Relational Query Optimization

Chapters 14
An SQL query is parsed into a collection of **query blocks**, and these are optimized one block at a time.

Nested blocks are usually treated as calls to a subroutine, made once per outer tuple. (This is an over-simplification, but serves for now.)

For each block, the plans considered are:

- All available access methods, for each reln in FROM clause.
- All *left-deep join trees* (i.e., all ways to join the relations one-at-a-time, with the inner reln in the FROM clause, considering all reln permutations and join methods.)
Every SQL query block can be expressed as an essential $\sigma \pi \times$ expression:
- SELECT $\to \pi$
- WHERE $\to \sigma$
- FROM $\to \times$

The remaining operations are carried out on the $\sigma \pi \times$ expression.
- GROUP BY
- HAVING
- Final SELECT $\to \pi$
Cost Estimation

- For each plan considered, must estimate cost:
  - Must estimate cost of each operation in plan tree.
    - Depends on input cardinalities.
    - We’ve already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Must estimate size of result for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.
Statistics and Catalogs

- Need information about the relations and indexes involved. **Catalogs** typically contain at least:
  - # tuples (NTuples) and # pages (NPages) for each relation.
  - # distinct key values (NKeys) and NPages for each index.
  - Index height, low/high key values (Low/High) for each tree index.

- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.

- More detailed information (e.g., histograms of the values in some field) are sometimes stored.
Size Estimation and Reduction Factors

- Consider a query block:

  $$\text{SELECT attribute list FROM relation list WHERE } \text{term}_1 \text{ AND ... AND } \text{term}_k$$

- 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.

  - Implicit assumption that terms are independent!
  - col=value: RF = 1/NKeys(I), given index I on col
  - col1=col2: RF = 1/\text{MAX}(NKeys(I1), NKeys(I2))
  - col>value: RF = (High(I)-value)/(High(I)-Low(I))
Relational Algebra Equivalences

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

- **Selections:** \( \sigma_{c_1 \land \ldots \land c_n}(R) \equiv \sigma_{c_1}(\ldots \sigma_{c_n}(R)) \) \hspace{1cm} (Cascade)
  \[
  \sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R)) \hspace{1cm} (Commute)
  \]

- **Projections:** \( \pi_{a_1}(R) \equiv \pi_{a_1}(\ldots(\pi_{a_n}(R))) \) \hspace{1cm} (Cascade)

- **Joins:** \( R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \) \hspace{1cm} (Associative)
  \[
  (R \bowtie S) \equiv (S \bowtie R) \hspace{1cm} (Commute)
  \]

+ 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.
  \[ \pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R)) \]

- Selection between attributes of the two arguments of a cross-product converts cross-product to a join.
  \[ R \bowtie_c S \equiv \sigma_c(R \times S) \]

- 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 by retaining only attributes of R (and S) that are needed for the join or are kept by the projection.
  \[ \pi_a(R \bowtie_c S) \equiv \pi_{a_1}(R) \bowtie_c \pi_{a_2}(S) \]
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).
Cost Estimates for Single-Relation Plans

- Index I on primary key matches selection:
  - Cost is $\text{Height}(I) + 1$ for a B+ tree, about 1.2 for hash index.

- Clustered index I matching one or more selects:
  - $(\text{NPages}(I) + \text{NPages}(R)) \times \text{product of RF's of matching selects}.$

- Non-clustered index I matching one or more selects:
  - $(\text{NPages}(I) + \text{NTuples}(R)) \times \text{product of RF's of matching selects}.$

- Sequential scan of file:
  - $\text{NPages}(R)$.

+ Note: Typically, no duplicate elimination on projections! (Exception: Done on answers if user says DISTINCT.)
Example

- If we have an index on *rating*: 
  - Clustered index: \( \frac{1}{N\text{Keys}(I)} \times (N\text{Pages}(I)+N\text{Pages}(S)) = \frac{1}{10} \times (50+500) \) pages are retrieved. (This is the cost.)
  - Unclustered index: \( \frac{1}{N\text{Keys}(I)} \times (N\text{Pages}(I)+N\text{Tuples}(S)) = \frac{1}{10} \times (50+40000) \) pages are retrieved.

- If we have an index on *sid*: 
  - Would have to retrieve all tuples/pages. With a clustered index, the cost is 50+500, with unclustered index, 50+40000.

- Doing a file scan: 
  - We retrieve all file pages (500).
Queries Over Multiple Relations

- 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).
Enumeration of Left-Deep Plans

- Left-deep plans differ only in 1) the order of relations, 2) the access method for each relation, and 3) the join method 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.
Enumeration of Plans (Contd.)

- ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an `interestingly ordered’ plan or an additional sorting operator.

- An N-1 way plan is not combined with an additional relation unless there is a join condition between them, unless all predicates in WHERE 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.
Example

Pass 1:
- **Sailors**: B+ tree matches $rating \geq 5$, and is probably cheapest. However, if this selection is expected to retrieve a lot of tuples, and index is unclustered, file scan may be cheaper.
  - Still, B+ tree plan is kept (because tuples are in $rating$ order).
- **Reserves**: B+ tree on $bid$ matches $bid = 500$; 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.
- For example, **Reserves as outer**: Hash index can be used to get Sailors tuples that satisfy $sid = \text{outer tuple’s } sid$ value.
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.

Example query:

```sql
SELECT S.sname
FROM Sailors S
WHERE EXISTS
  (SELECT *
   FROM Reserves R
   WHERE R.bid=103
   AND R.sid=S.sid)
```

Nested block to optimize:

```sql
SELECT *
FROM Reserves R
WHERE R.bid=103
AND R.sid=S.sid
```

Equivalent non-nested query:

```sql
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid
AND R.bid=103
```
Summary

- Query optimization is an important task in a relational DBMS.
- Must understand optimization in order to understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).

- Two parts to optimizing a query:
  - Consider 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:
  - 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 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`.