Overall Process

- Scanning: Identify query keywords, tokens
- Parsing: Verify syntax
- Validation: Verify all referenced relations etc. exist
- Query tree or DAG created
- Query optimiser: Execution plan devised
- Code generator: Low level code for execution plan generated
- Runtime database processor: Low level code executed

Decomposition into Blocks

- Query decomposed into basic SELECT-FROM-WHERE blocks
- May have HAVING/GROUP BY
- These units -> translated into relational algebra equivalent

Example :

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

Inner Block :

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

Outer Block :

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

Inner Block translated into :

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

Outer Block translated into :

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

Query optimiser chooses execution plan **for each block.**

Semi-join, Anti-join

- Exists to be more computationally efficient than doing an entire inner join

Semi

T1.x S= T2.y

- Returns the first row of T1 that matches T1.x and T2.y

Anti

T1.x A= T2.y

- Returns the first row of T1 that does not match T1.x and T2.y

Implementing SELECT

Linear Search - Brute Force

Load every disk block one by one into main mem -> search through every record in that disk block and check condition

Binary Search

- Cond: Equality on key attribute using which the file is ordered
- More efficient than linear search

Primary Index

- Cond: Equality on key attribute that is used as primary index for file contents
- Find index -> get pointer to relevant row
- Retrieves single record at most

Hash Key

- Cond: Equality on key attribute with hash key

- H(key_val) -> direct index
- Retrieves single record at most

Primary Index for Multiple Records

- Cond: <, >, <=, >= on key attribute that is used as primary index for file contents
- Find index -> relevant row
- Select all subsequent/preceding rows as well depending on operation

Implementing JOIN

Two-way / multiway joins

Nested Loop Join (Brute Force)

- Manually check against every possible value in the inner loop
  `Student.Did = Dept.Did`
  For every record in Student:
  Go through every record in Department:
  if Student.Did == Department.Did:
  return joined record

Index-Based Nested Loop Join

- Use an index to look up value in inner loop instead of searching
- Suppose index exists on Department.Did
  For every record in Student:
  matched_dept = index_lookup(Student.record.DiD)
  for each record in matched_dept:
  return joined record

Sort-Merge Join

- Join on attributes on which tables are ordered
- Merge = scan both tables -> compare ID (or whatever attr) sequentially -> merge records with matching IDs and output

Partition-Hash Join

- Two phases: partitioning and probing
- Hashing done using same hashing function on both tables
- Partitioning:
  - Single pass on the smaller table
  - Collection of records with the same value of H(A) -> same hash bucket/partition
  - We assume that after partitioning, every individual bucket can fit into main memory
- Probing:
  - Pass on other table and hash value
  - Probe the hash bucket corresponding to this hash value

