Extended Relational Algebra Operations

Some common database requests—which are needed in commercial applications for RDBMSs—cannot be performed with the original relational algebra operations. These are additional relational algebra operations or extended relational algebra operations to express these requests.

These operations enhance the expressive power of the original relational algebra.

Extended Relational algebra operations includes

- Generalized Projection
- Aggregate Functions
- Outer Join

Generalized Projection

The generalized projection operation extends the projection operation by allowing functions of attributes to be included in the projection list.

The generalized form can be expressed as:

$$\pi_{F1,\, F2,\, \ldots,\, Fn}\,(R)$$

where *F*1, *F*2, ..., *Fn* are functions over the attributes in relation *R* and may involve arithmetic operations and constant values.

consider the relation

EMPLOYEE (Ssn, Salary, Deduction, Years_service)

A report may be required to show

Net Salary = Salary – Deduction,

Bonus = 2000 * Years_service, and

Tax = 0.25 * Salary.

Then a generalized projection combined with renaming may be used as follows:

$$\text{REPORT} \leftarrow \rho_{(\text{Ssn, Net salary, Bonus, Tax})}(\pi_{\text{Ssn, Salary} - \text{Deduction, } 2000 * \text{Years service, } 0.25 * \text{Salary}}(\text{EMPLOYEE}))$$

Aggregate Functions and Grouping

Aggregate functions cannot be expressed in the basic relational algebra is to specify mathematical **aggregate functions** on collections of values from the database.

Examples of such functions include retrieving the average or total salary of all employees or the total number of employee tuples.

Aggregate functions include:

- **avg**: average value
- **min**: minimum value
- **max**: maximum value
- **sum**: sum of values
- **count**: number of values

Another common type of request involves grouping the tuples in a relation by the value of some of their attributes and then applying an aggregate function *independently to each group*.

An example would be to group EMPLOYEE tuples by Dno, so that each group includes the tuples for employees working in the same department.

We can define an AGGREGATE FUNCTION operation, using the symbol

$\mathfrak{F}$

pronounced as *script F*

Relational Algebra syntax for Aggregate Function and Grouping.

$$\langle \text{grouping attributes} \rangle \, \mathfrak{F} \, \langle \text{function list} \rangle \, (R)$$

where <grouping attributes> is a list of attributes of the relation specified in *R*, and <function list> is a list of (<function> <attribute>) pairs.

In each such pair, <function> is one of the allowed functions—such as SUM, AVG, MAX, MIN, COUNT—and <attribute> is an attribute of the relation specified by *R*.

For example, retrieve, the number of employees working in the company, and their average salary.

$$\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$$

e.g. Retrieve each department number, the number of employees in the department, and their average salary

$$\text{Dno}\ \Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$$

OUTER JOIN OPERATIONS

Basically 3 types of outer join operations

- **Left Outer Join**
- **Right Outer Join**
- **Full Outer Join**

(Already discussed, Refer to notes)

Query Processing

A query expressed in a high-level query language such as SQL must first be scanned, parsed, and validated.

The **scanner** identifies the query tokens—such as SQL keywords, attribute names, and relation names—that appear in the text of the query,

The **parser** checks the query syntax to determine whether it is formulated according to the syntax rules (rules of grammar) of the query language.

