Lecture 18: Optimizations

Thursday, March 1, 2001

Outline

- Optimization
  - Algebraic laws
  - Heuristics based optimizations
  - Dynamic programming
- Size estimation

Optimization Overview

- Main idea:
  - SQL → query tree
  - query tree → query tree need: *algebraic laws*
- Two philosophies:
  - Heuristics-based optimizations
  - Cost-based optimizations need: *size estimation*
- Cost-based optimizations:
  - Well-understood for joins
  - Much more complex for the rest

Algebraic Laws

- Commutative and Associative Laws
  - $R \cup S = S \cup R$, $R \cup (S \cup T) = (R \cup S) \cup T$
  - $R \cap S = S \cap R$, $R \cap (S \cap T) = (R \cap S) \cap T$
  - $R \bowtie S = S \bowtie R$, $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$
- Distributive Laws
  - $R \bowtie (S \cup T) = (R \bowtie S) \cup (R \bowtie T)$

Algebraic Laws

- Laws involving selection:
  - $\sigma_{C \text{ AND } C'}(R) = \sigma_C(\sigma_{C'}(R)) = \sigma_C(R) \cap \sigma_{C'}(R)$
  - $\sigma_{C \text{ OR } C'}(R) = \sigma_C(R) \cup \sigma_{C'}(R)$
  - $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$
    - When C involves only attributes of R
  - $\sigma_C(R - S) = \sigma_C(R) - S$
  - $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$
  - $\sigma_C(R \cap S) = \sigma_C(R) \cap S$

Algebraic Laws

- Example: R(A, B, C, D), S(E, F, G)
  - $\sigma_{F=3}(R \underset{D=E}{\bowtie} S) =$?
  - $\sigma_{A=5 \text{ AND } G=9}(R \underset{D=E}{\bowtie} S) =$?

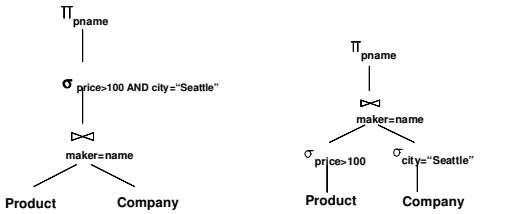
Algebraic Laws

- Laws involving projections
  - $\Pi_M(R \bowtie S) = \Pi_N(\Pi_P(R) \bowtie \Pi_Q(S))$
    - Where N, P, Q are appropriate subsets of attributes of M
  - $\Pi_M(\Pi_N(R)) = \Pi_{M,N}(R)$
- Example R(A,B,C,D), S(E, F, G)
  - $\Pi_{A,B,G}(R \underset{D=E}{\bowtie} S) = \Pi_{?}(\Pi_{?}(R) \underset{D=E}{\bowtie} \Pi_{?}(S))$

Heuristic Based Optimizations

- Query rewriting based on algebraic laws
- Result in better queries most of the time
- Main heuristics:
  - Push selections down the tree

Heuristic Based Optimizations



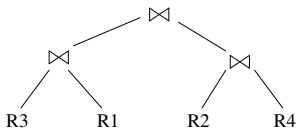
The earlier we process selections, less tuples we need to manipulate higher up in the tree (but may cause us to indexes).

Cost-Based Optimization

- Main optimization unit:
  - set of joins, i.e. single select-from-where block
  - Hence: *the join reordering problem*
- Optimization methods:
  - Dynamic programming (System R, 1977), for joins:
    - Conceptually cleanest
  - Rule-based optimizations, for arbitrary queries:
    - Volcano → SQL server
    - Starburst → DB2

Join Trees

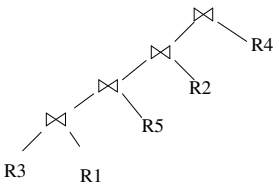
- R1 ⋈ R2 ⋈ …. ⋈ Rn
- Join tree:



- A join tree represents a plan. An optimizer needs to inspect many (all ?) join trees

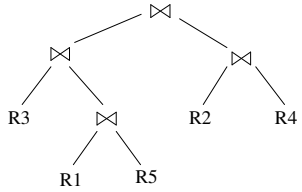
Types of Join Trees

- Left deep (or left-linear):



Types of Join Trees

- Bushy:

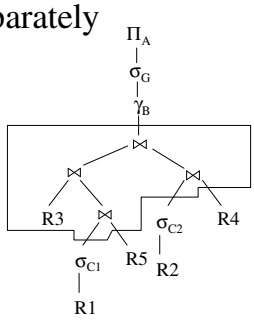


Problem

- Given: a query R1⋈ R2⋈ … ⋈Rn
- Assume we have a function cost() that gives us the cost of every join tree
- Find the best join tree for the query

Justification for Handling Joins Separately

Select A
From R_1, R_2, …, R_n
Where C
GroupBy B
Having G



Dynamic Programming

- Idea: for each subset of {R1, …, Rn}, compute the best plan for that subset
- Compute best plan as follows:
  - Step 1: for {R1}, {R2}, …, {Rn}
  - Step 2: for {R1,R2}, {R1,R3}, …, {Rn-1, Rn}
  - …
  - Step n: for {R1, …, Rn}
- A subset of {R1, …, Rn} is also called a *subquery*

