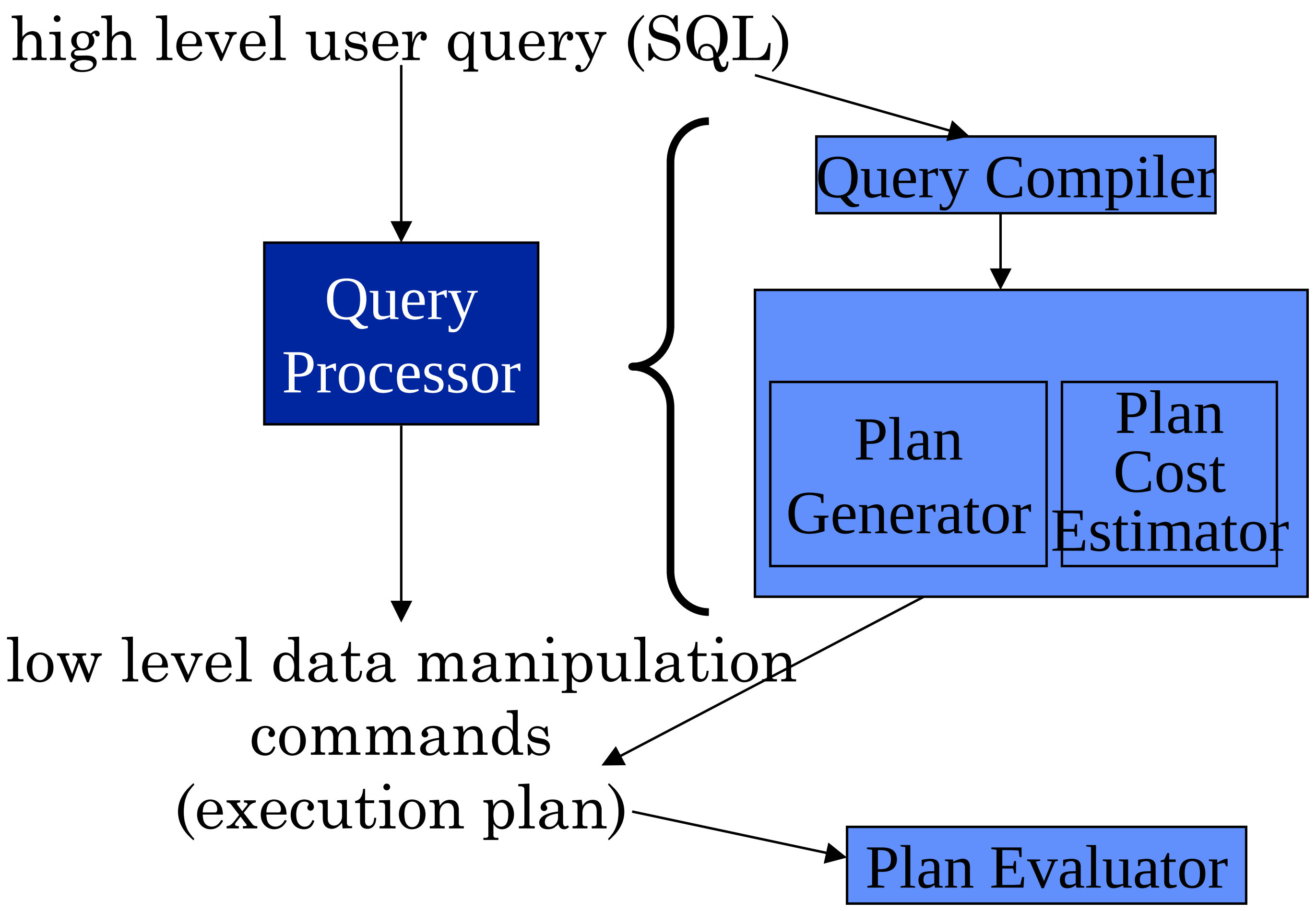
Query Processing


high level user query (SQL)
Query Processor
Query Compiler
Plan Generator
Plan Cost Estimator
low level data manipulation commands (execution plan)
Plan Evaluator

Query Processing Components

- Query language that is used
  - SQL: “intergalactic dataspeak”
- Query execution methodology
  - The steps that one goes through in executing high-level (declarative) user queries.
- Query optimization
  - How do we determine a good execution plan?

What are we trying to do?

- Consider query
  - “For each project whose budget is greater than $250000 and which employs more than two employees, list the names and titles of employees.”
- In SQL

```
SELECT  Ename, Title
FROM    Emp, Project, Works
WHERE   Budget > 250000
AND     Emp.Eno=Works.Eno
AND     Project.Pno=Works.Pno
AND     Project.Pno IN
        (SELECT  w.Pno
         FROM    Works w
         GROUP BY w.Pno
         HAVING  SUM(*) > 2)
```

