QUERY OPTIMIZATION

Chapter 19

Prof. Young-Kyoon Suh

Introduction

- Query optimization(질의 최적화)
  - Conducted by a ***query optimizer*** in a DBMS
  - Goal
    - Select the *best available strategy* for query evaluation
    - Arrive *within a reasonable amount of time* at the *most efficient and the cost-effective plan* using the available information about the schema and the content of relations involved
- Most RDBMSs use a *tree* as the internal representation of a query.

QUERY TREES AND HEURISTICS FOR QUERY OPTIMIZATION

Chapter 19.1

Heuristics-based Optimization Techniques

- Step 1: the scanner and parser of a query generates a data structure representing an *initial query tree presentation*
- Step 2: representation is *optimized* according to ***heuristic rules***.
  - Leads to an *optimized query representation*, corresponding to the query execution strategy
  - One of the main heuristic rules: to apply SELECT and PROJECT *before* JOIN or other binary operations.
    - Why? To reduce size of files to be joined.
- Step 3: a *query execution plan* is generated.
  - The plan executes groups of operations based on the access paths available on the files involved in the query

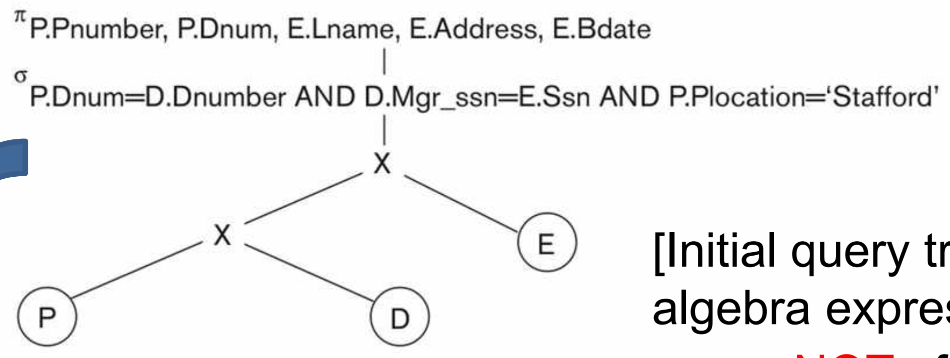
Query (Evaluation) Tree

- A tree data structure corresponding to an extended relational algebra expression
  - *Leaf nodes*: *input relations* of the query
  - *Internal nodes*: *relational algebra operations* (or, *query operators*)
- An execution of the query tree consists of:
  - Executing a *query operator node* whenever its operands are available
  - Replacing the node by *the resulting relation* produced by the operator
- The order of the execution:
  - *Starts at the leaf nodes*, or the input relation
  - *Ends at the root node*, or the final operation of the query

Example of a Query Tree

- Q1:

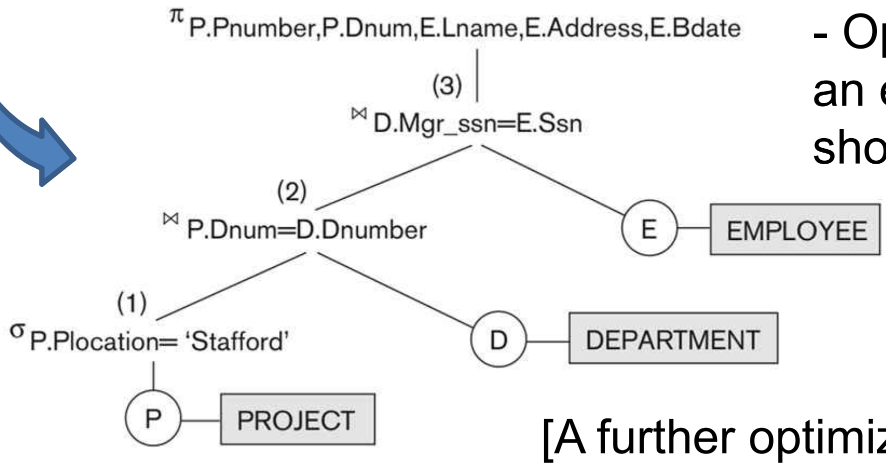
```
SELECT P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
FROM PROJECT P, DEPARTMENT D, EMPLOYEE E
WHERE P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND P.Plocation='Stafford'
```



[Initial query tree based on relational algebra expressions (RAEs)]

- NOT efficient; never executed
- Optimizer will transform into an equivalent final query tree shown below.

Applying the main heuristic rule: called ***early selection***



[A further optimized query tree based on RAE]

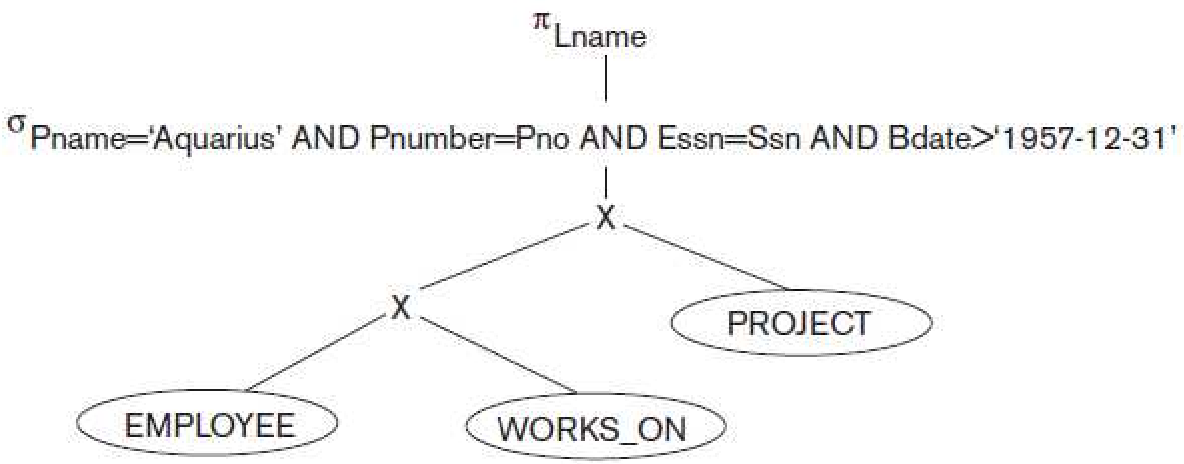
Query Transformation Example

- Many different query trees can be *semantically equivalent*;
  - They represent *the same query* and *produce the same results*.

```
Q:  SELECT  E.Lname
    FROM    EMPLOYEE E, WORKS_ON W, PROJECT P
    WHERE   P.Pname='Aquarius' AND P.Pnumber=W.Pno AND E.Essn=W.Ssn
            AND E.Bdate > '1957-12-31';
```

