The three components of an optimizer

We need three things in an optimizer:

- Algebraic laws
- An optimization algorithm
- A cost estimator

### Algebraic Laws

- **Commutative and Associative Laws**
  - $R \cup S = S \cup R$, $R \cup (S \cup T) = (R \cup S) \cup T$
  - $R \times S = S \times R$, $R \times (S \times T) = (R \times S) \times T$
  - $R \times (S \cup T) = (R \times S) \cup (R \times T)$

- **Distributive Laws**
  - $R \times (S \cup T) = (R \times S) \cup (R \times T)$

### Example

- $R(A, B, C, D), S(E, F, G)$
  - $\sigma_{D=3}(R \mid D=E) S = ?$
  - $\sigma_{A=5 \land G=9}(R \mid D=E) S = ?$

### Laws involving selection:

- $\sigma_{C \land D}(R) = \sigma_{C}(\sigma_{D}(R)) = \sigma_{C}(R) \cap \sigma_{C}(R)$
- $\sigma_{C \lor D}(R) = \sigma_{C}(R) \cup \sigma_{C}(R)$
- $\sigma_{C}(R \mid S) = \sigma_{C}(R) \mid S$

- When $C$ involves only attributes of $R$
  - $\sigma_{C}(R - S) = \sigma_{C}(R) - S$
  - $\sigma_{C}(R \cup S) = \sigma_{C}(R) \cup \sigma_{C}(S)$
  - $\sigma_{C}(R \mid S) = \sigma_{C}(R) \mid S$

### Laws involving projections

- $\Pi_{A}(R \mid S) = \Pi_{A}(\Pi_{D}(R \mid D=E) S)$
- $\Pi_{A}(\Pi_{D}(R)) = \Pi_{A,N}(R)$

### Example

- $R(A, B, C, D), S(E, F, G)$
  - $\Pi_{A,B,G}(R \mid D=E) S = \Pi_{A}(\Pi_{D}(R) \mid D=E) \Pi_{G}(S)$
### Algebraic Laws

- Laws involving grouping and aggregation:
  \[ \delta_{\gamma_{agg}(R)}(\gamma_{agg}(R)) = \gamma_{agg}(R) \]
  \[ \gamma_{agg}(R) = \gamma_{agg}(R) \] if agg is “duplicate insensitive”

- Which of the following are “duplicate insensitive”? sum, count, avg, min, max

\[ \gamma_{agg}(R) \]

\[ \gamma_{agg}(R) \]

\[ \gamma_{agg}(R) \] if agg is “duplicate insensitive”

### Optimization Algorithms

- Heuristic based
- Cost based
  - Dynamic programming: System R
  - Rule-based optimizations: DB2, SQL-Server

### 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

- For each company, find the maximal price of its products.
- 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.

### Dynamic Programming

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

```sql
SELECT list
FROM list
WHERE cond, AND cond, AND ... AND cond,
```

- Heuristics: selections down, projections up
- Dynamic programming: join reordering
Join Trees

- $R_1 \times R_2 \times \ldots \times R_n$
- Join tree:

```
           I-1
          /   \
         I-2    \
        /      \
       R_1    R_2
        \      /  \ 
         \    /    R_3
          \  I-3   \
           \       
            \     R_4
```

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

Types of Join Trees

- Left deep:

```
           I-1
          /   \
         I-2    \
        /      \
       R_1    R_2
        \      /  \ 
         \    /    R_3
        \   I-3    \
         \      
          \     
           \    R_4
```

- Right deep:

```
           I-1
          /   \
         I-2    \
        /      \
       R_1    R_2
        \      
         \    I-3   
          \      
           \   
            \  
             \ R_3
```

Types of Join Trees

- Bushy:

```
           I-1
          /   \
         I-2    \
        /      \
       R_1    R_2
        \      /  \ 
         \    /    R_3
        \   I-3    \
         \      
          \     
           \    R_4
```

Dynamic Programming

- Given: a query $R_1 \times R_2 \times \ldots \times R_n$
- Assume we have a function cost() that gives us the cost of every join tree
- Find the best join tree for the query

Dynamic Programming

- Idea: for each subset of $\{R_1, \ldots, R_n\}$, compute the best plan for that subset
- In increasing order of set cardinality:
  - Step 1: for $\{R_1\}, \{R_2\}, \ldots, \{R_n\}$
  - Step 2: for $\{R_1, R_2\}, \{R_1, R_3\}, \ldots, \{R_{n-1}, R_n\}$
  - ...
  - Step n: for $\{R_1, \ldots, R_n\}$
- It is a bottom-up strategy
- A subset of $\{R_1, \ldots, R_n\}$ is also called a subquery
Dynamic Programming

- For each subquery $Q \subseteq \{R_1, \ldots, R_n\}$ compute the following:
  - $\text{Size}(Q)$
  - A best plan for $Q$: $\text{Plan}(Q)$
  - The cost of that plan: $\text{Cost}(Q)$