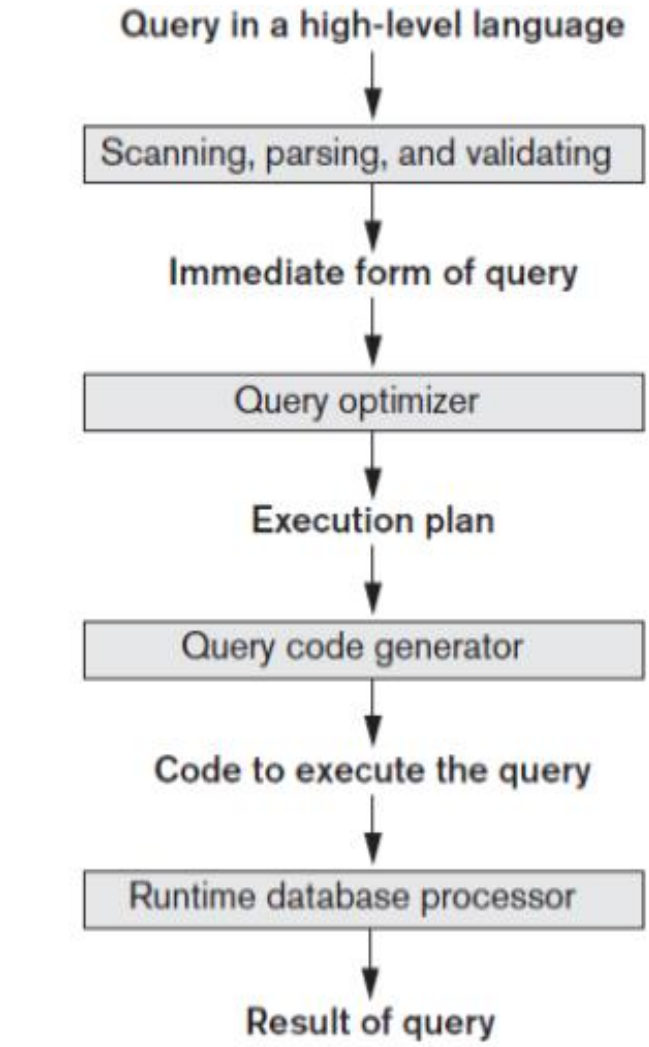
The query must also be **validated** by checking that all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried.

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** or **query plan** for retrieving the results 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 a good execution plan, and the **code generator** generates the code to execute that plan. The **runtime database processor** has the task of running (executing) 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.



Translating SQL Queries into Relational Algebra

SQL is the query language that is used in most commercial RDBMSs. An SQL query is first translated into an equivalent extended relational algebra expression, represented as a query tree data structure, that is then optimized.

Typically, SQL queries are decomposed into *query blocks,* which form the basic units that can be translated into the algebraic operators and optimized.

A **query block** contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses if these are part of the block. Hence, nested queries within a query are identified as separate query blocks.

Consider the following SQL query to retrieve the names of employees (from any department in the company) who earn a salary that is greater than the *highest salary in department 5.*

```
SELECT  Lname, Fname
FROM    EMPLOYEE
WHERE   Salary > ( SELECT   MAX (Salary)
                   FROM     EMPLOYEE
                   WHERE    Dno=5 );
```

The inner block is to retrieves the highest salary in department 5.

```
( SELECT   MAX (Salary)
  FROM     EMPLOYEE
  WHERE    Dno=5 )
```

The outer query block is:

```
SELECT  Lname, Fname
FROM    EMPLOYEE
WHERE   Salary > c
```

where c represents the result returned from the inner block

The inner block could be translated into the following extended relational algebra expression:

$$\Im_{\text{MAX Salary}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE}))$$

and the outer block into the expression

$$\pi_{\text{Lname,Fname}}(\sigma_{\text{Salary}>c}(\text{EMPLOYEE}))$$

The *query optimizer* would then choose an execution plan for each query block

Heuristics in Query Optimization

This is the 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 scanner and parser of an SQL query first generate a data structure that corresponds to an *initial query representation,* which is then optimized according to heuristic rules.

One of the main **heuristic rules** is to apply SELECT and PROJECT operations *before* applying the JOIN or other binary operations, 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.

Notation for Query Trees and Query Graphs

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 order of execution of operations *starts at the leaf nodes*, which represents the input database relations for the query, and *ends at the root node*, which represents the final operation of the query.

The execution terminates when the root node operation is executed and produces the result relation for the query.

For example , the query to retrieve For every project located in ‘Stafford’, retrieve the project number, the controlling department number, and the department manager’s last name, address, and birthdate.