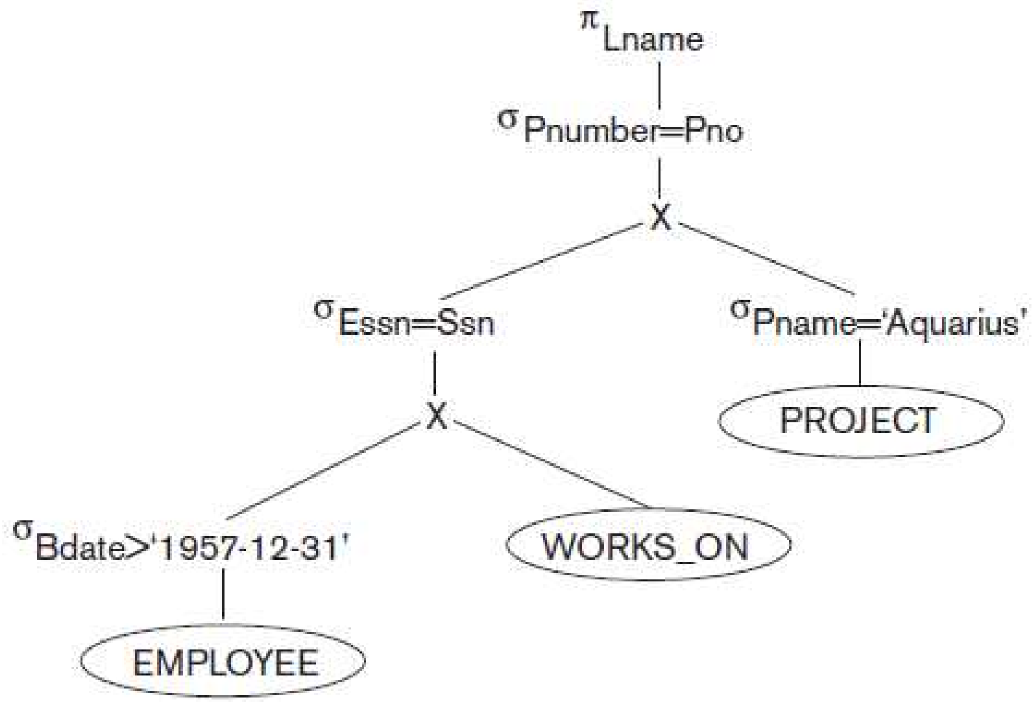
Converts to an initial query tree

(a)



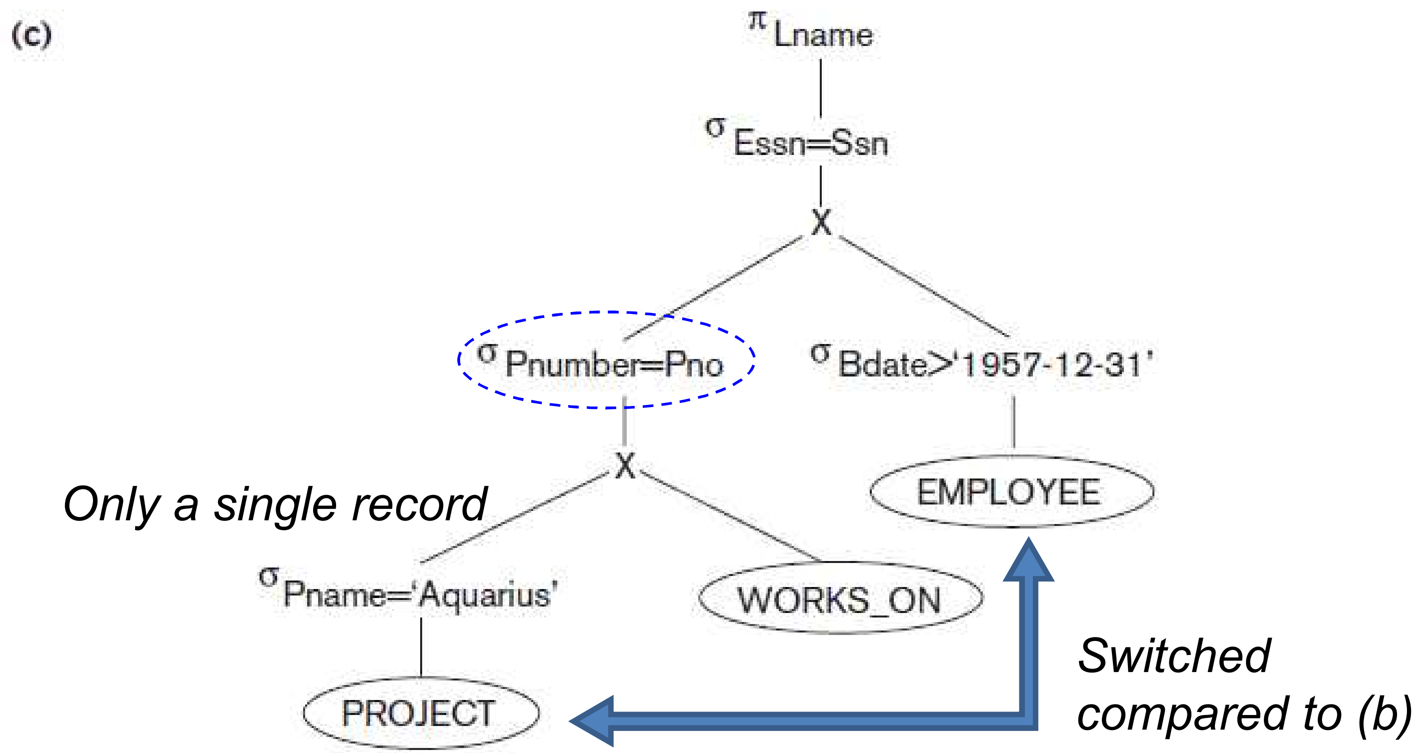
Query Transformation Example (Cont'd)

(b)



