❖ Introduction :-

- In this chapter we discuss the techniques used by a DBMS to process, optimize and execute high-level queries.
- This chapter discus the techniques used to split complex queries into multiple simple operations and methods of implementing these low-level operations.
- The query optimization techniques are used to chose an efficient execution plan that will minimize the runtime as well as many other types of resources such as number of disk I/O, CPU time and so on.

❖ Query Processing :-

- Query Processing is a **procedure of transforming a high-level** query (such as SQL) **into a correct and efficient execution plan** expressed in low-level language.
- A query processing **select a most appropriate plan** that is used in responding to a database request.
- When a database system receives a query for update or retrieval of information, it goes through a series of compilation steps, called ***execution plan***.
- In the **first phase** called **syntax checking phase**, the system parses the query and checks that it follows the syntax rules or not.
- It then matches the objects in the query syntax with the view tables and columns listed in the system table.
- Finally it performs the appropriate query modification. During this phase the system validates the user privileges and that the query does not disobey any integrity rules.
- The execution plan is finally execute to generate a response.
- So query processing is a **stepwise process**.
- The user gives a query request, which may be in **QBE** or other form. This is **first transformed into a standard high-level query language**, such as SQL.
- This SQL query is read by syntax analyzer so that it can be **check for correctness**. At this step the syntax **analyzer use the grammar** of SQL as input and the **parser** portion of the query processor check the syntax and verify whether the relation and attributes of the requested query are defined in database. At this stage the SQL query is translated in to an algebraic expression using various rules.
- **So that the process of transforming a high-level SQL query into a relational algebraic form is called Query Decomposition.**
- **The relational algebraic expression now passes to the query optimizer.** Here optimization is performed by substituting equivalent expression depends on the factors such that the existence of certain database structures, whether or not a given file is stored, the presence of different indexes & so on.
- Query optimization module work in tandem with the join manager module to improve the order in which joins are performed.

- At this stage the cost model and several other estimation formulas are used to rewrite the query.
- The modified query is written to utilize system resources so as to bring the optimal performance.
- The query optimizer then generates an action plan also called a execution plan.
- This action plans are converted into a query codes that are finally executed by a run time database processor.
- The run time database processor estimate the cost of each action plan and chose the optimal one for the execution.

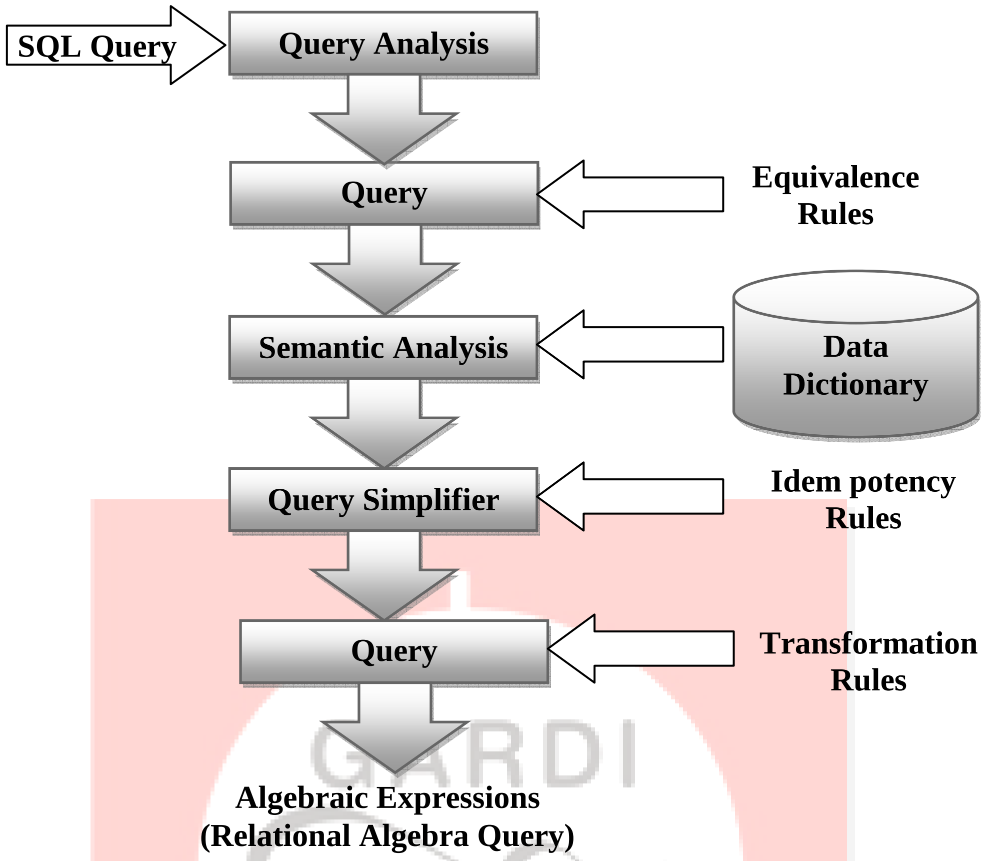
Query Analyzer

- The syntax analyzer takes the query from the users, parses it into tokens and analyses the tokens and their order to make sure they follow the rules of the language grammar.
- Is an error is found in the query submitted by the user, it is rejected and an error code together with an explanation of why the query was rejected is return to the user.
- A simple form of the language grammar that could use to implement SQL statement is given bellow :

QUERY	= SELECT + FROM + WHERE
SELECT	= 'SELECT' + <CLOUMN LIST>
FROM	= 'FROM' + <TABLE LIST>
WHERE	= 'WHERE' + VALUE1 OP VALUE2
VALUE1	= VALUE / COLUMN NAME
VALUE2	= VALUE / COLUMN NAME
OP	= >, <, >=, <=, =, <>

Query Decomposition

- The query decomposition is the first phase of the query processing whose aims are to transfer the high-level query into a relational algebra query and to check whether that query is syntactically and semantically correct.
- Thus the query decomposition is start with a high-level query and transform into query graph of low-level operations, which satisfy the query.
- The SQL query is decomposed into query blocks (low-level operations), which form the basic unit.
- Hence nested queries within a query are identified as separate query blocks.
- The query decomposer goes through five stages of processing for decomposition into low-level operation and translation into algebraic expressions.



1) Query Analysis :-

- During the query analysis phase, the query is syntactically analyzed using the programming language compiler (parser).
- A syntactically legal query is then validated, using the system catalog, to ensure that all data objects (relations and attributes) referred to by the query are defined in the database.
- The type specification of the query qualifiers and result is also checked at this stage.
- Let us consider the following query :
  SELECT emp_nm FROM EMPLOYEE WHERE emp_desg>100
- This query will be rejected because the comparison ">100" is incompatible with the data type of emp_desg which is a variable character string.

❖ QUERY TREE NOTATIONS :-

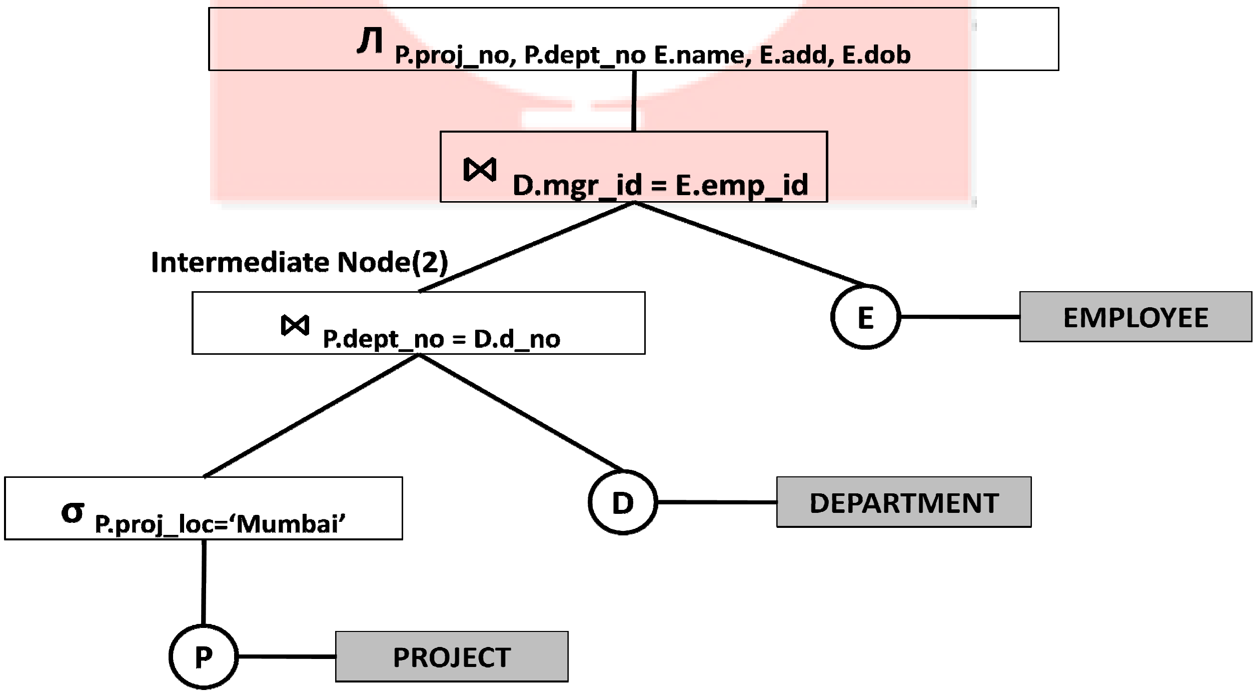
- At the end of query analysis phase, the high-level query (SQL) is transformed into some internal representation that is more suitable for processing. This internal representation is typically a kind of query tree.

- A Query Tree is a tree data structure that corresponds to a relational algebra expression.
- A Query Tree is also called a relational algebra tree.
    - **Leaf node** of the tree, representing the base input relations of the query.
    - **Internal nodes** of the tree representing an intermediate relation which is the result of applying an operation in the algebra.
    - ***Root*** of the tree representing a result of the query.
    - ***Sequence*** of operations is directed from ***leaf to root***.
- Query tree is executed by executing an **internal node** operation.
- The internal is then replaced by the resulting relation.
- The execution is terminated when the root node is executed and produces a result relation for the query.