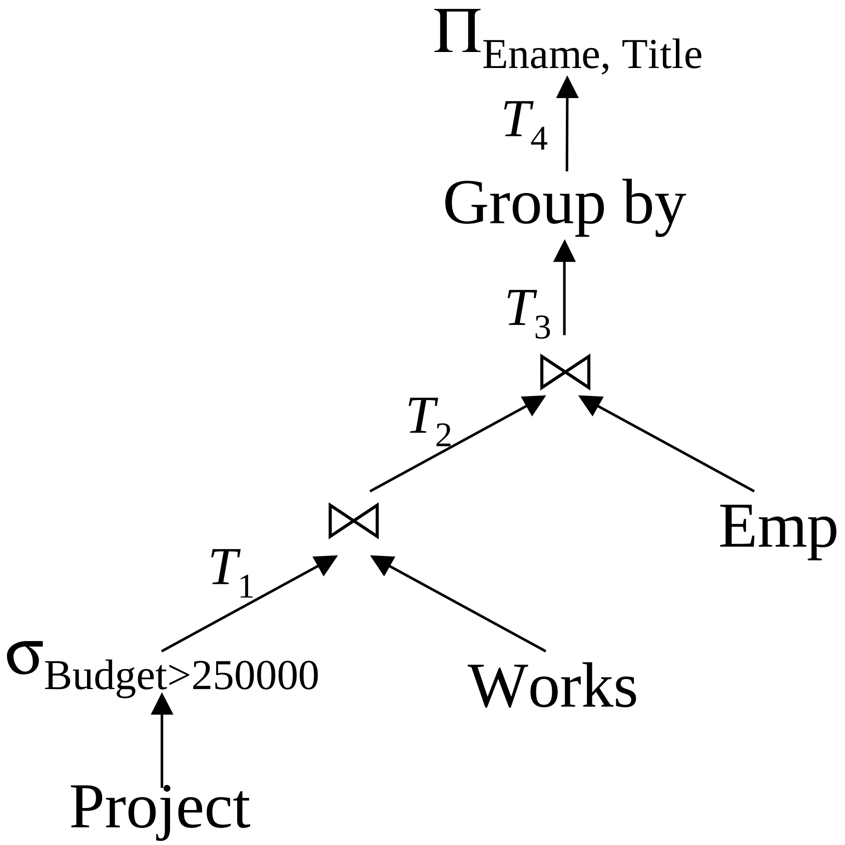
- How to execute this query?

A Possible Execution Plan

1. $T_1 \leftarrow$ Scan `Project` table and select all tuples with `Budget` value > 250000
2. $T_2 \leftarrow$ Join T_1 with the `Works` relation
3. $T_3 \leftarrow$ Join T_2 with the `Emp` relation
4. $T_4 \leftarrow$ Group tuples of T_3 over `Pno`
5. Scan tuples in each group of T_4 and for groups that have more than 2 tuples, Project over `Ename`, `Title`

Note: Overly simplified – we'll detail later.

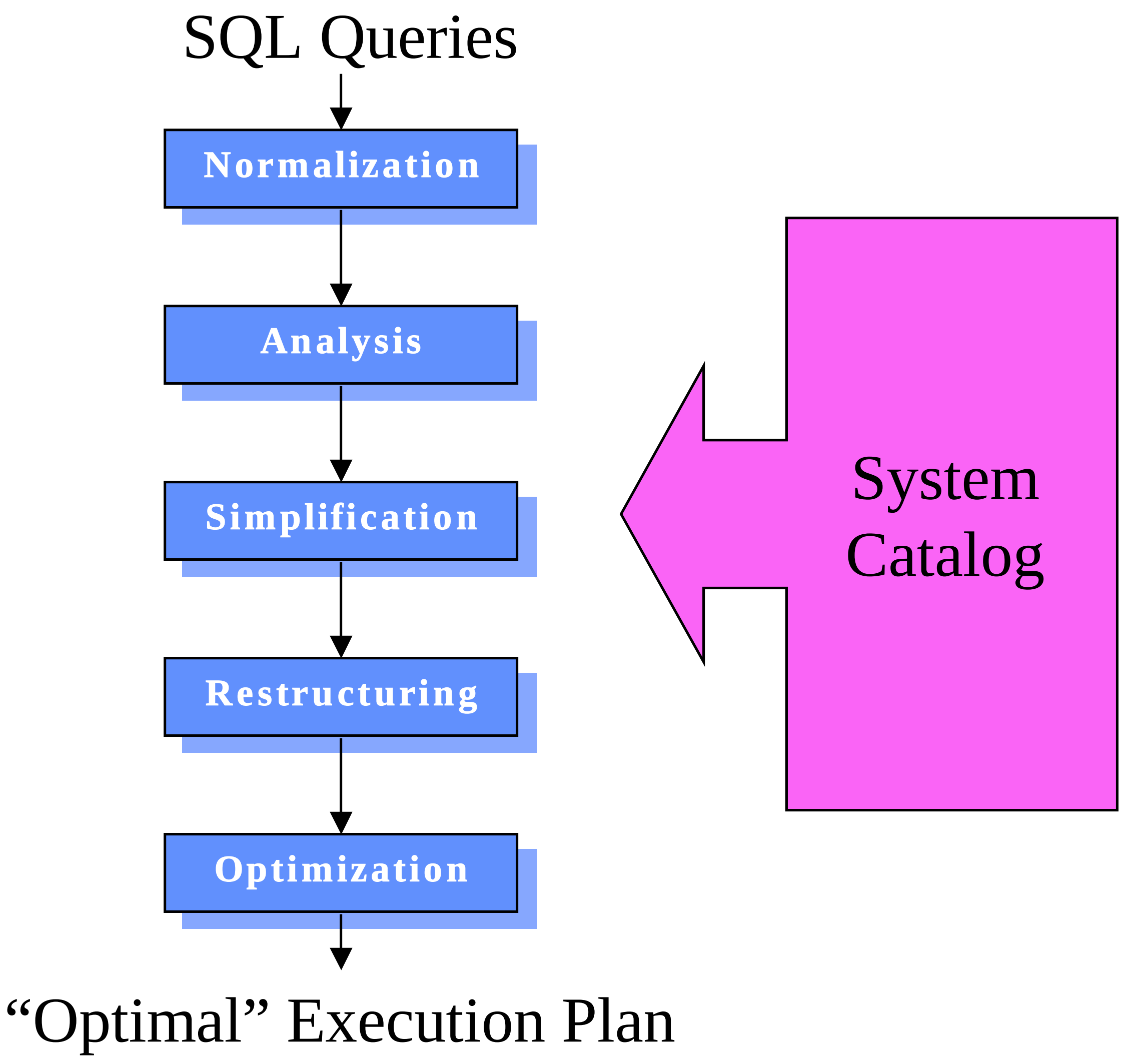
Pictorial Representation



1. How do we get this plan?
2. How do we execute each of the nodes?

$$\Pi_{\text{Ename, Title}}(\text{Group}_{\text{Pno,Eno}}(\text{Emp} \bowtie (\sigma_{\text{Budget}>250000}\text{Project} \bowtie \text{Works})))$$