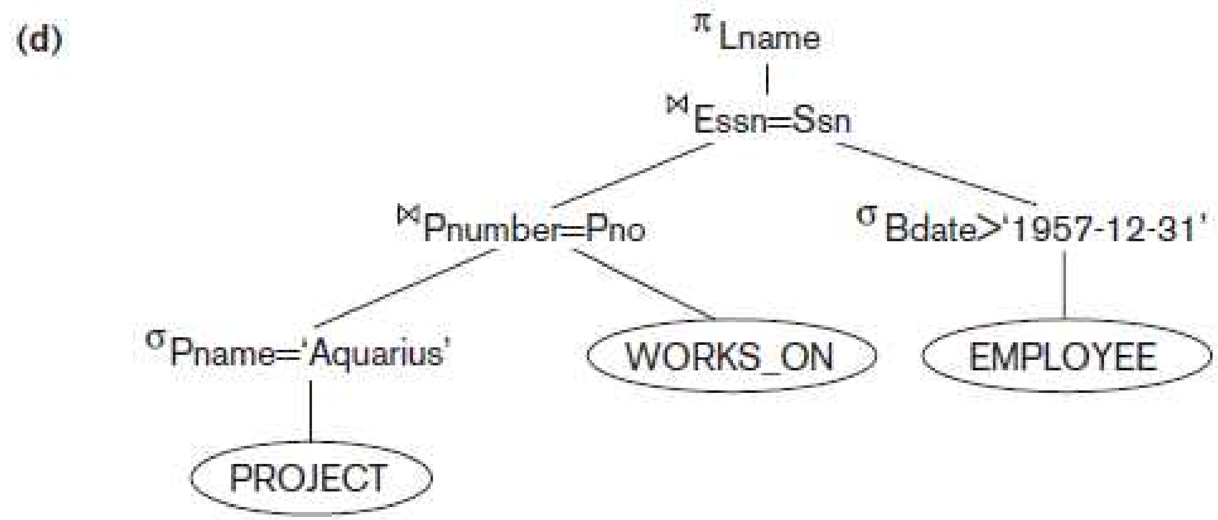
[Moving SELECT operations down the query tree]

Query Transformation Example (Cont'd)



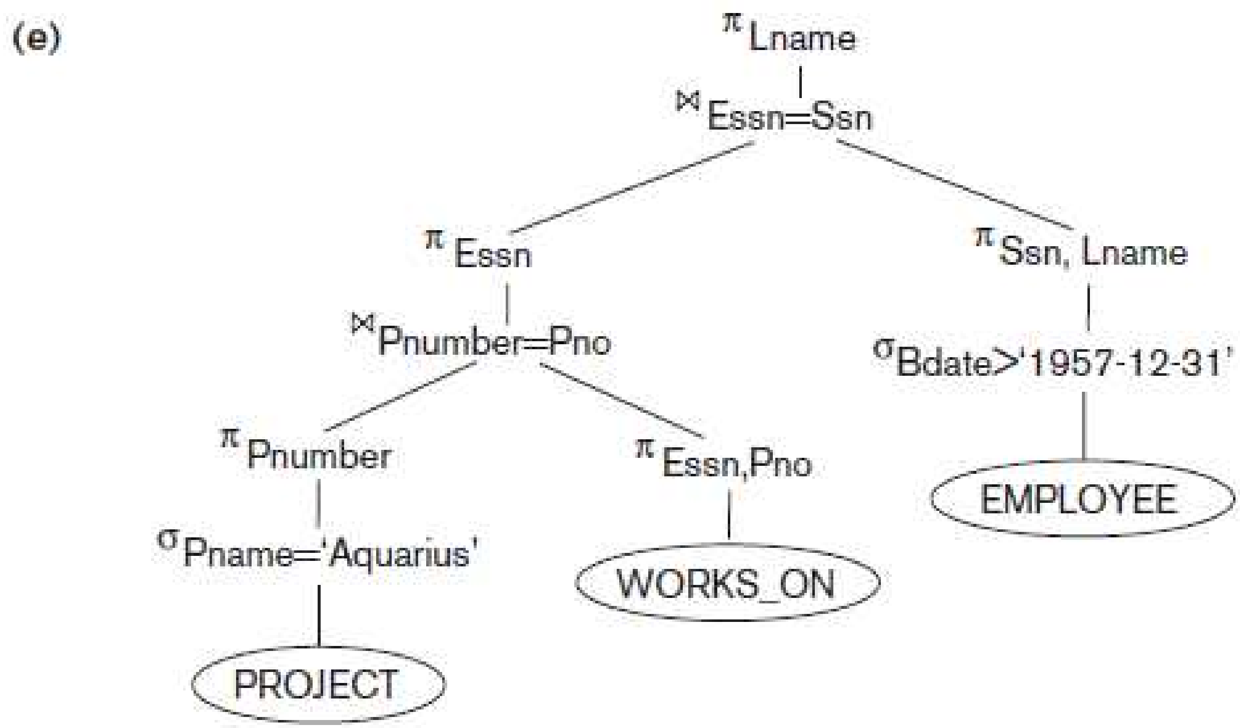
[Applying the *more restrictive* SELECT operation first]

Query Transformation Example (Cont'd)



[Replacing CARTESIAN PRODUCT and SELECT with JOIN operations]

Query Transformation Example (Cont'd)



[Moving PROJECT operations down the query tree]

Summary of Heuristics for Algebraic Optimization

- Apply first operations that *reduce the size of intermediate results*
  - Perform SELECT and PROJECT operations **as early as possible** to reduce the number of tuples and attributes
  - The SELECT and JOIN operations that are most restrictive should be ***executed before*** other similar operations.
    - So that fewer intermediate results can be joined if any

COST-BASED OPTIMIZATION

Chapter 19.3

Cost-based Query Optimization (QO)

- Applied to *compiled queries* rather than *interpreted queries*.
- Refers to the following approach:
  - An optimizer *enumerates* possible *execution plans* for a given SQL query.
  - It then *estimates and compares execution costs for* a query using different execution strategies and algorithms.
  - It then *chooses the execution plan with the lowest cost*.
- The scope of QO: a single query block
  - The optimizer consider various information: *various table and index access paths*, *join orders*, *join methods*, *group-by methods*, etc.
- Note: a query optimizer does *not* depend *solely* on heuristic rules or query transformations.

Cost Components for Query Execution

- *Disk I/O cost*
  - Access cost to secondary storage
- *Space (disk storage) cost*
  - Cost of storing on disk any intermediate results
- *CPU (computation) cost*
  - Cost of performing in-memory operations on records within buffer
- *Memory usage cost*
  - Cost pertaining to # buffers needed for query execution

