CS411
Database Systems

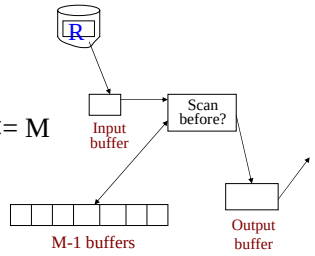
12: Query Optimization

Kazuhiro Minami

One-pass Algorithms

Duplicate elimination $\delta(R)$

- Need to keep a dictionary in memory:
  - balanced search tree
  - hash table
  - etc
- Cost: B(R)
- Assumption: $B(\delta(R)) <= M$



One-pass Algorithms

Grouping: $\gamma_{city,\ sum(price)}$ (R)

- Need to keep a dictionary in memory
- Also store the sum(price) for each city
- Cost: B(R)
- Assumption: number of cities fits in memory

Optimization

- Step 1: convert the SQL query to some logical plan
  - Remove subqueries from conditions
  - Map the SFW statement into RA expression
- Step 2: find a better logical plan, find an associated physical plan
  - Algebraic laws:
    - foundation for every optimization
  - Two approaches to optimizations:
    - Heuristics: apply laws that *seem* to result in cheaper plans
    - Cost based: estimate size and cost of intermediate results, search systematically for best plan

SQL –> Logical Query Plans

Converting from SQL to Logical Plans

```
Select a1, …, an
From R1, …, Rk
Where C
```

$\Pi_{a1,\ldots,an}(\sigma_C(R_1 \times R_2 \times \ldots \times R_k))$

```
Select a1, …, an
From R1, …, Rk
Where C
Group by b1, …, bl
```

$\Pi_{a1,\ldots,an}(\gamma_{b1,\ldots,bm,aggs}(\sigma_C(R_1 \times R_2 \times \ldots \times R_k)))$

Some nested queries can be flattened

```
Select distinct product.name
From product
Where product.maker in (Select company.name
                        From company
                        where company.city="Urbana")
```

```
Select distinct product.name
From product, company
Where product.maker = company.name AND
      company.city="Urbana"
```

Converting Nested Queries

Q: Give a list of product-manufacture pairs where the color of the product is blue and its prices is the highest among the products with blue color from that manufacture.

```
Select distinct x.name, x.maker
From product x
Where x.color= "blue"
  AND x.price >= ALL (Select y.price
                      From  product y
                      Where x.maker = y.maker
                        AND y.color="blue")
```

Q: How do we convert this one to logical plan ?

Converting Nested Queries

Let's compute the complement first:

```
Select distinct x.name, x.maker
From product x
Where x.color= "blue"
  AND x.price < SOME (Select y.price
                      From  product y
                      Where x.maker = y.maker
                        AND y.color="blue")
```

Converting Nested Queries

This one becomes a query without subqueries:

```
Select distinct x.name, x.maker
From product x, product y
Where x.color= "blue" AND x.maker = y.maker
  AND y.color="blue"  AND x.price < y.price
```

This returns exactly the products we DON'T want, so…

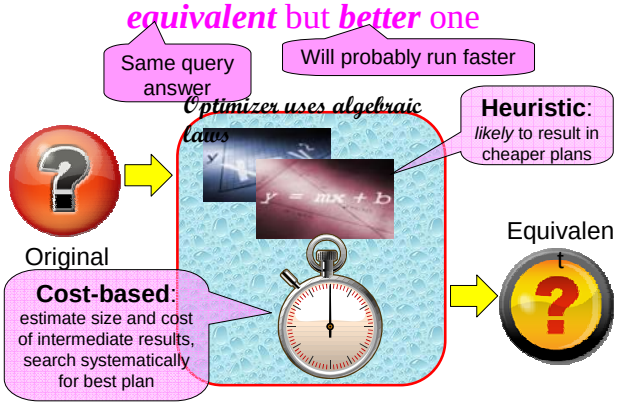
A set difference operator finishes the job

```
(Select x.name, x.maker
 From product x
 Where x.color = "blue")

EXCEPT

(Select x.name, x.maker
 From product x, product y
 Where x.color= "blue" AND x.maker = y.maker
   AND y.color="blue"  AND x.price < y.price)
```

