CSE 444: Database Internals

Lectures 11-12
Query Optimization (part 2)

Query Optimization Algorithm

- Enumerate alternative plans (logical & physical)
- Compute estimated cost of each plan
  - Compute number of I/Os
  - Compute CPU cost
- Choose plan with lowest cost
  - This is called cost-based optimization

Lessons

- Need to consider several physical plans
  - Even for one, simple logical plan
- No magic “best” plan: depends on the data
- In order to make the right choice
  - Need to have statistics over the data
  - The B’s, the T’s, the V’s

Outline

- Search space
- Algorithm for enumerating query plans

Relational Algebra Equivalences

- Selections
  - Commutative: $\sigma_{c_1}(\sigma_{c_2}(R))$ same as $\sigma_{c_2}(\sigma_{c_1}(R))$
  - Cascading: $\sigma_{c_1 \land c_2}(R)$ same as $\sigma_{c_2}(\sigma_{c_1}(R))$
- Projections
  - Cascading
- Joins
  - Commutative: $R \bowtie S$ same as $S \bowtie R$
  - Associative: $R \bowtie (S \bowtie T)$ same as $(R \bowtie S) \bowtie T$

Left-Deep Plans and Bushy Plans
Commutativity, Associativity, Distributivity

\[ R \cup S = S \cup R, \quad R \cup (S \cup T) = (R \cup S) \cup T \]

\[ R \bowtie S = S \bowtie R, \quad R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T \]

Laws Involving Selection

\[ \sigma_{C AND C'}(R) = \sigma_C(\sigma_{C'}(R)) = \sigma_C(R) \cap \sigma_{C'}(R) \]

\[ \sigma_{C OR C'}(R) = \sigma_C(R) \cup \sigma_{C'}(R) \]

\[ \sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S \]

\[ \sigma_C(R - S) = \sigma_C(R) - S \]

\[ \sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S) \]

\[ \sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S \]

Example: Simple Algebraic Laws

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

Laws Involving Projections

\[ \Pi_{M}(R \bowtie S) = \Pi_{M}(\Pi_{P}(R) \bowtie \Pi_{Q}(S)) \]

\[ \Pi_{M}(\Pi_{P}(R)) = \Pi_{M}(R) \]

/* note that \( M \subseteq N \) */

Example \( R(A,B,C,D), S(E, F, G) \)
\[ \Pi_{A,B,C}(R \bowtie D=E S) = \Pi_{A,B,C}(\Pi_{E}(R) \bowtie \Pi_{F}(S)) \]

Laws involving grouping and aggregation

\[ \lambda(\gamma_{A, agg}(R)) = \gamma_{A, agg}(R) \]

\[ \gamma_{A, agg}(R) = \gamma_{A, agg}(R) \]

Which of the following are "duplicate insensitive"?
sum, count, avg, min, max

\[ \gamma_{A, agg}(R) = \gamma_{A, agg}(R) \]

Laws Involving Constraints

Product(pid, pname, price, cid)
Company(cid, cname, city, state)

\[ \Pi_{pid, price}(\Pi_{cid, cname}(\Pi_{city, state}(Product \bowtie Company))) = \Pi_{pid, price}(Product) \]
Search Space Challenges

• Search space is huge!
  – Many possible equivalent trees
  – Many implementations for each operator
  – Many access paths for each relation
    • File scan or index + matching selection condition

• Cannot consider ALL plans
  – Heuristics: only partial plans with “low” cost

Outline

• Search space
  • Algorithm for enumerating query plans

Key Decisions

Logical plan
• What logical plans do we consider (left-deep, bushy ?); Search Space

• Which algebraic laws do we apply, and in which context(s) ?; Optimization rules

• In what order do we explore the search space ?; Optimization algorithm

Key Decisions

Physical plan
• What physical operators to use?

• What access paths to use (file scan or index)?

• Pipeline or materialize intermediate results?

These decisions also affect the search space

Two Types of Optimizers

• Heuristic-based optimizers:
  – Apply greedily rules that always improve plan
    • Typically: push selections down
  – Very limited: no longer used today

• Cost-based optimizers:
  – Use a cost model to estimate the cost of each plan
  – Select the “cheapest” plan
  – We focus on cost-based optimizers

Three Approaches to Search Space Enumeration

• Complete plans

• Bottom-up plans

• Top-down plans
Select * from R, S, T where R.B = S.B and S.C = T.C and R.A < 40

Why is this search space inefficient?

R(A,B)
S(B,C)
T(C,D)

Why is this better?