(Communication cost

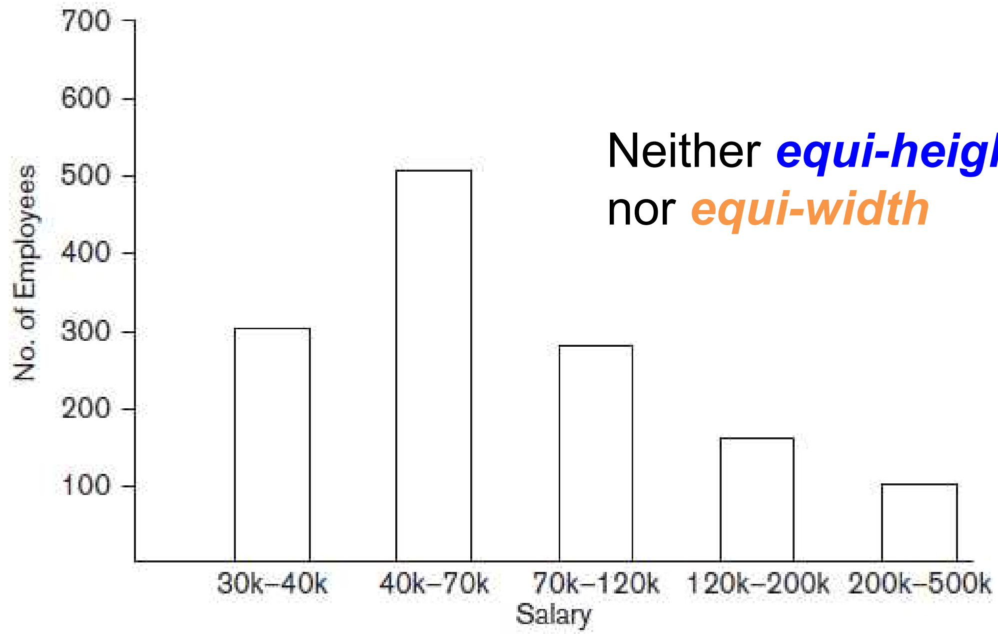
- query shipping and receiving results from a remote site)

Catalog Information Used in Cost Functions

- To estimate the costs of various plans, we leverage *cost functions*, and any information needed for the functions must be kept.
- The catalog information used by query optimizer
  1) File size
     - # records, the (average) record size, # blocks for a relation file, *bfr*
  2) File organizations
     - Primary organizations: unordered, ordered, hashed, tree-structured, etc.
       - Primary, clustering indexes with their indexing attributes
       - Secondary indexes with their indexing attributes , etc.
  3) Number of levels of each multi-level index
  4) Number of distinct values of an attribute
  5) Attribute selectivity: avg. # records satisfying equality condition

Histograms

- Tables or data structures that record information about *data distribution*
- RDBMS stores histograms for most important attributes



Neither ***equi-height*** (typically, better) nor ***equi-width***

[An Example of Histogram of salary in the relation `EMPLOYEE`]

COST FUNCTIONS FOR SELECT OPERATION

Chapter 19.4

Notations Used in Cost Formulas

C_{Si}: Cost for method Si in block accesses
r_X: Number of records (tuples) in a relation X
b_X: Number of blocks occupied by relation X (also referred to as b)
bfr_X: Blocking factor (i.e., number of records per block) in relation X
sl_A: Selectivity of an attribute A for a given condition
sA: Selection cardinality of the attribute being selected ($= sl_A * r$)
xA: Number of levels of the index for attribute A
$b_{I1}A$: Number of first-level blocks of the index on attribute A
NDV (A, X): Number of distinct values of attribute A in relation X

Cost Functions for SELECT

- S1: *Linear search* (brute force approach)
  - Search all file blocks to retrieve all records

    $C_{S1a}=b$
  - For equality condition on a key attribute, on average *one-half* the records are searched

    $C_{S1b}=\frac{b}{2}$
- S2: *Binary search*

  $C_{S2}=\log_2 b+[\frac{s}{bfr}]-1$
  - Reduces to $\log_2 b$ if equality condition is on a key attribute
    - s: selection cardinality

Cost Functions for SELECT (Cont'd)

- S3a: Using a *primary index* to retrieve a single record

$$C_{S3a} = x + 1$$

- S3b: Using a *hash key* to retrieve a single record

$$C_{S3b} = 1$$

- S4: Using an *ordering index* to retrieve multiple records

$$C_{S4} = x + \frac{b}{2}$$

- S5: Using a *clustering index* to retrieve multiple records

$$C_{S5} = x + [\frac{s}{bfr}]$$

- S6: Using a *secondary*(B+ tree) *index* to retrieve multiple records

$$C_{S6a} = x + 1 + s \text{ (worst case)}, C_{S6b} = x + \frac{b_{I1}}{2} + \frac{r}{2} \text{ (for range queries)}$$

Cost Functions for SELECT (Cont'd)

- ***Dynamic programming*** may be considered to find the optimal plan (with the least execution cost) without considering all possible execution plans.
  - Cost-based optimization approach
  - Sub-problems are solve only once.
  - Applicable when a problem may be broken down into subproblems that themselves have subproblems.
    - E.g. The cost of a root can be broken into the cost of each subtree of the tree, etc.

COST FUNCTIONS FOR THE JOIN OPERATION

Chapter 19.5

Cost Functions for JOIN

- Cost functions involve estimate of file size that results from the JOIN operation
  - Recall join may leave intermediate results during processing, leading to nontrivial I/O cost
- *Join selectivity* (js)
  - Ratio of the size of resulting file to size of CARTESIAN PRODUCT file
  - Simple formula for join selectivity on $R \bowtie_{R.A=S.B} S$
    - $js = \dfrac{1}{max(NDV(A,R), NDV(B,S))}$, when A or B is a key of either R or S, each.
- *Join cardinality* (jc)
  - The size of the resulting file after join: $jc = js$ x $|R|$ x $|S|$

Cost Functions for JOIN (Cont'd)

- J1: ***Nested-loop join***
  - For three memory buffer blocks:

$$C_{J1} = b_R + (b_R * b_S) + ((js * |R| * |S|)/bfr_{RS})$$

  - For n_B memory blocks:

$$C_{J1} = b_R + (\lceil b_R/(n_B - 2)\rceil * b_S) + ((js * |R| * |S|)/bfr_{RS})$$

- J2: ***Index-based nested-loop join***
  - For a secondary index with selection cardinality S_B for join attribute $S.B$

$$C_{J2a} = b_R + (|R| * (x_B + 1 + s_B)) + ((js * |R| * |S|)/bfr_{RS})$$

$((js * |R| * |S|)/bfr_{RS})$: the cost of writing results to disk

Cost Functions for JOIN (Cont'd)

- J3: ***Sort-merge join***
  - For files already sorted on the join attributes (without external sort)

$$C_{J3a} = b_R + b_S + ((js * |R| * |S|)/bfr_{RS})$$

    - Cost of sorting must be added if sorting needed like external sort

- J4: ***Partition-hash join***

$$C_{J4} = \boxed{3 * (b_R + b_S)} + ((js * |R| * |S|)/bfr_{RS})$$

R and S are read twice for partitioning via hashing.
Finally, partitioned R and S are written once.

