CS 525: Advanced Database Organization

11: Query Optimization

Physical

Boris Glavic

Slides: adapted from a course taught by Hector Garcia-Molina, Stanford InfoLab
SQL query

parse

parse tree

convert

logical query plan

apply laws

"improved" l.q.p

estimate result sizes

l.q.p. + sizes

consider physical plans

{P1, P2, ....}

answer

execute

pick best

{(P1, C1), (P2, C2), ...}

estimate costs

statistics
Cost of Query

• Parse + Analyze
• Optimization – Find plan
• Execution
• Return results to client
Cost of Query

- Parse + Analyze
  - Can parse MB of SQL code in milisecs

- Optimization – Find plan
  - Generating plans, costing plans

- Execution
  - Execute plan

- Return results to client
  - Can be expensive but not discussed here
Physical Optimization

• Apply after applying heuristics in logical optimization

• 1) Enumerate potential execution plans
   – All?
   – Subset

• 2) Cost plans
   – What cost function?
Physical Optimization

• To apply pruning in the search for the best plan
  – Steps 1 and 2 have to be interleaved
  – Prune parts of the search space
• if we know that it cannot contain any plan that is better than what we found so far
Example Query

```
SELECT e.name
FROM Employee e,
     EmpDep ed,
     Department d
WHERE e.name = ed.emp
  AND ed.dep = d.dep
  AND d.dep = 'CS'
```
Example Query – Possible Plan

```
SELECT e.name
FROM Employee e,
    EmpDep ed,
    Department d
WHERE e.name = ed.emp
    AND ed.dep = d.dep
    AND d.dep = 'CS'
```
Cost Model

• Cost factors
  – #disk I/O
  – CPU cost
  – Response time
  – Total execution time

• Cost of operators
  – I/O as discussed in query execution (part 10)
  – Need to know size of intermediate results (part 09)
Example Query – Possible Plan

```
SELECT e.name
FROM Employee e, EmpDep ed, Department d
WHERE e.name = ed.emp
  AND ed.dep = d.dep
  AND d.dep = 'CS'
```

Cost?

Need input size!

Diagram:

- \( \pi \) on \( \text{name} \)
- \( \sigma \) on \( \text{dep} = \text{CS} \)
- \( \Join \) on \( \text{name} = \text{emp} \)
- \( \Join \) on \( \text{dep} = \text{dep} \)
- \( \Join \) on \( \text{emp} \)
- \( \Join \) on \( \text{dep} = \text{CS} \)
Cost Model Trade-off

- **Precision**
  - Incorrect cost-estimation -> choose suboptimal plan

- **Cost of computing cost**
  - Cost of costing a plan
    - We may have to cost millions or billions of plans
  - Cost of maintaining statistics
    - Occupies resources needed for query processing
Plan Enumeration

• For each operator in the query
  – Several implementation options

• Binary operators (joins)
  – Changing the order may improve performance a lot!

• -> consider both different implementations and order of operators in plan enumeration
Example Join Ordering
Result Sizes

\[ \sigma_{\text{dep}=\text{CS}} \bowtie \sigma_{\text{name}=\text{emp}} \]

\[ \bowtie \text{dep} = \text{dep} \]

\[ \bowtie \text{dep} = \text{dep} \]

\[ 
\begin{array}{c}
E \\
10000 \\
\text{name}=\text{emp} \\
10000 \\
\text{dep}=\text{dep} \\
10000 \\
\sigma_{\text{dep}=\text{CS}} \\
1 \\
30 \\
D \\
1 \\
500 \\
D \\
1 \\
500 \\
E \\
10000 \\
\end{array}
\]
Example Join Ordering
Cost (only NL)

\[ S(E) = S(ED) = S(D) = \frac{1}{10} \text{ block} \]
\[ M = 101 \]
\[ S(E) = S(ED) = S(D) = \frac{1}{10} \text{ block} \]
\[ M = 101 \]
I/O costs only
No pipelining, write all results to disk

\[
1100 \times 10 + 101 \times 10 + 3 \text{ (operator costs)} + 1000 + 1 + 50 \text{ (write results)} = 13064 \text{ I/Os}
\]

\[
1001 + 1050 + 3 \text{ (operator costs)} + 1 + 50 + 50 = 2155 \text{ I/Os}
\]
Plan Enumeration

• **All**
  – Consider all potential plans of a certain type (discussed later)
  – Prune only if sure

• **Heuristics**
  – Apply heuristics to prune search space

• **Randomized Algorithms**
Plan Enumeration Algorithms

- **All**
  - Dynamic Programming (System R)
  - A* search

- **Heuristics**
  - Minimum Selectivity, Intermediate result size, ...
  - KBZ-Algorithm, AB-Algorithm

- **Randomized**
  - Genetic Algorithms
  - Simulated Annealing
Reordering Joins Revisited

• Equivalences (Natural Join)
  1. $R \bowtie S \equiv S \bowtie R$
  2. $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$

