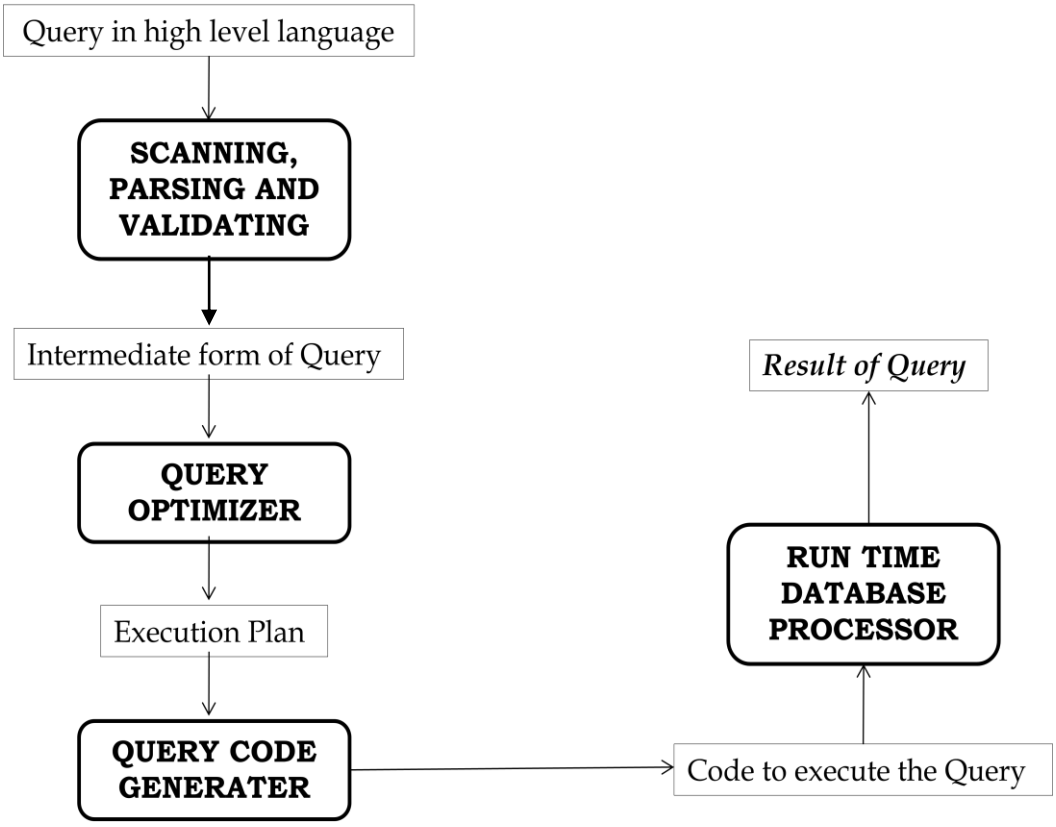
Chapter 8: Query Optimization

In this chapter we discuss the techniques used by a DBMS to process, optimize, and execute high-level queries. A query expressed in a high-level query language such as SQL must first be scanned, parsed, and validated. The scanner identifies the language tokens—such as SQL keywords, attributes names and relation names—in the text of the query, whereas the parser checks the query syntax to determine whether it is formulated according to the syntax rules (rules of grammar) of the query language. The query must also be validated, by checking that all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried. An internal representation of the query is then created, usually as a tree data structure called a query tree. It is also possible to represent the query using a graph data structure called a query graph. The DBMS must then devise an execution strategy for retrieving the result of the query from the database files. A query typically has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as query optimization.

The query optimizer module has the task of producing an execution plan, and the code generator generates the code to execute that plan. The runtime database processor has the task of running the query code, whether in compiled or interpreted mode, to produce the query result. If a runtime error results, an error message is generated by the runtime database processor.



The term optimization is actually a misnomer because in some cases the chosen execution plan is not the optimal (best) strategy—it is just a reasonably efficient strategy for executing the query. Finding the optimal strategy is usually too time-consuming except for the simplest of queries and may require information on how the files are implemented and even on the contents of the files—information that may not be fully available in the DBMS catalog. Hence, planning of an execution strategy may be a more accurate description than query optimization.

Heuristics in Query Optimization

In this section we discuss optimization techniques that apply heuristic rules to modify the internal representation of a query—which is usually in the form of a query tree or a query graph data Structure—to improve its expected performance. The parser of a high-level query first generates an Initial internal representation, which is then optimized according to heuristic rules. Following that, a Query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.

One of the main heuristic rules is to apply SELECT and PROJECT operations before applying the JOIN or other binary operations. This is because the size of the file resulting from a binary operation— such as JOIN—is usually a multiplicative function of the sizes of the input files. The SELECT and PROJECT operations reduce the size of a file and hence should be applied before a join or other binary operation.