Example Table for Calculating Execution Cost

Cost Estimate Items	Number
$r_{\mathrm{E(MPLOYEE)}}$	10,000 records
$b_{\mathrm{E(MPLOYEE)}}$	2,000 blocks
$r_{\mathrm{D(DEPARTMENT)}}$	125 records
$b_{\mathrm{D(DEPARTMENT)}}$	13 blocks
x_{Dnumber} (Number of levels of a primary index on the `Dnumber` attribute in `DEPARTMENT`)	1
x_{Dno} (Number of levels of a secondary index on the `Dno` attribute in `EMPLOYEE`)	2
s_{Dno} (Selectivity cardinality on `Dno`)	80 (records selected)
Join selectivity (js) on **EMPLOYEE** $\bowtie_{\mathrm{Dno=Dnumber}}$ **DEPARTMENT**	$\frac{1}{max(NDV(\mathrm{Dno,EMPLOYEE}),\ NDV(\mathrm{Dnumber,DEPARTMENT}))} = \frac{1}{125}$
Blocking factor of the resulting join (bfr_{ED})	4 records

`DEPARTMENT.Dnumber`: a key attribute of `DEPARTMENT`

Example of Join Optimization Based on Cost Functions

$$\texttt{EMPLOYEE (E)} \bowtie_{\texttt{DNO=Dnumber}} \texttt{DEPARTMENT (D)}$$

1) Nested-Loop Join (NLJ) with `EMPLOYEE` as outer loop:

- $C_{J1} = b_E + (b_E * b_D) + ((js * r_E * r_D) / bfr_{ED})$
  $= 2{,}000 + (2{,}000*13) + (((1/125) * 10{,}000 * 125) / 4) = 30{,}500$

2) Nested-Loop Join (NLJ) with `DEPARTMENT` as outer loop:

- $C_{J1} = b_D + (b_D * b_E) + ((js * r_E * r_D) / bfr_{ED})$
  $= 13 + (13*2{,}000) + (((1/125) * 10{,}000 * 125) / 4) = 28{,}513$

Example of Join Optimization Based on Cost Functions (Cont'd)

$$\texttt{EMPLOYEE(E)} \bowtie_{\texttt{DNO=Dnumber}} \texttt{DEPARTMENT(D)}$$

3) Indexed-based NLJ with `EMPLOYEE` as outer loop:

- Using the primary index on `DEPARTMENT` in inner loop
- $C_{J2c} = b_{\mathrm{E}} + (r_{\mathrm{E}} * (x_{\mathrm{Dnumber}} + 1)) + ((js * r_{\mathrm{E}} * r_{\mathrm{D}}) / bfr_{\mathrm{ED}})$
  $= 2{,}000 + (10{,}000*(1+1)) + (((1/125) * 10{,}000 * 125) / 4) = 24{,}500$

4) Indexed-based NLJ with `DEPARTMENT` as outer loop:

- Using the secondary index on `EMPLOYEE` in inner loop
- $C_{J2a} = b_{\mathrm{D}} + (r_{\mathrm{D}} * (x_{\mathrm{Dno}} + s_{\mathrm{Dno}})) + ((js * r_{\mathrm{E}} * r_{\mathrm{D}}) / bfr_{\mathrm{ED}})$
  $= 13 + (125*(2+80)) + (((1/125) * 10{,}000 * 125) / 4) = 24{,}500$

Example of Join Optimization Based on Cost Functions (Cont'd)

$$\text{EMPLOYEE(E)} \bowtie_{\text{DNO=Dnumber}} \text{DEPARTMENT(D)}$$

5) Partition-hash join

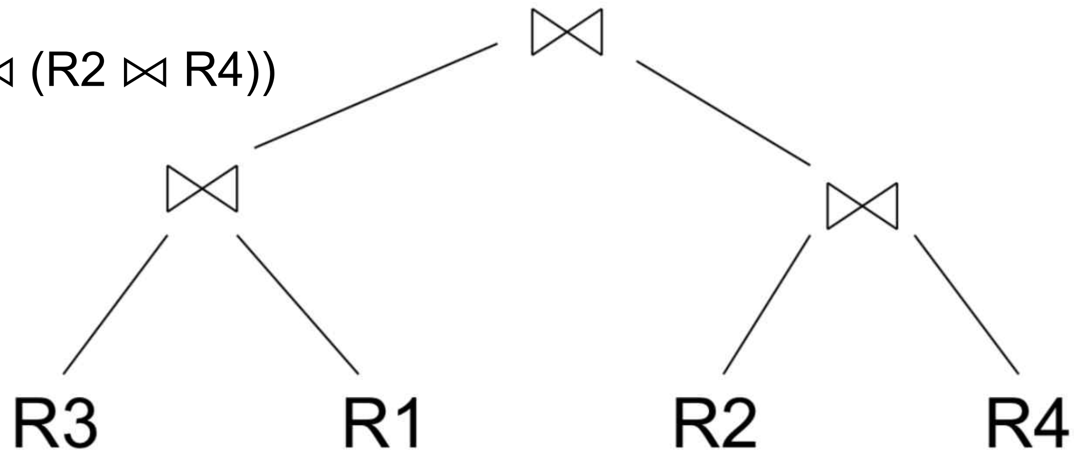
- $C_{J4} = 3 * (b_E + b_D)) + ((js * r_E * r_D) / bfr_{ED})$
  $= 3 * (13 + 2{,}000) + (((1/125) * 10{,}000 * 125) / 4) = 8{,}539$

Lowest cost!!!

Multi-way Join Queries & Join Tree

- Multi-way join queries: $R1 \bowtie R2 \bowtie \ldots \bowtie Rn$
  - TOO MANY join orders are possible on *N*-way joins.
- Join tree:

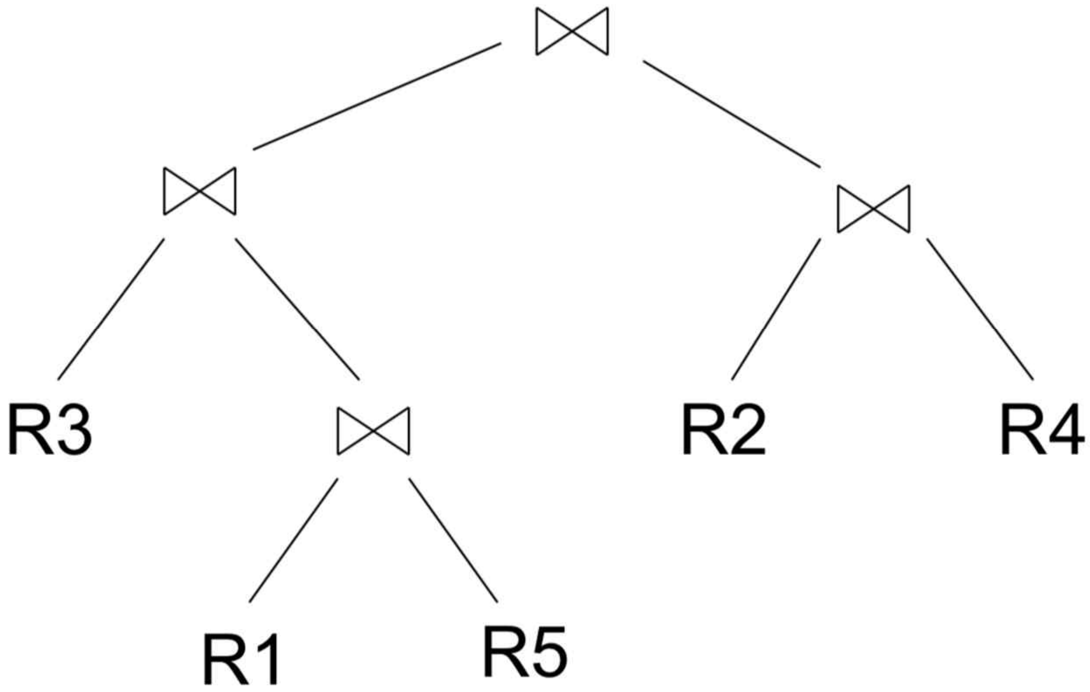
  $((R3 \bowtie R1) \bowtie (R2 \bowtie R4))$



