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

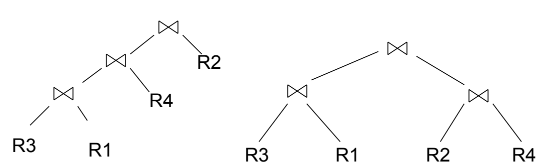
Outline

- Search space
- Algorithm for enumerating query plans

Relational Algebra Equivalences

- Selections
  - Commutative: $\sigma_{c1}(\sigma_{c2}(R))$ same as $\sigma_{c2}(\sigma_{c1}(R))$
  - Cascading: $\sigma_{c1 \wedge c2}(R)$ same as $\sigma_{c2}(\sigma_{c1}(R))$
- 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



Left-deep plan

Bushy plan

Commutativity, Associativity, Distributivity

$R \cup S = S \cup R$, $R \cup (S \cup T) = (R \cup S) \cup T$
$R \bowtie S = S \bowtie R$, $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$

$R \bowtie (S \cup T) = (R \bowtie S) \cup (R \bowtie T)$

Laws Involving Selection

$\sigma_{C \text{ AND } C'}(R) = \sigma_C(\sigma_{C'}(R)) = \sigma_C(R) \cap \sigma_{C'}(R)$
$\sigma_{C \text{ OR } C'}(R) = \sigma_C(R) \cup \sigma_{C'}(R)$
$\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$

$\sigma_C(R - S) = \sigma_C(R) - S$
$\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$
$\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$

Assuming C on attributes of R

Example: Simple Algebraic Laws

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

Laws Involving Projections

$\Pi_M(R \bowtie S) = \Pi_M(\Pi_P(R) \bowtie \Pi_Q(S))$

$\Pi_M(\Pi_N(R)) = \Pi_M(R)$
/* note that $M \subseteq N$ */

- Example R(A,B,C,D), S(E, F, G)
  $\Pi_{A,B,G}(R \bowtie_{D=E} S) = \Pi_{?}(\Pi_{?}(R) \bowtie_{D=E} \Pi_{?}(S))$

Laws involving grouping and aggregation

$\delta(\gamma_{A,\, agg(B)}(R)) = \gamma_{A,\, agg(B)}(R)$

$\gamma_{A,\, agg(B)}(\delta(R)) = \gamma_{A,\, agg(B)}(R)$
if agg is "duplicate insensitive"

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

$\gamma_{A,\, agg(D)}(R(A,B) \bowtie_{B=C} S(C,D)) =$
$\gamma_{A,\, agg(D)}(R(A,B) \bowtie_{B=C} (\gamma_{C,\, agg(D)} S(C,D)))$

Laws Involving Constraints

Foreign key

Product(<u>pid</u>, pname, price, cid)
Company(<u>cid</u>, cname, city, state)

$\Pi_{pid,\, price}(\text{Product} \bowtie_{cid=cid} \text{Company}) = \Pi_{pid,\, price}(\text{Product})$

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

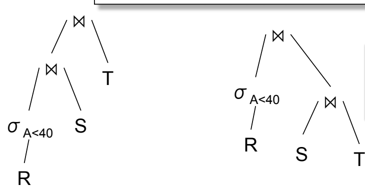
Three Approaches to Search Space Enumeration

- Complete plans
- Bottom-up plans
- Top-down plans

Complete Plans

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

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```



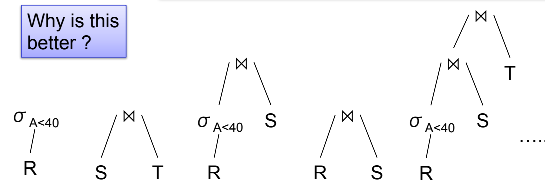
Why is this search space inefficient ?

Bottom-up Partial Plans

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

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```

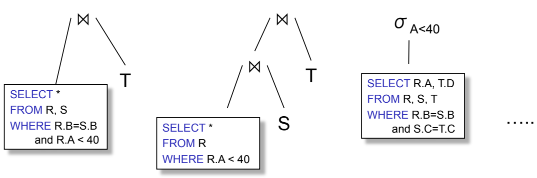
Why is this better ?