- **Example :-**

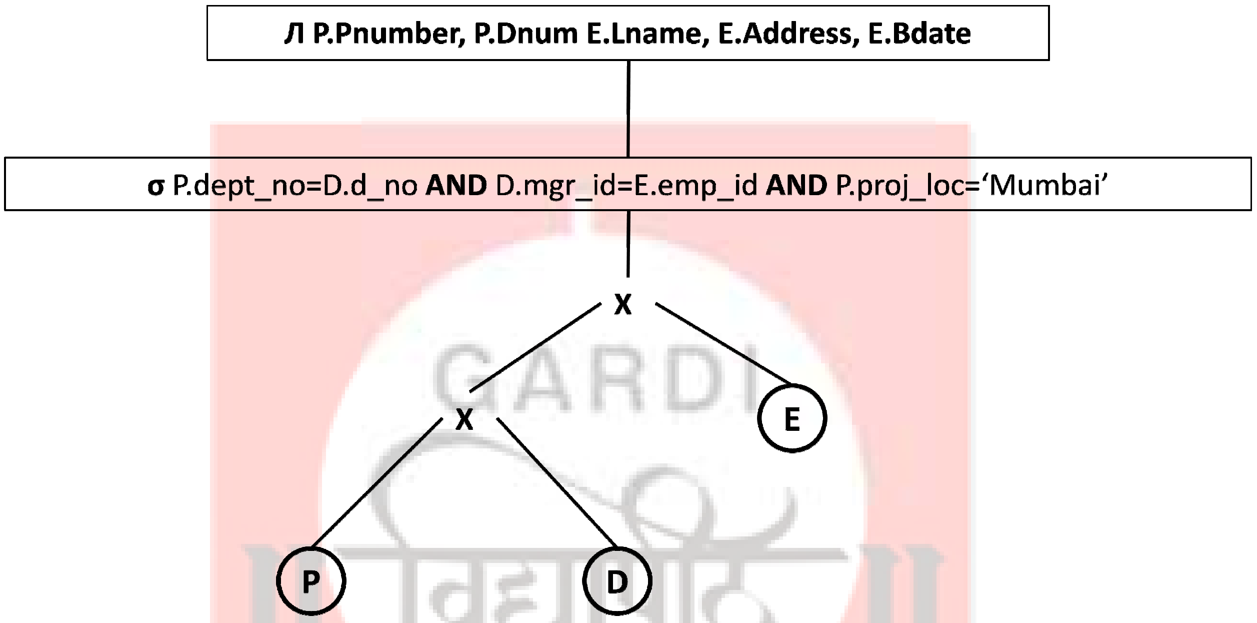
```
SELECT   (P.proj_no, P.dept_no, E.name, E.add, E.dob)
FROM     PROJECT P, DEPARTMENT D, EMPLOYEE E
WHERE    P.dept_no = D.d_no AND D.mgr_id = E.emp_id
         AND P.proj_loc = 'Mumbai' ;
```

Mumbai-PROJ $\leftarrow \sigma_{\text{proj\_loc}}$ = ‘Mumbai’ (PROJECT)
CONT-DEPT $\leftarrow$ (Mumbai-PROJ $\bowtie_{\text{dept\_no = d\_no}}$ DEPARTMENT)
PROJ-DEPT-MGR$\leftarrow$(CONT-DEPT $\bowtie_{\text{mgr\_id=emp\_id}}$ EMPLOYEE)
RESULT$\leftarrow\Pi_{\text{proj_no, dept_no, name, add, dob}}$ (PROJ-DEPT-MGR)



- The three relations PROJECT, DEPARTMENT, EMPLOYEE are represent as a leaf nodes P, D and E, while the relational algebra operations of the expression are represented by internal tree nodes.
- Same SQL query can have man different relational algebra expressions and hence many different query trees.
- The query parser typically generates a standard initial (canonical) query tree.

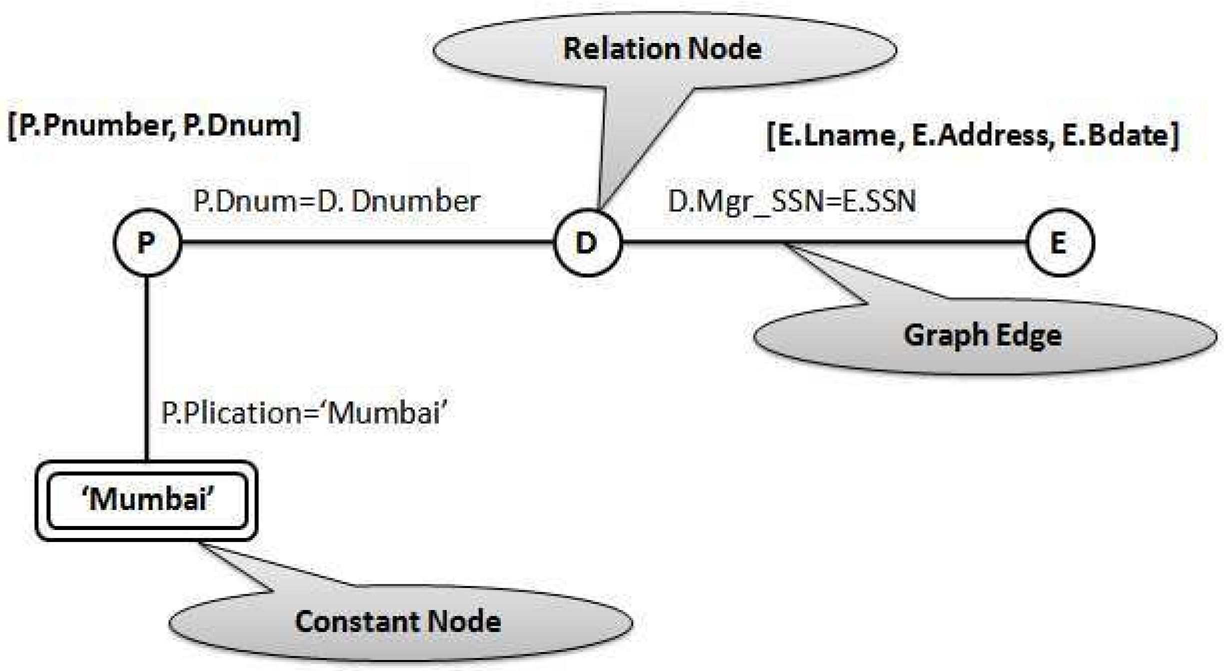
Initial (Canonical) Query Tree


Л P.Pnumber, P.Dnum E.Lname, E.Address, E.Bdate
σ P.dept_no=D.d_no AND D.mgr_id=E.emp_id AND P.proj_loc='Mumbai'
X
X
E
P
D

❖ QUERY GRAPH NOTATIONS :-

- Query graph is sometimes also used for representation of a query. In query graph representation, the relations in the query are represented as relation nodes. These **relation nodes** are displayed as a **single circle**.
- The constant values from the query selection (proj_loc = 'Mumbai') are represented by **constant node**, displayed as a **double circle**.
- The **selection and join conditions** are represented as a **graph edge** (e.g. P.dept_no = p.dept_num).
- Finally the attributes retrieve from the relation are displays in square brackets (P.proj_num, P.dept_no).

❑ Tree Graph Representation for given SQL Query :-



➢ **Disadvantages of Query Graph :-**

- It does not indicate an order on which operations to be perform first, as is the case with query tree.
- Therefore a query tree representation is preferred over the query graph in practice.
- There is only one graph corresponding to each query.
- Query tree and query graph notations are used as the basis for the data structures that are used for internal representation of query.

2) Query Normalization :-

➢ The primary phase of the normalization is to avoid redundancy. The normalization phase converts the query into a normalized form that can be more easily manipulated.

➢ In the normalization phase, a set of equivalency rules are applied so that the projection and selection operations included on the query are simplified to avoid redundancy.

➢ The projection operation corresponds to the SELECT clause of SQL query and the selection operation correspond to the predicate found in WHERE clause.

- The equivalency transformation rules that are applied to SQL query is shown in following table, in which UNARYOP means UNARY operation, BINOP means BINARY operation and REL1, REL2, REL3 are the relations.

	Rule Name	**Rule Description**
1.	Commutative of UNARY operation	UNARYOP1 UNARYOP2 REL ↔ UNARYOP2 UNARYOP1 REL
2.	Commutative of BINARY operation	REL1 BINOP (REL2 BINOP REL3) ↔ (REL1 BINOP REL2) BINOP REL3
3.	Idem potency of UNARY operations UNARYOP1 UNARYOP2 REL	UNARYOP REL
4.	Distributivity of UNARY operations	UNARYOP1 (REL1 BINOP REL2) ↔ UNARYOP (REL1) BINOP UNARYOP (REL2)
5.	Factorization of UNARY operations	UNARIOP (REL1) BINOP UNARYOP (REL2) ↔ UNARYOP (REL1 BINOP REL2)

- By applying these equivalency rules, the normalization phase rewrite the query into a normal form which can be easily manipulate in later steps.
- The **predicates** are converted into one of the following two normal forms :
  - Conjunctive Normal Form.
  - Disjunctive Normal Form.
- ***Conjunctive Normal Form*** is a sequence of conjuncts that are connected with the ‘AND’ (∧) operator.
- A conjunctive selection contains only those tuples that satisfy all conjuncts.
- **For example :-**

(emp_desig=‘Programmer’ ∨ emp_salary > 1200)
∧
Location = ‘Mumbai’

- ***Disjunctive Normal Form*** is a sequence of disjuncts that are connected with the OR (∨) operator. Each disjuncts contains one or more terms connected by the AND (∧) operator.
- A disjunctive selection contains those tuples formed by the union of all tuples that satisfied the disjuncts.
- **For example :-**

(emp_desig = ‘Programmer’ ∧ Location = ‘Mumbai)
∨
(emp_salary > 40000 ∧ Location = ‘Mumbai’)

- Disjunctive Normal Form is most often used, as it allows the query to be broken into a series of independent sub queries linked by union.

3) Semantic Analyzer :-

- The objective of this phase of query processing is to reduce the number of predicates.
- The semantic analyzer rejects the normalized queries that are incorrectly formulated.
- A **query is incorrectly formulated** if components do not contribute to the generation of result. This happens in case of missing join specification.
- A **query is contradictory** if its predicate cannot satisfy by any tuple in the relation.
- The semantic analyzer examine the relational calculus query (SQL) to make sure it contains only data objects that is table, columns, views, indexes that are defined in the database catalog.
- It makes sure that each object in the query is referenced correctly according to its data type.
- In case of missing join specifications the components do not contribute to the generation of the results, and thus, a query may be incorrect formulated.
- A query is contradictory if its predicates cannot be satisfied by any of the tuple.
- **For example :-**
  (emp_des = ‘Programmer’ $\wedge$ emp_des = ‘Analyst’)
- As an employee cannot be both ‘Programmer’ and ‘Analyst’ simultaneously, the above predicate on the EMPLOYEE relation is contradictory.

- **Example of Correctness and Contradictory :-**

- **Incorrect Formulated :-** (Missing Join Specification)

```
SELECT  p.projno, p.proj_location
FROM     project p, viewing v, department d
WHERE   d.dept_id = v.dept_id AND d.max_budj >= 8000
                    AND d.mgr = ‘Methew’
```

- **Contradictory :-** (Predicate cannot satisfy)

```
SELECT   p.proj_no, p.proj_location
FROM     project p, cost_proj c, department d
WHERE  d.max_budj >80000 AND d.max_budj < 50000 AND
                    d.dept_id = v.dept_id AND v.proj_no = p.proj_no
```

4) Query Simplifier :-

- The objectives of this phase are to detect redundant qualification, eliminate common sub-expressions and transform sub-graph to semantically equivalent but more easy and efficiently computed form.

- **Why to simplify? :-**
  - Commonly integrity constraints, view definitions and access restrictions are introduced into the graph at this stage of analysis so that the query must be simplified as much as possible.
  - Integrity constraints defines constants which must holds for all state of database, so any query that contradict an integrity constraints must be avoid and can be rejected without accessing the database.
- The final form of simplification is obtain by applying **idem potency rules** of Boolean algebra.