Query Processing Methodology



Query Normalization

- Lexical and syntactic analysis
  - check validity (similar to compilers)
  - check for attributes and relations
  - type checking on the qualification
- Put into (query normal form
  - Conjunctive normal form

    $(p_{11} \vee p_{12} \vee \ldots \vee p_{1n}) \wedge \ldots \wedge (p_{m1} \vee p_{m2} \vee \ldots \vee p_{mn})$
  - Disjunctive normal form

    $(p_{11} \wedge p_{12} \wedge \ldots \wedge p_{1n}) \vee \ldots \vee (p_{m1} \wedge p_{m2} \wedge \ldots \wedge p_{mn})$
  - OR's mapped into union
  - AND's mapped into join or selection

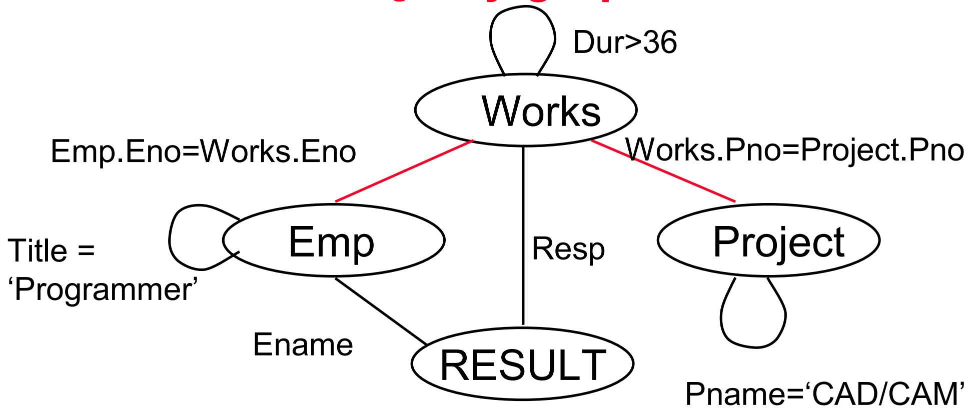
Analysis

- Refute incorrect queries
- Type incorrect
  - If any of its attribute or relation names are not defined in the global schema
  - If operations are applied to attributes of the wrong type
- Semantically incorrect
  - Components do not contribute in any way to the generation of the result
  - Only a subset of relational calculus queries can be tested for correctness
  - Those that do not contain disjunction and negation
  - To detect
    - connection graph (query graph)
    - join graph

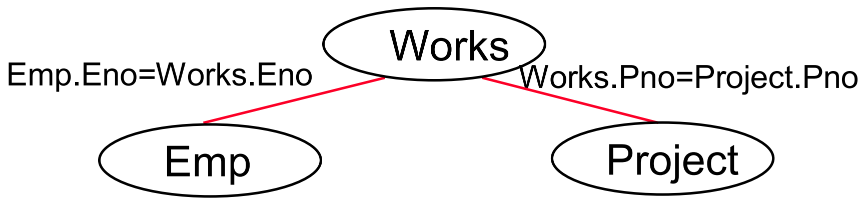
Analysis – Example

```
SELECT   Ename,Resp
FROM     Emp, Works, Project
WHERE    Emp.Eno = Works.Eno
AND      Works.Pno = Project.Pno
AND      Pname = ‘CAD/CAM’
AND      Dur > 36
AND      Title = ‘Programmer’
```

Query graph



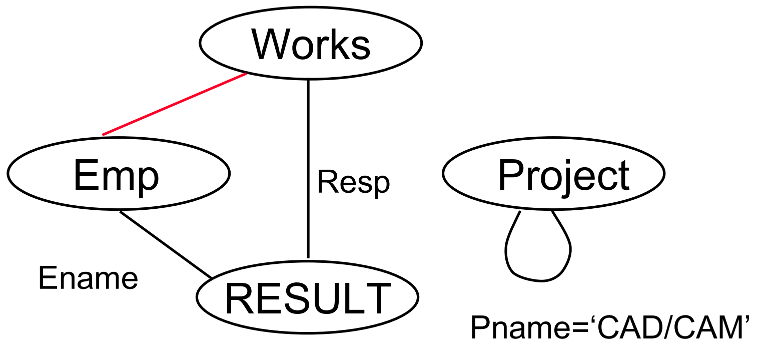
Join graph



Analysis

If the query graph is not connected, the query may be wrong.

```
SELECT Ename,Resp
FROM   Emp, Works, Project
WHERE  Emp.Eno = Works.Eno
AND    Pname = ‘CAD/CAM’
AND    Dur > 36
AND    Title = ‘Programmer’
```



Simplification

- Why simplify?
  - The simpler the query, the easier (and more efficient) it is to execute it
- How? Use transformation rules
  - elimination of redundancy
    - idempotency rules

      $p_1 \land \neg(p_1) \Leftrightarrow \text{false}$

      $p_1 \land (p_1 \lor p_2) \Leftrightarrow p_1$

      $p_1 \lor \text{false} \Leftrightarrow p_1$

      …
  - application of transitivity
  - use of integrity rules

Simplification – Example

```
SELECT      Title
FROM        Emp
WHERE       Ename = ‘J. Doe’
OR          (NOT(Title = ‘Programmer’)
AND         (Title = ‘Programmer’
OR          Title = ‘Elect. Eng.’)
AND         NOT(Title = ‘Elect. Eng.’))
```