  - Several types: bushy, linear, right-deep, and left-deep
- A (query) plan involving join = a join tree
- A partial plan = a subtree of a join tree

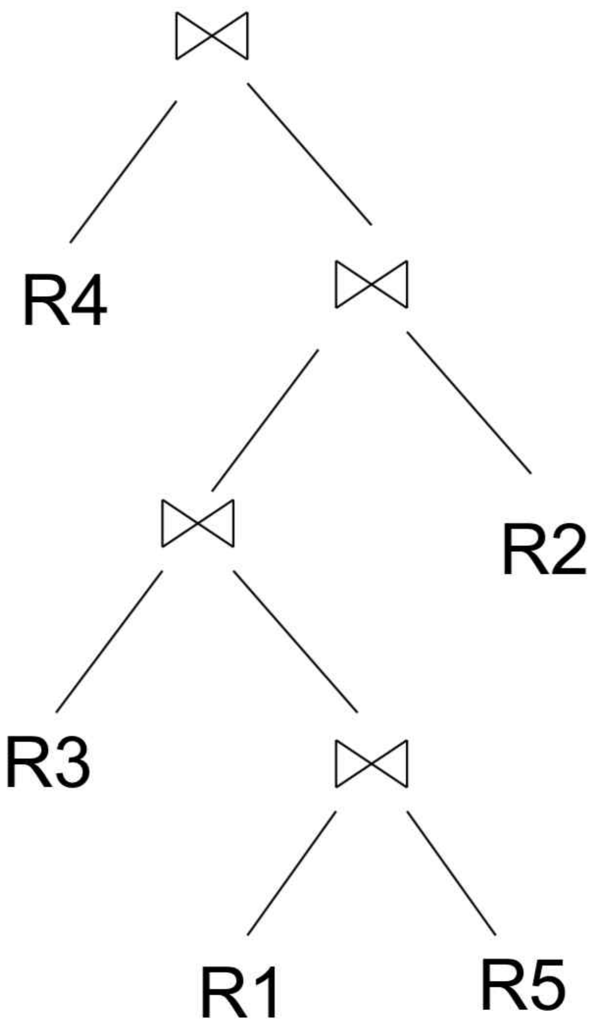
Types of Join Trees

- ***Bushy***(무성한):



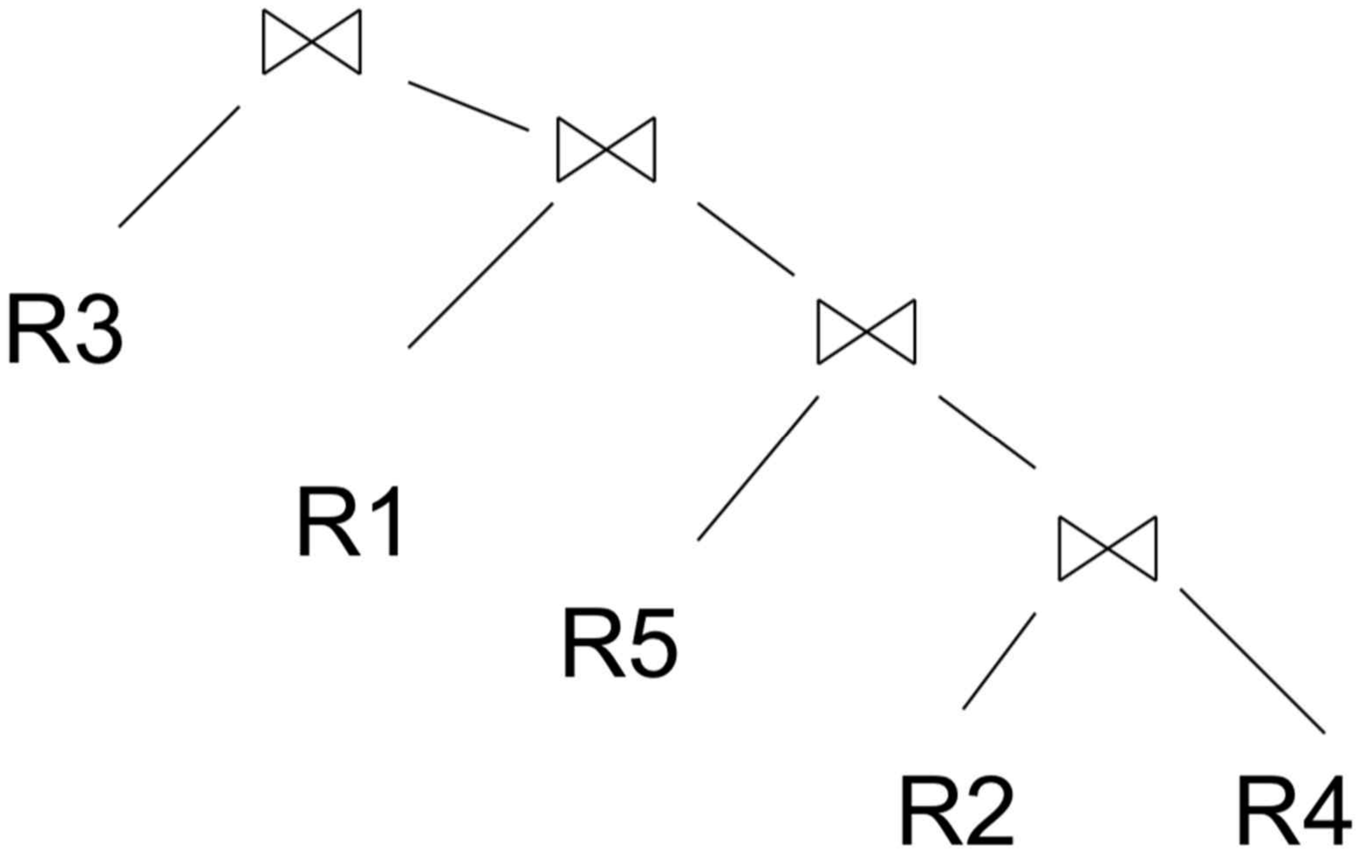
Types of Join Trees (Cont'd)