- Combine record with matching records from the partition

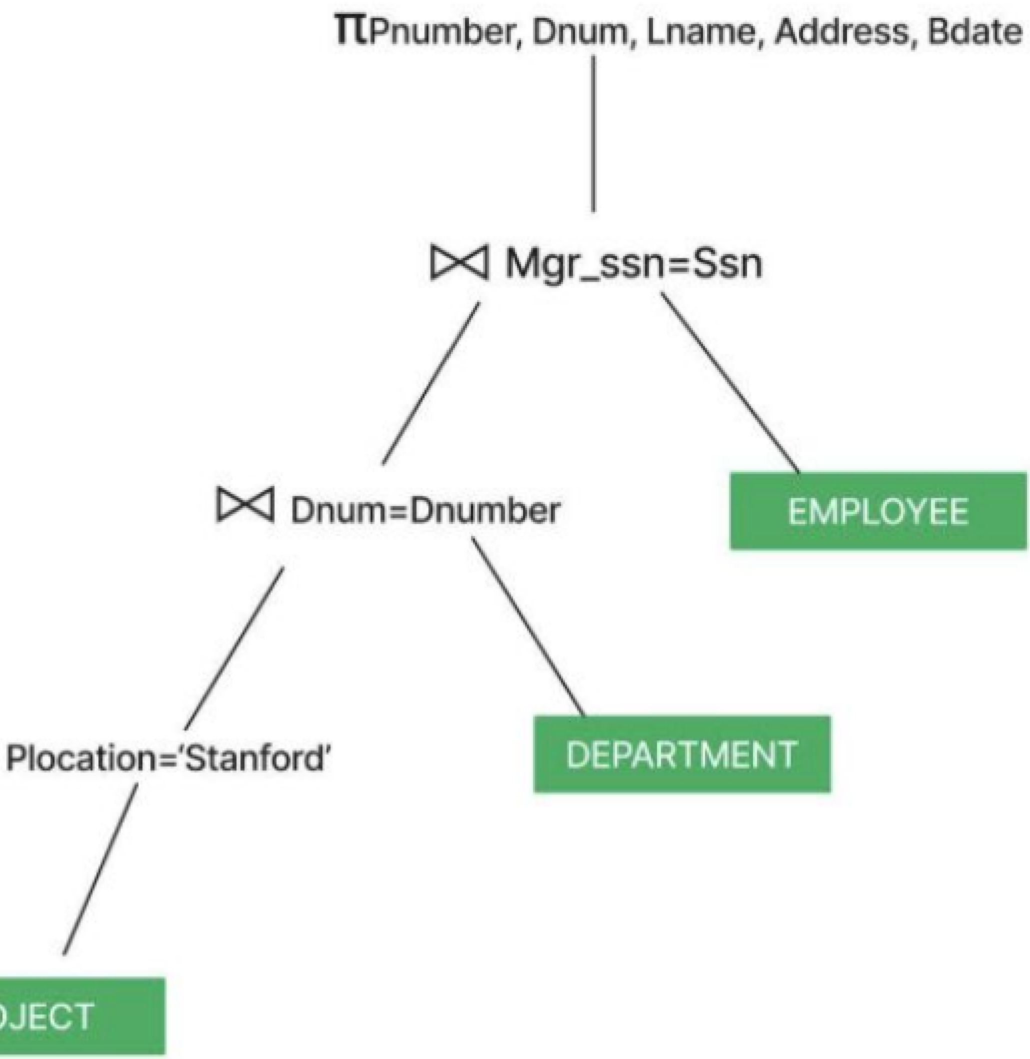
Query Optimisation

High level SQL query -> query tree/graph -> heuristics to remove unnecessary operations -> calculate cost of each operation -> cost model compares plans -> finally selected plan is executed

Query Tree

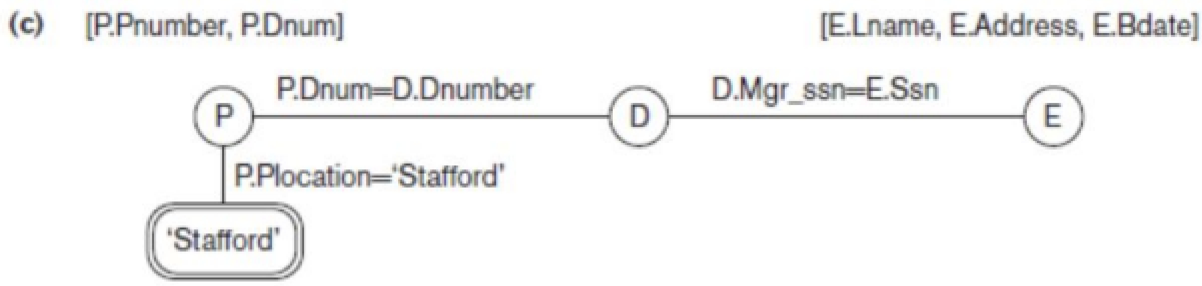
- Also called Query Evaluation/Execution Tree
- Leaf nodes = relations
- Internal nodes = relational algebra operators
- Root node = output

```
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';
```



Query Graphs

- Query tree preferred since it shows a definite order
- Relations = Circles
- Constants = Double circles
- Select/Join conditions = edges
- Attributes to be retrieved = written in square brackets