• Equivalences Equi-Join
  1. $R \bowtie_{a=b} S \equiv S \bowtie_{a=b} R$
  2. $(R \bowtie_{a=b} S) \bowtie_{c=d} T \equiv R \bowtie_{a=b} (S \bowtie_{c=d} T)$?
  3. $\sigma_{a=b} (R \times S) \equiv R \bowtie_{a=b} S$?
Equi-Join Equivalences

- \((R \bowtie_{a=b} S) \bowtie_{c=d} T \equiv R \bowtie_{a=b} (S \bowtie_{c=d} T)\)
  - What if \(c\) is attribute of \(R\)?
  \((R \bowtie_{a=b} S) \bowtie_{c=d} T \equiv R \bowtie_{a=b \land c=d} (S \times T)\)

- \(\sigma_{a=b} (R \times S) \equiv R \bowtie_{a=b} S?\)
  - Only useful if \(a\) is from \(R\) and \(S\) from \(b\) (vice-versa)
Why Cross-Products are bad

• We discussed efficient join algorithms
  – Merge-join $O(n)$ resp. $O(n \log(n))$
  – Vs. Nested-loop $O(n^2)$

• $R \times S$
  – Result size is $O(n^2)$
    • Cannot be better than $O(n^2)$
  – Surprise, surprise: merge-join doesn’t work
    no need to sort, but degrades to nested loop
Agenda

• Given some query
  – How to enumerate all plans?
• Try to avoid cross-products
• Need way to figure out if equivalences can be applied
  – Data structure: Join Graph
Join Graph

• Assumptions
  – Only equi-joins \(a = b\)
    • \(a\) and \(b\) are either constants or attributes
  – Only conjunctive join conditions (AND)
Join Graph

- Nodes: Relations $R_1, \ldots, R_n$ of query
- Edges: Join conditions
  - Add edge between $R_i$ and $R_j$ labeled with $C$
    - if there is a join condition $C$
    - That equates an attribute from $R_i$ with an attribute from $R_j$
  - Add a self-edge to $R_i$ for each simple predicate
Join Graph Example

SELECT e.name
FROM Employee e,
    EmpDep ed,
    Department d
WHERE e.name = ed.emp
AND ed.dep = d.dep
AND d.dep = 'CS'
Join Graph Example

```sql
SELECT e.name
FROM Employee e, EmpDep ed, Department d
WHERE e.name = ed.emp
  AND ed.dep = d.dep
  AND d.dep = 'CS'
```
Notes on Join Graph

• Join Graph tells us in which ways we can join without using cross products

• However, ...
  – Only if transitivity is considered
Join Graph Shapes

- Chain queries
- Star queries
- Tree queries
- Cycle queries
- Clique queries
Join Graph Shapes

SELECT *
FROM R,S,T
WHERE R.a = S.b
AND S.c = T.d

Chain queries
Join Graph Shapes

Star queries

```
SELECT * 
FROM R,S,T,U
WHERE R.a = S.a 
   AND R.b = T.b 
   AND R.c = U.c
```
Join Graph Shapes

```
SELECT * 
FROM R,S,T,U,V 
WHERE R.a = S.a 
    AND R.b = T.b 
    AND T.c = U.c 
    AND T.d = V.d 
```

Tree queries
Join Graph Shapes

Cycle queries

```
SELECT *
FROM R,S,T
WHERE R.a = S.a
    AND S.b = T.b
    AND T.c = R.c
```
Join Graph Shapes

\[
\text{SELECT } * \\
\text{FROM } R,S,T \\
\text{WHERE } R.a = S.a \\
\quad \text{AND } S.b = T.b \\
\quad \text{AND } T.c = R.c
\]
How many join orders?

• Assumption
  – Use cross products (can freely reorder)
  – Joins are binary operations
    • Two inputs
    • Each input either join result or relation access
How many join orders?

- Example 3 relations R, S, T
  - 12 orders
How many join orders?

- A join over \( n+1 \) relations requires \( n \) binary joins.
- The root of the join tree joins \( k \) with \( n - k - 1 \) join operators (\( 0 \leq k \leq n-1 \)).
How many join orders?

- This are the **Catalan numbers**

\[ C_n = \sum_{k=0}^{n-1} C_k \times C_{n-k-1} = \frac{(2n)!}{(n+1)!n!} \]

\[ C_0 = 1 \]
How many join orders?

• This are the **Catalan numbers**
• For each such tree we can permute the input relations \((n+1)!\) Permutations

\[
(2n)! / (n+1)!n! * (n+1)! = (2n)!/n!
\]
How many join orders?

<table>
<thead>
<tr>
<th>#relations</th>
<th>#join trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>120</td>
</tr>
<tr>
<td>5</td>
<td>1,680</td>
</tr>
<tr>
<td>6</td>
<td>30,240</td>
</tr>
<tr>
<td>7</td>
<td>665,280</td>
</tr>
<tr>
<td>8</td>
<td>17,297,280</td>
</tr>
<tr>
<td>9</td>
<td>17,643,225,600</td>
</tr>
<tr>
<td>10</td>
<td>670,442,572,800</td>
</tr>
<tr>
<td>11</td>
<td>28,158,588,057,600</td>
</tr>
</tbody>
</table>
How many join orders?