- ***Linear***:



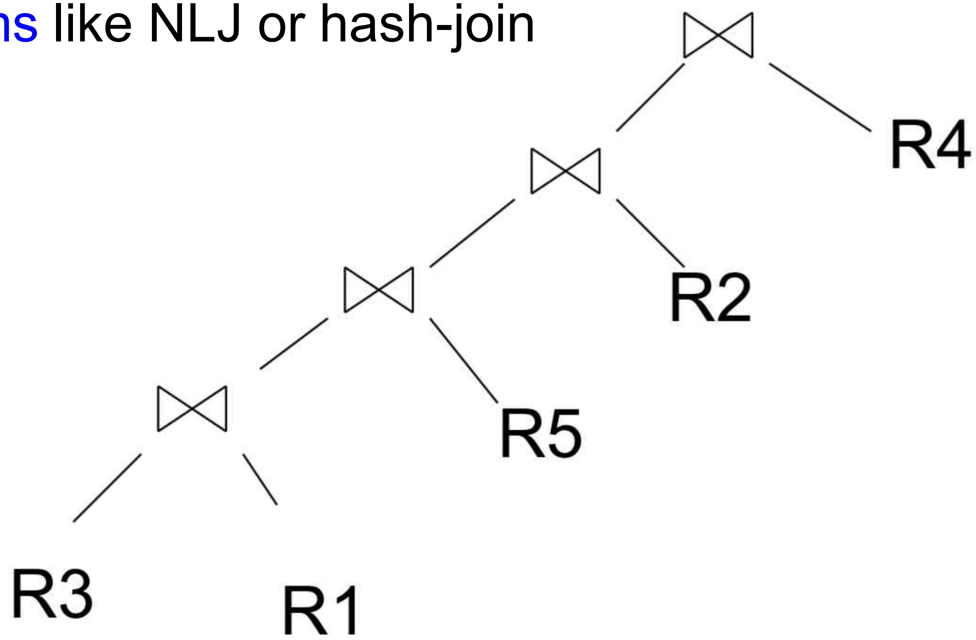
Types of Join Trees (Cont'd)

- ***Right-deep***:



Types of Join Trees (Cont'd)

- ***Left-deep***: gives the following advantages
  - Works well existing algorithms like NLJ or hash-join
  - Facilitates pipelining
  - Dynamic programming can be used with all left-deep trees
    - Optimal solution structure is developed
    - Value of the optimal solution is recursively defined
    - Optimal solution is computed and its value developed in a bottom-up fashion



Types of Join Trees (Cont'd)

- Number of permutations of left-deep and bushy join trees of N relations

No. of Relations N	No. of Left-Deep Trees $N!$	No. of Bushy Shapes $S(N)$	No. of Bushy Trees $(2N-2)!/(N-1)!$
2	2	1	2
3	6	2	12
4	24	5	120
5	120	14	1,680
6	720	42	30,240
7	5,040	132	665,280

EXAMPLE TO ILLUSTRATE COST-BASED QUERY OPTIMIZATION

Chapter 19.6

Consider the Following Query: Q2

```
SELECT P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
FROM PROJECT P, DEPARTMENT D, EMPLOYEE E
WHERE P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND P.Plocation='Stafford'
```

- Assume optimizer considers only “left-deep” trees.
- Possible join orderings?
  1) ((PROJECT ⋈ DEPARTMENT) ⋈ EMPLOYEE)
  2) ((DEPARTMENT ⋈ PROJECT) ⋈ EMPLOYEE)
  3) ((DEPARTMENT ⋈ EMPLOYEE) ⋈ PROJECT)
  4) ((EMPLOYEE ⋈ DEPARTMENT) ⋈ PROJECT)
- Now evaluate each of the potential join orders based on join cost

Sample Statistical Information for Relations in Q2

(a): Column information
(b): Table information
(c): Index information

(a)

Table_name	Column_name	Num_distinct	Low_value	High_value
PROJECT	Plocation	200	1	200
PROJECT	Pnumber	2000	1	2000
PROJECT	Dnum	50	1	50
DEPARTMENT	Dnumber	50	1	50
DEPARTMENT	Mgr_ssn	50	1	50
EMPLOYEE	Ssn	10000	1	10000
EMPLOYEE	Dno	50	1	50
EMPLOYEE	Salary	500	1	500

(b)

Table_name	Num_rows	Blocks
PROJECT	2000	100
DEPARTMENT	50	5
EMPLOYEE	10000	2000

(c)

Index_name	Uniqueness	Blevel*	Leaf_blocks	Distinct_keys
PROJ_PLOC	NONUNIQUE	1	4	200
EMP_SSN	UNIQUE	1	50	10000
EMP_SAL	NONUNIQUE	1	50	500

*Blevel is the number of levels without the leaf level.

Summary

- Query trees
- Heuristic approaches used to improve efficiency of query execution
- Reorganization of query trees
- Cost-based optimization approach

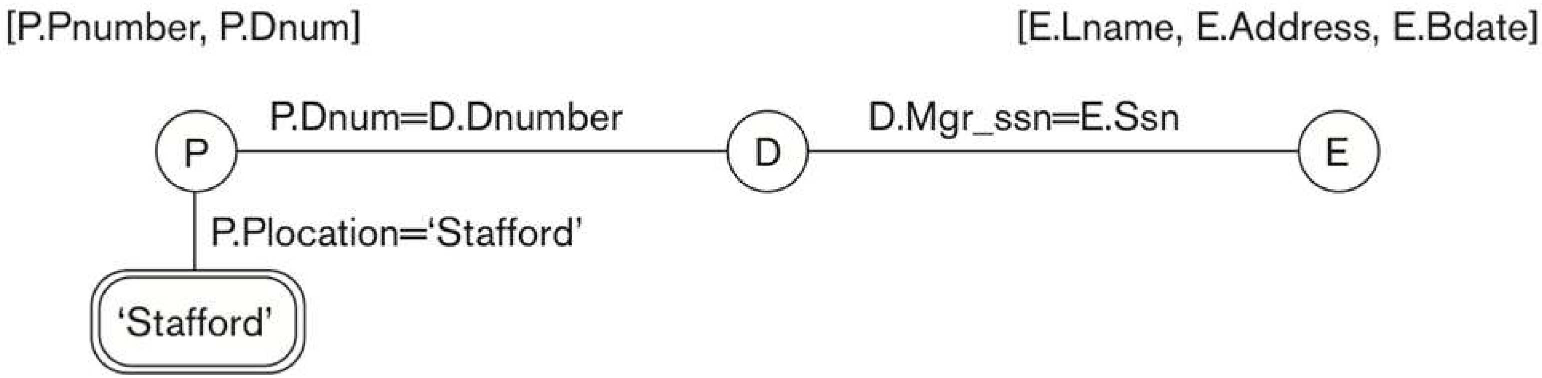
References

- https://courses.cs.washington.edu/courses/cse444/14sp/lectures/lecture11-optimization-part2.pdf

