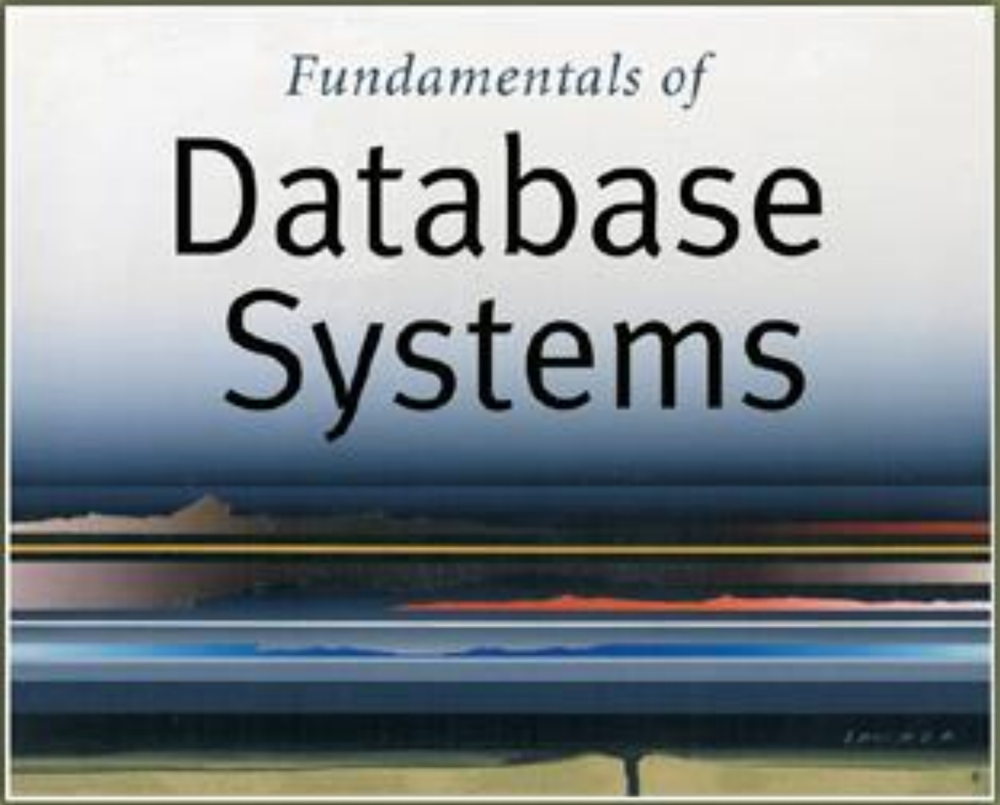

Fundamentals of

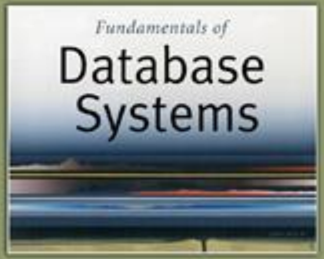
Database Systems

5th Edition

Elmasri / Navathe

Chapter 15

Algorithms for Query Processing and Optimization



5th Edition

Elmasri / Navathe

External Sorting

- **Sorting** is one of the primary algorithms used in query processing. For example,
  - whenever an SQL query specifies an **ORDER BY-clause**, the query result must be sorted.
- Sorting is also a key component in sort-merge algorithms used for **JOIN** and other operations (such as **UNION and INTERSECTION**), and in **duplicate elimination** algorithms for the **PROJECT** operation

External Sorting

- **External sorting**:
  - Refers to sorting algorithms that are suitable for **large files** of records stored on disk that do **not fit entirely in main memory**, such as most database files.
- **Sort-Merge strategy**:
  - Starts by **sorting** small subfiles (**runs**) of the main file and then **merges the sorted runs**, creating larger sorted subfiles that are merged in turn.

External Sorting

- The basic algorithm consists of **two phases**: the sorting phase and the merging phase.
  - In the **sorting phase, runs (portions or pieces) of the file that can fit in the available** buffer space are read into main memory, sorted using an ***internal sorting algorithm***, and written back to disk as temporary sorted subfiles (or runs).
  - In the **merging phase, the sorted runs are merged during one or more merge passes. Each merge pass can have one or more merge steps.**
- The buffer space is divided into individual buffers, where each **buffer is the same size in bytes as the size** of one disk block. Thus, one buffer can hold the contents of exactly *one disk block.*