• If for each join we consider $k$ join algorithms then for $n$ relations we have
  – Multiply with a factor $k^{n-1}$

• Example consider
  – Nested loop
  – Merge
  – Hash
### How many join orders?

<table>
<thead>
<tr>
<th>#relations</th>
<th>#join trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>108</td>
</tr>
<tr>
<td>4</td>
<td>3240</td>
</tr>
<tr>
<td>5</td>
<td>136,080</td>
</tr>
<tr>
<td>6</td>
<td>7,348,320</td>
</tr>
<tr>
<td>7</td>
<td>484,989,120</td>
</tr>
<tr>
<td>8</td>
<td>37,829,151,360</td>
</tr>
<tr>
<td>9</td>
<td>115,757,203,161,600</td>
</tr>
<tr>
<td>10</td>
<td>13,196,321,160,422,400</td>
</tr>
<tr>
<td>11</td>
<td>1,662,736,466,213,222,400</td>
</tr>
</tbody>
</table>
Too many join orders?

- Even if costing is cheap
  - Unrealistic assumption 1 CPU cycle
  - Realistic are thousands or millions of instructions

- Cost all join options for 11 relations
  - 3GHz CPU, 8 cores
  - 69,280,686 sec > 2 years
How to deal with excessive number of combinations?

• Prune parts based on optimality
  – Dynamic programming
  – A*-search

• Only consider certain types of join trees
  – Left-deep, Right-deep, zig-zag, bushy

• Heuristic and random algorithms
Dynamic Programming

• Assumption: **Principle of Optimality**
  – To compute the *global* optimal plan it is only necessary to consider the optimal solutions for its *sub-queries*

• Does this assumption hold?
  – Depends on cost-function
What is dynamic programming?

- Recall data structures and algorithms 101!
- Consider a **Divide-and-Conquer** problem
  - Solutions for a problem of size $n$ can be build from solutions for sub-problems of smaller size (e.g., $n/2$ or $n-1$)

- **Memoize**
  - Store solutions for sub-problems
  - $\rightarrow$ Each solution has to be only computed once
  - $\rightarrow$ Needs extra memory
Example Fibonacci Numbers

- \( F(n) = F(n-1) + F(n-2) \)
- \( F(0) = F(1) = 1 \)

```
Fib(n) {
    if (n = 0) return 0
    else if (n = 1) return 1
    else return Fib(n-1) + Fib(n-2)
}
```
Example Fibonacci Numbers

F(4)

F(3)     F(2)

F(2)     F(1)     F(1)     F(0)

F(1)     F(0)
Complexity

- Number of calls
  - $C(n) = C(n-1) + C(n-2) + 1 = \text{Fib}(n+2)$
  - $O(2^n)$
Using dynamic programming

```cpp
Fib(n) {
    int[] fib;
    fib[0] = 1;
    fib[1] = 1;

    for(i = 2; i < n; i++)
        fib[i] = fib[i-1] + fib[i-2]

    return fib[n];
}
```
Example Fibonacci Numbers

\[ \begin{align*}
F(4) \\
F(3) \\
F(2) \\
F(1) \\
F(0)
\end{align*} \]
What do we gain?

- $O(n)$ instead of $O(2^n)$
Dynamic Programming for Join Enumeration

- Find cheapest plan for n-relation join in n passes
- For each $i$ in $1 \ldots n$
  - Construct solutions of size $i$ from best solutions of size $< i$
DP Join Enumeration

\[
\text{optPlan} \leftarrow \text{Map}([R],\{\text{plan}\})
\]

\[
\text{find\_join\_dp}(q(R_1,...,R_n))
\]
\[
\begin{align*}
& \text{for } i=1 \text{ to } n \\
& \quad \text{optPlan}[\{R_i\}] \leftarrow \text{access\_paths}(R_i) \\
& \text{for } i=2 \text{ to } n \\
& \quad \text{foreach } S \subseteq \{R_1,...,R_n\} \text{ with } |S|=i \\
& \quad \quad \text{optPlan}[S] \leftarrow \emptyset \\
& \quad \quad \text{foreach } O \subset S \text{ with } O \neq \emptyset \\
& \quad \quad \quad \text{optPlan}[S] \leftarrow \text{optPlan}[S] \cup \text{possible\_joins}(\text{optPlan}(O), \text{optPlan}(S\setminus O)) \\
& \quad \quad \text{prune\_plans}(\text{optPlan}[S]) \\
& \quad \text{return } \text{optPlan}[\{R_1,...,R_n\}]
\end{align*}
\]
Dynamic Programming for Join Enumeration

- **access_paths (R)**
  - Find cheapest access path for relation R

- **possible_joins(plan, plan)**
  - Enumerate all joins (merge, NL, ...) variants between the input plans

- **prune_plans({plan})**
  - Only keep cheapest plan from input set
DP-JE Complexity

• Time: $O(3^n)$
• Space: $O(2^n)$
• Still too much for large number of joins (10-20)
Types of join trees

- **Left-deep**
  - Diagram: [Diagram]

- **zig-zag**
  - Diagram: [Diagram]

- **bushy**
  - Diagram: [Diagram]
  - Additional details: +left, + zig-zag, +right

- **Right-deep**
  - Diagram: [Diagram]
Number of Join-Trees

- Number of join trees for $n$ relations
- Left-deep: $n!$
- Right-deep: $n!$
- Zig-zag: $2^{n-2}n!$
### How many join orders?