Now rewrite the logical plan to an ***equivalent*** but ***better*** one



Algebraic Laws

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 about selections

$\sigma_{C \text{ AND } D}(R) = \sigma_C(\sigma_D(R)) = \sigma_C(R) \cap \sigma_D(R)$

$\sigma_{C \text{ OR } D}(R) = \sigma_C(R) \cup \sigma_D(R)$

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

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

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

$\sigma_C(R \cap S) = \sigma_C(R) \cap S$

if C involves only attributes of R

R(A,B,C,D) S(E,F,G)

$\sigma_{F=3}(R \bowtie_{D=E} S) = (R \bowtie_{D=E} \sigma_{F=3}(S))$

$$\begin{aligned}\sigma_{A=5 \text{ AND } G=9}(R \bowtie_{D=E} S) &= \sigma_{A=5}(\sigma_{G=9}(R \bowtie_{D=E} S)) \\ &= \sigma_{A=5}(R \bowtie_{D=E} \sigma_{G=9}(S)) \\ &= \sigma_{A=5}(R) \bowtie_{D=E} \sigma_{G=9}(S)\end{aligned}$$

Algebraic laws for projection

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

where N, P, Q are appropriate subsets of attributes of M

$\Pi_M(\Pi_N(R)) = \Pi_{M,N}(R)$

R(A,B,C,D) **S(E,F,G)**

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

Algebraic laws for 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*

~~SUM~~
~~COUNT~~
~~AVG~~
MIN
MAX

The book describes additional algebraic laws, but even the book doesn't cover them all.

Heuristics-based Optimization

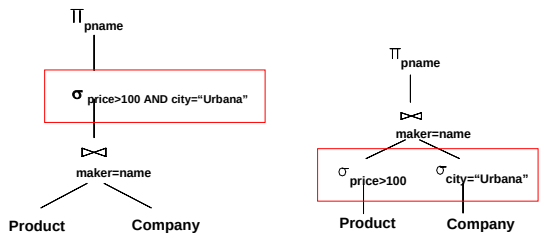
- or –

Do projections and selections as early as possible

Heuristic Based Optimizations

- Query rewriting based on algebraic laws
- Result in better queries most of the time
- Heuristics number 1:
  - Push selections down
- Heuristics number 2:
  - Sometimes push selections up, then down

Predicate Pushdown



(but may cause us to lose an important ordering of the tuples, if we use indexes).

For each company with a product costing more than $100, find the max price of its products

```
Select   y.name, y.address,
         Max(x.price)
From     product x, company y
Where    x.maker = y.name
GroupBy  y.name
Having Max(x.price) > 100
```

```
Select y.name, y.address,
       Max(x.price)
From   product x, company y
Where  x.maker=y.name  and
       x.price > 100
GroupBy  y.name
Having Max(x.price) > 100
```

- Advantage: the size of the join will be smaller.
- Requires transformation rules specific to the grouping/aggregation operators.
- **Won't work if we replace Max by Min.**

Pushing predicates up

```
Select   V2.category, V2.cname, V2.price
From     V1, V2
Where    V1.category = V2.category  and
         V1.p = V2.price
```

V1: categories and the cheapest price of the product in that category under $20

```
Create View V1 AS
Select   x.category,
         Min(x.price) AS p
From     product x
Where    x.price < 20
GroupBy  x.category
```

V2: Company name, product category, and the price of the product made by that company

```
Create View V2 AS
Select   y.cname, x.category, x.price
From     product x, company y
Where    x.maker=y.cname
```