Top-down Partial Plans

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

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```



Two Types of Plan Enumeration Algorithms

- Dynamic programming (in class)
  - Based on System R (aka Selinger) style optimizer[1979]
  - Limited to joins: *join reordering algorithm*
  - Bottom-up
- Rule-based algorithm (will not discuss)
  - Database of rules (=algebraic laws)
  - Usually: dynamic programming
  - Usually: top-down

Dynamic Programming

Originally proposed in System R [1979]

- Only handles single block queries:

SELECT list
FROM R1, …, Rn
WHERE $cond_1$ AND $cond_2$ AND . . . AND $cond_k$

- Some heuristics for search space enumeration:
  - Selections down
  - Projections up
  - Avoid cartesian products

Dynamic Programming

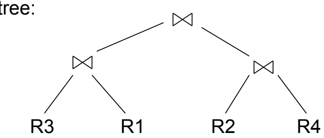
- Search space = join trees
- Algebraic laws = commutativity, associativity
- Algorithm = dynamic programming ☺

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

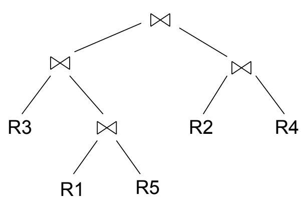
- R1 ⋈ R2 ⋈ …. ⋈ Rn
- Join tree:



- A plan = a join tree
- A partial plan = a subtree of a join tree

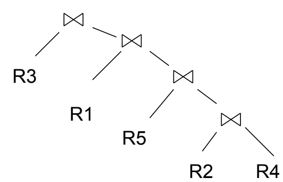
Types of Join Trees

- Bushy:



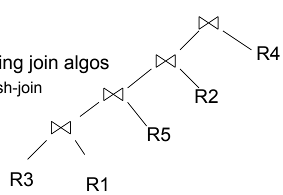
Types of Join Trees

- Right deep:



Types of Join Trees

- Left deep:
  - Work well with existing join algos
    - 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 R1 ⋈ R2 ⋈ . . . ⋈ Rn
- Find optimal order
- Assume we have a function cost() that gives us the cost of every join tree

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

Dynamic Programming

- For each subquery Q ⊆{R1, …, Rn} compute the following:
  - Size(Q) = the estimated size of Q
    - I.e., the cardinality of the result of Q
  - Plan(Q) = a best plan for Q
  - 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 condk
```


Dynamic Programming

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

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


Dynamic Programming

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

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


Dynamic Programming

- **Step 3**: Return Plan($\{R_1, …, R_n\}$)

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


Example

We use cost model for logical plans:

- Cost(P_1 ⋈ P_2) = Cost(P_1) + Cost(P_2) + size(intermediate results for P_1, P_2)
- Cost of a scan = 0

Example

```
SELECT *
FROM   R, S, T, U
WHERE cond1 AND cond2 AND . . .
```

- R ⋈ S ⋈ T ⋈ U
- Assumptions:

All join selectivities = 1%

T(R) = 2000
T(S) = 5000
T(T) = 3000
T(U) = 1000

T(R ⋈ S) = 0.01*T(R)*T(S)
T(S ⋈ T) = 0.01*T(S)*T(T)
etc.

T(R) = 2000
T(S) = 5000
T(T) = 3000
T(U) = 1000

Subquery	Size	Cost	Plan
RS			
RT			
RU			
ST			
SU			
TU			
RST			
RSU			
RTU			
STU			
RSTU			

T(R) = 2K
T(S) = 5K
T(T) = 3K
T(U) = 1K

Subquery	Size	Cost	Plan
RS	100k	0	RS
RT	60k	0	RT
RU	20k	0	UR
ST	150k	0	TS
SU	50k	0	US
TU	30k	0	UT
RST	3M	60k	(RT)S
RSU	1M	20k	(UR)S
RTU	600K	20k	(UR)T
STU	1.5M	30k	(UT)S
RSTU	30M	60k +50k=110k	(RT)(SU)

Reducing the Search Space

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

R(A,B) ⋈ S(B,C) ⋈ T(C,D)

Plan: (R(A,B)⋈T(C,D)) ⋈ 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 GY
  - 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