<table>
<thead>
<tr>
<th>#relations</th>
<th>#bushy join trees</th>
<th>#left-deep join trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>120</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>1,680</td>
<td>120</td>
</tr>
<tr>
<td>6</td>
<td>30,240</td>
<td>720</td>
</tr>
<tr>
<td>7</td>
<td>665,280</td>
<td>5040</td>
</tr>
<tr>
<td>8</td>
<td>17,297,280</td>
<td>40,230</td>
</tr>
<tr>
<td>9</td>
<td>17,643,225,600</td>
<td>362,880</td>
</tr>
<tr>
<td>10</td>
<td>670,442,572,800</td>
<td>3,628,800</td>
</tr>
<tr>
<td>11</td>
<td>28,158,588,057,600</td>
<td>39,916,800</td>
</tr>
</tbody>
</table>
DP with Left-deep trees only

- Reduced search-space
- Each join is with input relation
  - can use index joins
  - easy to pipe-line
- DP with left-deep plans was introduced by system R, the first relational database developed by IBM Research
Revisiting the assumption

• Is it really sufficient to only look at the best plan for every sub-query?
• Cost of merge join depends whether the input is already sorted
  → A sub-optimal plan may produce results ordered in a way that reduces cost of joining above
  – Keep track of interesting orders
Interesting Orders

• Number of interesting orders is usually small
• -> Extend DP join enumeration to keep track of interesting orders
  – Determine interesting orders
  – For each sub-query store best-plan for each interesting order
Example Interesting Orders

Left-deep best plans: 3-way \{R,S,T\}

Left-deep best plans: 2-way

\{R,S\}  \{R,T\}  \{S,T\}
Example Interesting Orders

Left-deep best plans: 3-way \( \{R,S,T\} \)

Left-deep best plans: 2-way

\[ \{R,S\} \]

Not best

\[ \{R,T\} \]

\[ \{S,T\} \]
Greedy Join Enumeration

- Heuristic method
  - Not guaranteed that best plan is found
- Start from single relation plans
- In each iteration greedily join to plans with the minimal cost
- Until a plan for the whole query has been generated
Greedy Join Enumeration

\[
\text{plans} \leftarrow \text{list}\{\text{plan}\}
\]

\[
\text{find_join_dp}(q(R_1,\ldots,R_n))
\{
\text{for } i=1 \text{ to } n
\quad \text{plans} \leftarrow \text{plans} \cup \text{access_paths}(R_i)
\text{for } i=n \text{ to } 2
\quad \text{cheapest} = \text{argmin}_{j,k\in \{1,\ldots,n\}} \text{ (cost}(P_j \bowtie P_k))
\quad \text{plans} \leftarrow \text{plans} \setminus \{P_j,P_k\} \cup \{P_j \bowtie P_k\}
\}
\text{return plans} \quad // \quad \text{single plan left}
\]
Greedy Join Enumeration

• Time: $O(n^3)$
  – Loop iterations: $O(n)$
  – In each iterations looking of pairs of plans in of max size n: $O(n^2)$

• Space: $O(n^2)$
  – Needed to store the current list of plans
Randomized Join-Algorithms

- Iterative improvement
- Simulated annealing
- Tabu-search
- Genetic algorithms
Transformative Approach

• Start from (random) complete solutions
• Apply transformations to generate new solutions
  – Direct application of equivalences
    • Commutativity
    • Associativity
  – Combined equivalences
    • E.g., $(R \bowtie S) \bowtie T \equiv T \bowtie (S \bowtie R)$
Concern about Transformative Approach

- Need to be able to generate random plans fast
- Need to be able to apply transformations fast
  - Trade-off: space covered by transformations vs. number and complexity of transformation rules
Iterative Improvement

```plaintext
improve(q(R_1, ..., R_n))
{
    best ← random_plan(q)
    while (not reached time limit)
        curplan ← random_plan(q)
        do
            prevplan ← curplan
            curplan ← apply_random_trans(prevplan)
            while (cost(curplan) < cost(prevplan))
                if (cost(prevplan) < cost(best))
                    best ← prevplan
        return best
}
```
Iterative Improvement

- Easy to get stuck in local minimum
- **Idea:** Allow transformations that result in more expensive plans with the hope to move out of local minima
  - -> Simulated Annealing
Simulated Annealing

\( \text{SA}(q(R_1, \ldots, R_n)) \)