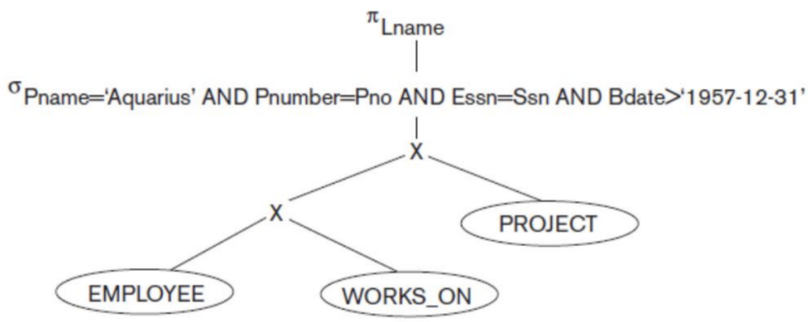
Query Transformation

- Use heuristics to optimise query execution

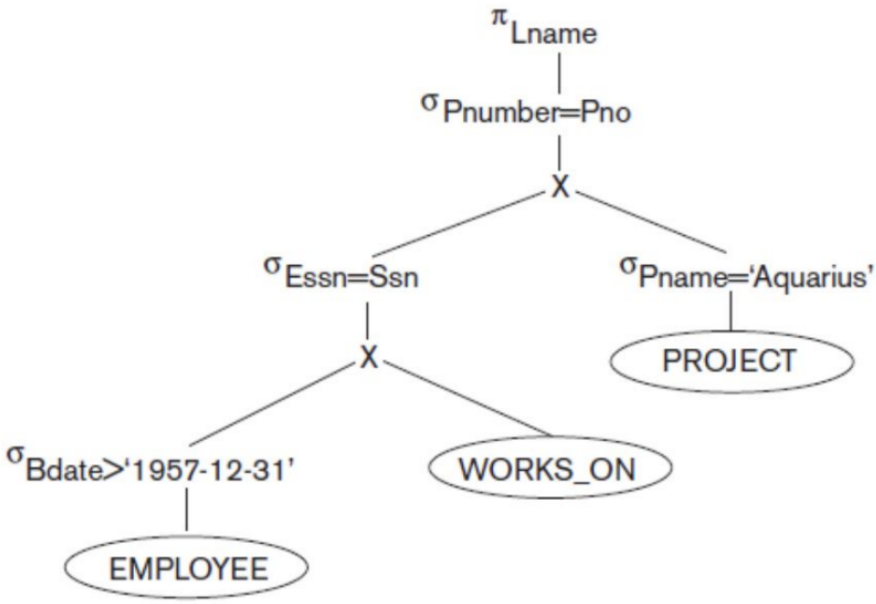
Consider the query Q

```
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';
```

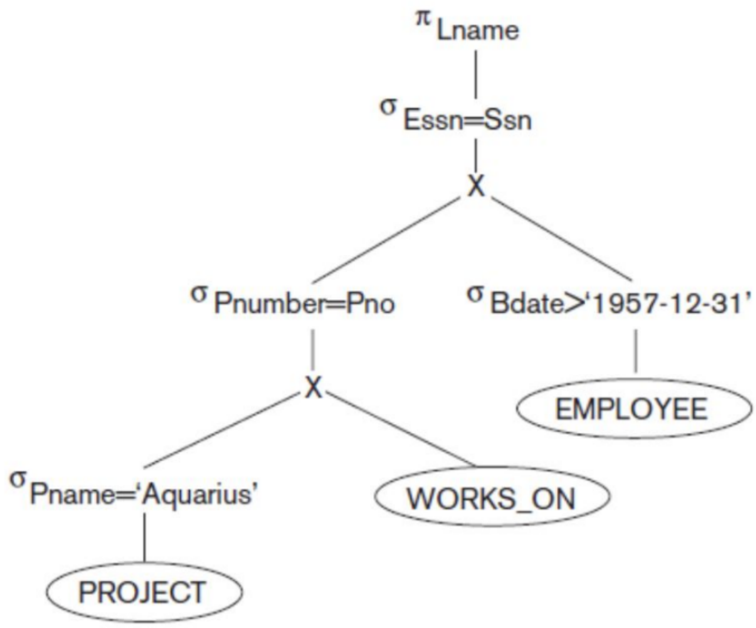
Step 1. Initial (canonical) query tree for SQL query Q.