- ***Example :***

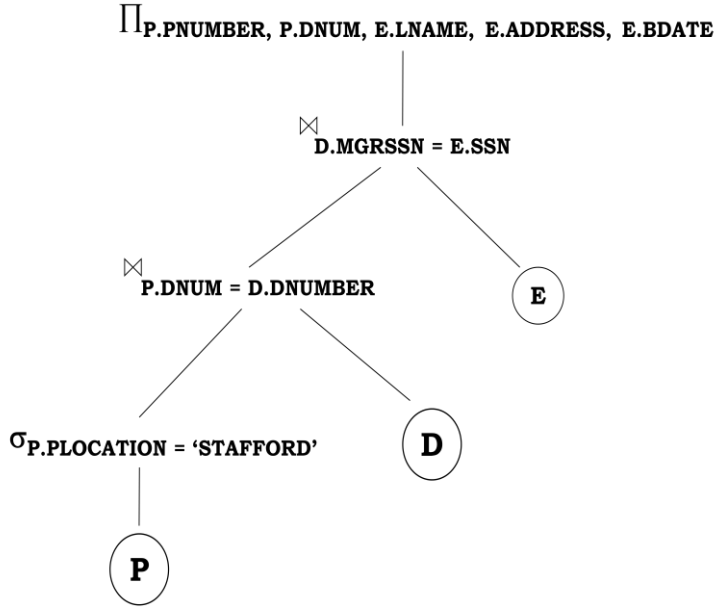
SQL Query:
```
SELECT   P.PNUMBER, D.DNUM, E.LNAME, E.ADDRESS, E.BDATE
FROM     PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E
WHERE    D.MGRSSN = E.SSN AND P.DNUM = D.DUMBER
           AND P.PLOCATION = 'STAFFORD'
```

Relational Algebra:

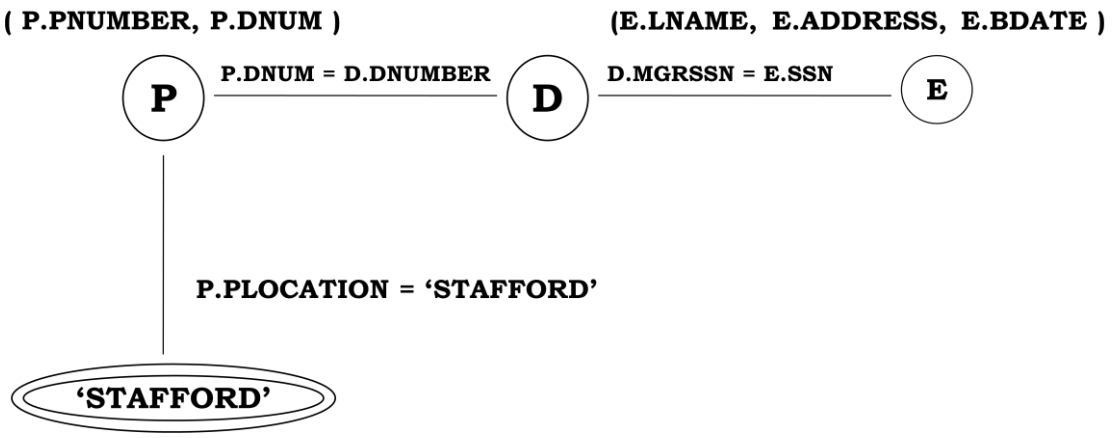
$$\Pi_{\text{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}} (\sigma_{\text{PLOCATION = 'STAFFORD'}} (\text{PROJECT}) \bowtie_{\text{DNUM = DNUMBER}} (\text{DEPARTMENT}) \bowtie_{\text{MGRSSN = SSN}} (\text{EMPLOYEE}))$$

- ***Query Tree***

A query tree is a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as leaf nodes of the tree, and represents the relational algebra operations as internal nodes. An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation. The execution terminates when the root node is executed and produces the result relation for the query.



- ***Query Graph***



General Transformation Rules for Relational Algebra Operations.

There are many rules for transforming relational algebra operations into equivalent ones. Here we are interested in the meaning of the operations and the resulting relations. Hence, if two relations have the same set of attributes in a different order but the two relations represent the same information, we consider the relations equivalent. We gave an alternative definition of relation that makes order of attributes unimportant; we will use this definition here. We now state some transformation rules that are useful in query optimization, without proving them:

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

$$\sigma_{c1 \text{ and } c2 \text{ and } \ldots\ldots \text{ cn}}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots\ldots(\sigma_{cn}(R))\ldots\ldots))$$

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

$$\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$$

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

$$\Pi_{list1}(\Pi_{list2}(\ldots\ldots(\Pi_{listn}(R)) \ldots\ldots\ldots)) \equiv \Pi_{list1}(R)$$

4. ***Commuting σ with Π****:* If the selection condition c involves only those attributes A1,….., An in the projection list, the two operations can be commuted.

$$\Pi_{A1, A2, \ldots\ldots, An}(\sigma_c(R)) \equiv \sigma_c(\Pi_{A1, A2, \ldots\ldots, An}(R))$$

5. ***Commutative of ⋈ (and X) :*** The ⋈ operation is commutative , as is the **X** operations:

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

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 order of attributes is not important in the alternative definition of relation.