	Description	**Rule Format**
1.	**PRED** AND **PRED** = **PRED**	P ∧ (P) = P
2.	**PRED** AND TRUE = **PRED**	P ∨ TRUE = P
3.	**PRED** AND FALSE = FALSE	P ∧ FALSE = FALSE
4.	**PRED** AND NOT(**PRED**) = FALSE	P ∧ (~P) = FALSE
5.	**PRED1** AND **(PRED1** OR **PRED2) = PRED1**	P1 ∧ (P1 ∨ P2) = P1
6.	**PRED** OR **PRED** = **PRED**	P ∨ (P) = P
7.	**PRED** OR TRUE = TRUE	P ∨ TRUE = TRUE
8.	**PRED** OR FALSE = **PRED**	P ∨ FALSE = P
9.	**PRED** OR NOT(**PRED**) = TRUE	P ∨ (~P) = TRUE
10.	**PRED1** OR (**PRED1** AND **PRED2**) = **PRED1**	P1 ∨ (P1 ∧ P2) = P1

SELECT D.DEPT_ID, M.BRANCH_MGR, M.BRANCH_ID, B.BRANCH_ID, B.BRANCH_LOC, E.EMP_NAME, E.EMP_SALARY
FROM DEPARTMENT **AS** D, MANAGER **AS** M, BRANCH **AS** B
WHERE D.DEPT_IT = M.DEPT_ID **AND** M.BRANCH_ID = B.BRANCH_ID
AND M.BRANCH_MGR = E.EMPID **AND** B.BRANCH_LOC = ‘Mumbai’
AND NOT (B.BRANCH_LOC = ‘Mumbai’ **AND** B.BRANCH_LOC = ‘Delhi’**)**
AND B.PROFIT_TO_DATE > 1000000 **AND** E.EMP_SALARY >85000
AND NOT (BRANCH_LOC = ‘Delhi’**) AND** D.DEPT_LOC = ‘Bangalore’

Let us examine the following part of the above query

AND B.BRANCH_LOC = ‘Mumbai’
AND NOT (B.BRANCH_LOC = ‘Mumbai’ **AND** B.BRANCH_LOC = ‘Delhi’)

In above query statements, let us equate as follows:

B.BRANCH_LOC = ‘Mumbai’ = PRED1
B.BRANCH_LOC = ‘Mumbai’ = PRED2
B.BRANCH_LOC = ‘Delhi’ = PRED3

Now the above part of the query can be represented in the form of idem potency rules of Boolean algebra as follows:

PRED1 **AND NOT** (PRED2 **AND** PRED3) = P1 ∧ ~ (P2 ∧ P3)
PRED1 AND **((NOT** PRED1) **AND** (**NOT** PRED3)**) (because (PRED1 = PRED2))**

P1 ∧ ((~P1) ∧ (~P3)) = (P1 ∧ (~P1)) ∧ (~P3)

(P1 ∧ (~P1)) ∧ (~P3) = **FALSE ∧ (~P3)** because (P1 ∧ (~P1)) will always false

FALSE ∧ (~P3) = ~ P3

So, we can write **NOT (B.BRANCH_LOC = 'Delhi')** instead of the condition like:
AND B.BRANCH_LOC = 'Mumbai'
AND NOT (B.BRANCH_LOC = 'Mumbai' **AND** B.BRANCH_LOC = 'Delhi')

So, the above whole query will simplify in following form:

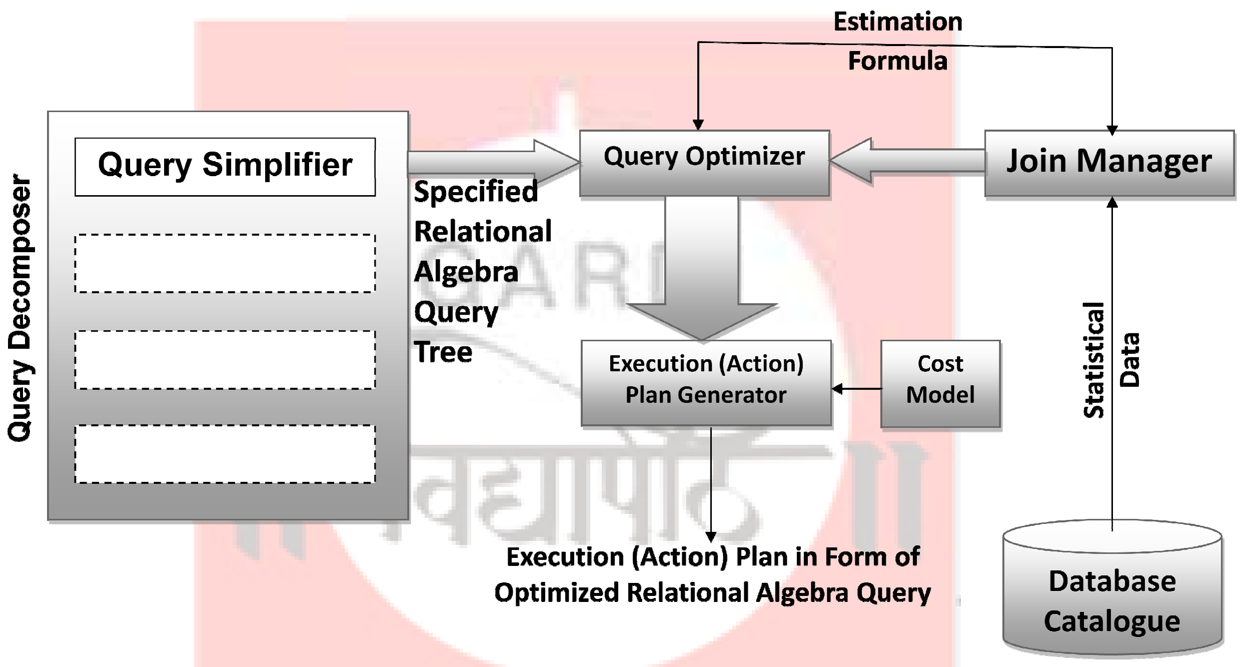
```
SELECT   D.DEPT_ID, M.BRANCH_MGR, M.BRANCH_ID, B.BRANCH_ID,
         B.BRANCH_LOC, E.EMP_NAME, E.EMP_SALARY
FROM     DEPARTMENT AS D, MANAGER AS M, BRANCH AS B
WHERE    D.DEPT_IT = M.DEPT_ID AND M.BRANCH_ID = B.BRANCH_ID
         AND M.BRANCH_MGR = E.EMPID AND NOT (B.BRANCH_LOC = 'Delhi')
         AND B.PROFIT_TO_DATE > 1000000 AND E.EMP_SALARY >85000
         AND NOT (BRANCH_LOC = 'Delhi') AND D.DEPT_LOC = 'Bangalore'
```

5) Query Restructuring :-

- In the final stage of the query decomposition, the query can be restructured to give a more efficient implementation.
- Transformation rules are used to convert one relational algebra expression into an equivalent form that is more efficient.
- The query can now be regarded as a relational algebra program, consisting of a series of operations on relation.

Query Optimization

- The primary goal of query optimization is of choosing an efficient execution strategy for processing a query.
- The query optimizer attempts to minimize the use of certain resources (mainly the number of I/O and CPU time) by selecting a best execution plan (access plan).
- A query optimization start during the validation phase by the system to validate the user has appropriate privileges.
- Now an action plan is generate to perform the query.



Detail Block Diagram of Query Optimization

- Relational algebra query tree generated by the query simplifier module of query decomposer.
- Estimation formulas used to determine the cardinality of the intermediate result table.
- A cost Model
- Statistical data from the database catalogue.

- The output of the query optimizer is the execution plan in form of optimized relational algebra query.

- 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 basic issues in Query Optimization are :**
    - How to use available indexes.
    - How to use memory to accumulate information and perform immediate steps such as sorting.
    - How to determine the order in which joins should be performed.
- The term query optimization does not mean giving always an optimal (best) strategy as the execution plan.
- It is just a responsibly efficient strategy for execution of the query.
- The decomposed query block of SQL is translating into an equivalent extended relational algebra expression and then optimized.

- **There are two main techniques for implementing Query Optimization:**
    - The **first** technique is based on ***Heuristic Rules*** for ordering the operations in a query execution strategy.
    - The **second** technique involves the ***systematic estimation*** of the cost of the different execution strategies and choosing the execution plan with the *lowest cost*.
- Semantic query optimization is used with the combination with the heuristic query transformation rules.
- It uses constraints specified on the database schema such as unique attributes and other more complex constraints, in order to modify one query into another query that is more efficient to execute.

1. Heuristic Rules :-

- The heuristic rules are used as an optimization technique to modify the internal representation of query.
- Usually, heuristic rules are used in the form of query tree of query graph data structure, to improve its performance.
- One of the main heuristic rule is to apply SELECT operation 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 multi-value function of the sizes of the input files.
- The SELECT and PROJECT reduced the size of the file and hence, should be applied before the JOIN or other binary operation.

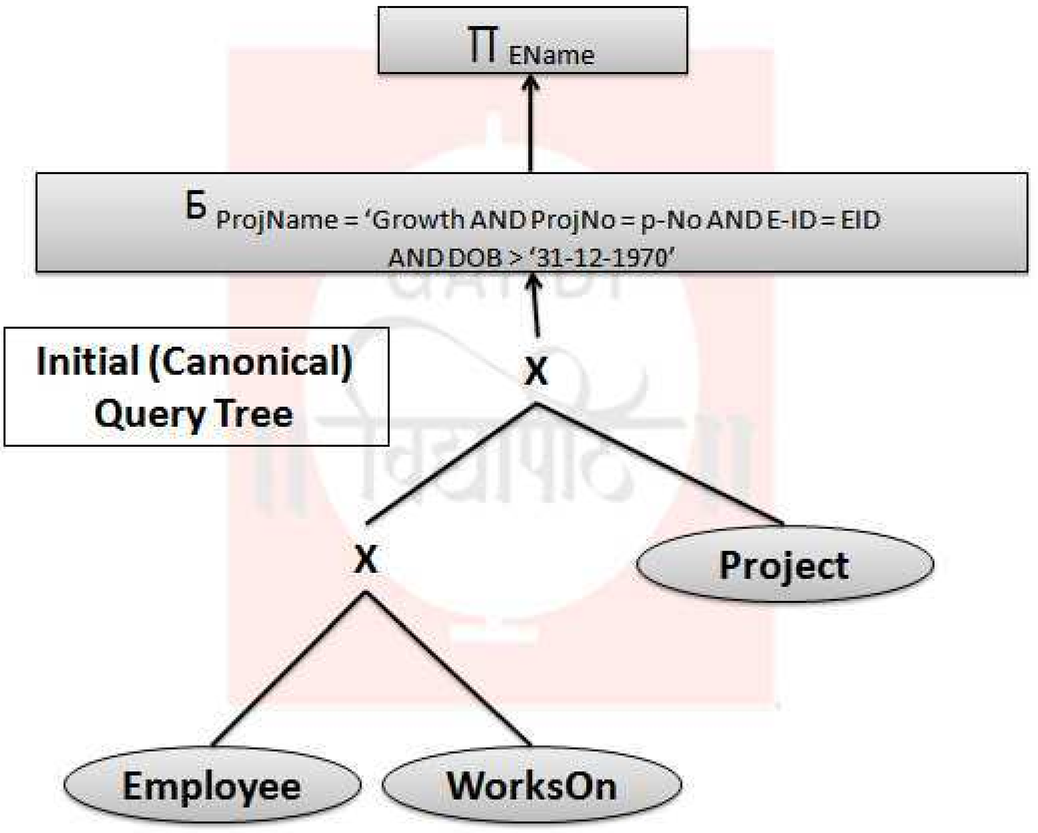
- Heuristic query optimizer transforms the initial (canonical) query tree into final query tree using equivalence transformation rules.
- This final query tree is efficient to execute.