Query Rewrite: Pushing predicates up

```
Select   V2.cname, V2.price
From     V1, V2
Where    V1.category = V2.category  and
         V1.p = V2.price AND V1.p < 20
```

```
Create View V1 AS
Select   x.category,
         Min(x.price) AS p
From     product x
Where    x.price < 20
GroupBy  x.category
```

```
Create View V2 AS
Select   y.cname, x.category, x.price
From     product x, company y
Where    x.maker=y.cname
```

Query Rewrite: Pushing predicates up

```
Select   V2.cname, V2.price
From     V1, V2
Where    V1.category = V2.category  and
         V1.p = V2.price AND V1.p < 20
```

```
Create View V1 AS
Select   x.category,
       Min(x.price) AS p
From     product x
Where  x.price < 20
GroupBy  x.category
```

```
Create View V2 AS
Select   y.cname, x.category, x.price
From     product x, company y
Where  x.maker=y.cname
   AND x.price < 20
```

Cost-based Optimization

Cost-based Optimizations

- Main idea: apply algebraic laws, until estimated cost is minimal
- Practically: start from partial plans, introduce operators one by one
  - Will see in a few slides
- Problem: there are too many ways to apply the laws, hence too many (partial) plans

Often: generate a partial plan, optimize it, then add another operator, …

Top-down: the partial plan is a top fragment of the logical plan

Bottom up: the partial plan is a bottom fragment of the logical plan

Search Strategies

- **Branch-and-bound**:
  - Remember the cheapest complete plan P seen by using heuristics so far and its cost C
  - Stop generating partial plans whose cost is > C
  - If a cheaper complete plan P is found, replace C with P
- **Hill climbing**:
  - Find nearby plans that have lower cost by making small changes to the plan
- **Dynamic programming**:
  - Compute cheapest partial plans of the smallest and compute cheapest partial plans of larger size next
  - Remember the all cheapest partial plans

Algebraic Laws for Joins

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

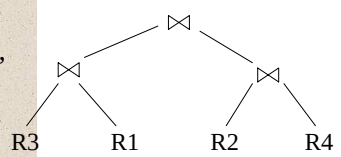
The order that relations are joined in has a huge impact on performance

Given:

query **R1 ⋈ … ⋈ Rn**,

function **cost( )**,

find the best join tree for the query

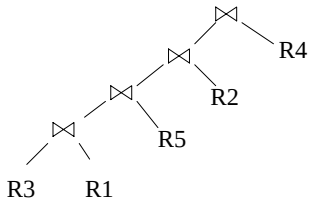


Plan = tree

Partial plan = subtree

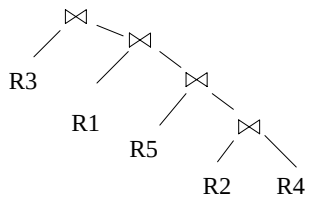
Types of Join Trees

- Left deep (all right children are leaves)



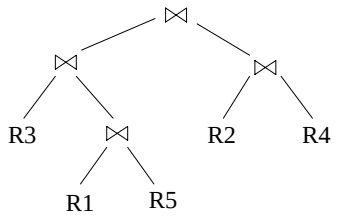
Types of Join Trees

- Right deep (all left children are leaves)



Types of Join Trees

- Bushy if neither left-deep nor right-deep



Dynamic programming is a good (bottom-up) way to choose join ordering

Find the best plan for each subquery Q of {R1, …, Rn}:

1. {R1}, …, {Rn}
2. {R1, R2}, {R1, R3}, …, {Rn-1, Rn}
3. {R1, R2, R3}, {R1, R2, R4}, …
4. …
5. {R1, …, Rn}

Output:

1. Size(Q)
2. A best plan Plan(Q)
3. Cost(Q)

The *i*th step of the dynamic program

For each Q ⊆ {R1, …, Rn} of size *i* do:

1. Compute Size(Q) (later…)
2. For every pair Q1, Q2 such that Q = Q1 ∪ Q2, compute cost(Plan(Q1) ⋈ Plan(Q2))

   Cost(Q) = the smallest such cost

   Plan(Q) = the corresponding plan

Dynamic Programming

- Return Plan({R1, …, Rn})