Algorithms for SELECT Operation

- Implementing the SELECT Operation

- Examples:
  - (OP1): $\sigma_{SSN='123456789'}$ (EMPLOYEE)
  - (OP2): $\sigma_{DNUMBER>5}$(DEPARTMENT)
  - (OP3): $\sigma_{DNO=5}$(EMPLOYEE)
  - (OP4): $\sigma_{DNO=5 \text{ AND } SALARY>30000 \text{ AND } SEX=F}$(EMPLOYEE)
  - (OP5): $\sigma_{ESSN=123456789 \text{ AND } PNO=10}$(WORKS_ON)

- Implementing the SELECT Operation:
- Search Methods for Simple Selection:
  - S1 **Linear search** (brute force):
    - Retrieve every record in the file, and test whether its attribute values satisfy the selection condition.
  - S2 **Binary search**:
    - If the selection condition involves an **equality comparison on a key attribute on which the file is ordered**, binary search (which is more efficient than linear search) can be used. (See OP1).
  - S3 **Using a primary index to retrieve a single record**:
    - If the selection condition involves an equality comparison on a key attribute with a primary index, use the primary index to retrieve the record.

- Implementing the SELECT Operation (contd.):
- Search Methods for Simple Selection:
  - S4 **Using a primary index to retrieve multiple records**:
    - If the comparison condition is >, ≥, <, or ≤ on a key field with a primary index, use the index to find the record satisfying the corresponding equality condition, **then retrieve all subsequent records in the (ordered) file.**
  - S5 **Using a clustering index to retrieve multiple records**:
    - If the selection condition involves an equality comparison on a **non-key attribute** with a clustering index, use the clustering index to retrieve all the records satisfying the selection condition.
  - S6 **Using a secondary index**:
    - On an equality comparison, this search method can be used to retrieve a single record if the indexing field has unique values (is a key) or to retrieve multiple records if the indexing field is not a key.
    - In addition, it can be used to retrieve records on conditions involving >,>=, <, or <=. (FOR RANGE QUERIES)

- Implementing the SELECT Operation (contd.):
- Search Methods for Complex Selection (**conjunctive condition—t**hat is, if it is made up of several simple conditions connected with the **AND** logical connective):
  - S7 **Conjunctive selection using an individual index**:
    - If an attribute involved in any single simple condition in the conjunctive condition has an access path that permits the use of one of the methods S2 to S6, use that condition to retrieve the records and then check whether each retrieved record satisfies the remaining simple conditions in the conjunctive condition.
  - S8 **Conjunctive selection using a composite index**
    - If two or more attributes are involved in equality conditions in the conjunctive condition and a composite index exists on the combined field, we can use the index directly.

- Implementing the SELECT Operation (contd.):
- Search Methods for Complex Selection:
  - S9 **Conjunctive selection by intersection of record pointers**:
    - This method is possible if secondary indexes are available on all (or some of) the fields involved in equality comparison conditions in the conjunctive condition and if the indexes include record pointers (rather than block pointers).
    - Each index can be used to retrieve the record pointers that satisfy the individual condition.
    - The intersection of these sets of record pointers gives the record pointers that satisfy the conjunctive condition, which are then used to retrieve those records directly.
    - If only some of the conditions have secondary indexes, each retrieved record is further tested to determine whether it satisfies the remaining conditions.

- **Disjunctive selection conditions (where simple conditions are connected by the OR** logical connective rather than by AND) is much harder to process and optimize. A little optimization can be done, because the records satisfying the disjunctive condition are the *union of the records satisfying the individual* conditions. Hence, if any one of the conditions does not have an access path, we are compelled to use the brute force, linear search approach.
- Only if an access path exists on *every simple condition in the disjunction can we optimize the selection by* retrieving the records satisfying each condition—or their record ids—and then applying the *union operation to eliminate duplicates.*

Algorithms for JOIN Operation

- The JOIN operation is one of the most time-consuming operations in query processing.
- Join (EQUIJOIN, NATURAL JOIN)
  - two–way join: a join on two files
  - e.g. $R \bowtie_{A=B} S$
  - multi-way joins: joins involving more than two files.
  - e.g. $R \bowtie_{A=B} S \bowtie_{C=D} T$
- Examples
  - (OP6): EMPLOYEE $\bowtie_{DNO=DNUMBER}$ DEPARTMENT
  - (OP7): DEPARTMENT $\bowtie_{MGRSSN=SSN}$ EMPLOYEE

- Implementing the JOIN Operation (contd.):
- Methods for implementing joins:
  - J1 **Nested-loop join** (brute force):
    - For each record t in R (outer loop), retrieve every record s from S (inner loop) and test whether the two records satisfy the join condition t[A] = s[B].
  - J2 **Single-loop join** (Using an access structure to retrieve the matching records):
    - If an index exists for one of the two join attributes — say, B of S — retrieve each record t in R, one at a time, and then use the access structure to retrieve directly all matching records s from S that satisfy s[B] = t[A].