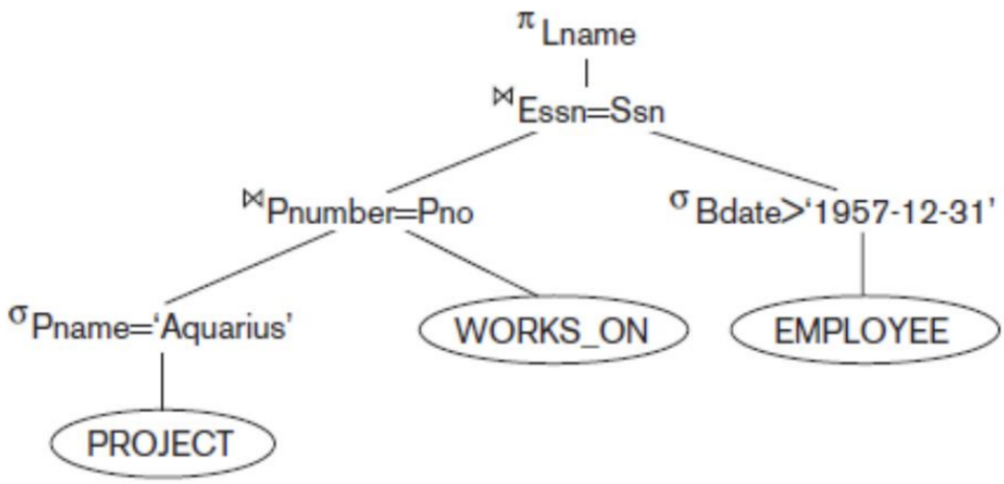
Step 2. Moving SELECT operations down the query tree.



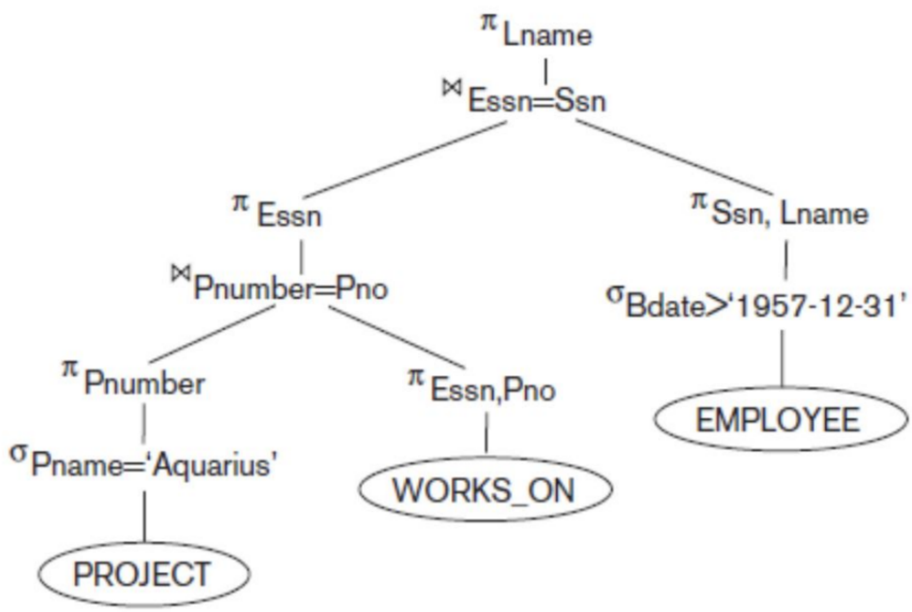
Step 3. Applying the more restrictive SELECT operation first.



Step 4. Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.



Step 5. Moving PROJECT operations down the query tree.



Types of Query Optimisation

1. Heuristics Based

- Defined rules used to refine the query tree structure
  - Execute SELECT and PROJECT early
  - Replace Cartesian product + SELECT with JOIN
  - Execute more restrictive filters first, etc.
- No cost estimation, only heuristics

2. Cost-Based

- Calculate a cost for each query execution plan (estimate)
- Pick the one with the lowest one
- Use statistics from the DBMS catalog: total relations, file size, histogram etc

3. Semantic Query Optimisation

- Rewrite query to be more efficient
- Utilise semantic information derived from schema and integrity constraints
- Uses schema rules/constraints to eliminate redundant operations

4. Dynamic Query Optimisation

- Changes plan at execution
- Adjusts join order, access paths, memory usage dynamically
- Uses runtime feedback loops to refine plans

Indexing

```
CREATE INDEX idx_salary ON Employee(salary);
EXPLAIN SELECT * FROM Employee WHERE salary > 50000;
```

```
match(p:Movie) where p.name="Movie3" detach delete p
```