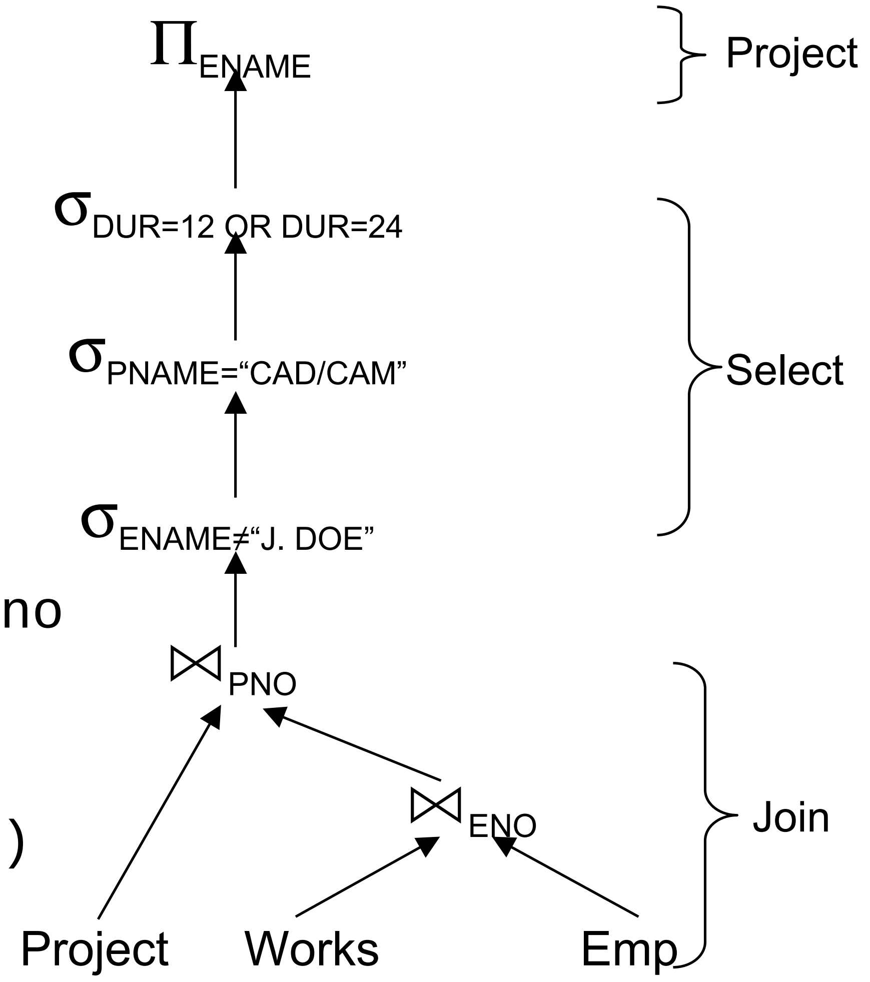
$\Downarrow$

```
SELECT      Title
FROM        Emp
WHERE       Ename = ‘J. Doe’
```

Restructuring

- Convert SQL to relational algebra
- Make use of query trees
- Example

```
SELECT  Ename
FROM    Emp, Works, Project
WHERE   Emp.Eno = Works.Eno
AND     Works.Pno = Project.Pno
AND     Ename <> 'J. Doe'
AND     Pname = 'CAD/CAM'
AND     (Dur = 12 OR Dur = 24)
```



Π_{ENAME}
Project
$\sigma_{DUR=12 \text{ OR } DUR=24}$
$\sigma_{PNAME=\text{“CAD/CAM”}}$
Select
$\sigma_{ENAME \neq \text{“J. DOE”}}$
$\bowtie_{PNO}$
$\bowtie_{ENO}$
Join
Project
Works
Emp

How to implement operators

- Selection (assume *R* has *n* pages)
  - Scan without an index – O(*n)*
  - Scan with index
    - B^+ index – O(log*n*)
    - Hash index – O(1)
- Projection
  - Without duplicate elimination – O(*n*)
  - With duplicate elimination
    - Sorting-based – O(*n*log*n*)
    - Hash-based – O(*n*+*t*) where *t* is the result of hashing phase

How to implement operators (cont'd)

- Join
  - Nested loop join: $R \bowtie S$

```
foreach tuple r∈R do
  foreach tuple s∈S do
    if r==s then add <r,s> to result
```

  - O($n*m$)
  - Improvements possible by
    - page-oriented nested loop join
    - block-oriented nested loop join

How to implement operators (cont'd)

- Join
  - Index nested loop join: $R \bowtie S$

```
foreach tuple r∈R do
  use index on join attr. to find tuples of S
  foreach such tuple s∈S do
    add <r,s> to result
```

  - Sort-merge join
    - Sort *R* and *S* on the join attribute
    - Merge the sorted relations
  - Hash join
    - Hash *R* and *S* using a common hash function
    - Within each bucket, find tuples where $r=s$

Index Selection Guidelines

- Hash vs tree index
  - Hash index on inner is very good for Index Nested Loops.
    - Should be clustered if join column is not key for inner, and inner tuples need to be retrieved.
  - Clustered B+ tree on join column(s) good for Sort-Merge.

Example 1

```
SELECT e.Ename, w.Dur
FROM   Emp e, Works w
WHERE  w.Resp='Mgr'
AND    e.Eno=w.Eno
```

- Hash index on w.Resp supports ‘Mgr’ selection.
- Hash index on w.Eno allows us to get matching (inner) Emp tuples for each selected (outer) Works tuple.
- What if WHERE included: “AND e.Title=`Programmer’’”?
  - Could retrieve Emp tuples using index on e.Title, then join with Works tuples satisfying Resp selection.