- **For example consider the following relations :**

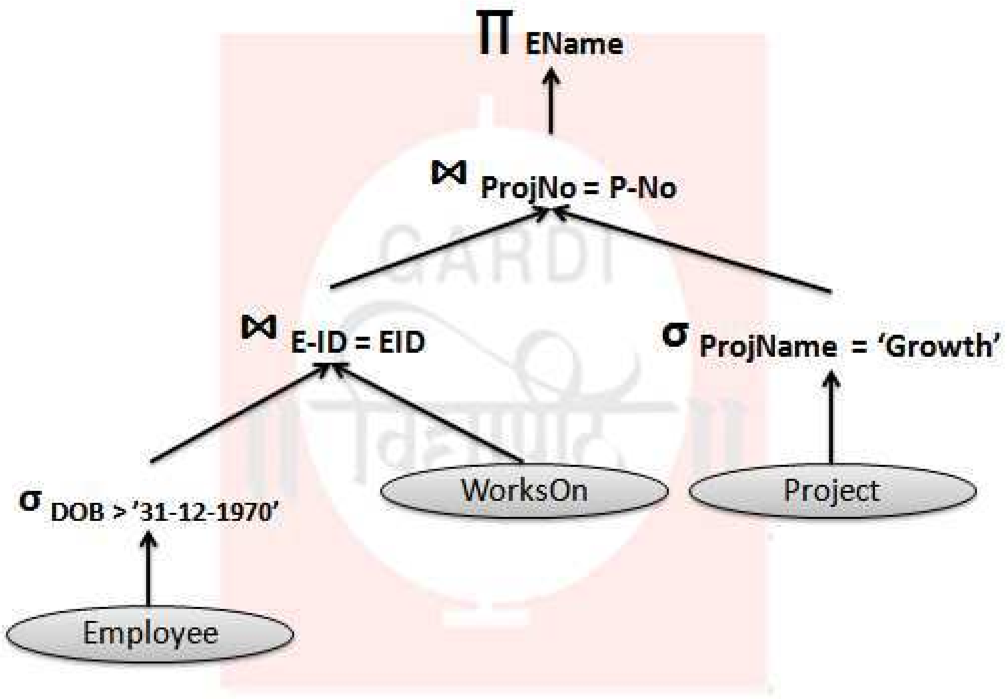
  Employee (EName, EID, DOB, EAdd, Sex, ESalary, EDeptNo)
  Department (DeptNo, DeptName, DeptMgrID, Mgr_S_date)
  DeptLoc (DeptNo, Dept_Loc)
  Project (ProjName, ProjNo, ProjLoc, ProjDeptNo)
  WorksOn (E-ID, P-No, Hours)
  Dependent (E-ID, DependName, Sex, DDOB, Relation)

- Now let us consider the query in the above database to find the name of employees born after 1970 who work on a project named 'Growth'.

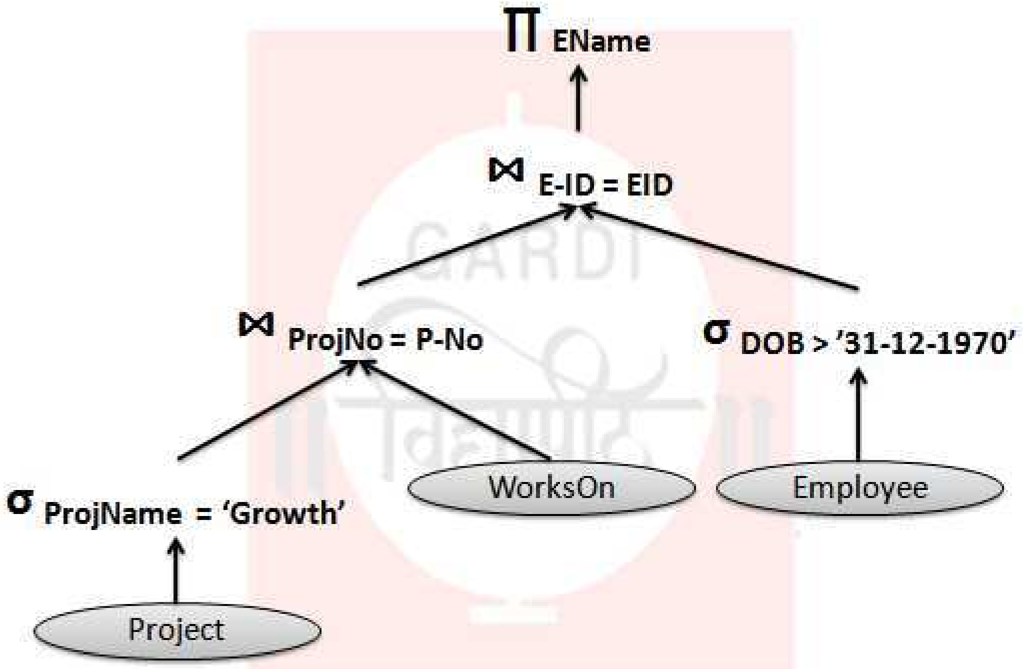
```
SELECT      EName
FROM        Employee, WorksOn, Project
WHERE       ProjName = 'Growth' AND ProjNo = P-No
                 AND EID = E-ID AND DOB > '31-12-1970';
```



Π EName
Б ProjName = 'Growth AND ProjNo = p-No AND E-ID = EID AND DOB > '31-12-1970'
Initial (Canonical) Query Tree
X
X
Project
Employee
WorksOn



Improved Query tree by applying SELECT Operation



Improved Query tree by applying more restrictive SELECT Operation

❖ General Transformation Rules :-

- Transformation rules are used by the query optimizer to transform one relational algebra expression into an equivalent expression that is more efficient to execute.
- A relation is consider as equivalent of another relation if two relations have the same set of attributes in a different order but representing the same information.
- These transformation rules are used to restructure the initial (canonical) relational algebra query tree attributes during query decomposition.

1. **Cascade of σ :-**
   σ_{c1} AND $_{c2}$ AND …AND $_{cn}$ (R) = $\sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$

2. **Commutativity of σ :-**
   $\sigma_{C1}(\sigma_{C2}(R)) = \sigma_{C2}(\sigma_{C1}(R))$

3. **Cascade of Л :-**
   $Л_{List1}(Л_{List2}(\ldots(Л_{List\ n}(R))\ldots)) = Л_{List1}(R)$

4. **Commuting σ with Л :-**
   $Л_{A1,A2,A3\ldots An}(\sigma_{C}(R)) = \sigma_{C}(Л_{A1,A2,A3\ldots An}(R))$

5. **Commutativity of ⋈ AND x :-**
   → $R \bowtie_c S = S \bowtie_c R$
   → R x S = S x R

6. **Commuting σ with or x :-**

➢ If all attributes in selection condition c involved only attributes of one of the relation schemas (R).
   $\sigma_c (R \bowtie S) = (\sigma_c (R)) \bowtie S$

➢ Alternatively, selection condition c can be written as (c1 AND c2) where condition c1 involves only attributes of R and condition c2 involves only attributes of S then :
   $\sigma_c (R \bowtie S) = (\sigma_{c1} (R)) \bowtie (\sigma_{c2} (S))$

7. **Commuting Л with ⋈ or x :-**
   → The projection list L = {A1,A2,..An,B1,B2,…Bm}.
   → A1…An attributes of R and B1…Bm attributes of S.
   → Join condition C involves only attributes in L then :
   $Л_L (R \bowtie_c S) = (Л_{A1,...An} (R)) \bowtie_c (Л_{B1,...Bm}(S))$

8. **Commutativity of SET Operation :-**
   → R ∪ S = S ∪ R
   → R ∩ S = S ∩ R
   Minus (R-S) is not commutative.

9. **Associatively of ⋈, x, ∩, and ∪ :-**
   → If ∅ stands for any one of these operation throughout the expression then :
   (R ∅ S) ∅ T = R ∅ (S ∅ T)

10. **Commutativity of σ with SET Operation :-**
   → If ∅ stands for any one of three operations (∪,∩,and-) then :
   $\sigma_c (R \emptyset S) = (\sigma_c (R)) \cup (\sigma_c (S))$
   $Л_c (R \emptyset S) = (Л_c (R)) \cup (Л_c (S))$

11. **The Л operation comute with ∪ :-**
   $Л_L (R \cup S) = (Л_L(R)) \cup (Л_L(S))$

12. **Converting a (σ,x) sequence with ∪**
   $(\sigma_c (R \times S)) = (R \bowtie_c S)$

❖ Example of Transformation Rule :-

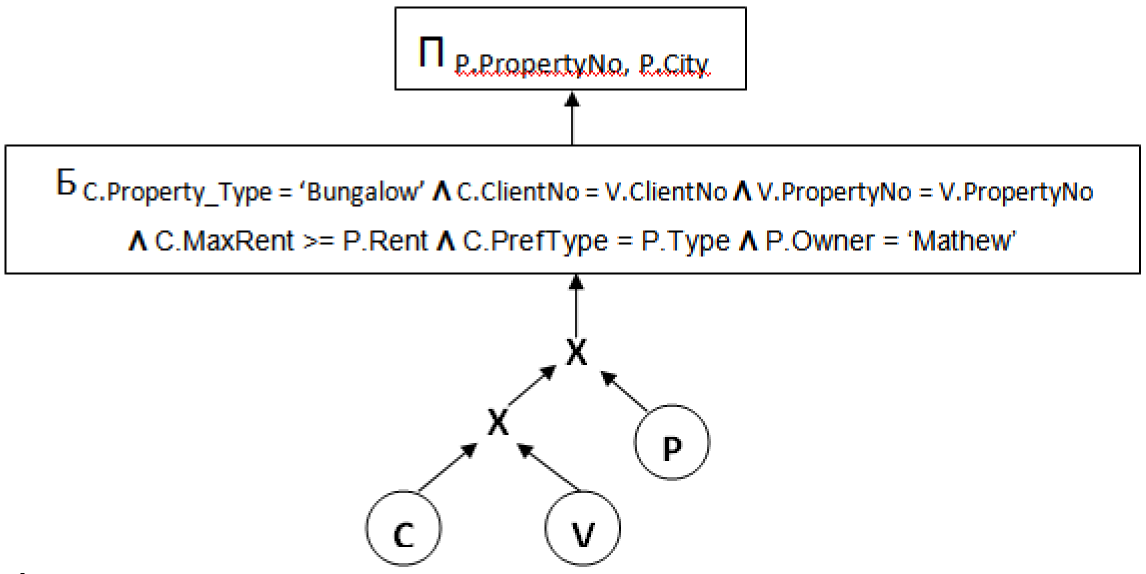
- Let us consider the SQL query in which the prospectives are looking for 'Bungalow'. Now we have to develop a query to find out the properties thet match their requirements and are owned by owner 'Mathew'.

