CS 443
Evaluation of Relational Operations Beyond Joins
Chapter 14

Winter 2013 **Renée J. Miller**

Relational Operations

- We will consider how to implement:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Join* ($\bowtie$) Allows us to combine two relations.
  - *Set-difference* ($-$) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\cup$) Tuples in reln. 1 and in reln. 2.
  - *Aggregation* (SUM, MIN, etc.) and GROUP BY
- Since each op returns a relation, ops can be *composed*! After we cover the operations, we will discuss how to *optimize* queries formed by composing them.

Using an Index for Selections

- Cost depends on #qualifying tuples, and clustering.
  - Cost of finding qualifying data entries (typically small) plus cost of retrieving records (could be large w/o clustering).
  - Assuming uniform distribution of names, about 10% of tuples qualify (e.g., 100 pages, 10000 tuples). With a clustered index, cost is little more than 100 I/Os; if unclustered, up to 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. This ensures that each data page is looked at just once (though # of such pages likely to be higher than with clustering).

Two Approaches to General

- First approach: Find the *most selective access path*, retrieve tuples using it, and 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.
  - Terms that match this index reduce the number of tuples *retrieved*; other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched.
  - Consider *day<8/9/94 AND bid=5 AND sid=3*. A B+ tree index on *day* can be used; then, *bid=5* and *sid=3* must be checked for each retrieved tuple. Similarly, a hash index on *<bid, sid>* could be used; *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 (we'll discuss intersection soon!)
  - Retrieve the records and apply any remaining terms.
  - Consider *day<8/9/94 AND bid=5 AND sid=3*. If we have a B+ tree index on *day* and an index on *sid*, both using Alternative (2), we can retrieve rids of records satisfying *day<8/9/94* using the first, rids of recs satisfying *sid=3* using the second, intersect, retrieve records and check *bid=5*.

The Projection Operation

```
SELECT  DISTINCT
        R.sid, R.bid
FROM    Reserves R
```

- An approach based on sorting:
  - Modify Pass 0 of external sort to eliminate unwanted fields. Thus, runs of about 2B pages are produced, but tuples in runs are smaller than input tuples. (Size ratio depends on # and size of fields that are dropped.)
  - Modify Pass 0 & merging passes to eliminate duplicates. Thus, number of result tuples smaller than input. (Difference depends on # of duplicates.)
  - Cost: In Pass 0, read original relation (size M), write out same number of smaller (distinct) tuples. In merging passes, fewer tuples written out in each pass. Reserves: 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.
  - If partition does not fit in memory, can apply hash-based projection algorithm recursively to this partition.
- 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 similar; we'll do union.
- Sorting based approach to union:
  - Sort both relations (on combination of all attributes).
    - Remove duplicates? (if not base relations)
  - Scan sorted relations and merge them.
  - *Alternative*: Merge runs from Pass 0 for *both* relations.
- 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 corresponding R-partition and add tuples to table while discarding duplicates.

Aggregate Operations (AVG, MIN,...)

- Without grouping:
  - In general, requires scanning the relation.
  - Given index whose search key includes all attributes in the SELECT (if there is no WHERE), can do index-only scan.
- With grouping:
  - Sort on group-by attributes, then scan relation and compute aggregate for each group. (Can improve upon this by combining sorting and aggregate computation.)
  - Similar approach based on hashing on group-by attributes.
  - Given tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, can do index-only scan; if group-by attributes form prefix of search key, can retrieve data entries/tuples in group-by order.

Impact of Buffering

- If several operations are executing concurrently, estimating the number of available buffer pages is guesswork.
- Repeated access patterns interact with buffer replacement policy.
  - e.g., Inner relation is scanned repeatedly in Simple Nested Loop Join. With enough buffer pages to hold inner, replacement policy does not matter. Otherwise, MRU is best, LRU is worst (*sequential flooding*).
  - Does replacement policy matter for Block Nested Loops?
  - What about Index Nested Loops? Sort-Merge Join?

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.