Example 2

```
SELECT e.Ename, w.Resp
FROM   Emp e, Works w
WHERE  e.Age BETWEEN 45 AND 60
AND    e.Title='Programmer'
AND    e.Eno=w.Eno
```

- Clearly, Emp should be the outer relation.
  - Suggests that we build a hash index on w.Eno.
- What index should we build on Emp?
  - B+ tree on e.Age could be used, OR an index on e.Title could be used. Only one of these is needed, and which is better depends upon the selectivity of the conditions.
    - As a rule of thumb, equality selections more selective than range selections.
- As both examples indicate, our choice of indexes is guided by the plan(s) that we expect an optimizer to consider for a query. Have to understand optimizers!

Examples of Clustering

```
SELECT e.Title
FROM   Emp e
WHERE  e.Age > 40
```

- B+ tree index on e.Age can be used to get qualifying tuples.
  - How selective is the condition?
  - Is the index clustered?

Clustering and Joins

```
SELECT e.Ename, p.Pname
FROM   Emp e, Project p
WHERE  p.Budget='350000'
AND    e.City=p.City
```

- Clustering is especially important when accessing inner tuples in Index Nested Loop join.
  - Should make index on e.City clustered.
- Suppose that the WHERE clause is instead:

  ```
  WHERE  e.Title='Programmer' AND e.City=p.City
  ```

  - If many employees are Programmers, Sort-Merge join may be worth considering. A clustered index on p.City would help.
- Summary: Clustering is useful whenever many tuples are to be retrieved.

Selecting Alternatives

```
SELECT  Ename
FROM    Emp e,Works w
WHERE   e.Eno = w.Eno
AND     w.Dur > 37
```

Strategy 1

$$\Pi_{\text{ENAME}}(\sigma_{\text{DUR}>37 \wedge \text{EMP.ENO}=\text{ASG.ENO}}\,(\text{Emp} \times \text{Works}))$$

Strategy 2

$$\Pi_{\text{ENAME}}(\text{Emp} \bowtie_{\text{ENO}} (\sigma_{\text{DUR}>37}\,(\text{Works})))$$

- Strategy 2 is “better” because
  - It avoids Cartesian product
  - It selects a subset of Works before joining
- How to determine the “better” alternative?

Query Optimization Issues – Types of Optimizers

- “Exhaustive” search
  - cost-based
  - optimal
  - combinatorial complexity in the number of relations
- Heuristics
  - not optimal
  - regroup common sub-expressions
  - perform selection, projection as early as possible
  - reorder operations to reduce intermediate relation size
  - optimize individual operations

Query Optimization Issues – Optimization Granularity

- Single query at a time
    - cannot use common intermediate results
- Multiple queries at a time
    - efficient if many similar queries
    - decision space is much larger

Query Optimization Issues – Optimization Timing

- Static
  - compilation ⇒ optimize prior to the execution
  - difficult to estimate the size of the intermediate results ⇒ error propagation
  - can amortize over many executions
- Dynamic
  - run time optimization
  - exact information on the intermediate relation sizes
  - have to reoptimize for multiple executions
- Hybrid
  - compile using a static algorithm
  - if the error in estimate sizes > threshold, reoptimize at run time

Query Optimization Issues – Statistics

- Relation
  - cardinality
  - size of a tuple
  - fraction of tuples participating in a join with another relation
  - …
- Attribute
  - cardinality of domain
  - actual number of distinct values
  - …
- Common assumptions
  - independence between different attribute values
  - uniform distribution of attribute values within their domain

Query Optimization Components

- Cost function (in terms of time)
  - I/O cost + CPU cost
  - These might have different weights
  - Can also maximize throughput
- Solution space
  - The set of equivalent algebra expressions (query trees).
- Search algorithm
  - How do we move inside the solution space?
  - Exhaustive search, heuristic algorithms (iterative improvement, simulated annealing, genetic,…)

Cost Calculation

- Cost function takes CPU and I/O processing into account
  - Instruction and I/O path lengths
- Estimate the cost of executing each node of the query tree
  - Is pipelining used or are temporary relations created?
- Estimate the size of the result of each node
  - Selectivity of operations – “reduction factor”
  - Error propagation is possible

Intermediate Relation Sizes

Selection

$$size(R) = card(R) * length(R)$$

$$card(\sigma_F(R)) = SF_\sigma(F) * card(R)$$

where

$$SF_\sigma(A = value) = \frac{1}{card(\prod_A(R))}$$

$$SF_\sigma(A > value) = \frac{max(A) - value}{max(A) - min(A)}$$

$$SF_\sigma(A < value) = \frac{value - min(A)}{max(A) - min(A)}$$

$$SF_\sigma(p(A_i) \wedge p(A_j)) = SF_\sigma(p(A_i)) * SF_\sigma(p(A_j))$$

$$SF_\sigma(p(A_i) \vee p(A_j)) = SF_\sigma(p(A_i)) + SF_\sigma(p(A_j)) - (SF_\sigma(p(A_i)) * SF_\sigma(p(A_j)))$$

$$SF_\sigma(A \in value) = SF_\sigma(A = value) * card(\{values\})$$

Intermediate Relation Sizes

Projection

$card(\Pi_A(R)) = card(R)$

Cartesian Product

$card(R \times S) = card(R) * card(S)$

Union

upper bound: $card(R \cup S) = card(R) + card(S)$

lower bound: $card(R \cup S) = max\{card(R), card(S)\}$

Set Difference

upper bound: $card(R – S) = card(R)$

lower bound: 0

Intermediate Relation Size

Join

- Special case: A is a key of R and B is a foreign key of S;