6. ***Commuting σ with ⋈ (or X):*** 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:

$$\sigma_c (R \bowtie S) \equiv (\sigma_{c1} (R)) \bowtie (\sigma_{c2} (S))$$

The same rules apply if the is replaced by a X operation

7. ***Commuting Π with ⋈ (or X):*** Suppose that the projection list is L = { A_1, ……. An, B_1, …….. Bm }, where A_1 , ..., An are attributes of *R* and B_1,...,Bm are attributes of *S*. If the join condition *c* involves only attributes in *L,* the two operations can be commuted as follows:

$$\Pi_L (R \bowtie_c S) \equiv (\Pi_{A1, \ldots\ldots An}(R)) \bowtie_c (\Pi_{B1, \ldots\ldots, Bm} (S))$$

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},……,A_{n+k} of *R* and B_{m+1} , ….. , 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:

$$\Pi_L(R \bowtie_c S) \equiv \Pi_L(\Pi_{A1,\ldots\ldots An, An+1, \ldots\ldots,An+k}(R)) \bowtie_c (\Pi_{B1,\ldots\ldots,Bm, Bm+1 , \ldots\ldots , Bm+p} (S)).$$

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

8. ***Commutativity of set operations***: The set operations ∩ and **U** are commutative but **-** is not.

9. ***Associativity of ⋈ , X, U, and ∩ :*** These four operations are individually associative; that is, if **θ** stands for 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 **U**, **∩**, 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 U:***

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

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

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

There are other possible transformations. For example, a selection or join condition *c* can be converted into an equivalent condition by using the following rules (DeMorgan's laws):

NOT (c1 AND c2) M (NOT c1) OR (NOT c2)

NOT (c1 OR c2) M (NOT c1) AND (NOT c2)

Heuristic Optimization of Query Trees

In general, many different relational algebra expressions—and hence many different query trees—can be equivalent; that is, they can correspond to the same query. The query parser will typically generate a standard initial query tree to correspond to an SQL query, without doing any optimization. For example, for a select–project–join query, such as Q2, The CARTESIAN PRODUCT of the relations specified in the FROM clause is first applied; then the selection and join conditions of the WHERE clause are applied, followed by the projection on the SELECT clause attributes. Such a canonical query tree represents a relational algebra expression that is very inefficient if executed directly, because of the CARTESIAN PRODUCT (X) operations. For example, if the PROJECT, DEPARTMENT, and EMPLOYEE relations had record sizes of 100, 50, and 150 bytes and contained 100, 20, and 5000 tuples, respectively, the result of the CARTESIAN PRODUCT would contain 10 million tuples of record size 300 bytes each. It is now the job of the heuristic query optimizer to transform this initial query tree into a final query tree that is efficient to execute.

The optimizer must include rules for equivalence among relational algebra expressions that can be applied to the initial tree. The heuristic query optimization rules then utilize these equivalence expressions to transform the initial tree into the final, optimized query tree. We first discuss informally how a query tree is transformed by using heuristics. Then we discuss general transformation rules and show how they may be used in an algebraic heuristic optimizer.

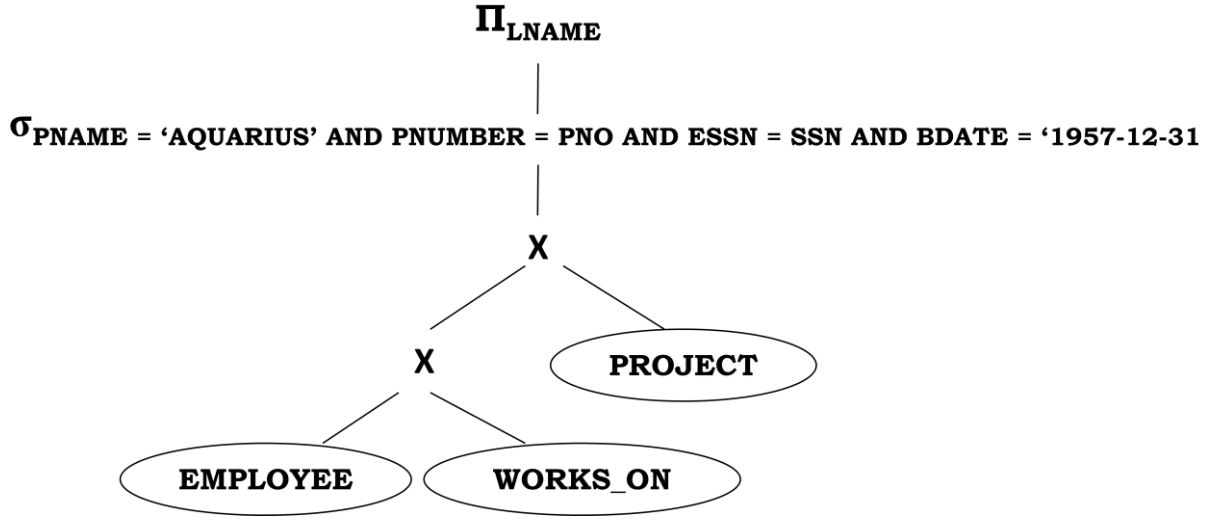
SQL Query:

```
SELECT  LNAME
FROM    EMPLOYEE, WORKS_ON, PROJECT
WHERE   PNAME = 'AQUARIUS' AND PNUMBER = PNO AND  ESSN = SSN AND
        BDATE = '1957-12-31';
```

