1 Query Optimization

Query optimization is selecting the most efficient query-evaluation plan from the set of equivalent plans possible for processing a given query.

The optimizer first generates a set of equivalent relational algebra expressions, then create different strategies for processing the query, choose the most effecient algorithm, index, etc. Then it estimates the cost of each, and then choose the most efficient one. It must also select a strategy

2 Size Estimation

Estimations of the size of an evaluation plan are measured in disk block accesses, as this is the performance bottleneck. The DBMS **catalog** maintains information about each relation, tuple count n_r, block count b_r, tuple size l_r (in bytes), tuples per block f_r, number of distinct values in relation for an attribute $V(A, r)$, and the minimum and maximum values for an attribute $\min(V(A, r))$ and $\max(V(A, r))$. The catalog is updated in idle periods.

A subset of the estimation rules of **selection** result size are:

$$|\sigma_{A=\text{value}}(r)| \approx \frac{n_r}{V(A, r)}$$
$$|\sigma_{A\leq\text{value}}(r)| \approx n_r \cdot \frac{v - \min(A, r)}{\max(A, r) - \min(A, r).}$$

The estimation of cartesian product $r \times s$ is $n_r \cdot n_s \cdot (l_r + l_s)$. Estimation rules of **natural join** $r \bowtie s$ result size are:

- if the relations have no attributes in common, then $|r \bowtie s| \approx |r \times s|$
- if the common attributes are a superkey for r, then each tuple in s will join with at most one tuple from r, so the estimation is $n_s \cdot (l_r + l_s)$.
- in other cases, the following is a reasonable estimate:
$$\frac{n_r \cdot n_s}{V(R \cap S, r)}$$

Estimating the size of a theta join can be done using a transformation rule
$$r \bowtie_\theta s = \sigma_\theta(r \times s).$$

Projection is estimated to $\Pi_A(r) \approx V(A, r)$ since projection eliminates duplicates.

3 Transforming Relational Algebra Expressions

2 relational algebra expressions are **equivalent** if, for every database instance, the two expressions always generate the same set of tuples.

A subset of the equivalence rules:

- Conjunctive selections can be cascaded $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
- Selection is **commutative** $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
- Iterative projection can be simplified $\Pi_{A_1}(\Pi_{A_2}(...(\Pi_{A_n}(E))...)) = \Pi_{A_1}(E)$
- Set union and intersection are commutative and associative, e.g. $E_1 \cup E_2 = E_2 \cup E_1 \qquad (E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$
- Natural joins are **associative** $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$.

The last rule is very important, as joins are expensive. Therefore we will always **join the smallest relations first** and **select before join**.

$$\begin{aligned} & \Pi_{\text{customer-name}}((\sigma_{\text{branch-city="Aalborg"}}(\text{branch})) \bowtie \text{account} \bowtie \text{depositor}) \\ = \; & \Pi_{\text{customer-name}}(((\sigma_{\text{branch-city="Aalborg"}}(\text{branch})) \bowtie \text{depositor}) \bowtie \text{account}) \end{aligned}$$

The optimizer use equivalence rules to systematically generate all expressions equivalent to the given expression by trying to match a subexpression with one side of an equivalence rule.

4 Cost-based Optimization

A **cost-based optimizer** generates a range of query-evaluation plans from a query, and chooses the one with the least cost. The search for best plan is very large, but cost values can be saved for subplans, enabling a **dynamic**

programming algorithm for selecting the optimum plan. The algorithm can also discard sets of subplans if they are more expensive than the cheapest one found yet. However, the optimization algorithm has exponential complexity!

5 Heuristic Optimization

Heuristic optimization uses general rules instead of calculating cost of individual queries. Some typical rules:

1. Cascade conjunctive selections
2. Select early - move selections down the query tree
3. Small selections and joins first
4. Replace cartesian product followed by selection with join
5. Project early - move projections down the query tree
6. Pipeline where possible