$$card(R \bowtie_{A=B} S) = card(S)$$

- More general:

$$card(R \bowtie S) = SF_J * card(R) * card(S)$$

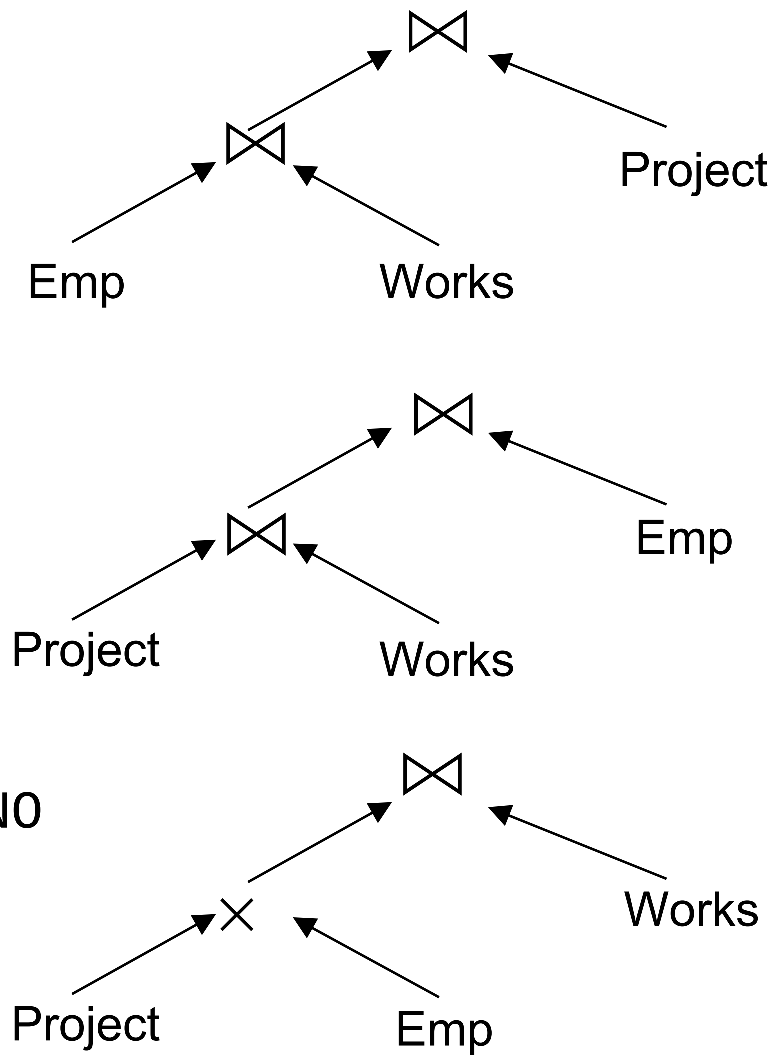
Search Space

- Characterized by “equivalent” query plans
  - Equivalence is defined in terms of equivalent query results
- Equivalent plans are generated by means of algebraic transformation rules
- The cost of each plan may be different
- Focus on joins

Search Space – Join Trees

- For *N* relations, there are O(*N*!) equivalent join trees that can be obtained by applying *commutativity* and *associativity* rules

```
SELECT    Ename,Resp
FROM      Emp, Works, Project
WHERE     Emp.Eno=Works.Eno
AND       Works.PNO=Project.PNO
```



Transformation Rules

- Commutativity of binary operations
  - $R \times S \Leftrightarrow S \times R$
  - $R \bowtie S \Leftrightarrow S \bowtie R$
  - $R \cup S \Leftrightarrow S \cup R$
- Associativity of binary operations
  - $(R \times S) \times T \Leftrightarrow R \times (S \times T)$
  - $(R \bowtie S) \bowtie T \Leftrightarrow R \bowtie (S \bowtie T)$
- Idempotence of unary operations
  - $\Pi_{A'}(\Pi_{A''}(R)) \Leftrightarrow \Pi_{A'}(R)$
  - $\sigma_{p_1(A_1)}(\sigma_{p_2(A_2)}(R)) = \sigma_{p_1(A_1) \wedge p_2(A_2)}(R)$

  where $R[A]$ and $A' \subseteq A$, $A'' \subseteq A$ and $A' \subseteq A''$

Transformation Rules

- Commuting selection with projection
- Commuting selection with binary operations
  - $\sigma_{p(A)}(R \times S) \Leftrightarrow (\sigma_{p(A)}(R)) \times S$
  - $\sigma_{p(A_i)}(R \bowtie_{(A_j,B_k)} S) \Leftrightarrow (\sigma_{p(A_i)}(R)) \bowtie_{(A_j,B_k)} S$
  - $\sigma_{p(A_i)}(R \cup T) \Leftrightarrow \sigma_{p(A_i)}(R) \cup \sigma_{p(A_i)}(T)$

  where A_i belongs to R and T
- Commuting projection with binary operations
  - $\Pi_C(R \times S) \Leftrightarrow \Pi_{A'}(R) \times \Pi_{B'}(S)$
  - $\Pi_C(R \bowtie_{(A_j,B_k)} S) \Leftrightarrow \Pi_{A'}(R) \bowtie_{(A_j,B_k)} \Pi_{B'}(S)$
  - $\Pi_C(R \cup S) \Leftrightarrow \Pi_C(R) \cup \Pi_C(S)$