Relational Algebra:

Π_{LNAME} ($\sigma_{PNAME = 'AQUARIUS' AND PNUMBER = PNO AND ESSN = SSN AND BDATE = '1957-12-31'}$ (EMPLOYEE X WORKS_ON X PROJECT))

STEP 1 :



➢ STEP 2 :

❖ Using Rules

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

$$\sigma_{c1 \text{ and } c2 \text{ and } \ldots\ldots \text{ cn}}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots\ldots(\sigma_{cn}(R))\ldots\ldots))$$

- ✓ ***Commutative of σ:*** The **σ** operation is commutative.

$$\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$$

- ✓ ***Commuting σ with Π***: If the selection condition c involves only those attributes A1, ….., An in the projection list, the two operations can be commuted.

$$\Pi_{A1, A2, \ldots\ldots, An}(\sigma_c(R)) \equiv \sigma_c(\Pi_{A1, A2, \ldots\ldots, An}(R))$$

- ***Commuting σ with ⋈ (or X):*** 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:

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

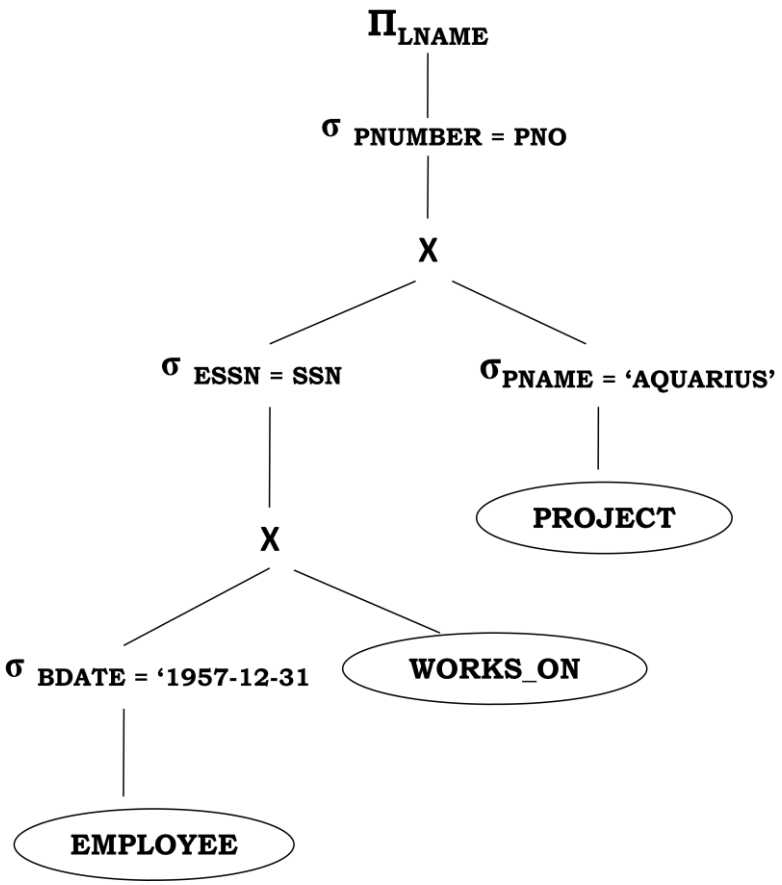
Alternatively, if the selection condition *c* can be written as (*c*1 AND *c*2), where condition *c*1 involves only the attributes of *R* and condition *c*2 involves only the attributes of *S*, the operations commute as follows:

$$\sigma_c (R \bowtie S) \equiv (\sigma_{c1} (R)) \bowtie (\sigma_{c2} (S))$$

The same rules apply if the is replaced by a X operation

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

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



STEP 3:

❖ Using Rules

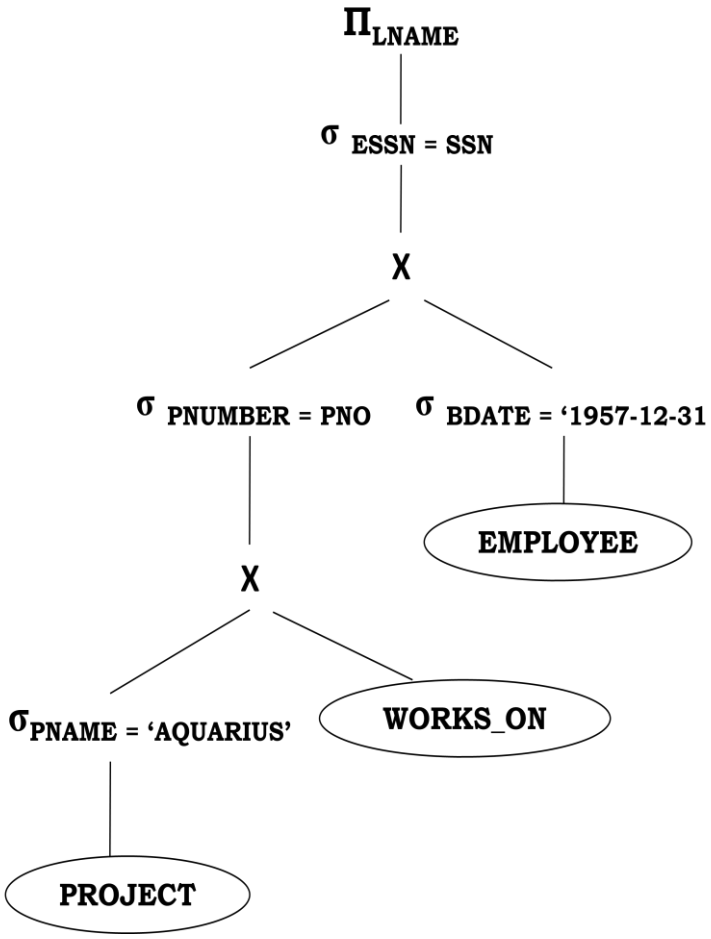
- ✓ ***Commutative of ⋈ (and X) :*** The ⋈ operation is commutative , as is the **X** operations:

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

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 order of attributes is not important in the alternative definition of relation.

- ✓ ***Associativity of ⋈ , X, U, and ∩ :*** These four operations are individually associative; that is, if **θ** stands for any one of these four operations (throughout the expression), we have:

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

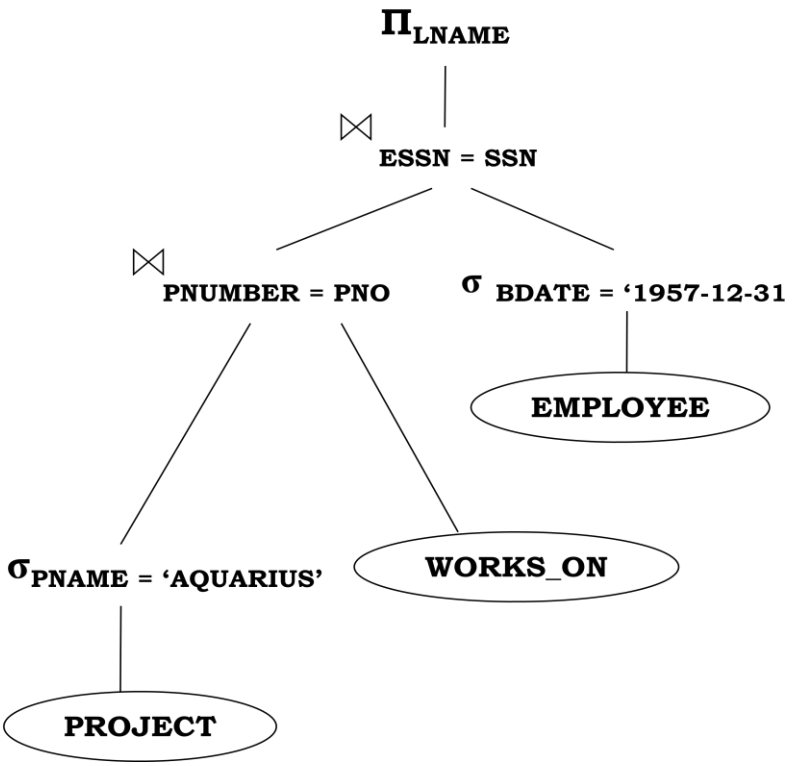


STEP 4:

❖ Using Rules

- ✓ ***Converting a (σ, X) sequence into ⋈ :*** If the condition *c* of a **σ** that follows a **X** corresponds to a join condition, convert the (**σ**, X) sequence into a as follows:

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



STEP 5:

❖ Using Rules

- ✓ ***Cascade of Π*** *:* In a cascade (sequence) of **Π** operations, all but the last one can be ignored:

$$\Pi_{list1}\ (\Pi_{list2}\ (.....(\Pi_{listn}\ (R))\ .....)) \equiv \Pi_{list1}\ (R)$$

- ✓ ***Commuting σ with Π***: If the selection condition c involves only those attributes A1, ….., An in the projection list, the two operations can be commuted.

$$\Pi_{A1,\ A2,\ .....\ ,\ An}\ (\sigma_c\ (R)) \equiv \sigma_c\ (\Pi_{A1,\ A2,\ .....\ ,\ An}(R))$$