\{
  \text{best} \leftarrow \text{random\_plan}(q) \\
  \text{curplan} \leftarrow \text{best} \\
  t \leftarrow t_{\text{init}} // \text{“temperature”} \\
  \text{while} \ (t > 0) \\
    \text{newplan} \leftarrow \text{apply\_random\_trans}(\text{curplan}) \\
    \text{if} \ \text{cost(newplan)} < \text{cost(curplan)} \\
      \text{curplan} \leftarrow \text{newplan} \\
    \text{else if} \ \text{random()} < e^{-(\text{cost(newplan)}-\text{cost(curplan)})/t} \\
      \text{curplan} \leftarrow \text{newplan} \\
    \text{if} \ (\text{cost(curplan)} < \text{cost(best)}) \\
      \text{best} \leftarrow \text{curplan} \\
    \text{reduce}(t) \\
  \text{return} \ \text{best}
\}
Simulated Annealing

\[ SA(q(R_1,\ldots,R_n)) \]

\[
\text{best} \leftarrow \text{random\_plan}(q) \\
\text{curplan} \leftarrow \text{best} \\
t \leftarrow t_{\text{init}} // \text{“temperature”} \\
\text{while} \ (t > 0) \\
\quad \text{newplan} \leftarrow \text{apply\_random\_trans}(\text{curplan}) \\
\quad \text{if} \ \text{cost(newplan)} < \text{cost(curplan)} \\
\quad \quad \text{curplan} \leftarrow \text{newplan} \\
\quad \text{else if random()} < e^{-\frac{\text{cost(newplan)} - \text{cost(curplan)}}{t}} \\
\quad \quad \text{curplan} \leftarrow \text{newplan} \\
\quad \text{if} \ \text{cost(curplan)} < \text{cost(best)} \\
\quad \quad \text{best} \leftarrow \text{curplan} \\
\quad \text{reduce}(t) \\
\] 

\[
\text{return best} \\
\]

Until “cooled down”

Reduce Chance To “jump”

Probability to Take “bad” plan Based on temp.
Genetic Algorithms

• Represent solutions as sequences (strings) = genome

• Start with random population of solutions

• Iterations = Generations
  – Mutation = random changes to genomes
  – Cross-over = Mixing two genomes
Genetic Join Enumeration for Left-deep Plans

• A left-deep plan can be represented as a permutation of the relations
  – Represent each relation by a number
  – E.g., encode this tree as “1243”
Mutation

- Switch random two random positions
- Is applied with a certain fixed probability
- E.g., “1342” -> “4312”
Cross-over

• Sub-set exchange
  – For two solutions find subsequence
    • equals length with the same set of relations
  – Exchange these subsequences

• Example
  – \( J_1 = "5632478" \) and \( J_2 = "5674328" \)
  – Generate \( J' = "5643278" \)
Survival of the fittest

• Probability of survival determined by rank within the current population
• Compute ranks based on costs of solutions
• Assign Probabilities based on rank
  – Higher rank -> higher probability to survive
• Roll a dice for each solution
Genetic Join Enumeration

- Create an initial population $P$ random plans
- Apply crossover and mutation with a fixed rate
  - E.g., crossover 65%, mutation 5%
- Apply selection until size is again $P$
- Stop once no improvement for at least $X$ iterations
Comparison Randomized Join

- **Iterative Improvement**
  - Towards local minima (easy to get stuck)

- **Simulated Annealing**
  - Probability to “jump” out of local minima

- **Genetic Algorithms**
  - Random transformation
  - Mixing solutions (crossover)
  - Probabilistic chance to keep solution based on cost
Join Enumeration Recap

• Hard problem
  – Large problem size
    • Want to reduce search space
  – Large cost differences between solutions
    • Want to consider many solution to increase chance to find a good one.
Join Enumeration Recap

• Tip of the iceberg
  – More algorithms
  – Combinations of algorithms
  – Different representation subspaces of the problem
  – Cross-products / no cross-products
  – …
From Join-Enumeration to Plan Enumeration

- So far we only know how to reorder joins
- What about other operations?
- What if the query does consist of several SQL blocks?
- What if we have nested subqueries?
SQL query

parse

parse tree

convert

logical query plan

apply laws

“improved” l.q.p

estimate result sizes

l.q.p. + sizes

consider physical plans

\{P_1, P_2, \ldots \}\n
\{(P_1, C_1), (P_2, C_2), \ldots \}\n
answer

execute

\Pi_i

pick best

estimate costs

statistics
From Join-Enumeration to Plan Enumeration

• Lets reconsider the input to plan enumeration!
  – We briefly touched on **Query graph models**
  – We discussed briefly why relational algebra is not sufficient
Query Graph Models

- Represents an SQL query as query blocks
  - A query block corresponds to an SQL query block (SELECT FROM WHERE ...)
  - Data type/operator/function information
    - Needed for execution and optimization decisions
  - Structured in a way suited for optimization
QGM example

SELECT name, city
FROM
  (SELECT * FROM person) AS p,
  (SELECT * FROM address) AS a
WHERE p.addrId = a.id
Postgres Example

```
{QUERY

:commandType 1
:querySource 0
:canSetTag true
:utilityStmt <>
:resultRelation 0
:intoClause <>
:hasAggs false
:hasSubLinks false
:rtable (  
   {RTE
    :alias
     {ALIAS
      :aliasname p
      :colnames <>
     }
    :eref
     {ALIAS
      :aliasname p
      :colnames ("name" "addrid")
     }
    :rtekind 1
    :subquery
     {QUERY
      :commandType 1
      :querySource 0
      :canSetTag true
     }
   )
}
```
How to enumerate plans for a QGM query