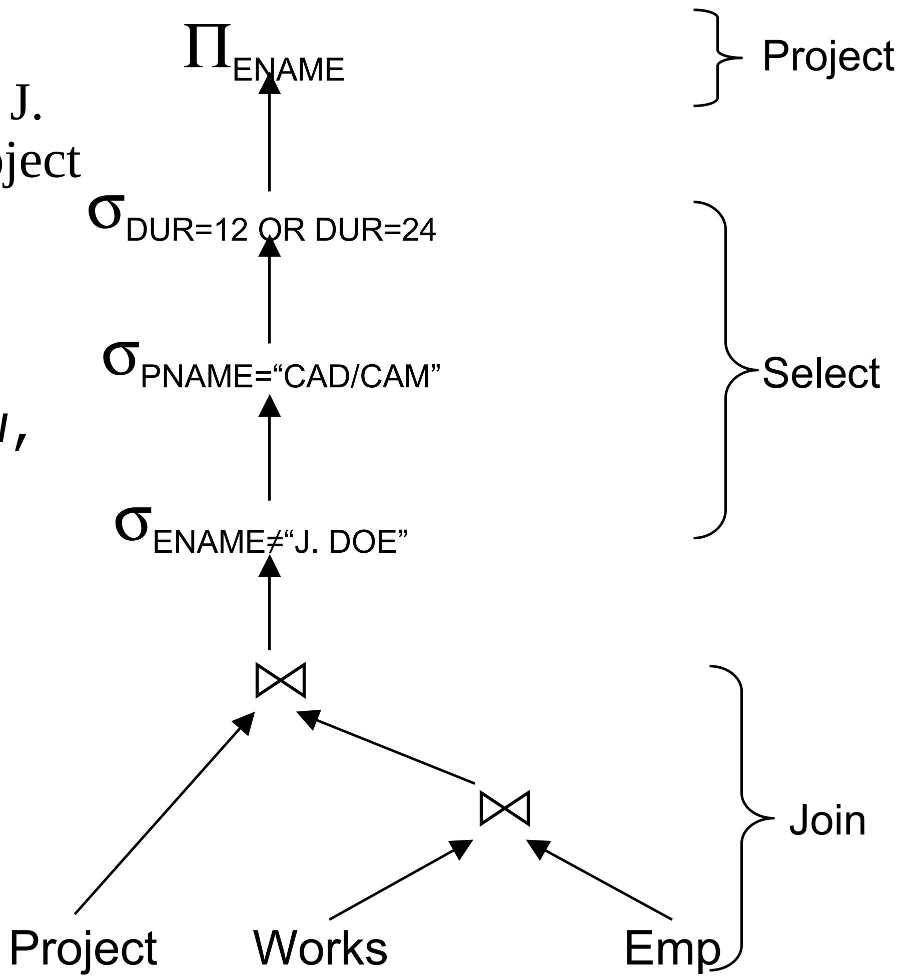
  where $R[A]$ and $S[B]$; $C = A' \cup B'$ where $A' \subseteq A$, $B' \subseteq B$, $A_j \subseteq A'$, $B_k \subseteq B'$

Example

Consider the query:

Find the names of employees other than J. Doe who worked on the CAD/CAM project for either one or two years.

```
SELECT    Ename
FROM      Project p, Works w,
          Emp e
WHERE     w.Eno=e.Eno
AND       w.Pno=p.Pno
AND       Ename<>`J. Doe'
AND       p.Pname=`CAD/CAM'
AND       (Dur=12 OR Dur=24)
```



Π_{ENAME}
$\sigma_{DUR=12 \text{ OR } DUR=24}$
$\sigma_{PNAME=\text{"CAD/CAM"}}$
$\sigma_{ENAME\neq\text{"J. DOE"}}$
$\bowtie$
$\bowtie$
Project
Works
Emp
Project
Select
Join

Equivalent Query

$$\begin{tikzcd}
 & \Pi_{\text{Ename}} & & \\
 & \sigma_{\text{Pname=`CAD/CAM'} \wedge (\text{Dur=12} \vee \text{Dur=24}) \wedge \text{Ename<>`J. DOE'}} \arrow[u] & & \\
 & \bowtie \arrow[u] & & \\
 & & \times \arrow[ul] & \\
\text{Works} \arrow[uur] & \text{Project} \arrow[ur] & & \text{Emp} \arrow[ul]
\end{tikzcd}$$

Another Equivalent Query

$$\begin{tikzcd}
 & & \Pi_{\text{Ename}} & & \\
 & & \bowtie \arrow[u] & & \\
 & & & \Pi_{\text{Pno,Ename}} \arrow[ul] & \\
 & & & \bowtie \arrow[u] & \\
\Pi_{\text{Pno}} \arrow[uuurr] & & \Pi_{\text{Pno,Eno}} \arrow[ur] & & \Pi_{\text{Eno,Ename}} \arrow[ul] \\
\sigma_{\text{Pname = `CAD/CAM'}} \arrow[u] & & \sigma_{\text{Dur =12} \wedge \text{Dur=24}} \arrow[u] & & \sigma_{\text{Ename <> `J. Doe'}} \arrow[u] \\
\text{Project} \arrow[u] & & \text{Works} \arrow[u] & & \text{Emp} \arrow[u]
\end{tikzcd}$$

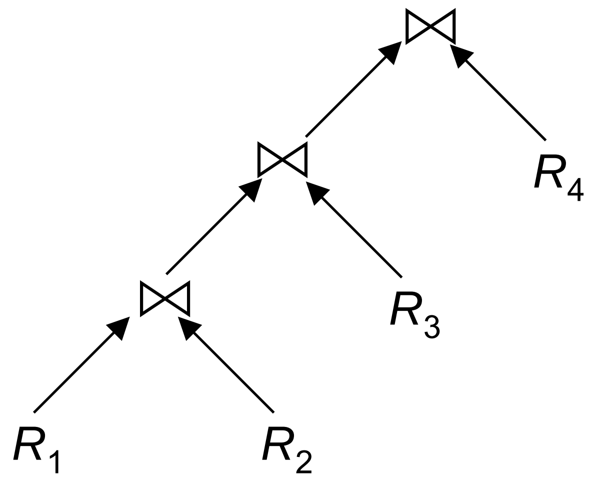
Search Strategy

- How to "move" in the search space.
- Deterministic
  - Start from base relations and build plans by adding one relation at each step
  - Dynamic programming: breadth-first
  - Greedy: depth-first
- Randomized
  - Search for optimalities around a particular starting point
  - Trade optimization time for execution time
  - Better when > 5-6 relations
  - Simulated annealing
  - Iterative improvement

