Databases and Information Systems
CS303

Query Optimization
02-11-2023

Query Optimization

- Goal :
  - Look at all equivalent relational algebraic expressions that are equivalent to the current query
  - Estimate the number of tuples generated at every intermediate step
    - Calculate the cost of evaluation depending on this estimate
  - Choose the expression that has the minimal cost

Recap : Selection size estimation

- $\sigma_{A = a}(r)$
  - Estimate of the number of tuples = n_r / $V(A,r)$
    - n_r is the number of tuples in r
    - $V(A,r)$ is the number of distinct values of A that occurs in r
  - This assumes all values of A occur with uniform frequency
    - Good approximate in real life
  - If histogram data is available then we can get a better estimate

Recap : Selection size estimation

- $\sigma_{A <= a}(r)$
  - Catalog maintains min(A,r) and max(A,r) be the lowest and highest values that A can take
  - Estimate of the number of tuples
    - If a < min(A,r) then estimate = 0
    - If a >= max(A,r) then estimate = n_r

$$n_r * \frac{a - \min(A,r)}{\max(A,r) - \min(A,r)}$$

  - If expression is part of query, value of v may not be available.
    - Then rough estimate is $n_r / 2$

Recap : Selection size estimation

- $\sigma_{\theta 1 \wedge \theta 2 \dots \wedge \theta n}(r)$
  - For each i if the estimate of $\sigma_{\theta i}(r)$ is s_i then probability of a tuple satisfying θ_i is s_i / n_r
  - Estimate

$$n_r * \frac{s_1 * s_2 * \dots s_n}{n_r^{nr}}$$

- $\sigma_{\theta 1 \vee \theta 2 \dots \vee \theta n}(r)$
  - Estimate $n_r * (1 - (1 - s_1/n_r)(1 - s_2 / n_r) \dots (1 - s_n / n_r))$

- $\sigma_{\neg\theta}(r)$ Exercise

Recap : Join size estimation

- Cartesian product $r \times s$ will have $n_r * n_s$ tuples
- Natural Join : Let R and S be the set of attributes of r and s respectively
  - If $R \cap S = \varnothing$ then Natural Join is same as cartesian product
  - If $R \cap S$ is key for R then we can have at most n_s tuples in the natural join
  - If $R \cap S$ is a key neither for R nor for S if $R \cap S = \{ A \}$ then
    - Every tuple t in R produces $n_s / V(A,s)$ many tuples
    - So total estimate is $n_r * n_s / V(A,s)$
    - Reversing the roles of r and s, we get the estimate $n_r * n_s / V(A,r)$
      - Estimates differ if $V(A,r) \neq V(A,s)$ then pick the minimum
- We can also estimate $r \bowtie_\theta s$ as $\sigma_\theta (r \times s)$

Size estimation for other operations

- Projection : $\Pi_A (r)$ has the estimate $V(A,r)$
  - Since project operation eliminates duplicates
- Set operations :
  - Convert $\sigma_{\theta 1} (r) \cup \sigma_{\theta 2} (r)$ to $\sigma_{\theta 1 \vee \theta 2} (r)$ for estimation
  - If union is over different relations then estimate is the sum of tuples from each operand
  - For intersection of different relations, estimate is the minimum number of both operands
- Outer Join :
  - For left outer join of r and s, estimate is size of $r \bowtie s$ plus size of s
  - Similar for right outer join and full outer join
  - Estimate provides upper bound

Estimation of number of distinct values

- $V(A, \sigma_\theta(r))$
  - Distinct possible values that attribute A can take
  - If θ is of the form $A = a$ then estimate = 1
  - If θ is of the form $A = a_1 \vee A = a_2 \vee \ldots A = a_n$ then estimate = n
  - If θ is of the form A op a where op is some comparison operator then estimate = $V(A,r) * s/n_r$ where s is the estimate of $\sigma_\theta(r)$
  - In all other cases
    - Assume the selection is independent of A

      Estimate is the minimum of $V(A,r)$ and estimate of $\sigma_\theta(r)$

Estimation of number of distinct values

- $V(A, r \bowtie s)$
  - If all attributes of A are from R then estimate is minimum of V(A,r) and estimate of $r \bowtie s$
  - If A contains A1 attributes from R and A2 attributes from S then
    Estimate is minimum of V(A1,r) * V(A2-A1, s) and V(A1-A2 , r) * V(A2, s) and estimate of $r \bowtie s$