- Implementing the JOIN Operation (contd.):
- Methods for implementing joins:
  - J3 **Sort-merge join**:
    - If the records of R and S are ***physically sorted*** (*ordered*) by **value of the join attributes** A and B, respectively, we can implement the join in the most efficient way possible.
    - Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for A and B.
    - In this method, the records of each file are scanned only once each for matching with the other file—
    - If the files are not sorted, they may be sorted first by using external sorting

Join Selection Factor

- Join selection factor **affects the performance of a join**, particularly the single-loop method J2, is the fraction of records in one file that will be joined with records in the other file.
- Example:
- Assume that the DEPARTMENT file consists of *rD = 50 records stored in bD = 10 disk* blocks and that the EMPLOYEE file consists of *rE = 6000 records stored in bE = 2000* disk blocks.
- consider the operation OP7, which joins each DEPARTMENT record with the EMPLOYEE record for the manager of that department.
- (OP7): DEPARTMENT $\bowtie_{\text{MGRSSN=SSN}}$ EMPLOYEE
- Here, each DEPARTMENT record (there are 50 such records in our example) will be joined with a *single EMPLOYEE record, but many EMPLOYEE records (the* 5,950 of them that do not manage a department) will not be joined with any record from DEPARTMENT.

- Suppose that secondary indexes exist on both the attributes Ssn of EMPLOYEE and Mgr_ssn of DEPARTMENT, with the number of index levels *xSsn = 4 and xMgr_ssn= 2*
- We have two options for implementing method J2.
- The first retrieves each EMPLOYEE record and then uses the index on Mgr_ssn of DEPARTMENT to find a matching DEPARTMENT record.
- In this case no matching record will be found for employees who do not manage a department. The number of block accesses for this case is approximately:
- *bE + (rE * (xMgr_ssn + 1)) = 2000 + (6000 * 3) = 20,000 block accesses*

- The second option retrieves each DEPARTMENT record and then uses the index on Ssn of EMPLOYEE to find a matching manager EMPLOYEE record. In this case, every DEPARTMENT record will have one matching
- *bD + (rD * (xSsn + 1)) = 10 + (50 * 5) = 260 block accesses*

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.
  - 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.

Query Optimization

- An **internal representation** of the query is **then** created, usually as a tree data structure called a **query tree**.
- 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.**
- **Query optimization**:
  - The process of choosing a suitable execution strategy for processing a query.