- ✓ ***Commuting Π with ⋈ (or X):*** Suppose that the projection list is L = { A_1, ……. An, B_1, …….. Bm }, where A_1 , ..., An are attributes of *R* and B_1,...,Bm are attributes of *S*. If the join condition *c* involves only attributes in *L,* the two operations can be commuted as follows:

$$\Pi_L (R \bowtie_c S) \equiv (\Pi_{A1, \ldots\ldots An}(R)) \bowtie_c (\Pi_{B1, \ldots\ldots , Bm}(S))$$

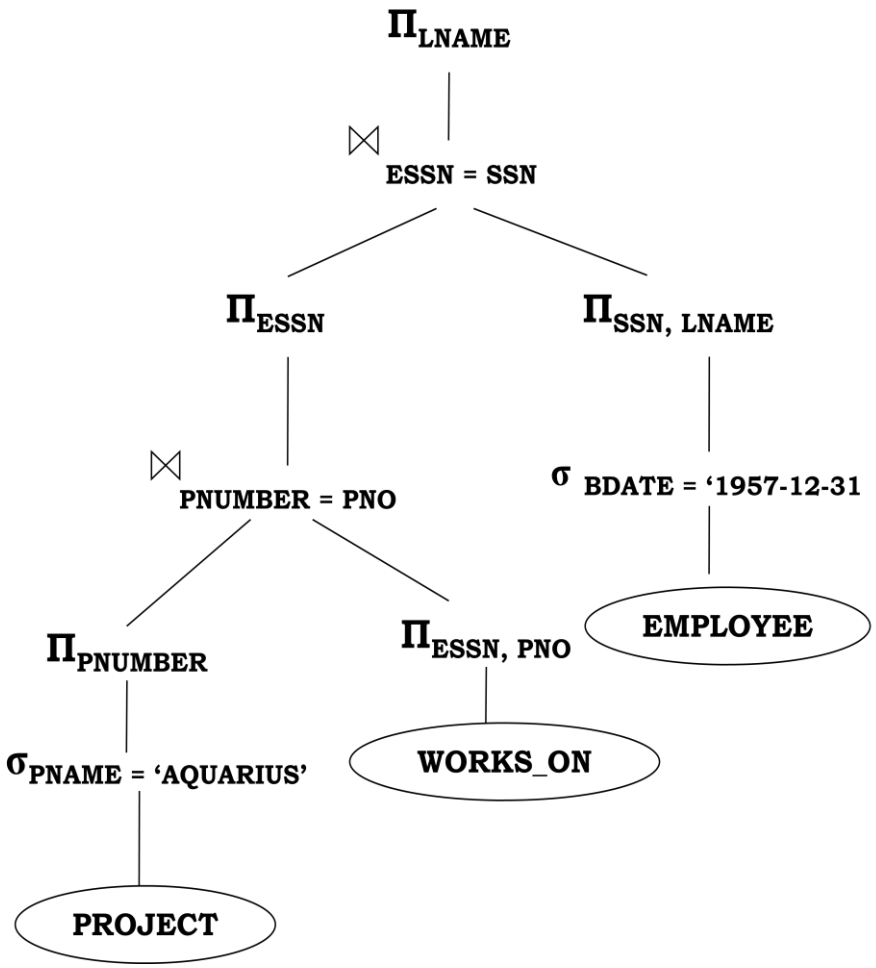
If the join condition *c* contains additional attributes not in *L,* these must be added to the projection list, and a final Π operation is needed. For example, if attributes A_{n+1},……,A_{n+k} of *R* and B_{m+1} , ….. , B_{m+p} of *S* are involved in the join condition *c* but are not in the projection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

$$\Pi_L(R \bowtie_c S) \equiv \Pi_L(\Pi_{A1,\ldots\ldots An, An+1, \ldots\ldots,An+k}(R)) \bowtie_c (\Pi_{B1,\ldots\ldots,Bm, Bm+1 , \ldots\ldots , Bm+p}(S)).$$

For $\mathbf{X}$, there is no condition *c,* so the first transformation rule always applies by replacing $\bowtie_c$ with $\mathbf{X}$.

- ✓ ***The Π operation commutes with U:***

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



We can now outline the steps of an algorithm that utilizes some of the above rules to transform an initial query tree into an optimized tree that is more efficient to execute (in most cases). The algorithm will lead to transformations similar to those discussed in our example. The steps of the algorithm are as follows:

1. Using Rule 1, break up any SELECT operations with conjunctive conditions into a cascade of SELECT operations. This permits a greater degree of freedom in moving SELECT operations down different branches of the tree.

2. Using Rules 2, 4, 6, and 10 concerning the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition.

3. Using Rules 5 and 9 concerning commutativity and associativity of binary operations, rearrange the leaf nodes of the tree using the following criteria. First, position the leaf node relations with the most restrictive SELECT operations so they are executed first in the query tree representation. The definition of most restrictive SELECT can mean either the ones that produce a relation with the fewest tuples or with the smallest absolute size . Another possibility is to define the most restrictive SELECT as the one with the smallest selectivity; this is more practical because estimates of selectivities are often available in the DBMS catalog. Second, make sure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operations; for example, if the two relations with the most restrictive SELECT do not have a direct join condition between them, it may be desirable to change the order of leaf nodes to avoid Cartesian products .