- Estimates for aggregates : sum / count / min / max / average
  - Exercise

Choice of Evaluation Plan

- Evaluation plan defines exactly what algorithm should be used for each operation
  - How the execution of the operations should be coordinated
- Cost based optimizer explores all query-evaluation plans that are equivalent to the given query
  - chooses the one with the least estimated cost

Cost based Join Order Selection

- $r_1 \bowtie r_2 \bowtie \dots \bowtie r_n$ can be reordered and the join can be performed

- With $n = 3$ there are 12 different join orderings
- For $n = 10$ we have 17.6 billion possible orderings

- With n different relations to join, how many join orderings are possible?
  - Exercise

$$
\begin{array}{llll}
r_1 \bowtie (r_2 \bowtie r_3) & r_1 \bowtie (r_3 \bowtie r_2) & (r_2 \bowtie r_3) \bowtie r_1 & (r_3 \bowtie r_2) \bowtie r_1 \\
r_2 \bowtie (r_1 \bowtie r_3) & r_2 \bowtie (r_3 \bowtie r_1) & (r_1 \bowtie r_3) \bowtie r_2 & (r_3 \bowtie r_1) \bowtie r_2 \\
r_3 \bowtie (r_1 \bowtie r_2) & r_3 \bowtie (r_2 \bowtie r_1) & (r_1 \bowtie r_2) \bowtie r_3 & (r_2 \bowtie r_1) \bowtie r_3
\end{array}
$$

Cost based Join Order Selection

- Not necessary to generate all join orderings

- Suppose we want to find the best join ordering of the form

  $$(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$$

  - Choose the best among the 12 orderings of $(r_1 \bowtie r_2 \bowtie r_3)$
  - Then join the result with r_4 and r_5
  - Total checks = 12 + 12 (instead of 144 with n = 5)

- This idea can be used to develop a dynamic-programming algorithm

```
procedure FindBestPlan(S)
    if (bestplan[S].cost ≠ ∞) /* bestplan[S] already computed */
        return bestplan[S]
    if (S contains only 1 relation)
        set bestplan[S].plan and bestplan[S].cost based on best way of accessing S
    else for each non-empty subset S1 of S such that S1 ≠ S
        P1 = FindBestPlan(S1)
        P2 = FindBestPlan(S − S1)
        A = best algorithm for joining results of P1 and P2
        cost = P1.cost + P2.cost + cost of A
        if cost < bestplan[S].cost
            bestplan[S].cost = cost
            bestplan[S].plan = "execute P1.plan; execute P2.plan;
                                join results of P1 and P2 using A"
    return bestplan[S]
```

Cost based optimization of equivalence rules

- Join order optimization technique handles the most common class of queries, which perform an inner join of a set of relations

- But queries use other features which are not addressed by join order selection
  - aggregation, outer join, and nested queries

- Algorithm to generate all possible equivalent relational expressions can be modified to generate all possible query execution plans
  - Example : Join can be annotated as has join, nested-block join etc

- But this is a costly process since it is a Brute-Force algorithm

Cost based optimization of equivalence rules

- Some techniques to make the algorithm efficient:
  - A space-efficient representation of expressions
    - Avoids making multiple copies of the same subexpressions when equivalence rules are applied.
  - Efficient techniques for detecting duplicate derivations of the same expression.
  - A form of dynamic programming based on memoization
    - stores the optimal query evaluation plan for a subexpression when it is optimized for the first time;
    - subsequent requests to optimize the same subexpression are handled by returning the already memoized plan.
  - Techniques to avoid generating all possible equivalent plans
    - By keeping track of the cheapest plan generated for any subexpression up to any point of time
    - Then pruning away any plan that is more expensive than the cheapest plan found so far for that subexpression

Heuristics in Optimization

- A drawback of cost-based optimization is the cost of optimization itself

- The number of different evaluation plans for a query can be very large
  - finding the optimal plan from this set requires a lot of computational effort.

- Hence, optimizers use heuristics to reduce the cost of optimization.