SELECT P.PropertyNo, P.City

FROM CLIENT **AS** C, VIEWING **AS** V, PROPERTY_RENT **AS** P

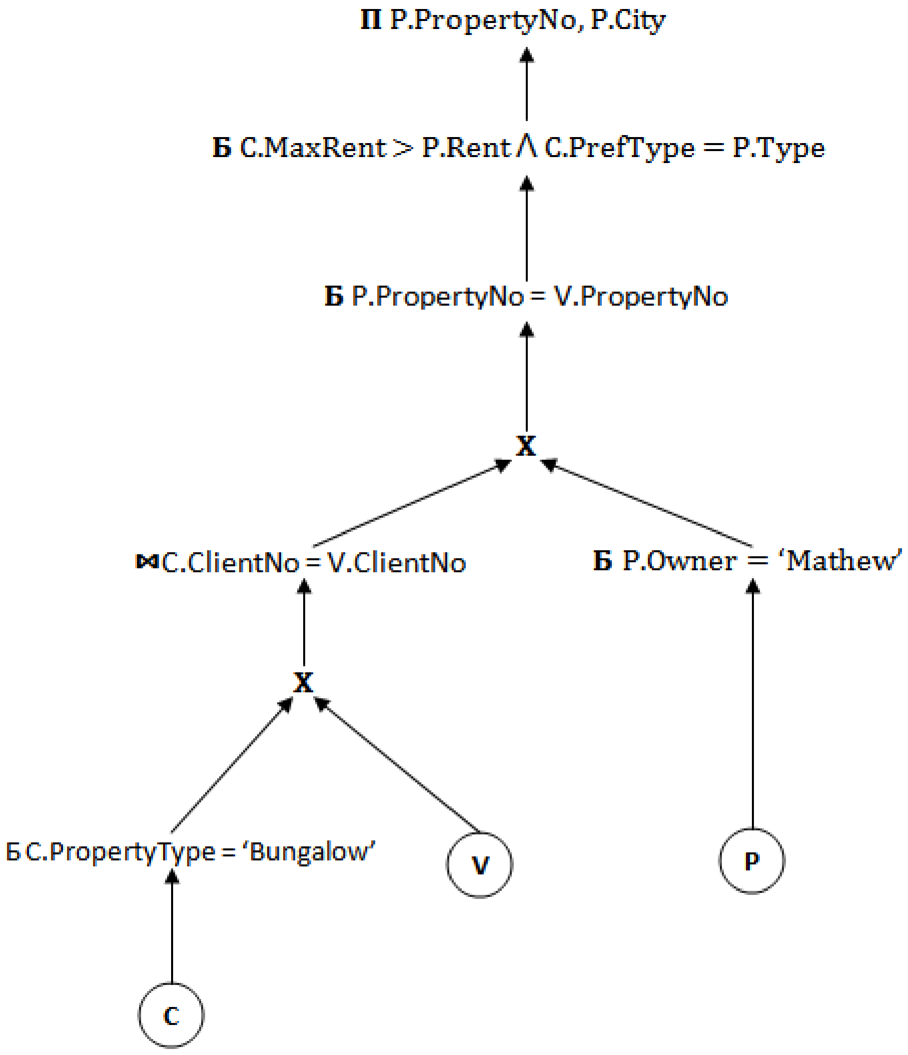
WHERE C.Property_Type = 'Bungalow' **AND** C.Client_No = V.Client_No **AND** V.PropertyNo = P.PropertyNo **AND** C.MaxRent >= P.Rent **AND** C.PrefType = P.Type **AND** P.Owner = 'Methew' **;**

Initial (Canonical) Query Tree

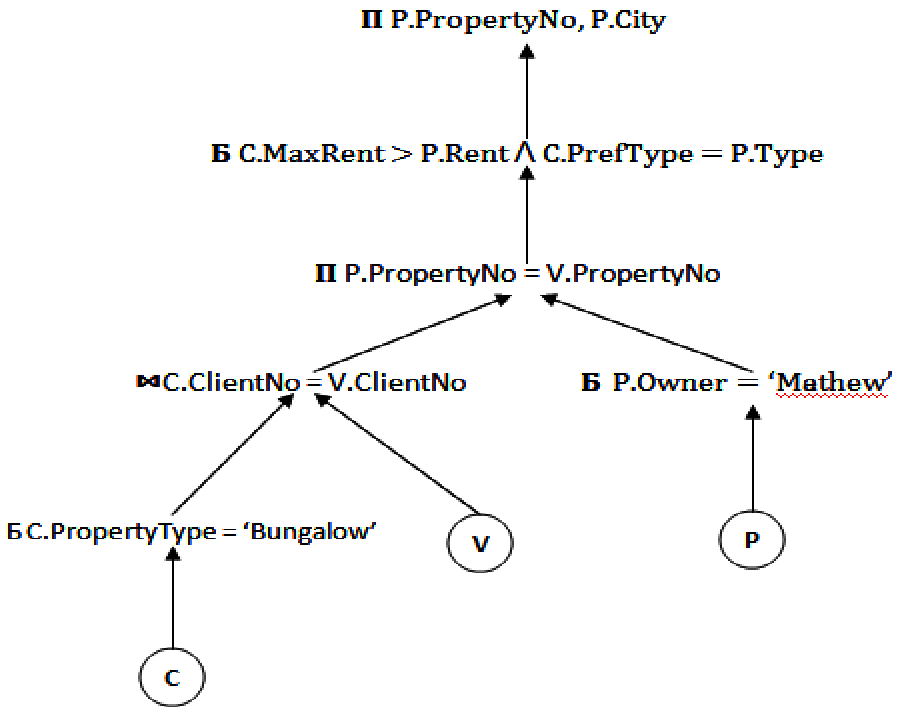

Π P.PropertyNo, P.City
Б C.Property_Type = 'Bungalow' ∧ C.ClientNo = V.ClientNo ∧ V.PropertyNo = V.PropertyNo ∧ C.MaxRent >= P.Rent ∧ C.PrefType = P.Type ∧ P.Owner = 'Mathew'
X
X
P
C
V

.

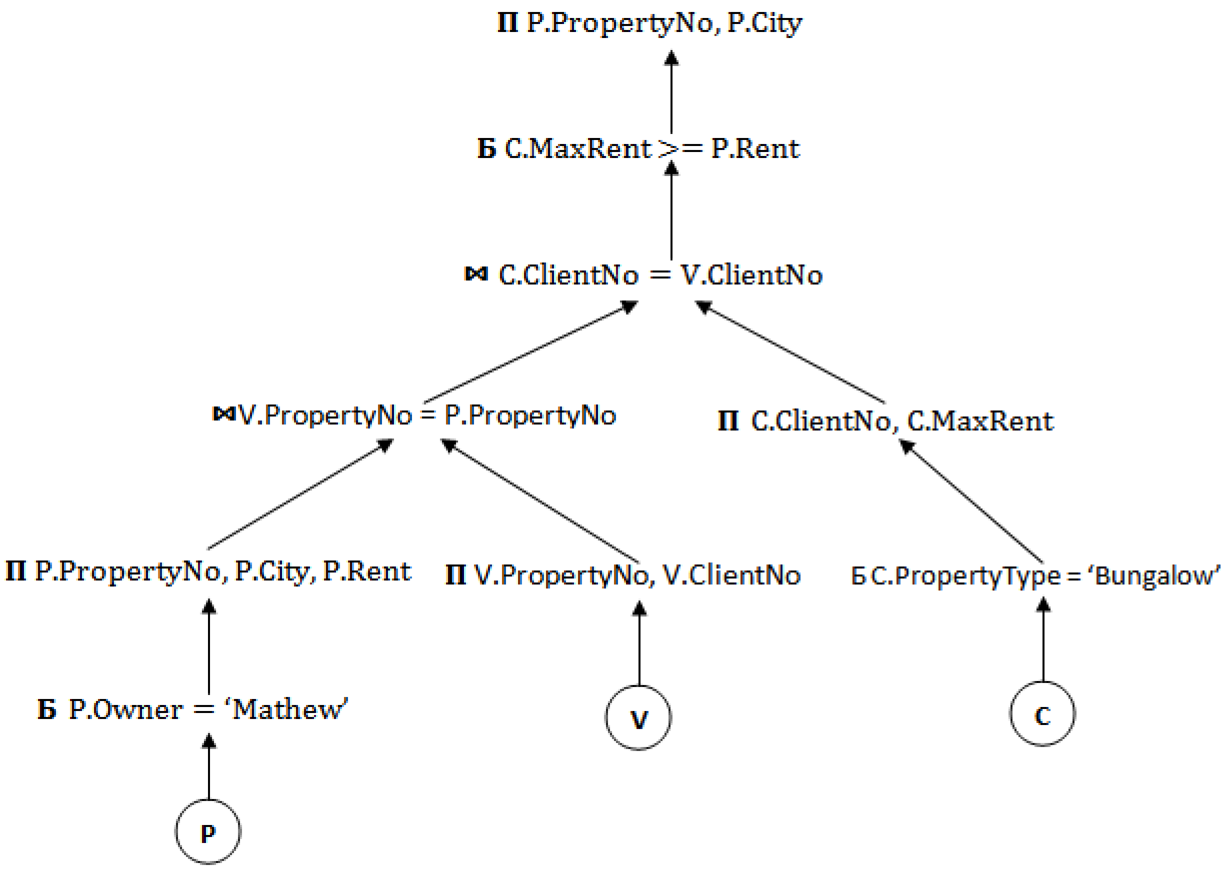
Initial (Canonical) query Tree


Π P.PropertyNo, P.City
Б C.MaxRent > P.Rent ∧ C.PrefType = P.Type
Б P.PropertyNo = V.PropertyNo
X
⋈C.ClientNo = V.ClientNo
Б P.Owner = 'Mathew'
X
Б C.PropertyType = 'Bungalow'
V
P
C

Improved Query Tree by applying SELECT operations


Π P.PropertyNo, P.City
Б C.MaxRent > P.Rent ∧ C.PrefType = P.Type
Π P.PropertyNo = V.PropertyNo
⋈C.ClientNo = V.ClientNo
Б P.Owner = 'Mathew'
Б C.PropertyType = 'Bungalow'
V
P
C

Improved Query tree by changing Cartesian Products to Equijoin


Π P.PropertyNo, P.City
Б C.MaxRent >= P.Rent
⋈ C.ClientNo = V.ClientNo
⋈V.PropertyNo = P.PropertyNo
Π C.ClientNo, C.MaxRent
Π P.PropertyNo, P.City, P.Rent
Π V.PropertyNo, V.ClientNo
Б C.PropertyType = 'Bungalow'
Б P.Owner = 'Mathew'
V
C
P

Final Reduced Relational Algebra Query Tree

❖ Heuristic Optimization Algorithm :-

- The Database Management System use Heuristic Optimization Algorithm that utilizes some of the transformation rules to transform an initial query tree into an optimized and efficient executable query tree.
- The steps of the heuristic optimization algorithm that could be applied during query processing and optimization are :

1. **Step-1 :-**
    - Perform SELECT operation to reduced the subsequent processing of the relation:
    - Use the transformation rule 1 to break up any SELECT operation with conjunctive condition into a cascade of SELECT operation.
2. **Step-2 :-**
    - Perform commutativity of SELECT operation with other operation at the earliest to move each SELECT operation down to query tree.
    - Use transformation rules 2, 4, 6 and 10 concerning the commutativity of SELECT with other operation such as unary and binary operations and move each select operation as far down the tree as is permitted by the attributes involved in the SELECT condition. Keep selection predicates on the same relation together.
3. **Step-3 :-**
    - Combine the Cartesian product with subsequent SELECT operation whose predicates represents the join condition into a JOIN operation.
    - Use the transformation rule 12 to combine the Cartesian product operation with subsequent SELECT operation.
4. **Step-4 :-**
    - Use the commutativity and associativity of the binary operations.
    - Use transformation rules 5 and 9 concerning commutaitivity and associativity to rearrange the leaf nodes of the tree so that the leaf node with the most restrictive selection operation are executed first in the query tree representation. The most restrictive SELECT operation means :
        - Either the one that produce a relation with a fewest tuples or with smallest size.
        - The one with the smallest selectivity.
5. **Step-5 :-**

- o Perform the projection operation as early as possible to reduce the cardinality of the relation and the subsequent processing of the relation, and move the Projection operations as far down the query tree as possible.