4. Using Rule 12, combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition.

5. Using Rules 3, 4, 7, and 11 concerning the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed. Only those attributes needed in the query result and in subsequent operations in the query tree should be kept after each PROJECT operation.

6. Identify sub trees that represent groups of operations that can be executed by a single algorithm.

Using Selectivity and Cost Estimates in Query Optimization

A query optimizer should not depend solely on heuristic rules; it should also estimate and compare the costs of executing a query using different execution strategies and should choose the strategy with the lowest cost estimate. For this approach to work, accurate cost estimates are required so that different strategies are compared fairly and realistically. In addition, we must limit the number of execution strategies to be considered; otherwise, too much time will be spent making cost estimates for the many possible execution strategies. Hence, this approach is more suitable for compiled queries where the optimization is done at compile time and the resulting execution strategy code is stored and executed directly at runtime. For interpreted queries, where the entire process occurs at runtime, a full-scale optimization may slow down the response time. A more elaborate optimization is indicated for compiled queries, whereas a partial, less time-consuming optimization works best for interpreted queries.

We call this approach cost-based query optimization, and it uses traditional optimization techniques that search the solution space to a problem for a solution that minimizes an objective (cost) function. The

cost functions used in query optimization are estimates and not exact cost functions, so the optimization may select a query execution strategy that is not the optimal one.

➢ *Cost Components for Query Execution*

The cost of executing a query includes the following components:

- ✓ Access cost to secondary storage: This is the cost of searching for, reading, and writing data blocks that reside on secondary storage, mainly on disk. The cost of searching for records in a file depends on the type of access structures on that file, such as ordering, hashing, and primary or secondary indexes. In addition, factors such as whether the file blocks are allocated contiguously on the same disk cylinder or scattered on the disk affect the access cost.

- ✓ Storage cost: This is the cost of storing any intermediate files that are generated by an execution strategy for the query.

- ✓ Computation cost: This is the cost of performing in-memory operations on the data buffers during query execution. Such operations include searching for and sorting records, merging records for a join, and performing computations on field values.

- ✓ Memory usage cost: This is the cost pertaining to the number of memory buffers needed during query execution.

- ✓ Communication cost: This is the cost of shipping the query and its results from the database site to the site or terminal where the query originated.

For large databases, the main emphasis is on minimizing the access cost to secondary storage. Simple cost functions ignore other factors and compare different query execution strategies in terms of the number of block transfers between disk and main memory. For smaller databases, where most of the data in the files involved in the query can be completely stored in memory, the emphasis is on minimizing computation cost. In distributed databases, where many sites are involved communication cost must be minimized also. It is difficult to include all the cost components in a (weighted) cost function because of the difficulty of assigning suitable weights to the cost components. That is why some cost functions consider a single factor only—disk access

➢ *Catalog Information Used in Cost Functions*

To estimate the costs of various execution strategies, we must keep track of any information that is needed for the cost functions. This information may be stored in the DBMS catalog, where it is accessed by the query optimizer. First, we must know the size of each file. For a file whose records are all of the same type, the number of records (tuples) (r), the (average) record size (R), and the number of blocks (b) (or close estimates of them) are needed. The blocking factor (bfr) for the file may also be needed. We must also keep track of the primary access method and the primary access attributes for each file. The file records may be unordered, ordered by an attribute with or without a primary or clustering index, or hashed on a key attribute. Information is kept on all secondary indexes and indexing attributes. The number of levels (x) of each multilevel index (primary, secondary, or clustering) is needed for cost functions that estimate the number of block accesses that occur during query execution. In some cost functions the number of first-level index blocks is needed.

Another important parameter is the number of distinct values (d) of an attribute and its selectivity (sl), which is the fraction of records satisfying an equality condition on the attribute. This allows

estimation of the selection cardinality (s = sl * r) of an attribute, which is the average number of records that will satisfy an equality selection condition on that attribute. For a key attribute, d = r, sl = 1/r and s = 1. For a nonkey attribute, by making an assumption that the d distinct values are uniformly distributed among the records, we estimate sl = (1/d) and so s = (r/d)

➢ *Cost Functions for SELECT*

- ✓ S1. Linear search (brute force) approach: We search all the file blocks to retrieve all records satisfying the selection condition; hence, C_{s1a} = b. For an equality condition on a key, only half the file blocks are searched on the average before finding the record, so C_{s1b} = (b/2) if the record is found; if no record satisfies the condition, C_{s1b} = b.

- ✓ S2. Binary search: This search accesses approximately $C_{s2} = \log_2 b + (s/bfr) - 1$ file blocks. This reduces to $\log_2 b$ if the equality condition is on a unique (key) attribute, because s = 1 in this case.

- ✓ S3. Using a primary index (S3a) or hash key (S3b) to retrieve a single record: For a primary index, retrieve one more block than the number of index levels; hence, C_{s3a} = x + 1. For hashing, the cost function is approximately C_{s3b} = 1 for static hashing or linear hashing, and it is 2 for extendible hashing.

