Evaluation of Relational Operations: Other Techniques

Chapter 14, Part B
Using an Index for Selections

- Cost depends on #qualifying tuples, and clustering.
  - Cost of finding qualifying data entries, plus
  - Cost of retrieving records (could be large w/o clustering).
  - E.g., assuming uniform distribution of names, about 10% of tuples qualify (100 pages, 10000 tuples). With a clustered index, cost is little more than 100 I/Os; if unclustered, upto 10000 I/Os!

- Important refinement for unclustered indexes:
  1. Find qualifying data entries.
  2. Sort the rid’s of the data records to be retrieved.
  3. Fetch rids in order.
Two Approaches to General Selections

- **First approach:**
  - Find the *most selective access path*, retrieve tuples using it,
  - Then, apply any remaining terms that don’t *match* the index:

- **Most selective access path:** An index or file scan that we estimate will require the fewest page I/Os.

- E.g., \( \text{day} < 8/9/94 \ \text{AND} \ \text{bid} = 5 \ \text{AND} \ \text{sid} = 3 \).
  - A B+ tree index on \( \text{day} \) can be used; then, \( \text{bid} = 5 \) and \( \text{sid} = 3 \) must be checked for each retrieved tuple.
  - Similarly, a hash index on \(<\text{bid}, \text{sid}>\) could be used; \( \text{day} < 8/9/94 \) must then be checked.
Intersection of Rids

- **Second approach** (if we have 2 or more matching indexes that use Alternatives (2) or (3) for data entries):
  - Get sets of rids of data records using each matching index.
  - Then *intersect* these sets of rids
  - Retrieve the records and apply any remaining terms.

- E.g., \( \text{day}<8/9/94 \ \text{AND \ bid=}5 \ \text{AND \ sid=}3 \)
  - If we have a B+ tree index on \( \text{day} \) and an index on \( \text{sid} \), both using Alternative (2), we can:
    - Retrieve rids of records satisfying \( \text{day}<8/9/94 \) using the first
    - Retrieve rids of recs satisfying \( \text{sid=}3 \) using the second
    - Intersect
    - Retrieve records and check \( \text{bid=}5 \).
The Projection Operation

- An approach based on **sorting**:
  - Modify Pass 0 of external sort to eliminate unwanted fields. Tuples in runs are smaller than input tuples (due to eliminated attributes).
  - Modify merging passes to eliminate duplicates. Thus, number of result tuples smaller than input (due to eliminated duplicates).
- **Cost:**
  - In Pass 0, read original relation (size M)
  - Write out same number of smaller tuples
  - In merging passes, fewer tuples written out in each pass.
  - E.g., using Reserves example, 1000 input pages reduced to 250 in Pass 0 if size ratio is 0.25
Projection Based on Hashing

- **Partitioning phase**: Read R using one input buffer. For each tuple, discard unwanted fields, apply hash function $h1$ to choose one of B-1 output buffers.
  - Result is B-1 partitions (of tuples with no unwanted fields). 2 tuples from different partitions guaranteed to be distinct.

- **Duplicate elimination phase**: For each partition, read it and build an in-memory hash table, using hash fn $h2$ ($<> h1$) on all fields, while discarding duplicates.

- **Cost**: For partitioning, read R, write out each tuple, but with fewer fields. This is read in next phase.
Discussion of Projection

- Sort-based approach is the standard; better handling of skew and result is sorted.
- If an index on the relation contains all wanted attributes in its search key, can do index-only scan.
  - Apply projection techniques to data entries (much smaller!)
- If an ordered (i.e., tree) index contains all wanted attributes as prefix of search key, can do even better:
  - Retrieve data entries in order (index-only scan), discard unwanted fields, compare adjacent tuples to check for duplicates.
Set Operations

- Intersection and cross-product special cases of join.
- Union (Distinct) and Except are similar.
- **Sorting** based approach to **union**:
  - Sort both relations (on combination of all attributes).
  - Scan sorted relations and merge them.
- **Hash** based approach to **union**:
  - Partition R and S using hash function $h$.
  - For each S-partition, build in-memory hash table (using $h2$), scan corr. R-partition and add tuples to table while discarding duplicates.
Aggregate Operations (AVG, MIN, etc.)

- **Without grouping:**
  - In general, requires scanning the relation.
  - Given index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan.

- **With grouping:**
  - **Two Approaches:**
    - **Sort** on group-by attributes, then scan relation and compute aggregate for each group.
    - **Hashing** on group-by attributes.
  - When can we do aggregation with group-by using an index-only plan?
Summary

- A virtue of relational DBMSs: *queries are composed of a few basic operators*; the implementation of these operators can be carefully tuned (and it is important to do this!).

- Many alternative implementation techniques for each operator; no universally superior technique for most operators.

- Must consider available alternatives for each operation in a query and choose best one based on system statistics, etc. This is part of the broader task of optimizing a query composed of several ops.