Introduction to Query Processing

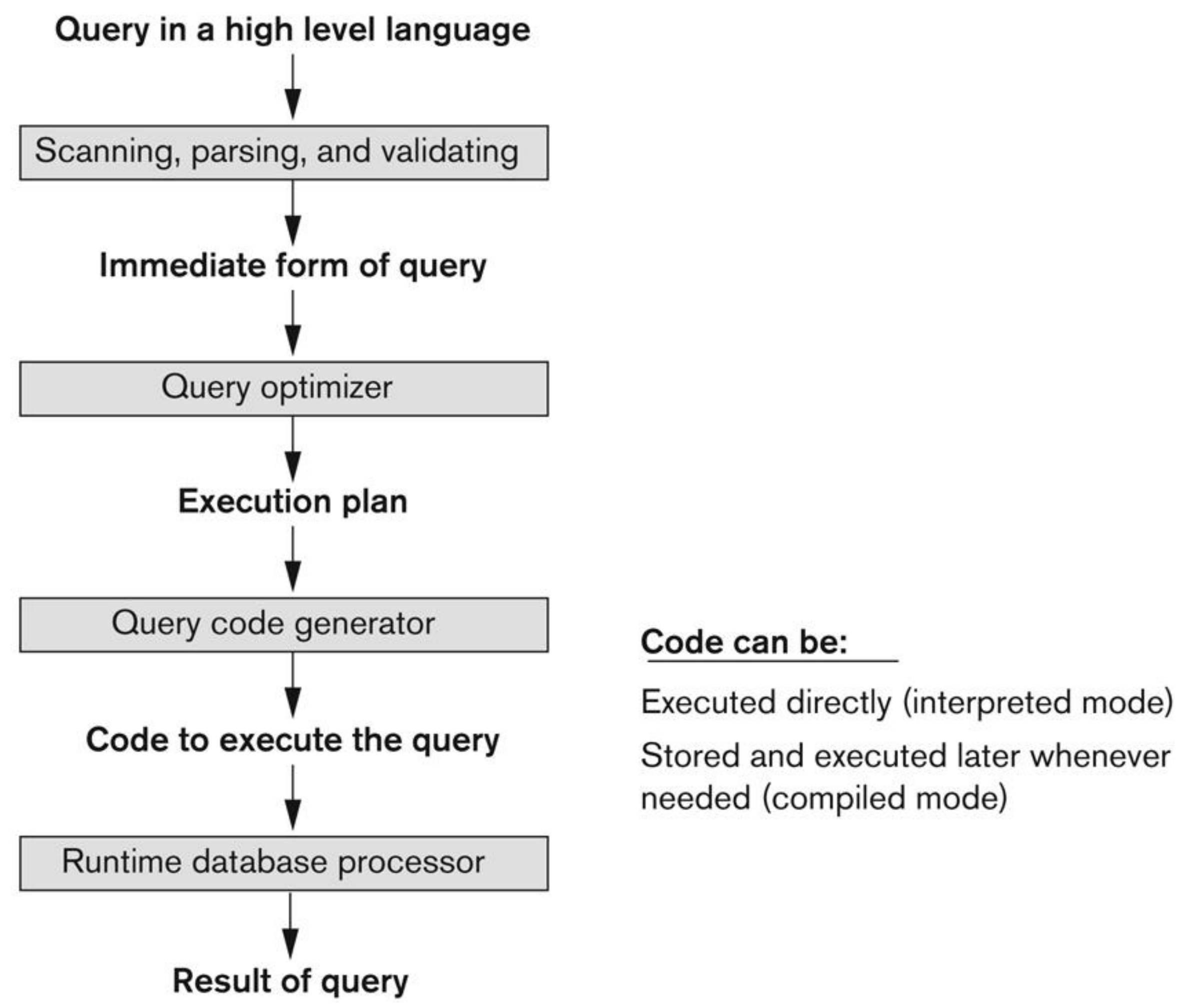

Query in a high level language
Scanning, parsing, and validating
Immediate form of query
Query optimizer
Execution plan
Query code generator
Code to execute the query
Runtime database processor
Result of query
Code can be:
Executed directly (interpreted mode)
Stored and executed later whenever needed (compiled mode)

Figure 15.1
Typical steps when processing a high-level query.

Compiled vs. Interpreted Queries

- The **optimizer 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.
- This approach is more suitable for **<u>compiled queries</u>** where the optimization is **done at compile time and the resulting execution strategy code is stored and executed directly at runtime.**
- For **<u>interpreted queries</u>,** where the entire process occurs at runtime, a full-scale optimization may slow **down the response time.**

Translating SQL Queries into Relational Algebra

- **Query block**:
  - The basic unit 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 clause if these are part of the block.
- **Nested queries** within a query are identified as separate query blocks.

Translating SQL Queries into Relational Algebra (2)

```
SELECT   LNAME, FNAME
FROM     EMPLOYEE
WHERE    SALARY > (  SELECT   MAX (SALARY)
                     FROM     EMPLOYEE
                     WHERE    DNO = 5);
```

```
SELECT   LNAME, FNAME
FROM     EMPLOYEE
WHERE    SALARY > C
```

```
SELECT   MAX (SALARY)
FROM     EMPLOYEE
WHERE    DNO = 5
```

$\pi_{\text{LNAME, FNAME}}(\sigma_{\text{SALARY>C}}(\text{EMPLOYEE}))$

$\mathcal{F}_{\text{MAX SALARY}}(\sigma_{\text{DNO=5}}(\text{EMPLOYEE}))$

- In general, a query tree gives a good **visual representation and understanding of the query in terms of the relational operations**