• Recall the correspondence between SQL query blocks and algebra expressions!
• If block is (A)SPJ
  – Determine join order
  – Decide which aggregation to use (if any)
• If block is set operation
  – Determine order
More than one query block

• Recursive create plans for subqueries
  – Start with leaf blocks

• Consider our example
  – Even if blocks are only SPJ we would not consider reordering of joins across blocks
  – -> try to “pull up” subqueries before optimization
Subquery Pull-up

SELECT name, city
FROM
  (SELECT *
   FROM person) AS p,
  (SELECT *
   FROM address) AS a
WHERE p.addrId = a.id

SELECT name, city
FROM
  person p,
  address a
WHERE p.addrId = a.id
Parameterized Queries

• Problem
  – Repeated executed of similar queries

• Example
  – Webshop
  – Typical operation: Retrieve product with all user comments for that product
  – Same query modulo product id
Parameterized Queries

• Naïve approach
  – Optimize each version individually
  – Execute each version individually

• Materialized View
  – Store common parts of the query
  – -> Optimizing a query with materialized views
  – -> Separate topic not covered here
Caching Query Plans

• Caching Query Plans
  – Optimize query once
  – Adapt plan for specific instances
  – Assumption: varying values do not effect optimization decisions
  – Weaker Assumption: Additional cost of “bad” plan less than cost of repeated planning
Parameterized Queries

- How to represent varying parts of a query
  - Parameters
  - Query planned with parameters assumed to be unknown
  - For execution replace parameters with concrete values
PREPARE statement

• In SQL
  – `PREPARE name (parameters) AS` query
  – `EXECUTE name (parameters)`
Nested Subqueries

SELECT name
FROM person p
WHERE EXISTS (SELECT newspaper
              FROM hasRead h
              WHERE h.name = p.name
              AND h.newspaper = 'Tribune')
How to evaluate nested subquery?

• If no correlations:
  – Execute once and cache results
• For correlations:
  – Create plan for query with parameters
• -> called nested iteration
Nested Iteration - Correlated

q ← outer query
q' ← inner query
result ← \texttt{execute}(q)
\texttt{foreach} tuple t \texttt{in} result
  qt ← q'(t) // parameterize q’ with values from t
  result' ← \texttt{execute} (qt)
  \texttt{evaluate\_nested\_condition} (t,result')
Nested Iteration - Uncorrelated

q ← outer query
q' ← inner query
result ← \texttt{execute}(q)
result' ← \texttt{execute}(q_t)
\texttt{foreach} tuple t in result
    evaluate\_nested\_condition (t, result')
Nested Iteration - Example

```sql
SELECT name
FROM person p
WHERE EXISTS (SELECT newspaper
               FROM hasRead h
               WHERE h.name = p.name
               AND h.newspaper = 'Tribune')
```

<table>
<thead>
<tr>
<th>person</th>
<th>hasRead</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name</strong></td>
<td><strong>newspaper</strong></td>
</tr>
<tr>
<td>Alice</td>
<td>Tribune</td>
</tr>
<tr>
<td>Bob</td>
<td>Courier</td>
</tr>
<tr>
<td>Joe</td>
<td>Courier</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>name</strong></th>
<th><strong>gender</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>female</td>
</tr>
<tr>
<td>Bob</td>
<td>male</td>
</tr>
<tr>
<td>Joe</td>
<td>male</td>
</tr>
</tbody>
</table>
Nested Iteration - Example

\[
q \leftarrow \text{outer query} \\
q' \leftarrow \text{inner query} \\
\text{result} \leftarrow \text{execute}(q) \\
\text{foreach tuple } t \text{ in } \text{result} \\
q_t \leftarrow q'(t) \\
\text{result}' \leftarrow \text{execute } (q_t) \\
\text{evaluate_nested_condition } (t, \text{result}')
\]

\[
\text{SELECT newspaper} \\
\text{FROM hasRead } h \\
\text{WHERE } h.\text{name} = p.\text{name} \\
\quad \text{AND } h.\text{newspaper} = \text{‘Tribune’}
\]

<table>
<thead>
<tr>
<th>person</th>
<th>hasRead</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Person Table" /></td>
<td><img src="#" alt="HasRead Table" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>gender</th>
<th>newspaper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>female</td>
<td>Tribune</td>
</tr>
<tr>
<td>Bob</td>
<td>male</td>
<td></td>
</tr>
<tr>
<td>Joe</td>
<td>male</td>
<td>Courier</td>
</tr>
</tbody>
</table>
Nested Iteration - Example

```
q ← outer query
q' ← inner query
result ← execute(q)
foreach tuple t in result
  qt ← q'(t)
  result' ← execute(qt)
evaluate_nested_condition(t, result')
```

```
SELECT newspaper
FROM hasRead h
WHERE h.name = 'Alice'
  AND h.newspaper = 'Tribune')
```
Nested Iteration - Example