Dynamic Programming

To illustrate, we will make the following simplifications:

- Cost(P1 ⋈ P2) = Cost(P1) + Cost(P2) + size(intermediate results for P1 and P2)
- Intermediate results:
  - If P1 is a join, then the size of the intermediate result is size(P1), otherwise the size is 0
  - Similarly for P2
- Cost of a scan = 0

Dynamic Programming

- Example:
- Cost(R5 ⋈ R7)
  = Cost(R5) + Cost(R7)
  + intermediate results for R5 and R7
  = 0 (no intermediate results)
- Cost((R2 ⋈ R1) ⋈ R7)
  = Cost(R2 ⋈ R1) + Cost(R7) + size(R2 ⋈ R1)
  = size(R2 ⋈ R1)

Dynamic Programming

- Relations: R, S, T, U
- Number of tuples: 2000, 5000, 3000, 1000
- Size estimation: T(A ⋈ B) = 0.01*T(A)*T(B)

Subquery	Size	Cost	Plan
RS	100k	0	RS
RT	60K	0	RT
RU			
ST	1		
SU			
TU	30K	0	TU
RST			
RSU			
RTU			
STU			
RSTU			

0.01 * T(R) * T(S)
= 0.01 * 2000 * 5000
= 100,000 = 100k

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

T(A ⋈ B)
= 0.01*T(A)*T(B)

Subquery	Size	Cost	Plan
RS	100K	0	RS
RT	60K	0	RT
RU	20K	0	RU
ST	150K	0	ST
SU	50K	0	SU
TU	30K	0	TU
RST	3M	60K	(RT)S
RSU	1M	20K	(RU)S
RTU	0.6M	20K	(RU)T
STU	1.5M	30K	(TU)S
RSTU			

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

T(A ⋈ B)
= 0.01*T(A)*T(B)

Cost((RS)T)
= Cost((RS)T)
= Cost(RS) + Cost(T)
+ size(RS)
= 100k

Cost((RT)S)
= Cost((RT)S)
= Cost(RT) + Cost(S)
+ size(RT)
= 60k

Cost((ST)R) = 150k

Subquery	Size	Cost	Plan
RS	100K	0	RS
RT	60K	0	RT
RU	20K	0	RU
ST	150K	0	ST
SU	50K	0	SU
TU	30K	0	TU
RST	3M	60K	(RT)S
RSU	1M	20K	(RU)S
RTU	0.6M	20K	(RU)T
STU	1.5M	30K	(TU)S
RSTU	30M	60K+50K	(RT)(SU)

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

T(A ⋈ B)
= 0.01*T(A)*T(B)

1. Cost(R(STU)) = 30K+1.5M
2. Cost(S(RTU)) = 20K+0.6M
3. Cost(T(RSU)) = 20K+1M
4. Cost(U(RST)) = 60K+3M
5. Cost((RS)(TU)) = 130K
6. Cost((RT)(SU)) = 110K
7. Cost((RU)(ST)) = 170K

Subquery	Size	Cost	Plan
RS	100k	0	RS
RT	60k	0	RT
RU	20k	0	RU
ST	150k	0	ST
SU	50k	0	SU
TU	30k	0	TU
RST	3M	60k	(RT)S
RSU	1M	20k	(RU)S
RTU	0.6M	20k	(RU)T
STU	1.5M	30k	(TU)S
RSTU	30M	60k+50k=110k	(RT)(SU)

What if we don't oversimplify?

- More realistic size/cost estimations!! (next lecture)
- Use heuristics to reduce the search space
  - Consider only left linear trees
  - No trees with cartesian products:

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

$(R \bowtie T) \bowtie S$ has a cartesian product

Summary of query optimization process so far

1. Parse your query into tree form
2. Move selections as far down the tree as you can
3. Project out unwanted attributes as early as you can, when you have their tuples in memory anyway
4. Pick a good join order, based on the **expected size** of intermediate results
5. Pick an **implementation** for each operation in the tree