Example

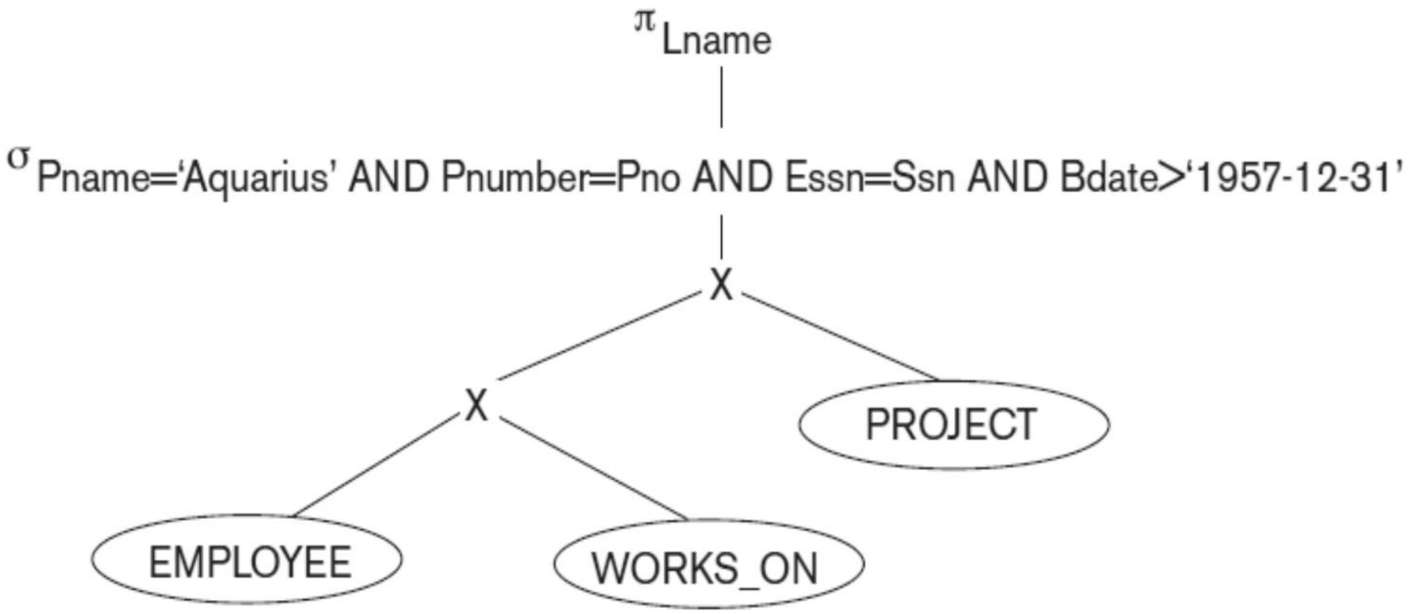
- Find the last names of employees born after 1957 who work on a project named ‘Aquarius’.

- SELECT Lname

  FROM EMPLOYEE, WORKS_ON, PROJECT

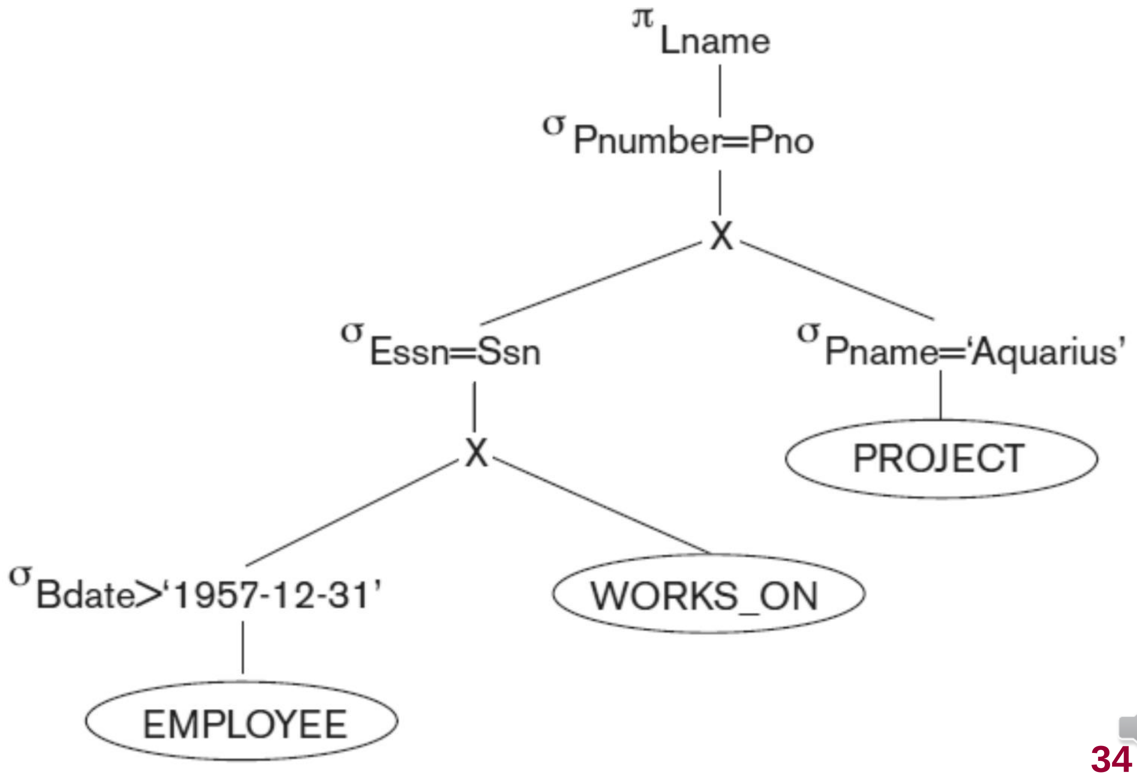
  WHERE Pname=‘Aquarius’ AND Pnumber=Pno AND Essn=Ssn AND Bdate > ‘1957-12-31’;