Dynamic Programming

- For each subquery Q□ {R1, …, Rn} compute the following:
  - Size(Q)
  - A best plan for Q: Plan(Q)
  - The cost of that plan: Cost(Q)
- Additional complication:
  - Consider *interesting orders*
  - For each subquery Q□ {R1, …, Rn}, generate one plan for each interesting order

Dynamic Programming

- Step 1: For each {Ri} do:
  - Size({Ri}) = B(Ri)
  - Plan({Ri}) = Ri
  - Cost({Ri}) = (cost of scanning Ri)

Dynamic Programming

- Step i: For each Q⊆{R1, …, Rn} of cardinality i do:
  - Compute Size(Q) (later…)
  - For every pair of subqueries Q', Q'' s.t. Q = Q' U Q'' compute cost(Plan(Q') ⋈ Plan(Q''))
  - Cost(Q) = the smallest such cost
  - Plan(Q) = the corresponding plan

Dynamic Programming

- Return Plan({R1, …, Rn})

Dynamic Programming

Heuristics for Reducing the Search Space

- Restrict to left linear trees
- Restrict to trees "without cartesian product"

Estimating Sizes

- Need size in order to estimate cost
- Example:
  - Cost of partitioned hash-join E1 ⋈ E2 is 3B(E1) + 3B(E2)
  - B(E1) = T(E1)/ block size
  - B(E2) = T(E2)/ block size
  - So, we need to estimate T(E1), T(E2)

Rule-based Optimizations

- Volcano:
  - Main idea: let programmers define rewrite rules, based on the algebraic laws
  - System searches for "best plan" by applying laws repeatedly
  - Need to avoid cycles, etc.
  - Join-reordering becomes harder, but can handle other operators too
- Starburst:
  - Same, but keep larger nodes, corresponding to one select-from-where block
  - Apply rewrite rules inter-blocks
  - Do dynamic programming inside blocks

Size Estimation

- Crucial for the optimizer's performance
- Idea:
  - Keep statistics for base tables
  - Use heuristics to compute size of operator's results
  - Works well for the first 1-2 levels, then it's way off
  - Hence: don't expect optimizers to return *best plan*, but avoid *worse plan*

Estimating Sizes

Estimating the size of a projection

- Easy: $T(\Pi_L(R)) = T(R)$
- This is because a projection doesn't eliminate duplicates

Estimating Sizes

Estimating the size of a selection

- $S = \sigma_{A=c}(R)$
  - T(S) san be anything from 0 to T(R) – V(R,A) + 1
  - Mean value: T(S) = T(R)/V(R,A)
- $S = \sigma_{A<c}(R)$
  - T(S) can be anything from 0 to T(R)
  - Heuristics: T(S) = T(R)/3

Estimating Sizes

Estimating the size of a natural join, $R \bowtie_A S$

- When the set of A values are disjoint, then $T(R \bowtie_A S) = 0$
- When A is a key in S and a foreign key in R, then $T(R \bowtie_A S) = T(R)$
- When A has a unique value, the same in R and S, then $T(R \bowtie_A S) = T(R)\ T(S)$

Estimating Sizes

Assumptions:

- *Containment of values*: if V(R,A) <= V(S,A), then the set of A values of R is included in the set of A values of S
  - Note: this indeed holds when A is a foreign key in R, and a key in S
- *Preservation of values*: for any other attribute B, $V(R \bowtie_A S, B) = V(R, B)$ (or V(S, B))

Estimating Sizes

Assume V(R,A) <= V(S,A)

- Then each tuple t in R joins *some* tuple(s) in S
  - How many ?
  - On average S/V(S,A)
  - t will contribute S/V(S,A) tuples in $R \bowtie_A S$
- Hence $T(R \bowtie_A S) = T(R)\ T(S) / V(S,A)$

In general: $T(R \bowtie_A S) = T(R)\ T(S) / \max(V(R,A),V(S,A))$

Estimating Sizes

Example:

- T(R) = 10000, T(S) = 20000
- V(R,A) = 100, V(S,A) = 200
- How large is $R \bowtie_A S$?

Answer: $T(R \bowtie_A S) = 10000\ 20000/200 = 1M$

Estimating Sizes

Joins on more than one attribute:

- $T(R \bowtie_{A,B} S) =$

  $T(R)\ T(S)/\max(V(R,A),V(S,A))\max(V(R,B),V(S,B))$

Histograms

- Statistics on data maintained by the RDBMS
- Makes size estimation much more accurate (hence, cost estimations are more accurate)

Histograms

Employee(ssn, name, salary, phone)

- Maintain a histogram on salary:

Salary:	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
Tuples	200	800	5000	12000	6500	500

- T(Employee) = 25000, but now we know the distribution

Histograms

Ranks(rankName, salary)

- Estimate the size of Employee $\bowtie_{Salary}$ Ranks

Employee	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
	200	800	5000	12000	6500	500

Ranks	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
	8	20	40	80	100	2

Histograms

- Assume:
  - V(Employee, Salary) = 200
  - V(Ranks, Salary) = 250
- Then $T(\text{Employee} \bowtie_{Salary} \text{Ranks}) =$
  $= \Sigma_{i=1,6}\ T_i\ T_i' / 250$
  $= (200x8 + 800x20 + 5000x40 +$
  $12000x80 + 6500x100 + 500x2)/250$
  $= \ldots.$