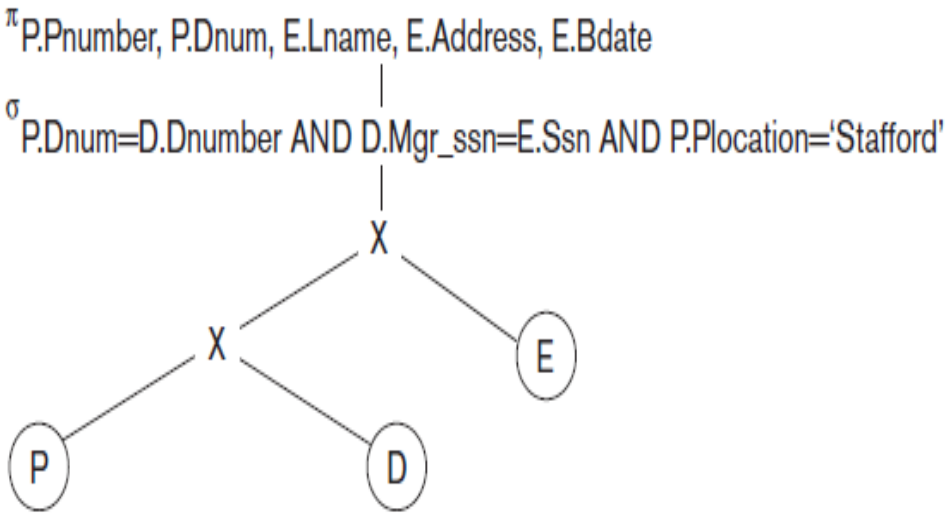
```
SELECT  P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
FROM    PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E
WHERE   P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND
        P.Plocation= ‘Stafford’;
```

The corresponding Relational Algebra expression is

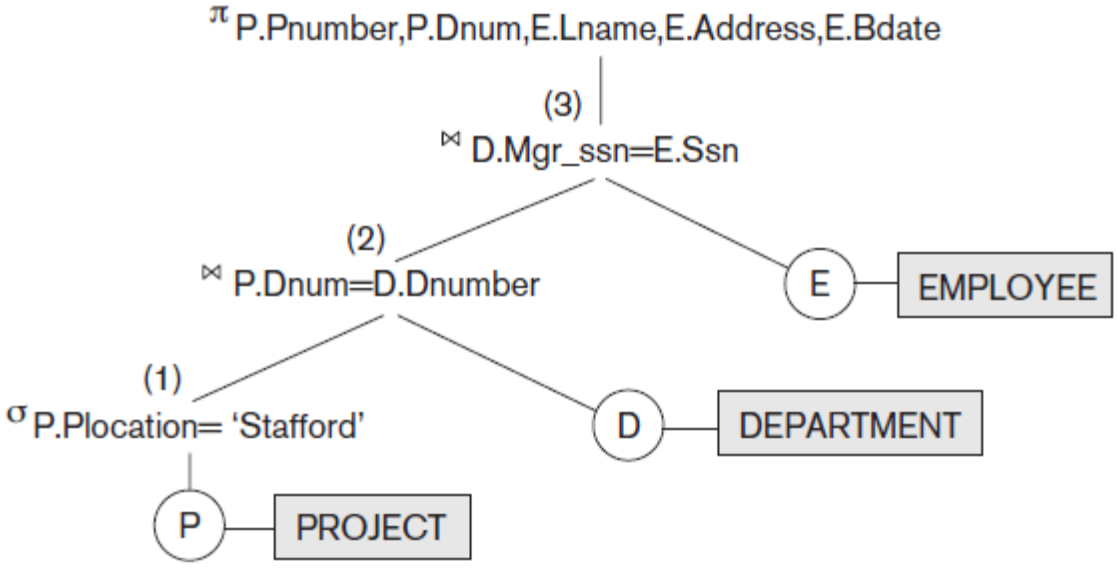
$$\pi_{Pnumber, Dnum, Lname, Address, Bdate}(((\sigma_{Plocation='Stafford'}(PROJECT)) \bowtie_{Dnum=Dnumber}(DEPARTMENT)) \bowtie_{Mgr\_ssn=Ssn}(EMPLOYEE))$$

The query tree representation is as follows

(a) Intial canonical query (corresponding to SQL representation)