APPENDIX

[Appendix] Example of a Query Graph

- Q1 can be represented by a *query graph* below:



  - Represents *relational calculus expression*
  - Relation nodes displayed as single circles
  - Constants represented by constant nodes: double circles (or ovals)
  - Selection or join conditions as edges
  - Attributes to be retrieved displayed in square brackets

Some Transformation Rules

...proving them: ... rules that are useful in query

1. **Cascade of σ.** A conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual σ operations:

   $\sigma_{c_1 \text{ AND } c_2 \text{ AND } \ldots \text{ AND } c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$

2. **Commutativity of σ.** The σ operation is commutative:

   $\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$

3. **Cascade of π.** In a cascade (sequence) of π operations, all but the last one can be ignored:

   $\pi_{\text{List}_1}(\pi_{\text{List}_2}(\ldots(\pi_{\text{List}_n}(R))\ldots)) \equiv \pi_{\text{List}_1}(R)$

4. **Commuting σ with π.** If the selection condition c involves only those attributes $A_1, \ldots, A_n$ in the projection list, the two operations can be commuted:

   $\pi_{A_1, A_2, \ldots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, A_2, \ldots, A_n}(R))$

5. **Commutativity of ⋈ (and ×).** The join operation is commutative, as is the × operation:

   $R \bowtie_c S \equiv S \bowtie_c R$

   $R \times S \equiv S \times R$

   Notice that although the order of attributes may not be the same in the relations resulting from the two joins (or two Cartesian products), the *meaning* is the same because the order of attributes is not important in the alternative definition of relation.

6. **Commuting σ with ⋈ (or ×).** If all the attributes in the selection condition c involve only the attributes of one of the relations being joined—say, R—the two operations can be commuted as follows:

   $\sigma_c(R \bowtie S) \equiv (\sigma_c(R)) \bowtie S$

   Alternatively, if the selection condition c can be written as (c_1 AND c_2), where condition c_1 involves only the attributes of R and condition c_2 involves only the attributes of S, the operations commute as follows:

   $\sigma_c(R \bowtie S) \equiv (\sigma_{c_1}(R)) \bowtie (\sigma_{c_2}(S))$

   The same rules apply if the ⋈ is replaced by a × operation.

7. **Commuting π with ⋈ (or ×).** Suppose that the projection list is $L = \{A_1, \ldots, A_n, B_1, \ldots, B_m\}$, where $A_1, \ldots, A_n$ are attributes of R and $B_1, \ldots, B_m$ are attributes of S. If the join condition c involves only attributes in L, the two operations can be commuted as follows:

   $\pi_L(R \bowtie_c S) \equiv (\pi_{A_1, \ldots, A_n}(R)) \bowtie_c (\pi_{B_1, \ldots, B_m}(S))$

   If the join condition c contains additional attributes not in L, these must be added to the projection list, and a final π operation is needed. For example, if attributes $A_{n+1}, \ldots, A_{n+k}$ of R and $B_{m+1}, \ldots, B_{m+p}$ of S are involved in the join condition c but are not in the projection list L, the operations commute as follows:

   $\pi_L(R \bowtie_c S) \equiv \pi_L((\pi_{A_1, \ldots, A_n, A_{n+1}, \ldots, A_{n+k}}(R)) \bowtie_c (\pi_{B_1, \ldots, B_m, B_{m+1}, \ldots, B_{m+p}}(S)))$

   For ×, there is no condition c, so the first transformation rule always applies by replacing $\bowtie_c$ with ×.

8. **Commutativity of set operations.** The set operations ∪ and ∩ are commutative, but − is not.

9. **Associativity of ⋈, ×, ∪, and ∩.** These four operations are individually associative; that is, if both occurrences of θ stand for the same operation that is any one of these four operations (throughout the expression), we have:

   $(R\ \theta\ S)\ \theta\ T \equiv R\ \theta\ (S\ \theta\ T)$

10. **Commuting σ with set operations.** The σ operation commutes with ∪, ∩, and −. If θ stands for any one of these three operations (throughout the expression), we have:

    $\sigma_c(R\ \theta\ S) \equiv (\sigma_c(R))\ \theta\ (\sigma_c(S))$

11. **The π operation commutes with ∪.**

    $\pi_L(R \cup S) \equiv (\pi_L(R)) \cup (\pi_L(S))$

12. **Converting a (σ, ×) sequence into ⋈.** If the condition c of a σ that follows a × corresponds to a join condition, convert the (σ, ×) sequence into a ⋈ as follows:

    $(\sigma_c(R \times S)) \equiv (R \bowtie_c S)$

13. **Pushing σ in conjunction with set difference.**

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

    However, σ may be applied to only one relation:

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

14. **Pushing σ to only one argument in ∩.**

    If in the condition σ_c all attributes are from relation R, then:

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

15. **Some trivial transformations.**

    If S is empty, then $R \cup S = R$

    If the condition c in σ_c is true for the entire R, then $\sigma_c(R) = R$.

NOT (c_1 AND c_2) ≡ (NOT c_1) OR (NOT c_2)

NOT (c_1 OR c_2) ≡ (NOT c_1) AND (NOT c_2)

Example Table for Calculating Execution Cost

Cost Estimate Items	Number
$r_{\mathrm{E(MPLOYEE)}}$	10,000 records
$b_{\mathrm{E(MPLOYEE)}}$	2,000 blocks
$r_{\mathrm{D(DEPARTMENT)}}$	125 records
$b_{\mathrm{D(DEPARTMENT)}}$	13 blocks
x_{Dnumber} (Number of level of a primary index on the Dnumber attribute in DEPARTMENT)	1
$x_{\mathrm{Mgr\_ssn}}$ (Number of level of a secondary index on the Mgr_ssn attribute in DEPARTMENT)	2
$s_{\mathrm{Mgr\_ssn}}$ (Selectivity cardinality on Mgr_ssn)	1
Join selectivity (js) on **EMPLOYEE** $\bowtie_{\mathrm{Dno=Dnumber}}$ **DEPARTMENT**	$\frac{1}{max(NDV(\mathrm{Dno,EMPLOYEE}), NDV(\mathrm{Dnumber,DEPARTMENT}))} = \frac{1}{125}$
Blocking factor of the resulting join (bfr_{ED})	4 records

DEPARTMENT.Dnumber: a key attribute of DEPARTMENT