Principles of Data Management

Lecture #12
Set operations, Aggregation

Instructor: Chen Li
Relational Set Operations

- Intersection and cross-product special cases of join.
- Union (\texttt{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.
  - \textit{Alternative}: Merge runs from Pass 0\textsuperscript{+} for \textit{both} relations (!).
- Hash-based approach to union (from Grace):
  - Partition both R and S using hash function \texttt{h1}.
  - For each S-partition, build in-memory hash table (using \texttt{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.
- In other words, SQL does not support these operations using the bag semantics.
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: `init()`, `next(value)`, `finalize()` $\rightarrow$ agg. value
- Consider the following query

```
SELECT E.dname, avg(E.sal)
FROM Emp E
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>

$\text{ Aggregate state (per unfinished group):}$
- 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).
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! 😊)