- o Use transformation rules 3, 4, 7 and 10 concerning the cascading and commuting of projection operations with other binary operation. Break down and move the projection attributes down the tree as far as needed. Keep the projection attributes in the same relation together.

6. **Step-6 :-**
   - o Compute common expression once.
   - o Identify sub-tree that represent group of operations that can be executed by a single algorithm.

2. Cost Estimation in Query Optimization :-

- ➢ The main aim of query optimization is to choose the most efficient way of implementing the relational algebra operations at the lowest possible cost.
- ➢ Therefore the query optimizer should not depend solely on heuristic rules, but, it should also estimate the cost of executing the different strategies and find out the strategy with the minimum cost estimate.
- ➢ The method of optimizing the query by choosing a strategy those result in minimum cost is called cost-based query optimization.
- ➢ The cost-based query optimization uses the formula that estimate the cost for a number of options and selects the one with lowest cost and the most efficient to execute.
- ➢ The cost functions used in query optimization are estimates and not exact cost functions.
- ➢ The cost of an operation is heavily dependent on its selectivity, that is, the proportion of select operation(s) that forms the output.
- ➢ In general the different algorithms are suitable for low or high selectivity queries. In order for query optimizer to choose suitable algorithm for an operation an estimate of the cost of executing that algorithm must be provided.
- ➢ The cost of an algorithm is depend of a cardinality of its input.
- ➢ To estimate the cost of different query execution strategies, the query tree is viewed as containing a series of basic operations which are linked in order to perform the query.
- ➢ It is also important to know the expected cardinality of an operation's output because this forms the input to the next operation.

❖ Cost Components of Query Execution :-

- ➢ The success of estimating the size and cost of intermediate relational algebra operations depends on the amount the accuracy of statistical data information stored with DBMS.

- The cost of executing the query includes the folowing components :-
  - Access cost to secondary storage.
  - Storage cost.
  - Computation cost.
  - Memory uses cost.
  - Communication cost.

1. **Access cost to secondary storage :-**
   - Access cost is the cost of searching for reading and writing data blocks (containing the number of tuples) that reside on secondary storage, mainly on disk of the DBMS.
   - The cost of searching for tuples in the database relations depends on the type of access structures on that relation, such ordering, hashing and primary or secondary indexes.

2. **Storage cost :-**
   - The storage cost is of storing any intermediate relations that are generated by the executing strategy for the query.

3. **Computation cost :-**
   - Computation cost is the cost of performing in-memory operations on the data buffers during query execution.
   - Such operations contain searching for and sorting records, merging records for a join and performing computation on a field value.

4. **Memory uses cost :-**
   - Memory uses cost is a cost pertaining to the number of memory buffers needed during query execution.

5. **Communication cost :-**
   - It is the cost of transferring query and its results from the database site to the site of terminal of query organization.

- Out of the above five cost components, the most important is the secondary storage access cost.
- The emphasis of the cost minimization depends on the **size** and **type** of database applications.
- For example in smaller database the emphasis is on the minimizing computing cost as because most of the data in the files involve in the query can be completely store in the main memory. For large database, the main emphasis is on minimizing the access cost to secondary device.

- For distributed database, the communication cost is minimized as because many sites are involved for the data transfer.
- To estimate the cost of various execution strategies, we must keep track of any information that is needed for the cost function. This information may be stored in database catalog, where it is accessed by the query optimizer.
- Typically the DBMS is expected to hold the following types of information in its system catalogue.
    - The **number of tuples** in relation as **R [nTuples(R)]**.
    - The average record size in relation R.
    - The **number of blocks** required to store relation R as **[nBlocks(R)]**.
    - The blocking factors in relation R (that is the number of tuples of R that fit into one block) as **[bFactor(R)]**.
    - Primary access method for each file.
    - Primary access attributes for each file.
    - The number of level of each multilevel index I (primary, secondary or clustering) as **[$nLevels_A(I)$]**.
    - The number of first level index blocks as **[$nBlocks_A$ (I)]**.
    - The number of distinct values that are appear for attribute A in relation R as **[$nDistinct_A(R)$]**.
    - The minimum and maximum possible values for attribute A in relation R as **[$min_A(R)$, $max_A(R)$]**.
    - The selectivity of an attribute, which is the fraction of records satisfying an equality condition on the attribute.
    - The selection cardinality of given attribute A in relation R as **[$SC_A(R)$]**. The selection cardinality is the average number of tuples that satisfied an equality condition on attribute A.

❖ **Cost functions for SELECT Operation :-**

- **Linear Search :-**
    - [nBlocks(R)/2], if the record is found.
    - [nBlocks(R)], if no record satisfied the condition.

- **Binary Search :-**
    - [log_2(nBlocks(R))], if equality condition is on key attribute, because $SC_A(R) = 1$ in this case.
    - [log_2(nBlocks(R))] + [SC_A(R)/bFactor(R)] – 1, otherwise.

- **Using primary index or hash key to retrieve a single record :-**
    - 1, assuming no overflow

- **Equity condition on Primary key :-**
  - $[nLevel_A(I) + 1]$

- **Equity condition on Non-Primary key :-**
  - $[nLevel_A(I) + 1] + [nBlocks(R)/2]$
- **Using inequality condition on a secondary index (B$^+$ Tree) :-**
  - $[nLevel_A(I) + 1] + [nLfBlocks_A(I)/2] + [nTuples(R)/2]$
  - **Equity condition on clustering index :-**
  - $[nLevel_A(I) + 1] + [SC_A(R)/bFactor(R)]$
  - **Equity condition on non-clustering index :-**
  - $[nLevel_A(I) + 1] + [SC_A(R)]$

❖ Example of Cost Estimation for SELECT Operation :-

- Let us consider the relation EMPLOYEE having following attributes :-
  EMPLOYEE (EMP-ID, DEPT-ID, POSITION, SALARY)

- Let us consider the following assumptions :-
  - o There is a hash index with no overflow on primary key attribute EMP-ID.
  - o There is a clustering index on foreign key attribute DEPT-ID.
  - o There is a B$^+$-Tree index on the SALARY attribute.
- Let us also assume that the EMPLOYEE relation has the following statistics in the system catalog :

nTuples(EMPLOYEE)	=	6,000
bFactor(EMPLOYEE)	=	60
nBlocks(EMPLOYEE)	= =	nTuples(EMPLOYEE) / bFactor(EMPLOYEE) 6,000 / 60 = 100
$nDistinct_{DEPT-ID}$(EMPLOYEE)	=	1,000
$SC_{DEPT-ID}$(EMPLOYEE)	= =	nTuples(EMPLOYEE) / $nDistinct_{DEPT-ID}$(EMPLOYEE) 6,000 / 1,000 = 6
$nDistinct_{POSITION}$(EMPLOYEE)	=	20
$SC_{POSITION}$(EMPLOYEE)	= =	nTuples(EMPLOYEE) / $nDistinct_{POSITION}$(EMPLOYEE) 6,000 / 20 = 300
$nDistinct_{SALARY}$(EMPLOYEE)	=	1,000
SC_{SALARY}(EMPLOYEE)	= =	nTuples(EMPLOYEE) / $nDistinct_{SALARY}$(EMPLOYEE) 6,000 / 1,000 = 6
min_{SALARY}(EMPLOYEE)	=	20,000
max_{SALARY}(EMPLOYEE)	=	80,000
$nLevels_{DEPT-ID}$(I)	=	2

nLevels$_{SALARY}$(I)	=	2
nLfBlocks$_{SALARY}$(I)	=	50

- Selection 1 :- б $_{EMP\text{-}ID = ‘106519’}$ (EMPLOYEE)
- Selection 2 :- б $_{POSOTION = ‘MANAGER’}$ (EMPLOYEE)
- Selection 3 :- б $_{DEPT\text{-}ID = ‘SAP\text{-}04’}$ (EMPLOYEE)
- Selection 4 :- б $_{SALARY = 30,000}$ (EMPLOYEE)
- Selection 5 :- б $_{POSOTION = ‘MANAGER’ \land DEPT\text{-}ID = ‘SPA\text{-}04’}$ (EMPLOYEE)

- Now we will choose the query execution strategies by comparing the cost as follows :

Selection -1	The selection operation contains an equality condition on the primary key EMP-ID of the relation EMPLOYEE. Therefore, as the attribute EMP-ID is hashed we can use the strategy 3 to estimate the cost as 1 block. The estimated cardinality of the result relation is **SC $_{EMP\text{-}ID}$ (EMPLOYEE) = 1**.
Selection -2	The attribute in the predicate is the non-key, non-indexed attribute. Therefore we can improve on the linear search method, giving an estimated cost of 100 blocks. The estimated cardinality of the result relation is **SC $_{POSITION}$ (EMPLOYEE) = 300**.
Selection -3	The attribute in the predicate is a foreign key with a clustering index. Therefore, we can use strategy 7 to estimate the cost as **(2 + (6/30)) = 3 blocks**. The estimated cardinality of result relation is **SC $_{DEPT\text{-}ID}$ (EMPLOYEE) = 6**.
Selection -4	The predicate here involves a range search on the SALARY attribute, which has the B^{+}-Tree index. Therefore we can use the strategy 6 to estimate the cost as (2 + (50/2) + (6,000/2)) = 3027 blocks. Thus the linear search strategy is used in this case, the estimated cardinality of the result relation is SC $_{SALARY}$ (EMPLOYEE) = [6000*(8000-2000*2)/(8000-2000)] = 4000.
Selection -5	While we are retrieving each tuple using the clustering index, we can check whether they satisfied the first condition (POSITION = ‘MANAGER’). We know that estimated cardinality of the second condition **SC $_{DEPT\text{-}ID}$ (EMPLOYEE) = 6**. Let us assume that this intermediate condition is S. then the number of distinct values of POSITION in S can be estimated as **[(6 + 20)/2] = 9**. Let us apply now the second condition using the clustering index on DEPT-ID, which has an estimated cost of 3 blocks. Thus, the estimated cardinality of the result relation will be SC $_{POSITION}$ (S) = 6/9 = 1, which would be correct if there is one manager for each branch.