Some Heuristics

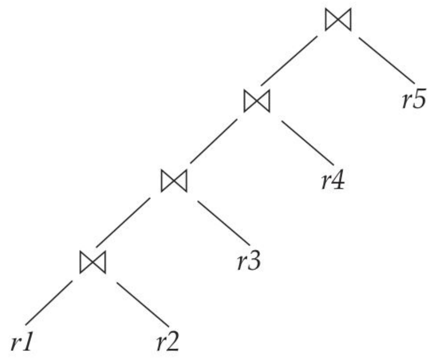
- Perform selection operation as early as possible
    - $\sigma_\theta(r \bowtie s)$ where θ is a condition over r then better to do $\sigma_\theta(r) \bowtie s$
- But if r is extremely small compared to s, and if there is an index on the join attributes of s, but no index on the attributes used by θ
    - Then it is probably a bad idea to perform the selection early
        - Performing the selection directly on s require doing a scan of all tuples in s
    - It is probably cheaper, in this case, to compute the join by using the index, and then to reject tuples that fail the selection.

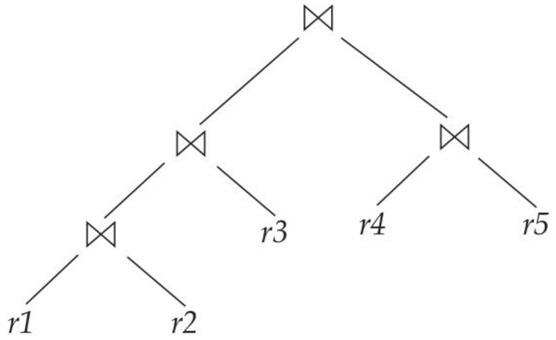
Some Heuristics

- Perform projection operation as early as possible
  - First do selection then do the projection
    - Selection enables the use of index
  - This might also not be the best always
    - Exercise : Come up with a scenario where doing projection later is useful

Some Heuristics

- Some optimizers use particular type of joins called left-deep join orders
- Better for pipelining
- Can find best order (among left-deep joins) faster

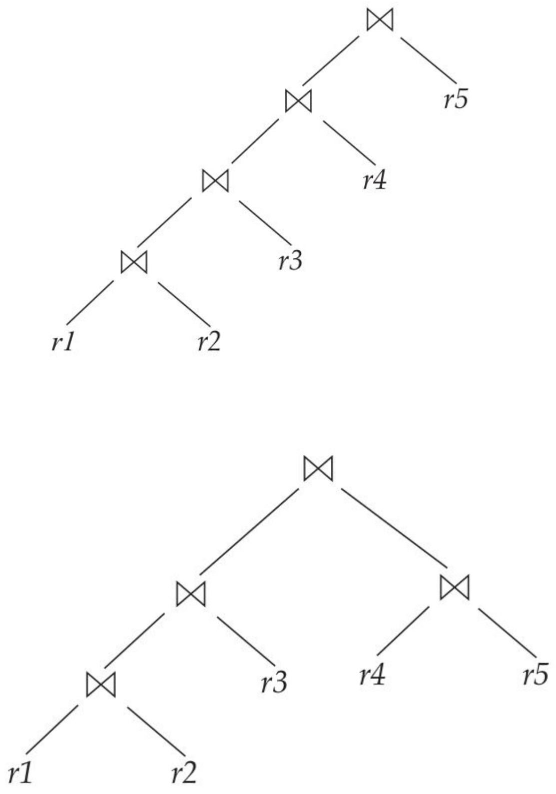




Some Heuristics

- Consider n different left-deep join orders, starting from every relation

- At every step choose the next best relation to perform join



Some Heuristics

- Some optimizers have a budget (time)
- If the budget runs out then return the best query evaluation plan found until then
    - First apply cheap heuristics to get a descent plan
    - In the remaining time apply complicated search to get the best plan

Some Heuristics

- Many applications execute the same query repeatedly but with different values for the constants.
    - Example: a university application may repeatedly execute a query to find the courses for which a student has registered, each time for a different student with a different value for the student ID.

- As a heuristic, many optimizers optimize a query once, with whatever values were provided for the constants when the query was first submitted
    - cache the query plan

- Whenever the query is executed again, the cached query plan is reused

- Optimal plan for the new instance may differ from the optimal plan for the earlier instance
    - but as a heuristic the cached plan is reused

- Caching and reuse of query plans is referred to as Plan caching.