1- Initial Query Tree


π_{Lname}
$\sigma_{\text{Pname='Aquarius' AND Pnumber=Pno AND Essn=Ssn AND Bdate>'1957-12-31'}}$
X
X
PROJECT
EMPLOYEE
WORKS_ON

2- Improved Query Tree

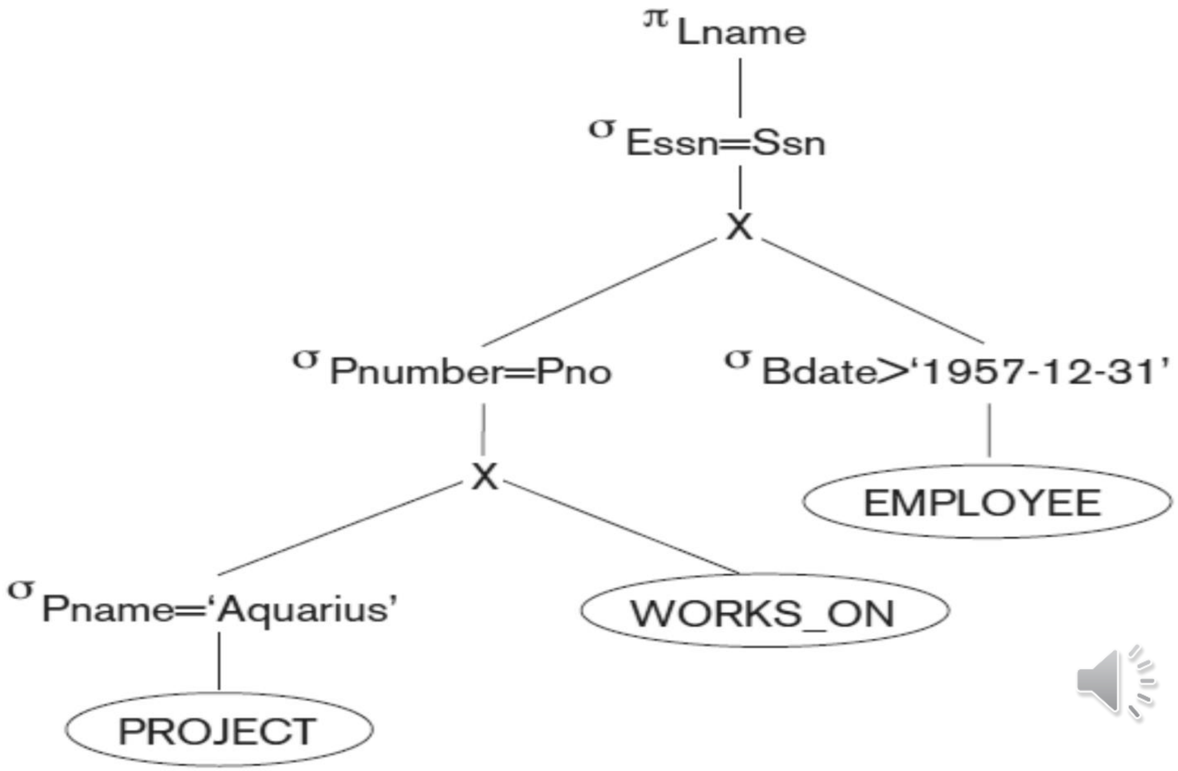
An **improved query tree** that first applies the SELECT operations to reduce the number of tuples that appear in the CARTESIAN PRODUCT



π_{Lname}
$\sigma_{\text{Pnumber=Pno}}$
X
$\sigma_{\text{Essn=Ssn}}$
$\sigma_{\text{Pname='Aquarius'}}$
X
PROJECT
$\sigma_{\text{Bdate>'1957-12-31'}}$
WORKS_ON
EMPLOYEE

