CS222: Principles of Data Management

Lecture #15
Query Optimization: Plan Generation

Instructor: Chen Li
System R Optimizer

- **Impact:**
  - Most widely used currently; works well for < 10 joins.

- **Cost estimation:** An approximate art at best (😊).
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes.
  - Considers combination of CPU and I/O costs.

- **Plan Space:** Too large, must be pruned.
  - Only the space of *left-deep plans* is considered.
    - Left-deep plans allow output of each operator to be *pipelined* into the next operator without storing it in a temporary relation.
  - Cartesian products avoided.
Overview of Query Optimization

- **Plan**: Tree of relational algebra ops annotated with the chosen algorithm for each op.
  - Then at runtime, when an operator is `pulled’ for its next output tuple, it `pulls’ on its inputs and computes them.

- Two main issues:
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?

- Ideally: Want to find very best available plan.
  (Reality: Avoid picking one of the worst plans!)

- We will study the System R approach.
Query Blocks: Units of Optimization

- An SQL query is parsed into a collection of *query blocks*, and they are optimized one block at a time.
- Nested blocks usually treated as calls to a subroutine, once per outer tuple. (Over-simplification, but will serve our purposes.)
- Query rewrite phase, before cost-based optimization phase, tries to “flatten” nested queries where it can (exposing joins).

```
SELECT S.sname
FROM Sailors S
WHERE S.age IN
  (SELECT MAX(age)
   FROM Sailors
   WHERE rating = S.rating)
```
For each block

- Fundamental decision in System R: **only left-deep join trees** are considered.
  - As the number of joins increases, the number of alternative plans grows rapidly; *we need to restrict the search space.*
  - Left-deep trees allow us to generate all **fully pipelined plans**.
    - Intermediate results not written to temporary files.
    - Not all left-deep trees are fully pipelined (e.g., SM join).
Relational Algebra Equivalences

- Allow us to choose different join orders and to `push’ selections and projections ahead of joins.

**Selections:** \( \sigma_{c_1 \wedge \ldots \wedge c_n}(R) \equiv \sigma_{c_1}(\ldots \sigma_{c_n}(R)) \) (Cascade)

\( \sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R)) \) (Commute)

**Projections:** \( \pi_{a_1}(R) \equiv \pi_{a_1}(\ldots (\pi_{a_n}(R))) \) (Cascade)

**Joins:** \( R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \) (Associative)

\( (R \bowtie S) \equiv (S \bowtie R) \) (Commute)

Show that: \( R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S \)
More Equivalences

- A projection commutes with a selection that only uses attributes retained by the projection.
- Selection between attributes of the two arguments of a cross-product converts the cross-product to a join.
- A selection on just attributes of R commutes with $R \bowtie S$. (i.e., $\sigma (R \bowtie S) \equiv \sigma (R) \bowtie S$).
- Similarly, if a projection follows a join $R \bowtie S$, we can push it down (earlier) by retaining only attributes of R (and S) that are needed for the join or are kept by the projection.
Enumeration of Alternative Plans

- There are two main cases:
  - Single-relation plans
  - Multiple-relation plans

- For queries over a single relation, queries consist of a combination of selects, projects, and aggregate ops:
  - Each available access path (file scan / index) is considered, and the one with the least estimated cost is chosen.
  - The different operations are essentially carried out together (e.g., if an index is used for a selection, projection is done for each retrieved tuple, and the resulting tuples are pipelined into the aggregate computation).
Enumeration of Left-Deep Plans

- Left-deep plans differ only in the order of relations, the access method for each relation, and the join method chosen for each join.

- Enumerated using N passes (if N relations joined):
  - Pass 1: Find best 1-relation plan for each relation.
  - Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation. (All 2-relation plans.)
  - Pass N: Find best way to join result of a (N-1)-relation plan (as outer) to the N’th relation. (All N-relation plans.)