1. Define outer query:
   \[ q \leftarrow \text{outer query} \]
2. Define inner query:
   \[ q' \leftarrow \text{inner query} \]
3. Execute outer query:
   \[ \text{result} \leftarrow \text{execute}(q) \]
4. Iterate over result:
   \[ \text{foreach} \ t \text{ in result} \]
   \[ q_t \leftarrow q'(t) \]
5. Execute inner query:
   \[ \text{result}’ \leftarrow \text{execute}(q_t) \]
6. Evaluate nested condition:
   \[ \text{evaluate_nested_condition(t, result’)} \]

SQL Example:

```
SELECT newspaper
FROM hasRead h
WHERE h.name = p.name
AND h.newspaper = 'Tribune'
```
Nested Iteration - Example

q ← outer query
q' ← inner query
result ← execute(q)
foreach tuple t in result
    q_t ← q'(t)
    result' ← execute(q_t)
evaluate_nested_condition(t, result')

EXISTS evaluates to true!
Output(Alice)

<table>
<thead>
<tr>
<th>person</th>
<th>hasRead</th>
<th>result'</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>gender</td>
<td>newspaper</td>
</tr>
<tr>
<td>Alice</td>
<td>female</td>
<td>Tribune</td>
</tr>
<tr>
<td>Bob</td>
<td>male</td>
<td>Courier</td>
</tr>
<tr>
<td>Joe</td>
<td>male</td>
<td>Courier</td>
</tr>
</tbody>
</table>
Nested Iteration - Example

q ← outer query
q' ← inner query
result ← execute(q)

foreach tuple t in result
    q_t ← q'(t)
    result' ← execute(q_t)
    evaluate_nested_condition(t,result')

Empty result set → EXISTS evaluates to false

<table>
<thead>
<tr>
<th>person</th>
<th>hasRead</th>
<th>result'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>name</td>
<td>newspaper</td>
</tr>
<tr>
<td>Alice</td>
<td>female</td>
<td>Tribune</td>
</tr>
<tr>
<td>Bob</td>
<td>male</td>
<td>Courier</td>
</tr>
<tr>
<td>Joe</td>
<td>male</td>
<td>Courier</td>
</tr>
</tbody>
</table>
Nested Iteration - Example

\[
q \leftarrow \text{outer query} \\
q' \leftarrow \text{inner query} \\
\text{result} \leftarrow \text{execute}(q) \\
\text{foreach tuple } t \text{ in result} \\
\quad q_t \leftarrow q'(t) \\
\quad \text{result'} \leftarrow \text{execute} (q_t) \\
\text{evaluate_nested_condition} (t, \text{result'})
\]

Empty result set $\rightarrow$ EXISTS evaluates to false
Nested Iteration - Discussion

- Repeated evaluation of nested subquery
  - If correlated
  - Improve:
    - Plan once and substitute parameters
    - EXISTS: stop processing after first result
    - IN/ANY: stop after first match

- No optimization across nesting boundaries
Unnesting and Decorrelation

- Apply equivalences to transform nested subqueries into joins

- **Unnesting:**
  - Turn a nested subquery into a join

- **Decorrelation:**
  - Turn correlations into join expressions
Equivalences

• Classify types of nesting
• Equivalence rules will have preconditions
• Can be applied heuristically before plan enumeration or using a transformative approach
N-type Nesting

• Properties
  – Expression ANY comparison (or IN)
  – No Correlations
  – Nested query does not use aggregation

• Example

```
SELECT name
FROM orders o
WHERE o.cust IN (SELECT cId
                     FROM customer
                     WHERE region = 'USA')
```
A-type Nesting

• Properties
  – Expression is ANY comparison (or scalar)
  – No Correlations
  – Nested query uses aggregation
  – No Group By

• Example

```sql
SELECT name
FROM orders o
WHERE o.amount = (SELECT max(amount)
                 FROM orders i)
```
J-type Nesting

• Properties
  – Expression is ANY comparison (IN)
  – Nested query uses equality comparison with correlated attribute
  – No aggregation in nested query

• Example

```sql
SELECT name
FROM orders o
WHERE o.amount IN (SELECT amount
                     FROM orders i
                     WHERE i.cust = o.cust
                     AND i.shop = 'New York')
```
JA-type Nesting

• Properties
  – Expression equality comparison
  – Nested query uses equality comparison with correlated attribute
  – Nested query uses aggregation and no GROUP BY

• Example

```sql
SELECT name
FROM orders o
WHERE o.amount = (SELECT max(amount)
                  FROM orders i
                  WHERE i.cust = o.cust)
```
Unnesting A-type

- Move nested query to FROM clause
- Turn nested condition (op ANY, IN) into op with result attribute of nested query
Unnesting N/J-type

- Move nested query to FROM clause
- Add DISTINCT to SELECT clause of nested query
- Turn equality comparison with correlated attributes into join conditions
- Turn nested condition (op ANY, IN) into op with result attribute of nested query
Example

1. To FROM clause

2. Add DISTINCT

3. Correlation to join

4. Nesting condition to join

```
SELECT name
FROM orders o
WHERE o.amount IN (SELECT amount
    FROM orders i
    WHERE i.cust = o.cust
    AND i.shop = 'New York')
```

```
SELECT name
FROM orders o,
    (SELECT amount
    FROM orders i
    WHERE i.cust = o.cust
    AND i.shop = 'New York') AS sub
```
Example

