Lecture #15: Query Planning & Optimization II

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Cost-based Query Optimization

The DBMS's optimizer will use an internal *cost model* to estimate the execution cost for a particular query plan. This provides an estimate to determine whether one plan is better than another without having to actually run the query (which would be slow to do for thousands of plans).

This estimate is an internal metric that (usually) is not comparable to real-world metrics, but it can be derived from estimating the usage of different resources:

- **CPU:** Small cost; tough to estimate.
- **Disk:** Number of block transferred.
- **Memory:** Amount of DRAM used.
- **Network:** Number of messages transfer ed.

To accomplish this, the DBMS stores internal statistics about tables, attributes, and indexes in its internal catalog. Different systems update the statistics at different times. Commercial DBMS have way more robust and accurate statistics compared to the open source systems. These are estimates and thus the cost estimates will often be inaccurate.

2 Statistics

For a relation R, the DBMS stores the number of tuples (N_R) and distinct values per attribute ($V(A, R)$).

The *selection cardinality* ($SC(A, R)$) is the average number of records with a value for an attribute A given $N_R/V(A, R)$.

Complex Predicates

- The *selectivity* (sel) of a predicate P is the fraction of tuples that qualify:

$$sel(A = constant) = SC(P)/V(A, R)$$

- For a range query, we can use: $sel(A >= a) = (A_{max} - a/(A_{max} - A_{min}))$.
- For negations: $sel(notP) = 1 - sel(P)$.
- The selectivity is the probability that a tuple will satisfy the predicate. Thus, assuming predicates are independent, then $sel(P1 \bigwedge P2) = sel(P1) * sel(P2)$.

Join Estimation

- Given a join of R and S, the estimated size of a join on non-key attribute A is approx

$$estSize \approx N_R * N_S/max(V(A, R), V(A, S))$$

Statistics Storage

Histograms: We assumed values were uniformly distributed. But in real databases values are not uniformly distributed, and thus maintaining a histogram is expensive. We can put values into buckets to reduce the size of the histograms. However, this can lead to inaccuracies as frequent values will sway the count of infrequent values. To counteract this, we can size the buckets such that their spread is the same. They each hold a similar amount of values.

Sampling: Modern DBMSs also employ **sampling** to estimate predicate selectivities. Randomly select and maintain a subset of tuples from a table and estimate the selectivity of the predicate by applying the predicate to the small sample.

3 Search Algorithm

The basic cost-based search algorithm for a query optimizer is the following:

1. Bring query in internal form into canonical form.
2. Generate alternative plans.
3. Generate costs for each plan.
4. Select plan with smallest cost.

It is important to pick the best access method (i.e., sequential scan, binary search, index scan) for each table accessed in the query. Simple heuristics are sometimes good enough for simple OLTP queries (i.e., queries that only access a single table). For example, queries where it easy to pick the right index to use are called *sargable* (Search Argument Able). Joins in OLTP queries are also almost always on foreign key relationships with small cardinality.

For multiple relation query planning, the number of alternative plans grows rapidly as number of tables joined increases. For an n-way join, the number of different ways to order the join operations is known as a Catalan number (approx 4^n). This is too large of a solution space and it is infeasible for the DBMS to consider all possible plans. Thus, we need a way to reduce the search complexity. For example, in IBM's System R, they only considered *left-deep* join trees. Left-deep joins allow you to pipeline data, and only need to maintain a single join table in memory.

4 Nested Sub-Queries

The DBMS treats nested sub-queries in the `WHERE` clause as functions that take parameters and return a single value or set of values.

Two Approaches:

1. Rewrite to decorrelate and/or flatten queries.
2. Decompose nested query and store result in sub-table.