (sum, avg)
2. Aggregate Functions – produce a set of values (group by).

Aggregation is very similar to duplicate removal – items must be brought together and compared. Three algorithms for aggregation and duplicate removal:

- **Nested Loop**: For each record, compare it to entire set of accumulated output so far. Typically very inefficient.
- **Sorting**: Brings equivalent items together – makes duplicate removal easy. Also good for group-bys.
- **Hashing**: Groups equivalent records in the same bucket, makes duplicate removal easy. Also good for group-bys.