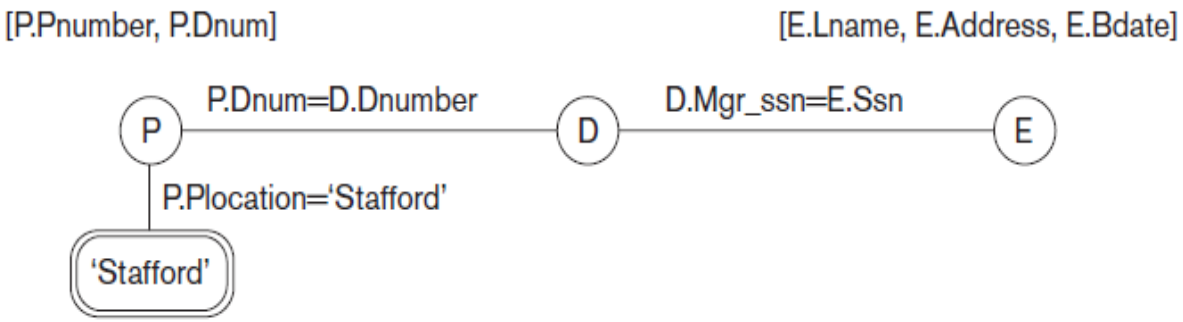
(b) Query tree corresponding to Relational Algebra expression



Query Graph

A Query graph is used to represent a relational calculus expression.

The query graph corresponding to the above query is as follows



General Transformation Rules for relational algebra operations or Heuristic Optimization Rules:

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 $\bowtie$ (and $\times$).** The join operation is commutative, as is the $\times$ 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 $\bowtie$ (or $\times$).** 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 $(c_1 \text{ AND } c_2)$, 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 $\bowtie$ is replaced by a $\times$ operation.

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

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

   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 $A_{n+1}, ..., A_{n+k}$ 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:

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

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

8. **Commutativity of set operations.** The set operations $\cup$ and $\cap$ are commutative but $-$ is not.

9. **Associativity of $\bowtie$, $\times$, $\cup$, and $\cap$.** 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 $\cup$, $\cap$, 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 $\cup$.**

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

12. **Converting a $(\sigma, \times)$ sequence into $\bowtie$.** If the condition c of a σ that follows a $\times$ corresponds to a join condition, convert the $(\sigma, \times)$ sequence into a $\bowtie$ 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 standard rules from Boolean algebra (DeMorgan's laws):

NOT $(c_1$ **AND** $c_2) \equiv$ (**NOT** c_1) **OR** (**NOT** c_2)

NOT $(c_1$ **OR** $c_2) \equiv$ (**NOT** c_1) **AND** (**NOT** c_2)

Outline of a Heuristic Algebraic Optimization Algorithm

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. If the condition involves attributes from *only one table*, which means that it represents a *selection condition*, the operation is moved all the way to the leaf node that represents this table. If the condition involves attributes from *two tables*, which means that it represents a *join condition*, the condition is moved to a location down the tree after the two tables are combined.

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 selectivity are often available in the DBMS catalog. Second, make sure that the ordering of leaf nodes do 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 subtrees that represent groups of operations that can be executed by a single algorithm.

Example of Transforming a Query

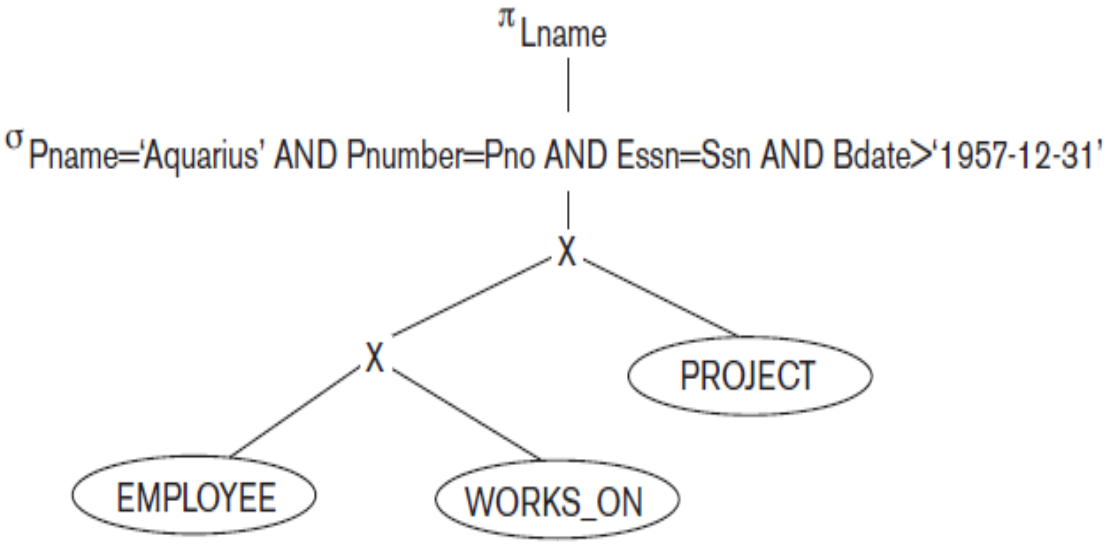
Consider the following query

Find the last names of employees born after 1957 who work on a project named 'Aquarius'