R(A,B)
S(B,C)
T(C,D)

Top-down Partial Plans

Dynamic Programming

Originally proposed in System R [1979]
- Only handles single block queries:

```
SELECT list
FROM R1, ..., Rn
WHERE cond1 AND cond2 AND ... AND condn
```
- Some heuristics for search space enumeration:
  - Selections down
  - Projections up
  - Avoid cartesian products

Dynamic Programming

- Search space = join trees
- Algebraic laws = commutativity, associativity
- Algorithm = dynamic programming
Selinger Optimizer Algorithm

- Original Selinger optimizer enumerates different logical and physical plans at the same time
- To simplify the discussion, we will first study the approach considering only logical plans
- We come back to the actual Selinger enumeration algorithm at the end of the lecture

Join Trees

- \( R_1 \Join R_2 \Join \ldots \Join R_n \)
- Join tree:

Types of Join Trees

- Bushy:

Types of Join Trees

- Left deep:
  - Work well with existing join algs
    - Nested-loop and hash-join
  - Facilitate pipelining
  - Selinger algorithm considers only those trees
  - Dynamic programming can be used with all trees

Dynamic Programming

Join ordering:

- Given: a query \( R_1 \Join R_2 \Join \ldots \Join R_n \)
- Find optimal order
- Assume we have a function \( \text{cost()} \) that gives us the cost of every join tree

SELECT list
FROM R_1, ..., R_n
WHERE cond_1 AND cond_2 AND ... AND cond_k
Dynamic Programming

- For each subquery \( Q \subseteq \{R_1, \ldots, R_n\} \) compute the following:
  - \( \text{Size}(Q) \) = the estimated size of \( Q \)
  - i.e., the cardinality of the result of \( Q \)
  - \( \text{Plan}(Q) \) = a best plan for \( Q \)
  - \( \text{Cost}(Q) \) = the estimated cost of that plan
- Note: we focus first on logical plans so we will use as cost estimate the sum of cardinalities of intermediate relations

```
SELECT list FROM R1, ..., Rn WHERE cond1 AND cond2 AND ... AND cond_k
```

Dynamic Programming

- **Step 1:** For each \( R_i \), set:
  - \( \text{Size}(R_i) = T(R_i) \)
  - \( \text{Plan}(R_i) = R_i \)
  - That’s the only alternative for a logical plan
  - \( \text{Cost}(R_i) = 0 \)
  - Remember that we are computing costs of logical plans

```
SELECT list FROM R1, ..., Rn WHERE cond1 AND cond2 AND ... AND cond_k
```

Dynamic Programming

- **Step 2:** For each \( Q \subseteq \{R_1, ..., R_n\} \) involving \( i \) relations:
  - \( \text{Size}(Q) \) = estimate it recursively
  - For every pair of subqueries \( Q', Q'' \) s.t. \( Q = Q' \cup Q'' \)
    - compute \( \text{cost}(\text{Plan}(Q') \Join \text{Plan}(Q'')) \)
    - \( \text{Cost}(Q) = \text{the smallest such cost} \)
  - \( \text{Plan}(Q) = \text{the least-cost plan} \)

```
SELECT list FROM R1, ..., Rn WHERE cond1 AND cond2 AND ... AND cond_k
```

Dynamic Programming

- **Step 3:** Return \( \text{Plan}((R_1, ..., R_n)) \)

```
SELECT list FROM R1, ..., Rn WHERE cond1 AND cond2 AND ... AND cond_k
```

Example

We use cost model for logical plans:

- \( \text{Cost}(P_1 \Join P_2) = \text{Cost}(P_1) + \text{Cost}(P_2) + \text{size(intermediate results for } P_1, P_2) \)
- Cost of a scan = 0

```
SELECT list FROM R1, ..., Rn WHERE cond1 AND cond2 AND ... AND cond_k
```

Example

- \( R \Join S \Join T \Join U \)
- Assumptions:
  - All join selectivities = 1%
  - \( T(R) = 2000 \)
  - \( T(S) = 5000 \)
  - \( T(T) = 3000 \)
  - \( T(U) = 1000 \)
  - \( T(R \Join S) = 0.01 \cdot T(R) \cdot T(S) \)
  - \( T(S \Join T) = 0.01 \cdot T(S) \cdot T(T) \)
  - etc.
Reducing the Search Space

- Restriction 1: only left linear trees (no bushy)
  Why?
- Restriction 2: no trees with cartesian product

\[ R(A,B) \bowtie S(B,C) \bowtie T(C,D) \]

Plan: \((R(A,B) \bowtie T(C,D)) \bowtie S(B,C)\)
has a cartesian product.
Most query optimizers will not consider it

Dynamic Programming: Summary

- Handles only join queries:
  - Selections are pushed down (i.e. early)
  - Projections are pulled up (i.e. late)
- Takes exponential time in general, BUT:
  - Left linear joins may reduce time
  - Non-cartesian products may reduce time further

Completing the Physical Query Plan

- Choose algorithm for each operator
  - How much memory do we have?
  - Are the input operand(s) sorted?
- Access path selection for base tables
- Decide for each intermediate result:
  - To materialize
  - To pipeline

More about the Selinger Algorithm

- Selinger enumeration algorithm considers
  - Different logical and physical plans at the same time
- Cost of a plan is IO + CPU
- Concept of interesting order during plan enumeration
  - Same order as that requested by ORDER BY or GROUP BY
  - Attributes that appear in equi-join predicates
    - They can speed-up a sort-merge join later
More about the Selinger Algorithm

- Step 1: Enumerate all access paths for a single relation
  - File scan or index scan
  - Keep the cheapest for each interesting order

- Step 2: Consider all ways to join two relations
  - Use result from step 1 as the outer relation
  - Consider every other possible relation as inner relation
  - Estimate cost when using sort-merge or nested-loop join
  - Keep the cheapest for each interesting order

- Steps 3 and later: Repeat for three relations, etc.

Selinger Algorithm Example

- On the white board