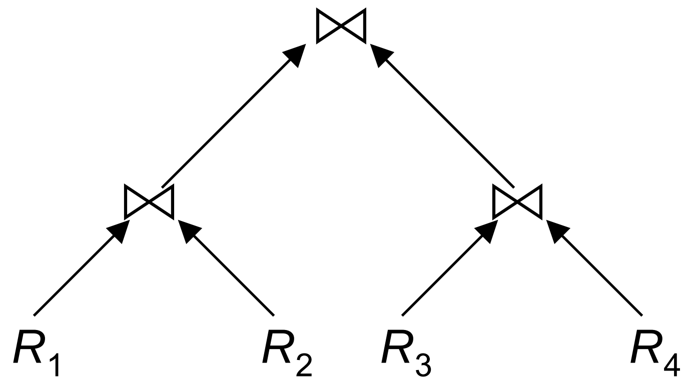
Search Algorithms

- Restrict the search space
  - Use heuristics
    - E.g., Perform unary operations before binary operations
  - Restrict the shape of the join tree
    - Consider only linear trees, ignore bushy ones

Linear Join Tree

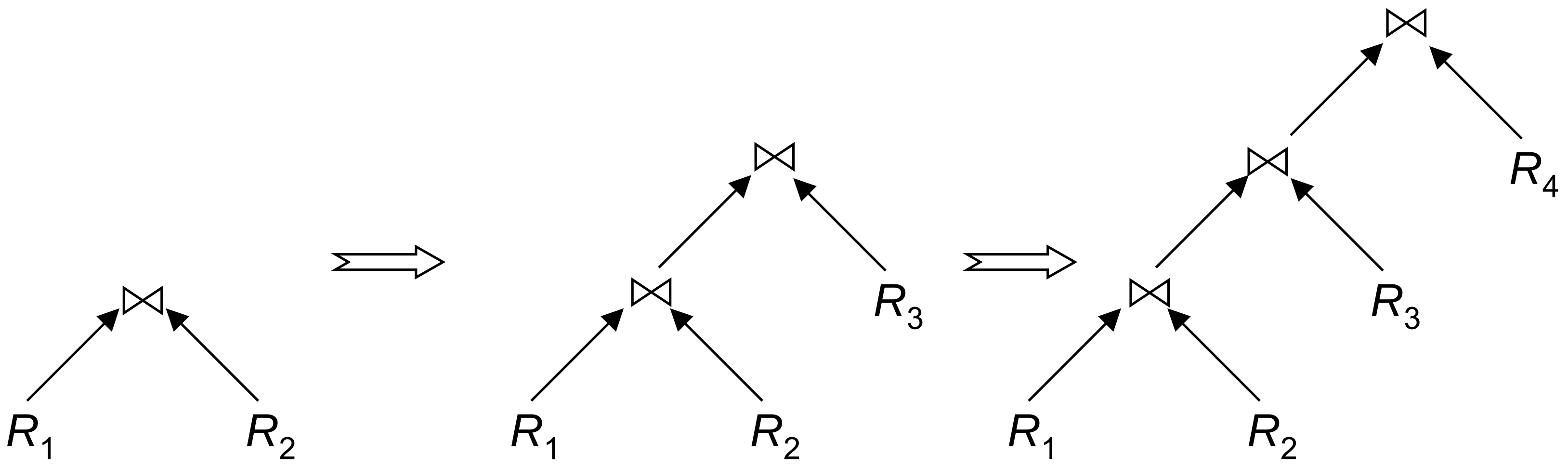


Bushy Join Tree

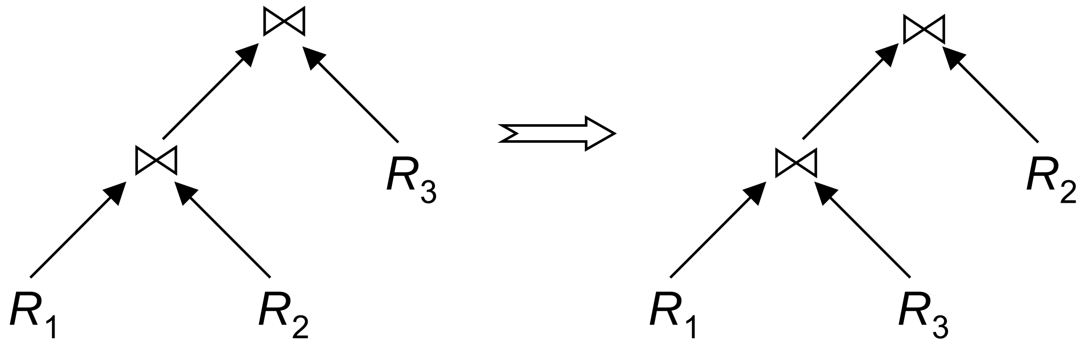


Search Strategies

- Deterministic



- Randomized



Summary

- Declarative SQL queries need to be converted into low level execution plans
- These plans need to be optimized to find the "best" plan
- Optimization involves
  - Search space: identifies the alternative plans and alternative execution algorithms for algebra operators
    - This is done by means of transformation rules
  - Cost function: calculates the cost of executing each plan
    - CPU and I/O costs
  - Search algorithm: controls which alternative plans are investigated