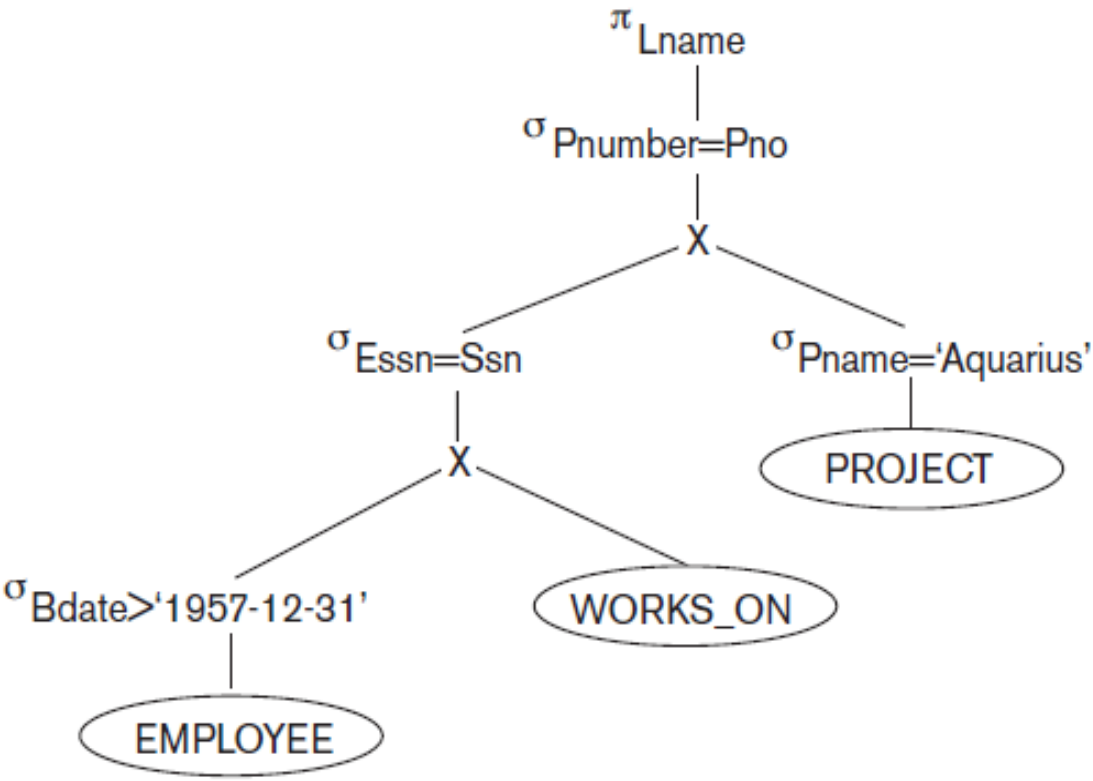
This query can be specified in SQL as follows:

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

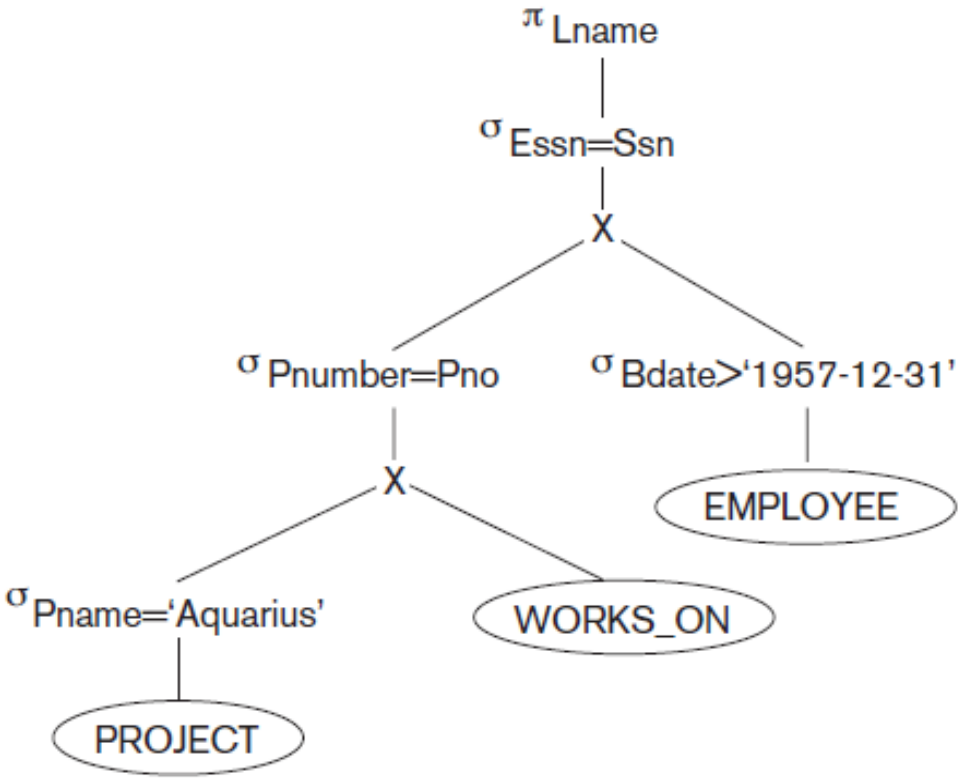
(a) Initial (canonical) query tree for SQL query



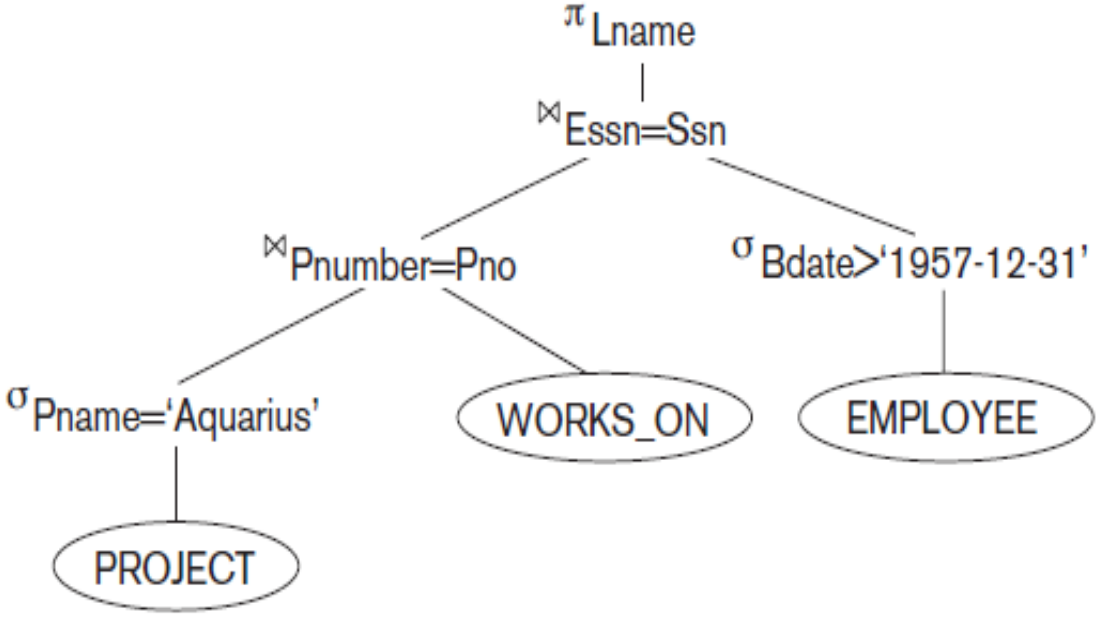
(b) Moving SELECT operations down the query tree.



(c) Applying the more restrictive SELECT operation first.



(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations



(e) Moving PROJECT operations down the query tree.