Cost of Query

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

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

```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'
```

Example Query – Possible Plan

```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'
```

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)

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
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

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**

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. \( \sigma_{a=b} (R \times S) \equiv R \times_{a=b} S \)
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'
```

**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
```

```
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
```
Join Graph Shapes

Cycle queries

\[
\text{SELECT * FROM R,S,T WHERE R.a = S.a AND S.b = T.b AND T.c = R.c}
\]

Clique queries

\[
\text{SELECT * FROM R,S,T WHERE R.a = S.a AND S.b = T.b AND T.c = R.c}
\]

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

\[
\frac{(2n)!}{(n+1)!n!} \times (n+1)! = \frac{(2n)!}{n!}
\]

---

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>17,63,225,600</td>
</tr>
<tr>
<td>10</td>
<td>670,942,572,800</td>
</tr>
<tr>
<td>11</td>
<td>28,158,588,057,600</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 built 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

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

```java
int[] fib = new int[2];

fib[0] = 1;
fib[1] = 1;

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

return fib[n];
```

Complexity

- Number of calls
  - \( C(n) = C(n-1) + C(n-2) + 1 = Fib(n+2) \)
  - \( O(2^n) \)

Using dynamic programming

```java
int[] fib = new int[2];

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

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 $< i$

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 Join Enumeration

```java
optPlan ← Map({R},{plan})
find_join_dp(q(R_1, \ldots, R_n))
{
  for i=1 to n
    optPlan[\{R_i\}] ← access_paths(R_i)
  for i=2 to n
    foreach S ⊆ \{R_1, \ldots, R_n\} with |S|=i
      optPlan[S] ← \∅
      foreach O ⊂ S with O ≠ \∅
        optPlan[S] ← optPlan[S] ∪ possible_joins(optPlan[O], optPlan[S\O])
      prune_plans(optPlan[S])
    return optPlan[\{R_1, \ldots, R_n\}]
}
```

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

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>17,267,280</td>
<td>46,230</td>
</tr>
<tr>
<td>9</td>
<td>17,643,230</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>21,150,580,057,600</td>
<td>39,916,800</td>
</tr>
</tbody>
</table>

DP with Left-deep trees only

- Reduced search-space
- Each join is with input relation
  - \( \rightarrow \) can use index joins
  - \( \rightarrow \) 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
  - \( \rightarrow \) 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
- \( \rightarrow \) 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 \(R,S,T\)

\[
\begin{align*}
\text{Left-deep best plans: 2-way} \\
(R,S) & \hspace{1cm} (R,T) & \hspace{1cm} (S,T) \\
\end{align*}
\]

Example Interesting Orders

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

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

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

\[
\text{find\_join\_dp}(q(R_1, \ldots, R_n))
\]

\[
\{
\begin{align*}
& \text{for } i=1 \text{ to } n \\
& \quad \text{plans} \leftarrow \text{plans} \cup \text{access\_paths}(R_i) \\
& \text{for } i=n \text{ to } 2 \\
& \quad \text{cheapest} = \min_{j,k} \left( \text{cost}(P_j \bowtie P_k) \right) \\
& \quad \text{plans} \leftarrow \text{plans} \setminus \{P_j, P_k\} \cup \{P_j \bowtie P_k\} \\
& \text{return } \text{plans} \text{ // single plan left}
\end{align*}
\}
\]

Greedy Join Enumeration

- Time: \(O(n^3)\)
  - Loop iterations: \(O(n)\)
  - In each iteration 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

```c
improve(q(R₁,…,Rₙ))
{
    best = random_plan(q)
    curplan = best
    do
        prevplan = curplan
        curplan = apply_random_trans(prevplan)
        while (cost(curplan) < cost(prevplan))
            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
  - $\rightarrow$ Simulated Annealing

Simulated Annealing

```c
SA(q(R₁,…,Rₙ))
{
    best = random_plan(q)
    curplan = best
    t = t_init // "temperature"
    while (t > 0)
        newplan = apply_random_trans(curplan)
        if cost(newplan) < cost(curplan)
            curplan = newplan
        else if random() < e^{-cost(newplan)-cost(curplan)}/t
            curplan = newplan
        if cost(curplan) < cost(best)
            best = curplan
        reduce(t)
    return best
}
```

Simulated Annealing

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 Enumeration

- **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?
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

Postgres Example

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

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 execution of similar queries

- **Example**
  - **Webshop**
  - Typical operation: Retrieve product with all user comments for that product
  - Same query modulo product id

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

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

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

PREPARE statement

- In SQL
  - `PREPARE name (parameters) AS query`
  - `EXECUTE name (parameters)`

Nested Subqueries

```sql
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 = execute(q)
foreach tuple t in result
q 't = q'(t) // parameterize q' with values from t
result' = execute(q 't)
evaluate_nested_condition (t,result')

Nested Iteration - Uncorrelated

q = outer query
q' = inner query
result = execute(q)
result' = execute (q)
foreach tuple t in result
q t = q'(t) // parameterize q' with values from t
result' = execute (q t)
evaluate_nested_condition (t,result')

Nested Iteration - Example

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

person
Alice female
Bob male
Joe male

hasRead
name   newspaper
Alice   Tribune
Alice   Courier
Joe     Courier

Nested Iteration - Example

SELECT newspaper
FROM hasRead
WHERE h.name = p.name
AND h.newspaper = 'Tribune'

person
Alice female
Bob male
Joe male

hasRead
name   newspaper
Alice   Tribune
Alice   Courier
Joe     Courier

Nested Iteration - Example

SELECT newspaper
FROM hasRead
WHERE h.name = 'Alice'
AND h.newspaper = 'Tribune'

person
Alice female
Bob male
Joe male

hasRead
name   newspaper
Alice   Tribune
Alice   Courier
Joe     Courier

Nested Iteration - Example

SELECT newspaper
FROM hasRead
WHERE h.name = 'Joe'
AND h.newspaper = 'Tribune'

person
Alice female
Bob male
Joe male

hasRead
name   newspaper
Alice   Tribune
Alice   Courier
Joe     Courier
**Nested Iteration - Example**

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

**Nested Iteration - Example**

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

**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
  ```sql
  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 DISTINCT amount,cust
FROM orders i
WHERE i.shop = 'New York') AS sub
WHERE sub.cust = o.cust
AND o.amount = sub.amount

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

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

Example 2.
To FROM clause
2. Introduce GROUP BY and join conditions
3. Nesting condition to join

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

Unnesting Benefits Example

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

Inner queries:
- One scan \(B(orders) = 100,000\) I/Os
- One scan \(B(orders) = 100,000\) I/Os
- One scan \(B(orders) = 1,000,000\) I/Os

Total cost: \(1,000,001 \times 100,000 \approx 10^{11}\) I/Os

Inner queries:
- One scan \(B(orders) = 100,000\) I/Os
- Aggregation: Sort (assume 1 pass) = \(3 \times 100,000 = 300,000\) I/Os
- 10,000 result tuples \(\Rightarrow + 1,000\) pages to write to disk
- The join: use merge – join during merge
- \(3 \times (1,000 + 100,000) = 303,000\) I/Os
- Total cost: 604,000 I/Os