CS222: Principles of Data Management

Notes #13
Set operations, Aggregation, Query Plans

Instructor: Chen Li
Relational 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).
  - Scan sorted relations and merge them.
  - *Alternative*: Merge runs from Pass 0+ for both relations (!).
- Hash-based approach to union (from Grace):
  - Partition both R and S using hash function $h1$.
  - For each S-partition, build in-memory hash table (using $h2$), then scan corresponding R-partition, adding truly new S tuples to hash table while discarding duplicates.
Sets versus Bags

- UNION, INTERSECT, and DIFFERENCE (EXCEPT or MINUS) use the set semantics.
- UNION ALL just combine two relations without considering any correlation.
Sets versus Bags

Example:

- \( R = \{1, 1, 1, 2, 2, 2, 2, 3\} \)
- \( S = \{1, 2, 2\} \)
- \( R \cup S = \{1, 2, 3\} \)
- \( R \cup \text{ALL} \ S = \{1, 1, 1, 2, 2, 2, 2, 2, 3\} \)
- \( R \cap S = \{1, 2\} \)
- \( R \setminus S = \{3\} \)

Notice that some DBs such as MySQL don't support some of these operations.
Aggregate Operations (AVG, MIN, ...)

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

- With grouping:
  - Sort on the group-by attributes, then scan sorted result and compute the aggregate for each group. (Can improve on this by combining sorting and aggregate computations.)
  - Or, similar approach via hashing on the group-by attributes.
  - Given tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, can do an index-only scan; if group-by attributes form prefix of search key, can retrieve data entries (tuples) in the group-by order.
Aggregation State Handling

- State: \( \text{init}( ) \), \( \text{next}(\text{value}) \), \( \text{finalize}( ) \) → agg. value
- Consider the following query

\[
\text{SELECT E.dname, avg(E.sal)} \\
\text{FROM Emp E} \\
\text{GROUP BY E.dname}
\]

```
<table>
<thead>
<tr>
<th>ename</th>
<th>sal</th>
<th>dname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>10K</td>
<td>Toy</td>
</tr>
<tr>
<td>Sue</td>
<td>20K</td>
<td>Shoe</td>
</tr>
<tr>
<td>Mike</td>
<td>13K</td>
<td>Shoe</td>
</tr>
<tr>
<td>Chris</td>
<td>25K</td>
<td>Toy</td>
</tr>
<tr>
<td>Zoe</td>
<td>50K</td>
<td>Book</td>
</tr>
</tbody>
</table>
```

\( h(\text{dname}) \mod 4 \)

Aggregate state (per unfinished group):
- Count: # values
- Min: min value
- Max: max value
- Sum: sum of values
- Avg: count, sum of values
Operator Buffering Considerations

- If several operations are executing concurrently, estimating the “right” number of buffer pages is tough. (Intra- vs. inter-query considerations, too!)
- Repeated access patterns can interact with buffer pool’s page replacement policy as well.
  - e.g., Inner relation scanned repeatedly in Simple Nested Loop Join. With enough buffer pages to hold the entire inner, replacement policy does not matter.
  - Otherwise MRU best, LRU worst (sequential flooding)!
- Memory management for multi-user DBMSs is still a challenging problem (too many “knobs” today).
DBMS Structure (from Lecture 1)

- Query Parser
- Query Optimizer
- Plan Executor
- Relational Operators (+ Utilities)
- Files of Records
- Access Methods (Indices)
- Buffer Manager
- Disk Space and I/O Manager

SQL

- Transaction Manager
- Lock Manager
- Log Manager

We are here...

Data Files
Index Files
Catalog Files

WAL
Component Roles

- **Query Parser**
  - Parse and analyze SQL query
  - Produce data structure capturing SQL statement and the “objects” that it refers to in the system catalogs

- **Query optimizer (often w/2 steps)**
  - Rewrite query logically (e.g., views/unnesting)
  - Perform cost-based optimization (*a la* System R)
  - Goal is to pick a “good” query plan considering
    - Physical table structures
    - Available access paths (indexes)
    - Data statistics (if known)
    - Cost model (for relational operations)

*(Cost differences can be orders of magnitude!)*
Component Roles (cont.)

- Plan Executor + Relational Operators
  - Runtime side of query processing
  - Usually based on a pull-based “tree of iterators” model, e.g.:

Nodes are relational operators (actually physical implementations of relational operators/combos)
**A Motivating Example**

\[
\text{SELECT S.sname} \\
\text{FROM Reserves R, Sailors S} \\
\text{WHERE R.sid=S.sid AND} \\
\text{R.bid=100 AND S.rating>5}
\]

- **Cost:** \(500+500\times1000 = 500,500\) I/Os
- *(Not the worst possible plan! 😊)*
- Misses several opportunities: selections could have been `pushed` earlier, no use is made of any available indexes, etc.
- **Goal of optimization:** To consider more efficient plans that compute the same logical answer.
Alternative Plan 1
(No Indexes)

- **Main difference:** push selects.
- With 5 buffers, cost of plan:
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution), with cost = 1010.
  - Scan Sailors (500) + write temp T2 (250 pages, if 10 ratings), cost = 750.
  - Join by sort T1 (2*2*10 = 40), sort T2 (2*3*250 = 1500), merge (10+250).
  - Total: **3560** page I/Os.
- If we use block NL join, join cost = 10+10*250, total cost = **4270**.
- If we “push” projections, T1 has only sid, T2 only sid and sname:
  - T1 now fits in 3 pages, so block NL cost drops lower, … (etc.)
Alternative Plan 2
(Using Indexes)

- With clustered index on bid of Reserves, we get 100,000/100 = 1000 tuples on 1000/100 = 10 pages.

- INL join with pipelining (outer not materialized).
  - (Projecting out unnecessary fields from outer doesn’t help.)

- Join column sid is a key for Sailors.
  - At most one matching Sailors tuple, so unclustered index on sid is OK.

- Decision not to push rating>5 before the join based on the availability of the sid index on Sailors.

- Cost: Selection of Reserves tuples (10 I/Os); then, for each one, must get matching Sailors tuple (1000*1.2); total = 1210 I/Os.
Relational Runtime Summary

- A virtue of relational DBMSs: *queries are composed of a few basic operators*; implementations 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.
- Thus, must consider available alternatives for each operation in a query and choose the best one based on system statistics, etc. This is part of the broader task of optimizing a query composed of several ops.

(→ OUR NEXT TOPIC! 😊)