Dynamic Programming

- **Step 1**: For each $\{R_i\}$ do:
  - $\text{Size}(R_i) = B(R_i)$
  - $\text{Plan}(R_i) = R_i$
  - $\text{Cost}(R_i) = (\text{cost of scanning } R_i)$

Dynamic Programming

- **Step $i$**: For each $Q \subseteq \{R_1, \ldots, R_n\}$ of cardinality $i$ do:
  - Compute $\text{Size}(Q)$ (later...)
  - For every pair of subqueries $Q'$, $Q''$
    s.t. $Q = Q' \cup Q''$
    compute $\text{cost}(\text{Plan}(Q') \mid \text{Plan}(Q''))$
  - $\text{Cost}(Q) = \text{the smallest such cost}$
  - $\text{Plan}(Q) = \text{the corresponding plan}$

Dynamic Programming

To illustrate, we will make the following simplifications:

- $\text{Cost}(P_1 \mid P_2) = \text{Cost}(P_1) + \text{Cost}(P_2) + \text{size}(\text{intermediate result}(s))$
- Intermediate results:
  - If $P_1$ is a join, then the size of the intermediate result is $\text{size}(P_1)$, otherwise the size is 0
  - Similarly for $P_2$
- $\text{Cost of a scan} = 0$

Dynamic Programming

- Example:
  - $\text{Cost}(R_5 \mid R_7) = 0$ (no intermediate results)
  - $\text{Cost}(R_2 \mid R_1 \mid R_7) = \text{Cost}(R_2 \mid R_1) + \text{Cost}(R_7) + \text{size}(R_2 \mid R_1)$
    $= \text{size}(R_2 \mid R_1)$
Dynamic Programming

• Relations: R, S, T, U
• Number of tuples: 2000, 5000, 3000, 1000
• Size estimation: \( T(A \mid \times B) = 0.01 \times T(A) \times T(B) \)

Reducing the Search Space

• Left-linear trees v.s. Bushy trees
• Trees without cartesian product

Example: \( R(A,B) \mid \times S(B,C) \mid \times T(C,D) \)

Plan: \( (R(A,B) \mid \times T(C,D)) \mid \times S(B,C) \) 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

Rule-Based Optimizers

• Extensible collection of rules
  Rule = Algebraic law with a direction
• Algorithm for firing these rules
  Generate many alternative plans, in some order
  Prune by cost
• Volcano (later SQL Sever)
• Starburst (later DB2)
Completing the Physical Query Plan

- Choose algorithm to implement each operator
  - Need to account for more than cost:
    - How much memory do we have?
    - Are the input operand(s) sorted?
- Decide for each intermediate result:
  - To materialize
  - To pipeline

Materialize Intermediate Results Between Operators

Question in class

Given B(R), B(S), B(T), B(U)

- What is the total cost of the plan?
  - Cost =
- How much main memory do we need?
  - M =

Pipeline Between Operators

Question in class

Given B(R), B(S), B(T), B(U)

- What is the total cost of the plan?
  - Cost =
- How much main memory do we need?
  - M =

Pipeline in Bushy Trees
Example

- Logical plan is:

```
  k blocks
   / \
  R(x) S(x)
   \ /
  5,000 blocks 10,000 blocks
```

- Main memory M = 101 buffers

Example

```
M = 101

  k blocks
   / \
  R(x) S(x)
   \ /
  5,000 blocks 10,000 blocks
```

Smarter:
- Step 1: hash R on x into 100 buckets, each of 50 blocks; to disk
- Step 2: hash S on x into 100 buckets; to disk
- Step 3: read each R, in memory (50 buffer) join with S, (1 buffer); hash result on y into 50 buckets (50 buffers) — here we pipeline
- Cost so far: 3B(R) + 3B(S)

Example

```
M = 101

  k blocks
   / \
  R(x) S(x)
   \ /
  5,000 blocks 10,000 blocks
```

Continuing:
- How large are the 50 buckets on y? Answer: k/50.
- If k <= 50 then keep all 50 buckets in Step 3 in memory, then:
  - Step 4: read U from disk, hash on y and join with memory
  - Total cost: 3B(R) + 3B(S) + B(U) = 55,000

Example

```
M = 101

  k blocks
   / \
  R(x) S(x)
   \ /
  5,000 blocks 10,000 blocks
```

Continuing:
- If 50 < k <= 5000 then send the 50 buckets in Step 3 to disk
  - Each bucket has size k/50 <= 100
  - Step 4: partition U into 50 buckets
  - Step 5: read each partition and join in memory
  - Total cost: 3B(R) + 3B(S) + 2k + 3B(U) = 73,000 + 2k

Example

```
M = 101

  k blocks
   / \
  R(x) S(x)
   \ /
  5,000 blocks 10,000 blocks
```

Continuing:
- If k > 5000 then materialize instead of pipeline
  - 2 partitioned hash-joins
  - Cost 3B(R) + 3B(S) + 4k + 3B(U) = 75000 + 4k
Example

Summary:
• If $k \leq 50$,  \quad cost = 55,000
• If $50 < k \leq 5000$,  \quad cost = 75,000 + 2k
• If $k > 5000$,  \quad cost = 75,000 + 4k