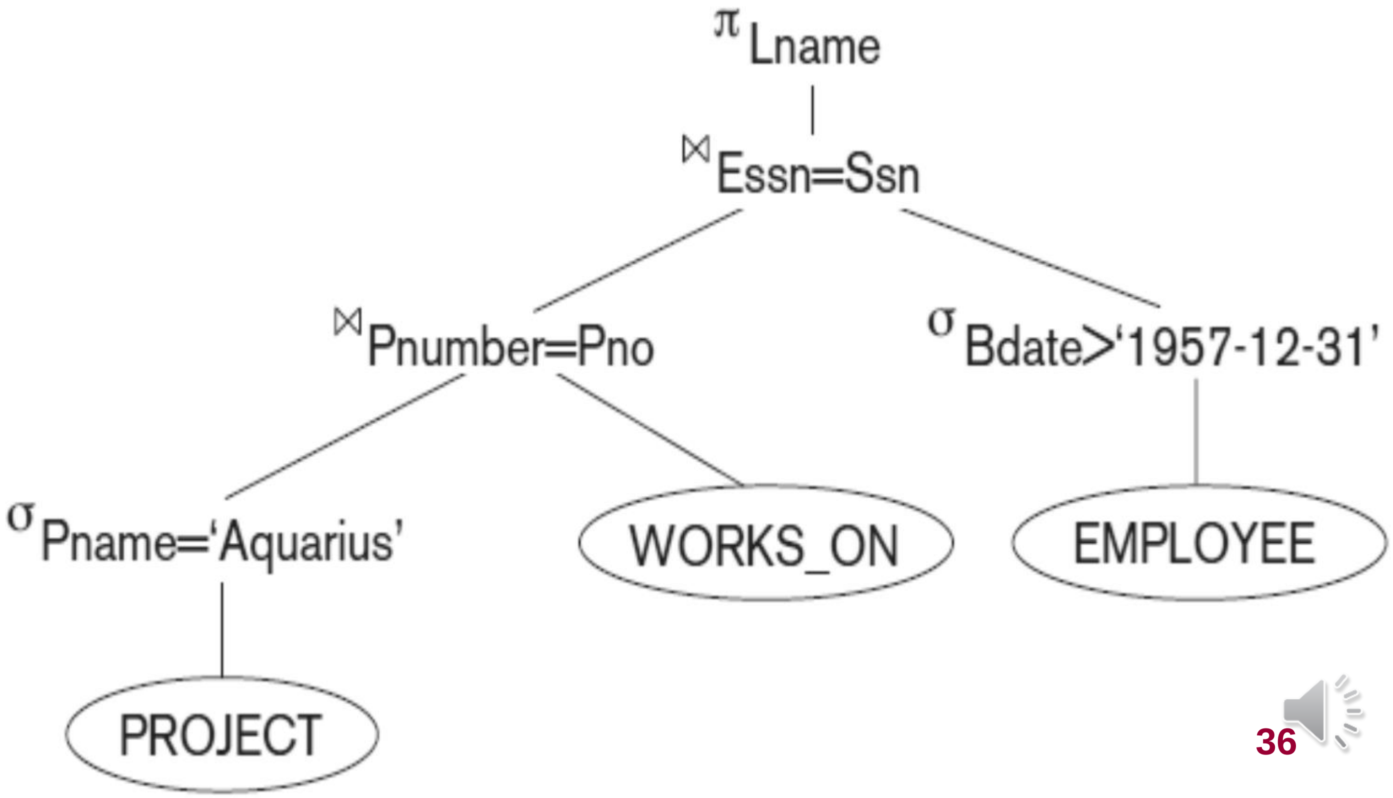
3- Improved Query Tree

A further improvement is achieved by **switching the positions** of the EMPLOYEE and PROJECT relations in the tree:



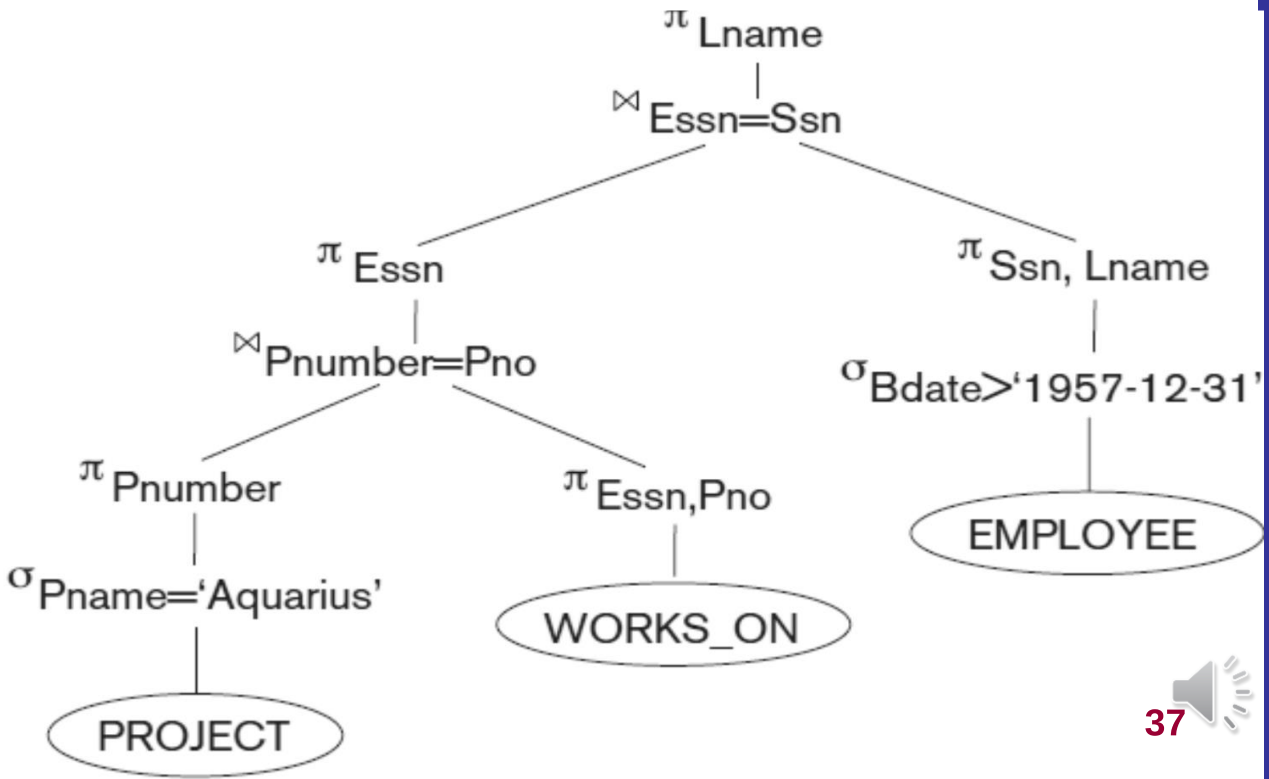
4- Improved Query Tree

We can further improve the query tree by **replacing any CARTESIAN PRODUCT operation** that is followed by a join condition **with a JOIN operation**:



5- Improved Query Tree

Another improvement is to **keep only the attributes needed** by subsequent operations in the intermediate relations, by **including PROJECT (π) operations as early as possible in the query tree**:



Rules

- 1. **Break up any SE**LECT 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. **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.
- 3. **Rearrange the leaf nodes** of the tree, 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**.

- 4. 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. **Break down and move lists of projection attributes down** the tree as far as possible by creating new PROJECT operations as needed.

- Summary of Heuristics for Algebraic Optimization:
  1. The main heuristic is to apply first the operations that reduce the size of intermediate results.
  2. Perform select operations as early as possible to reduce the number of tuples and perform project operations as early as possible to reduce the number of attributes. (This is done by moving select and project operations as far down the tree as possible.)
  3. The select and join operations that are most restrictive should be executed before other similar operations. (This is done by reordering the leaf nodes of the tree among themselves and adjusting the rest of the tree appropriately.)

Using Cost Estimates in Query Optimization

- **Cost-based query optimization**:
  - Estimate and compare the costs of executing a query using different execution strategies and choose the strategy with the lowest cost estimate.
  - (Compare to heuristic query optimization)

- Issues
  - Cost function
  - Number of execution strategies to be considered

- Cost Components for Query Execution
  1. Access cost to secondary storage
  2. Computation cost
  3. Communication cost

- Note: Different database systems may focus on different cost components.
  - Large vs. small dbs and distributed dbs

Catalog Information Used in Cost Functions

- Information about the size of a file
  - number of records (tuples) (r),
  - record size (R),
  - number of blocks (b)
  - blocking factor (bfr)
- Information about indexes and indexing attributes of a file
  - Number of levels (x) of each multilevel index
  - Number of first-level index blocks (bI1)
  - Number of distinct values (d) of an attribute
  - Selectivity (sl) of an attribute
  - Selection cardinality (s) of an attribute. (s = sl * r)

Selectivity

- **Selectivity** (sl) of an attribute: which **is the fraction of records satisfying an equality condition on the attribute.**
- The **selectivity (*sl*) *is defined as the ratio of the number*** of records (tuples) that satisfy the condition to the total number of records (tuples) in the file (relation), and thus is a number between zero and one.
- *Zero selectivity* means none of the records in the file satisfies the selection condition, and a selectivity of one means that all the records in the file satisfy the condition.

Selection Cardinality

- **Selection cardinality (s)** of an attribute. (s = sl * r) **the average number of records that will satisfy an equality selection condition on that attribute**. The smaller this estimate is, the higher the desirability of using that condition first to retrieve records.
- 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).*

- In certain cases, the actual distribution of records among the various distinct values of the attribute is kept by the DBMS in the form of a ***histogram****, in order to get more accurate estimates* of the number of records that satisfy a particular condition.
- For a nonkey attribute with *d distinct values, it is often the case that the records are* not uniformly distributed among these values.
- For example, suppose that a company has 5 departments numbered 1 through 5, and 200 employees who are distributed among the departments as follows: (1, 5), (2, 25), (3, 70), (4, 40), (5, 60). In such cases, the optimizer can store a **histogram that reflects the distribution of** employee records over different departments in a table with the two attributes (Dno, Selectivity), which would contain the following values for our example: (1, 0.025), (2,0.125), (3, 0.35), (4, 0.2), (5, 0.3).