- For each subset of relations, retain only:
  - Cheapest plan overall, plus
  - Cheapest plan for each interesting order of the tuples.
A Note on Interesting Orders

- A given data order is deemed “interesting” if it has the potential to save work (i.e., lower cost) later on.
  - Ordering on join attribute(s)
  - Ordering on GROUP BY attribute(s)
  - Ordering on DISTINCT attribute(s)
  - Ordering on ORDER BY attribute(s)

- Used to prune but not over-prune the search space.
  - Keep only the cheapest plan for each such ordering plus the cheapest unordered plan.
  - But, do indeed keep the cheapest plan for each one.
  - If a sorted plan is cheaper than any truly unordered plan, then toss the unordered one. (Q: Why?)
ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an “interestingly ordered” plan or via an additional sorting operator.

An N-1 way plan will not be combined with an additional relation unless there is a join condition between them, unless all WHERE predicates have been used up.

- i.e., avoid Cartesian products if possible!

In spite of pruning plan space, this approach is still exponential in the # of tables.
Cost Estimation for Multirelation Plans

- Consider a query block:
  ```sql
  SELECT attribute list
  FROM relation list
  WHERE term1 AND ... AND termk
  ```

- Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause.

- Reduction factor (RF) associated with each term reflects the impact of the term in reducing result size. Result cardinality = Max # tuples * product of all RF’s.

- Multirelation plans are built up by joining one new relation at a time.
  - Cost of join method, plus estimation of join cardinality, gives us both a cost estimate and a result size estimate.
Example

- **Pass 1:**
  - **Sailors:** B+ tree matches *rating* \(>5\), and might be the cheapest. However, if this selection is expected to retrieve a lot of tuples, and the index is unclustered, a file scan may be cheaper.
    - B+ tree plan possibly not kept (*rating* order *not* interesting!!).
  - **Reserves:** B+ tree on *bid* matches *bid*=100; cheapest.

- **Pass 2:**
  - We consider each plan retained from Pass 1 as the outer, and consider how to join it with the (only) other relation.
    - E.g., **Reserves as outer:** Hash index can be used to get Sailors tuples that satisfy *sid* = (outer tuple’s *sid* value).
System R Optimizer Recap

- Single-relation queries:
  - All access paths considered, cheapest chosen.
  - Issues: Selections that match index, whether index key has all needed fields or provides tuples in an interesting order.

- Multiple-relation queries:
  - All single-relation plans are first enumerated.
    - Selections/projections considered as early as possible.
  - Next, for each 1-relation plan, all ways of joining another relation (as inner) are considered.
  - Next, for each 2-relation plan that is retained, all ways of joining one more relation (as inner) are considered, etc.
  - At each level, for each subset of relations, only the best plan for each interesting order of tuples is retained.
Q: Nested Queries?

- Nested block is optimized independently, with the outer tuple considered as providing a selection condition.
- Outer block is optimized with the cost of `calling’ nested block computation taken into account.
- Implicit ordering of these blocks means that some good strategies are not considered. The non-nested version of the query is typically optimized better!
Summary

- Query optimization is an extremely important task in a relational DBMS.
- Must understand optimization to understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).
- Two parts to optimizing a query:
  - Explore a set of alternative plans.
    - Must prune search space; typically, left-deep plans only.
  - Must estimate cost of each plan that is considered.
    - Must estimate size of result and cost for each plan node.
    - Key issues: Statistics, indexes, operator implementations.
Summary (Contd.)

- Single-relation queries:
  - All access paths considered, cheapest is chosen.
  - Issues: Selections that match index, whether index key has all needed fields and/or provides tuples in a desired order.

- Multiple-relation queries (a la System R):
  - All single-relation plans are first enumerated.
    - Selections/projections considered as early as possible.
  - Next, for each 1-relation plan, all ways of joining one more relation (as inner) are considered.
  - Next, for each 2-relation plan that is retained, all ways of joining another relation (as inner) are considered, etc.
  - At each level, for each subset of relations, only best plan for each interesting order of tuples is ‘retained’.