❖ **Cost functions for JOIN Operation :-**

- Join operation is the most time consuming operation to process. An estimate for the size (number of tuples) of the file that results after the JOIN operation is required to develop reasonably accurate cost functions for JOIN operations.
- The JOIN operations define the relation containing tuples that satisfy a specific predicate F from the Cartesian product of two relations R and S.

- Following table shows the different strategies for JOIN operations.

Strategies		Cost Estimation
Block nested-loop JOIN	a)	**nBlocks(R) + (nBlocks(R) * nBlocks(S))** If the buffer has only one block.
	b)	**nBlocks(R) + [nBlocks(S) * (nBlocks(R)/(nBuffer-2))]** If (nBuffer-2) blocks is there for R
	c)	**nBlocks(R) + nBlocks(S)** If all blocks of R can be read into database buffer
Indexed nested-loop JOIN	a)	**nBlocks(R) + nTuples(R) * (nLevel$_A$(I) + 1)** If join attribute A in S is a primary key
	b)	**nBlocks(R) + nTuples(R) * (nLevel$_A$(I) + [SC$_A$(R) / bFactor(R)])** If clustering index I is on attribute A.
Sort-merge JOIN	a)	**nBlocks(R) * [log$_2$nBlocks(R)] + nBlocks(S) * [log$_2$nBlocks(R)]** For Sort
	b)	**nBlocks(R) + nBlocks(S)** For Merge
Hash JOIN	a)	**3 (nBlocks(R) + nBlocks(S))** If Hash index is in memory
	b)	**2 (nBlocks(R) + nBlocks(S)) * [log (nBlocks(S)) - 1] + nBlocks(R) + nBlocks(S)** Otherwise

❖ Example of Cost Estimation for JOIN Operation :-

- Let us consider the relations EMPLOYEE, DEPARTMENT and PROJECT having the following attributes :-
  EMPLOYEE (EMP-ID, DEPT-ID, POSITION, SALARY)
  DEPARTMENT (DEPT-ID, EMP-ID)
  PROJECT (PROJ-ID, DEPT-ID, EMP-ID)

- Let us consider the following assumptions :-
  - There are separate hash index with no overflow on the primary key attribute EMP-ID of relation EMPLOYEE and DEPT-ID on relation DEPARTMENT.

- o There are 200 database buffer blocks.

➢ Let us assume that the relations have the following statistics stored in database catalog:-.

nTuples(EMPLOYEE)	=	12,000
bFactor(EMPLOYEE)	=	120
nBlocks(EMPLOYEE)	= =	[nTuples(EMPLOYEE) / bFactor(EMPLOYEE)] 12,000 / 120 = 100
nTuples(DEPARTMENT)	=	600
bFactor(DEPARTMENT)	=	60
nBlocks(DEPARTMENT)	= =	[nTuples(DEPARTMENT) / bFactor(DEPARTMENT)] 600 / 60 = 10
nTuples(PROJECT)	=	80,000
bFactor(PROJECT)	=	40
nBlocks(PROJECT)	= =	[nTuples(PROJECT) / bFactor(PROJECT)] 80,000 / 40 = 2000

➢ Let us consider the following two JOINS and use the strategies that are mention in the table to improve the costs :-

JOIN-1 : EMPLOYEE $\bowtie_{\text{EMP-ID}}$ PROJECT
JOIN-2 : DEPARTMENT $\bowtie_{\text{DEPT-ID}}$ PROJECT

The estimated I/O cost of JOIN operations for the above two JOINS

Strategies	JOIN - 1	JOIN - 2	Comments
Block nested-loop JOIN	(a) 400200 (b) 4282 (c) N/A	(a) 20010 (b) N/A (c) 2010	(a) Buffer has only one block (b) (nBuffer-2) Blocks for R (c) All blocks of R can be read into database buffer
Indexed nested-loop JOIN	12200	610	Key Hashed
Sort-Merge JOIN	(a) 25800 (b) 2200	(a) 24240 (b) 2010	(a) Unsorted (b) Sorted
Hash JOIN	6600	6030	Hash table fit in memory

❖ **Algorithms for SELECT Operations :-**

1. Search Methods for SIMPLE SELECTION:-

➢ A number of search algorithms are possible for selecting records from file.

- These are known as File Scans, because they scan the records of the file to retrieve records that satisfy a selection condition.
- If search algorithm involves the use of index, the index search is called an Index Scan.
- The following search methods (S1 through S6) are example of some of the search algorithms that can be use to implement a SLECT operation.

S1:- Linear Search :-

Retrieving every records in the file, and test whether its attribute value satisfy the selection condition or not.

S2:- Binary Search

If the selection condition involves an equality comparison on a key attribute on which a file is ordered, binary search is more efficient then the linear search.
E.g.:- SSN is the ordering attribute of EMPLOYEE…
σ SSN = ‘12345’ (EMPLOYEE)

S3:- Using Primary Index :-

If the selection condition involves an equality comparison on a key attribute with a primary index, for example SSN = ‘12345’ in previous example, use a primary index to retrieve record. This condition retrieves ONLY SINGLE RECORD.

S4:- Using Primary Index to Retrieve Multiple Records :-

If the comparison condition is >,<,≥ or ≤ on a key field with a primary index, for example DNO < 5 in following example use the primary index to retrieve records satisfying condition then DNO<5 retrieve all the preceding records then of DNO = 5.
E.g.:- DNO is the primary attribute of EMPLOYEE uses the index…
σ DNO < ‘12345’ (EMPLOYEE)

S5:- Using Clustering Index to Retrieve Multiple Records :-

If the selection condition involves an equality comparison on an nonprime attribute with a clustering index, for example Mark2 = 85 use the index to retrieve all the records that satisfy the selection condition. σ Mark2 < 85 (MARKS)

S6:- Linear Search :-

This search method can be used to retrieve a single record if the indexing field is a key (has unique value) or to retrieve multiple records if the indexing field is not a key. This can also be use with the comparisons involving <,>,≤ or ≥.

2. Search Methods for COMPLEX SELECTION :-

- If the condition of SELECT operation is a conjunctive condition that is, if it is made up of several simple conditions connected with the AND logical connectives.

S7:- Conjunctive selection using an individual index.

If an attribute involve 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 satisfy the remaining simple condition in the conjunctive condition.

S8:- Conjunctive selection using composite condition.

If two or more attributes are involved n equality condition in the conjunctive condition and a composite index exist on the combined field, for example if an index has been created on the composite key (Essn, Pno) of the WORKS_ON file we can use the index directly.

$\sigma_{\text{Essn} < \text{'1234'}}\ \text{AND}_{\ \text{Pno} = 10}$ (WORKS_ON)

S9:- Conjunctive selection by intersection of records pointers.

If the secondary index is available on more then one of the fields involve in simple conditions in the conjunctive condition and if the indexes include record pointers, then each index can be used to retrieve the set of record pointers that satisfy the individual condition. The intersection of these set of record pointers gives the record pointers that satisfy the conjunctive condition, which are then used to retrieve those records directly.

❖ Algorithms for JOIN Operations :-

- The JOIN operation is one of the most time consuming operation in query processing.
- JOIN involves only two files are called two-way join and join in that there more than two files are involves are known as multiway join.
- The algorithms we consider are for join operations of the form: R $\bowtie_{A=B}$ S : where A & B are domain-compatible attributes of R and S.
  - EMPLOYEE $\bowtie_{Dno = Dnumber}$ DEPARTMENT
  - DEPARTMENT $\bowtie_{Mgr\_SSN = SSN}$ EMPLOYEE

J1 - Nested-loop Join :-

For each record t in R (outer loop) retrieve every records s in S (inner Join) and test whether two records satisfied the join condition t[A] = s[B].

J2 – Single-loop Join :-

If an Index exist for one of the two join attributes-say b of S retrieve each record t in R, one at a time (Single loop), and then use the access structure to retrieve directly all matching records s from S that satisfied s[B]=t[A].

J3 – Sort-merge Join :-

If the records of R and S are physically sorted by the value of Join attributes A and B, we can implement the join in most efficient way.
Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for A and B. If the files are not sorted, they must be sorted first by using external sorting.

J4 – Hash-Join :-

The records of file R and S are both hashed to the same hash file, using the same hashing function on the join attributes A of R and B of S as a hash keys.
First a single pass through a file (R) with a fewer records hashes its records to the hash file buckets; this is called Partitioning Phase, since the records of R are partitioned into a hash bucket.
In the second phase, called the Probing Phase, a single pass through the other file (S) then hashes each of its records to probe the appropriate bucket, and that records is combined with all matching records from R in that bucket.

This simplify description of hash-join assumes that the smaller of the two files fit entirely into memory bucket after the first phase.

❖ PIPELINING AND MATERIALIZING :-

- When a QUERY is composed of several relational algebra operations, the result of one operator is sometimes ***pipelined*** to another operator without creating a temporary relation to hold the intermediate result.
- When a input relation to a unary operation (for example SELECT or PROJECTION) is pipelined into it, it is sometimes said that the operation is applied ***on-the-fly***.
- Pipelining is sometime used to improve the performance of the query.
- As we know that the result of intermediate algebra operations are stored on the secondary storage or disk, which are temporary written.
- If an output of an operation is saved in a temporary relation for processing by the next operator, it is said that the tuple are ***materialized***.
- Thus, this process of temporarily writing intermediate algebra operation is called ***materialization***.
- The materialization process is start from the lowest level operation in the expression, which are at the bottom of the query tree.
- The inputs of the lowest level operations are the relations in the database.
- The lowest level operations on the input relations are executed and stored in temporary relation. Then these temporary relations are used to execute the operation at the next level up in the tree.
- Thus in materialization the output of one operation is stored in temporary relation for processing for the next relation.
- By repeating this process, the operation at the root of the tree is evaluated giving the final result of the expression.
- Alternatively the efficiency of the query evaluation can be improved by reducing the number of temporary files that are produced.
- Therefore the several relational operations are combined into a pipeline of operation in which, the result of one operation is pipelined into another operation without creating the intermediate relation to hold the intermediate result.
- A buffer is created for each pair of adjacent operations to hold the tuple being passed from first operation to second operation.
- Pipeline operation is eliminates the cost of reading and writing temporary relations.
- **Advantage :-**
  - The use of pipelining saves on the cost of creating temporary relations and reading the results back in again.
- **Disadvantage :-**

- The inputs to operations are not necessarily available all at once for processing. This can restrict the choice of algorithm.