- ✓ S4. Using an ordering index to retrieve multiple records: If the comparison condition is >, >=,<, or <= on a key field with an ordering index, roughly half the file records will satisfy the condition. This gives a cost function of C_{s4}= x + (b/2). This is a very rough estimate, and although it may be correct on the average, it may be quite inaccurate in individual cases.

- ✓ S5. Using a clustering index to retrieve multiple records: Given an equality condition, s records will satisfy the condition, where s is the selection cardinality of the indexing attribute. This means that (s/bfr) file blocks will be accessed, giving C_{s5} = x + (s/bfr).

Example to Illustrate Cost-Based Query Optimization

SQL Query:

```
SELECT  P.PNUMBER, D.DNUM, E.LNAME, E.ADDRESS, E.BDATE
FROM    PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E
WHERE   D.MGRSSN = E.SSN AND P.DNUM = D.DUMBER  AND
        P.PLOCATION = 'STAFFORD'
```

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	MGRSSN	50	1	50
EMPLOYEE	SSN	10000	1	10000
EMPLOYEE	DNO	50	1	50
EMPLOYEE	SALARY	500	1	500

TABLE_NAME	NUM_ROWS	BLOCKS
PROJECT	2000	100
DEPARTMENT	50	5
EMPLOYEE	10000	2000

INDEX_NAME	UNIQUENES	BLEVEL	LEAF_BLOCKS	DISTINCT_KEYS
PROJ_PLOC	NONUNIQUE	1	4	200
EMP_SSN	UNIQUE	1	50	10000
EMP_SAL	NONUNIQUE	1	50	500

The first cost-based optimization to consider is join ordering. As previously mentioned, we assume the optimizer considers only left-deep trees, so the potential join orders—without Cartesian product—are

PROJECT ⋈ DEPARTMENT ⋈ EMPLOYEE
DEPARTMENT ⋈ PROJECT ⋈ EMPLOYEE
DEPARTMENT ⋈ EMPLOYEE ⋈ PROJECT
EMPLOYEE ⋈ DEPARTMENT ⋈ PROJECT

Assume that the selection operation has already been applied to the PROJECT relation. If we assume a materialized approach, then a new temporary relation is created after each join operation. To examine the cost of join order (1), the first join is between PROJECT and DEPARTMENT. Both the join method and the access methods for the input relations must be determined. Since DEPARTMENT has no index according to Figure 18.08, the only available access method is a table scan (that is, a linear search). The PROJECT relation will have the selection operation performed before the join, so two options exist: table scan (linear search) or utilizing its PROJ_PLOC index, so the optimizer must compare their estimated costs. The statistical information on the PROJ_PLOC index shows the number of index levels x = 2 (root plus leaf levels). The index is nonunique (because PLOCATION is not a key of PROJECT), so the optimizer assumes a uniform data distribution and estimates the number of record pointers for each PLOCATION value to be 10. This is computed from the tables multiplying SELECTIVITY * NUM_ROWS, where SELECTIVITY is estimated by 1/NUM_DISTINCT. So the cost of using the index and accessing the records is estimated to be 12 block accesses (2 for the index and 10 for the data blocks). The cost of a table scan is estimated to be 100 block accesses, so the index access is more efficient as expected.

In the materialized approach, a temporary file TEMP1 of size 1 block is created to hold the result of the selection operation. The file size is calculated by determining the blocking factor using the formula NUM_ROWS/BLOCKS, which gives 2000/100 or 20 rows per block. Hence, the 10 records selected from the PROJECT relation will fit into a single block. Now we can compute the estimated cost of the first join. We will consider only the nested-loop join method, where the outer relation is the temporary file, TEMP1, and the inner relation is DEPARTMENT. Since the entire TEMP1 file fits in available buffer space, we need to read each of the DEPARTMENT table's five blocks only once, so the join cost is six block accesses plus the cost of writing the temporary result file, TEMP2. The optimizer would have to determine the size of TEMP2. Since the join attribute DNUMBER is the key for DEPARTMENT, any DNUM value from TEMP1 will join with at most one record from DEPARTMENT, so the number of rows in the TEMP2 will be equal to the number of rows in TEMP1, which is 10. The optimizer would determine the record size for TEMP2 and the number of blocks

needed to store these 10 rows. For brevity, assume that the blocking factor for TEMP2 is five rows per block, so a total of two blocks are needed to store TEMP2.

Finally, the cost of the last join needs to be estimated. We can use a single-loop join on TEMP2 since in this case the index EMP_SSN can be used to probe and locate matching records from EMPLOYEE. Hence, the join method would involve reading in each block of TEMP2 and looking up each of the five MGRSSN values using the EMP_SSN index. Each index lookup would require a root access, a leaf access, and a data block access (x+1, where the number of levels x is 2). So, 10 lookups require 30 block accesses. Adding the two block accesses for TEMP2 gives a total of 32 block accesses for this join.