1. To FROM clause
2. Add DISTINCT
3. Correlation to join
4. Nesting condition to join

```
SELECT name
FROM orders o
WHERE o.amount IN (SELECT amount
    FROM orders i
    WHERE i.cust = o.cust
    AND i.shop = 'New York')
```

```
SELECT name
FROM orders o,
    (SELECT DISTINCT amount
    FROM orders i
    WHERE i.cust = o.cust
    AND i.shop = 'New York') AS sub
```
Example

1. To FROM clause
2. Add DISTINCT
3. Correlation to join
4. Nesting condition to join

```
SELECT name
FROM orders o
WHERE o.amount IN (SELECT amount
                      FROM orders i
                      WHERE i.cust = o.cust
                      AND i.shop = 'New York')
```

```
SELECT name
FROM orders o,
     (SELECT DISTINCT amount, cust
      FROM orders i
      WHERE i.shop = 'New York') AS sub
WHERE sub.cust = o.cust
```
Example

1. To FROM clause
2. Add DISTINCT
3. Correlation to join
4. Nesting condition to join

```
SELECT name
FROM orders o
WHERE o.amount IN (SELECT amount
                     FROM orders i
                     WHERE i.cust = o.cust
                     AND i.shop = 'New York')
```

```
SELECT name
FROM orders o,
     (SELECT DISTINCT amount, cust
      FROM orders i
      WHERE i.shop = 'New York') AS sub
WHERE sub.cust = o.cust
    AND o.amount = sub.amount
```
Unnesting JA-type

• Move nested query to FROM clause
• Turn equality comparison with correlated attributes into
  – GROUP BY
  – Join conditions
• Turn nested condition (op ANY, IN) into op with result attribute of nested query
Example

1. To FROM clause

2. Introduce GROUP BY and join conditions

3. Nesting condition to join

```sql
SELECT name
FROM orders o
WHERE o.amount = (SELECT max(amount)
    FROM orders i
    WHERE i.cust = o.cust)
```

```sql
SELECT name
FROM orders o,
    (SELECT max(amount)
    FROM orders i
    WHERE i.cust = o.cust) sub
```
Example

1. To FROM clause

```
SELECT name
FROM orders o
WHERE o.amount = (SELECT max(amount)
    FROM orders i
    WHERE i.cust = o.cust)
```

2. Introduce GROUP BY and join conditions

```
SELECT name
FROM orders o,
    (SELECT max(amount) AS ma, i.cust
    FROM orders i
    GROUP BY i.cust) sub
WHERE i.cust = sub.cust
```

3. Nesting condition to join
Example

1. To FROM clause

2. Introduce GROUP BY and join conditions

3. Nesting condition to join

```
SELECT name
FROM orders o
WHERE o.amount = (SELECT max(amount)
                  FROM orders i
                  WHERE i.cust = o.cust)
```

```
SELECT name
FROM orders o,
     (SELECT max(amount) AS ma, i.cust
      FROM orders i
      GROUP BY i.cust) sub
WHERE sub.cust = o.cust
    AND o.amount = sub.ma
```
Unnesting Benefits Example

- $N(\text{orders}) = 1,000,000$
- $V(\text{cust,orders}) = 10,000$
- $S(\text{orders}) = 1/10 \text{ block}$

```
SELECT name
FROM orders o
WHERE o.amount = (SELECT max(amount)
    FROM orders i
    WHERE i.cust = o.cust)
```

```
SELECT name
FROM orders o,
    (SELECT max(amount) AS ma, i.cust
     FROM orders i
     GROUP BY i.cust) sub
WHERE sub.cust = o.cust
    AND o.amount = sub.ma
```
SELECT name
FROM orders o
WHERE o.amount = (SELECT max(amount)
                  FROM orders i
                  WHERE i.cust = o.cust)

• Inner query:
  – One scan B(orders) = 100,000 I/Os

• Outer query:
  – One scan B(orders) = 100,000 I/Os
  – 1,000,000 tuples

• Total cost: 1,000,001 x 100,000 =~ 10^{11} I/Os

• N(orders) = 1,000,000
• V(cust,orders) = 10,000
• S(orders) = 1/10 block
• M = 10,000
- \( N(\text{orders}) = 1,000,000 \)
- \( V(\text{cust,orders}) = 10,000 \)
- \( S(\text{orders}) = 1/10 \) block
- \( M = 10,000 \)

**Inner queries:**
- One scan \( B(\text{orders}) = 100,000 \) I/Os
  - 1,000,000 result tuples
- Aggregation: Sort (assume 1 pass) = \( 3 \times 100,000 = 300,000 \) I/Os
  - 10,000 result tuples -> + 1,000 pages to write to disk

**The join:** use merge – join during merge
- \( 3 \times (1,000 + 100,000) \) I/Os = 303,000 I/Os

**Total cost:** 604,000 I/Os

SELECT name
FROM orders o,
     (SELECT max(amount) AS ma, i.cust
      FROM orders i
      GROUP BY i.cust) sub
WHERE sub.cust = o.cust
    AND o.amount = sub.ma