❖ Structure of Query Evaluation Plans :-

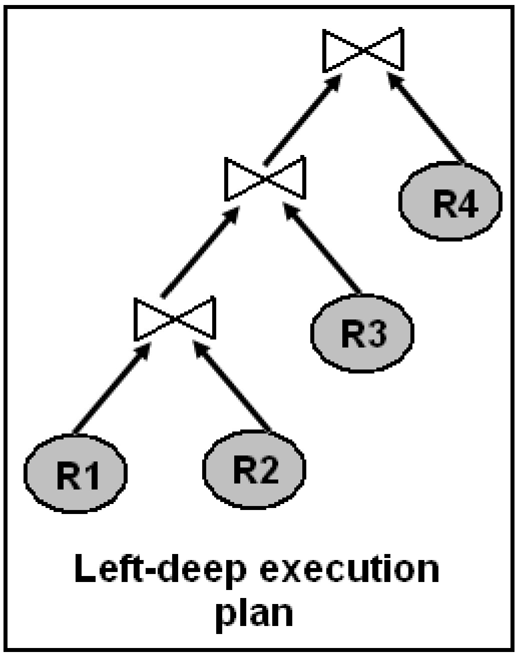
- An evaluation plan is used to define exactly what algorithm should be used for each operation and how the execution of the operation should be coordinate.
- So far we have discus mainly two basic approaches to choosing an efficient execution plan namely (a) heuristic optimization and (b) cost-based optimization. Most query optimizers combine the elements of both the approaches.

❖ Query execution plan :-

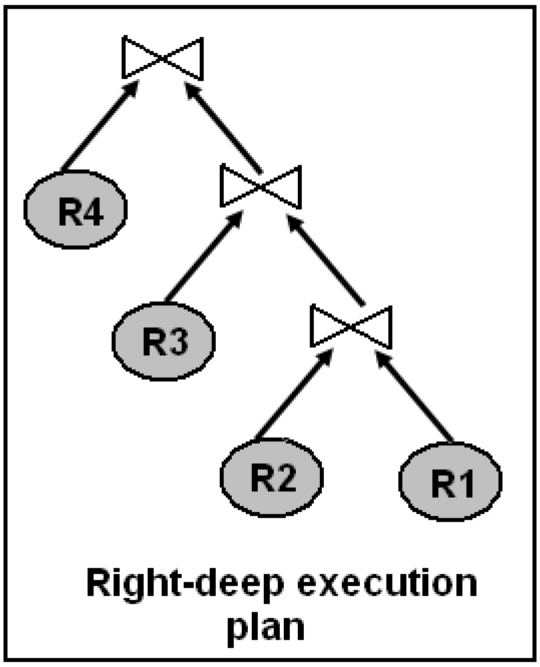
- The query execution plan may classified into the following :-
  - Left-deep (join) tree execution plan.
  - Right-deep query execution plan.
  - Linear tree query execution plan.
  - Bushy (non-linear) tree query execution plan.

1. Left-deep (join) tree query execution plan :-

- It starts with the relation and constructs the result by successively adding an operation involving a single relation until the query is completed.
- That is, only one input into a binary operation is an intermediate result.
- The term relates to how operations are combine to execute a query, for example, only the left hand side of join is allowed to be something that result from a previous join and hence the name left-deep tree.
- This plan has an **advantage** to reducing a search space and allowing the query optimizer to be based on dynamic programming technique.
- Left tree join plan are particularly convenient for pipelined evaluation, since the right operand is a stored relation, and thus only one input to each join is pipelined.
- The main **disadvantage** is that, in reducing the search space, many alternative execution strategies are not considered, some of which may be of lower cost than one found using the linear tree.



Left-deep execution plan



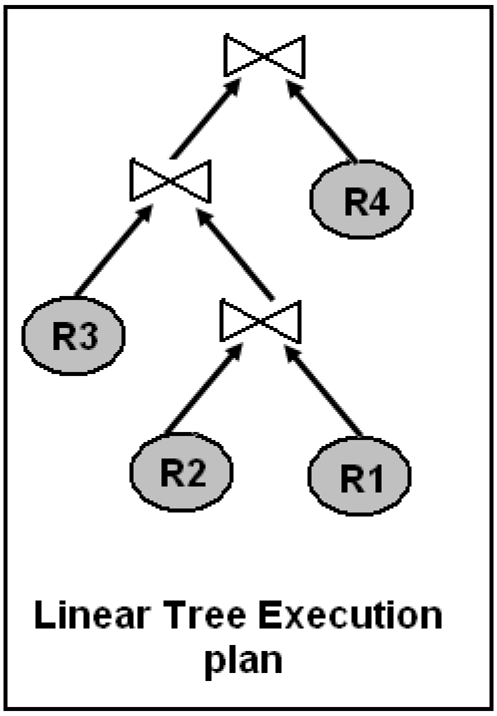
Right-deep execution plan

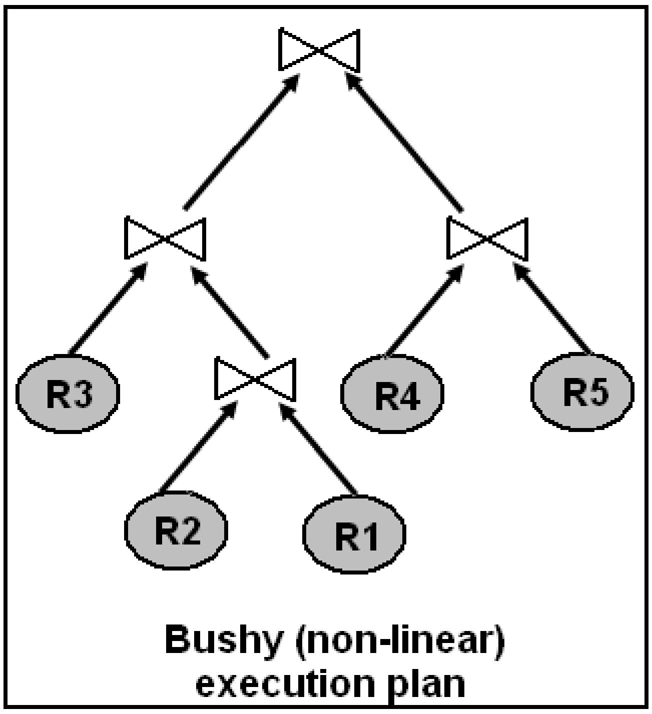
2. Right-deep query execution plan :-

- Right-deep execution plan have an application where there is a large main memory.

3. Linear tree query execution plan :-

- The combination of both Left-deep and Right-deep tree are also known as Linear tree. With linear tree, the relation of one side of each operation is based (stored) relation.
- Because however we need to examine entire inner relation for each tuple of the outer relation, inner relation must always be materialized.





4. Bushy (non-linear) tree query execution plan :-

- Bushy tree execution plans are the most general type of plan.
- They allow both inputs into binary operation to be intermediate results.

Algorithms

- Algorithm for JOIN operation :-

- **<u>Algorithm for Implementing</u> T ← R $\bowtie_{A=B}$ S**

Sort the tuples in R on A (* assume R has n tuples)
Sort the tuples in S on B (* assume S has m tuples)

```
set I ← 1, J ← 1;

while (I ≤ n) and (J ≤ m)
do { if R(I)[A] > S(J)[B]
        then set J ← J + 1;

    elseif R(I)[A] < S(J)[B]
        then set I ← I + 1;

    else { (* R(I)[A] = S(J)[B] so we output a matched tuple *)
            output the combined tuple <R(I), S(J)> to T;
            (*Output other tuples that match R(I), if any*)
            set L ← J + 1;

            while (L ≤ m) and (R(I)[A] = S(L)[B])
            do { output the combined tuple <R(I), S(J)> to T;
                set L ← L + 1;
               }

                (*Output other tuples that match S(J), if any*)
                set K ← I + 1;

                while (K ≤ n) and (R(K)[A] = S(J)[B])
                do { output the combined tuple <R(K), S(J)> to T;
                        set K ← K + 1;
                    }

                set I ← K, J ← L;
         }
   }
```

Algorithm for PROJECT Operation :→

- A PROJECT Operation π $_{\text{<attribute list>}}$ (R) is straightforward to implement if <attribute list> includes a key of relation R, because in this case the result of the operation will have the same number of tuples as R, but only the values for the attributes in <attribute list> in each tuple.
- If <attribute list> does not include the key of R, duplicate records must be eliminated. This is usually done by sorting the result of the operation and then eliminating duplicate tuples.

- Hashing can also be used to eliminating duplicates: as each records is hashed and inserted into a bucket of the hash file in memory, it is checked against those already in the bucket; if it is duplicate, it is not inserted.

- Algorithm for Implementing T ← π $_{\text{<attribute list>}}$ (R)

```
Create a tuple t[<attribute list>] in T' for each tuple t in R.

(*T' contain a projection results before duplicate elimination*)

if <attribute list> includes a key of R
        then T ← T';

else { sort the tuples in T';
          set I ← 1, j ← 2;

          while I ≤ n
                do { output the tuple T'[I] to T;

                     while T'[I] = T'[j] and j ≤ n
                        do j ← J + 1; (*eliminate duplicate*)
                       I ← j;
                       j ← I + 1;
                   }
     }

(*T contains the projection result after duplicate elimination*)
```

Algorithms for SET Operations :-

- Set operations-UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN are sometimes expensive to implement.
- In particular the CARTESIAN PRODUCT operation R X S is quit expensive because its result includes a record for each combination of records from R and S.
- Additionally the attributes of the result include all attributes of R and S.
- If R has N records and J attributes and

- S has M records and K attributes then
- The result relation will have N * M records and J * K attributes.
- Hence, it is avoided to implement the CARTESIAN PRODUCT operation.
- The other three set operations UNION, INTERSECTION and SET DIFFERENCE apply only to union-compatible relations, which have the same number of attributes and the same attribute domains.
- The way to implement these operations is to use variations of the **sort-merge technique:** the two relations are sorted to produce the result.
- We can implement the UNION operation, R ∪ S, by scanning and merging both sorted files, and whenever the same tuple exist in both relations, only one is kept in the merge result.
- For the INTERSECTION operation R ∩ S, we keep in the merge result only those tuples that appear in both relations.

- Algorithm for Implementing T ← R ∪ S

```
Sort the tuples in R and S using the same unique sort attributes;
set I ← 1, J ← 1;

while (I ≤ n) and (J ≤ m)
do { if R(I) > S(J)
            then { output S(J) to T;
                   set J ← J + 1;
                 }
     elseif R(I) < S(J)
            then { output R(I) to T;
                   set I ← I + 1;
                 }
     else J ← J + 1; (*R(I) = S(J) so we skip one of the duplicate tuple*)
   }

if(I ≤ n) then add tuples R(I) to R(n) to T;
if(J ≤ m) then add tuples S(J) to S(m) to T;
```

- **Algorithm for Implementing T ← R ∩ S**

```
Sort the tuples in R and S using the same unique sort attributes;
set I ← 1, J ← 1;

while (I ≤ n) and (J ≤ m)
```

```
do { if R(I) > S(J)
              then set J ← J + 1;
     elseif R(I) < S(J)
              then set I ← I + 1;
        else { output R(J) to T; (*R(I) = S(J) so we output the tuple*)
                   set I ← I + 1, J ← J + 1;
             }
   }
```

Algorithm for Implementing T ← R – S

```
Sort the tuples in R and S using the same unique sort attributes;
set I ← 1, J ← 1;

while (I ≤ n) and (J ≤ m)

do { if R(I) > S(J)
              then set J ← J + 1;
     elseif R(I) < S(J)
              then {  output R(I) to T;(*R(I) has no matching S(J) so we output the
                       tuple*)
                      set I ← I + 1;
                   }
        else set I ← I + 1, J ← J + 1;
       }
if(I ≤ n) then add tuples R(I) to R(n) to T;
```