Some Heuristics

- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead on query processing.
- However, the added cost of cost-based query optimization is usually more than offset by the saving at query-execution time.
- The difference in execution time between a good plan and a bad one may be huge, making query optimization essential.
- Most commercial systems include relatively sophisticated optimizers.

Optimizing nested Subqueries (Correlated queries)

- ```
  SELECT   name     FROM instructor
  WHERE EXISTS ( SELECT *   FROM teaches
                      WHERE instructor.ID = teaches.ID  AND teaches.year = 2009 )
  ```

- Subquery can be viewed as a function that takes a parameter (instructor.ID) and returns the set of all courses taught in 2009 by instructors with the same ID

- Not optimal to execute the inner query for every tuple from the outer query

- Get better code using decorrelation
  ```

Optimizing nested Subqueries (Correlated queries)

- SELECT ... FROM L_1
  WHERE P_1 AND EXISTS (SELECT * FROM L_2 WHERE P_2)

- CREATE TABLE t_1 AS SELECT DISTINCT V FROM L_2 WHERE P^1_2
  SELECT ... FROM L_1 , t_1 WHERE P_1 AND P^2_2
  - V is all attributes used in the condition P_2 from L_2
  - P^1_2 contains predicates in P_2 that talks only about L_2
    - P^2_2 contains predicates from P_2 that involves the correlation

- SELECT name FROM instructor
  WHERE EXISTS (SELECT * FROM teaches
  WHERE instructor.ID = teaches.ID AND teaches.year = 2009)

- CREATE TABLE t_1 AS SELECT DISTINCT ID FROM teachers WHERE teaches.year = 2009
  SELECT name FROM instructor, t_1 WHERE t_1.ID = instructor.ID

Optimizing nested Subqueries (Correlated queries)

- Decorrelation is more complicated when
  - The nested subquery uses aggregation
  - The result of the nested subquery is used to test for equality
  - The condition linking the nested subquery to the outer query is not exist
  - ....

- Decorrelation can be done, but complex

- Optimization of complex nested subqueries is a difficult task
  - It is best to avoid using complex nested subqueries, where possible
  - We cannot be sure that the query optimizer will succeed in converting them to a form that can be evaluated efficiently.

Other optimizations

- Materialized views
  - How to update them
    - Incremental view maintenance : Modify only the affected parts of materialized views
    - Immediate view maintenance or Differed view maintenance

  - Can be used for decorrelation and optimization

- Top K-optimization (LIMIT K)
  - If K is small, no point in computing full result
    - Pipelined plans can be generated in sorted order
    - Estimate what is the highest value on the sorted attributes that will appear in the top-K output
      - Introduce selection predicates that eliminate larger values
      - If extra tuples beyond the top-K are generated they are discarded
      - If too few tuples are generated then the selection condition is changed and the query is re-executed.

Other optimizations

- Join Minimization : Dropping a relation from a join without changing the result of the query
    - Can be done if we are creating a view using multiple relations but later using only some of those relations in the query

- Optimization of Updates : SET … WHERE ….
    - Cannot be done in parallel naively
    - Updates can affect the update query being executed in the presence of index (Halloween problem)
    - Problem can be avoided by executing the queries defining the update first, creating a list of affected tuples, and then updating the tuples and indices
    - Check if the Halloween problem occurs otherwise execute in parallel

Other optimizations

- Multiquery Optimization and Shared scans : Happens when a batch of queries are submitted together
  - Common subexpression elimination : If same subexpression is used by multiple queries, result can be reused
  - Instead of reading every relation per query, read it once and use it for all relevant queries
- Parametric Query Optimization (PQO) :
  - Query is generally optimized for some particular values (like SELECT ... FROM... WHERE ID = 1234)
  - PQO optimizes without specifying the value for parameters
  - The optimizer then outputs several plans, each optimal for a different parameter value
  - When a query is submitted with specific values for its parameters
    - Cheapest plan from the set of alternative plans computed earlier is used
    - Finding the cheapest such plan usually takes much less time than reoptimization

Reference:

Database System Concepts by Silberschatz, Korth and Sudarshan (6th edition)

Chapter 13