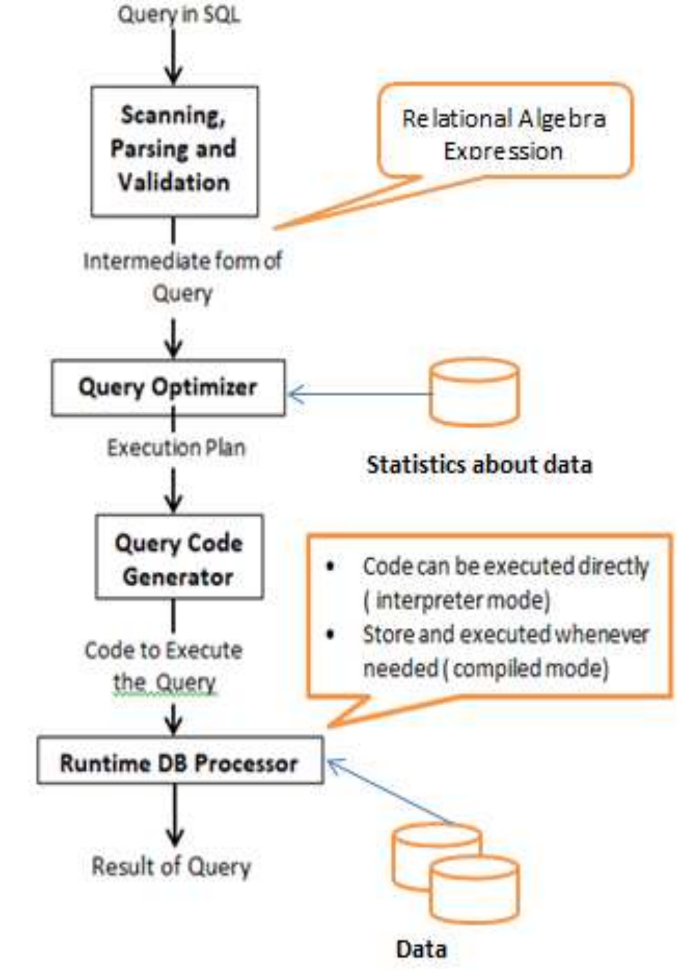
Process, Optimize, and Execute high-level query by DBMSs

Different steps of processing a high-level query

- A query expressed in SQL must first be scanned, parsed, and validated.
  - The scanner identifies tokens-such as SQL keywords, relation/attribute names
  - Parser checks the query syntax
  - Validator checks that all attribute and relation names in the schema.
- An internal representation of the query in RA is created, usually as a tree data structure called a **query tree** or a graph data structure called a **query graph**.
- The DBMS must then devise an execution plan for retrieving the result of the query from the database files. A query typically has many possible execution plans, and the process of choosing a suitable one is known as **query optimization**.

- The query optimizer module produces an execution plan, and the code generator generates the code to execute that plan. The runtime database processor has the task of running the query code.

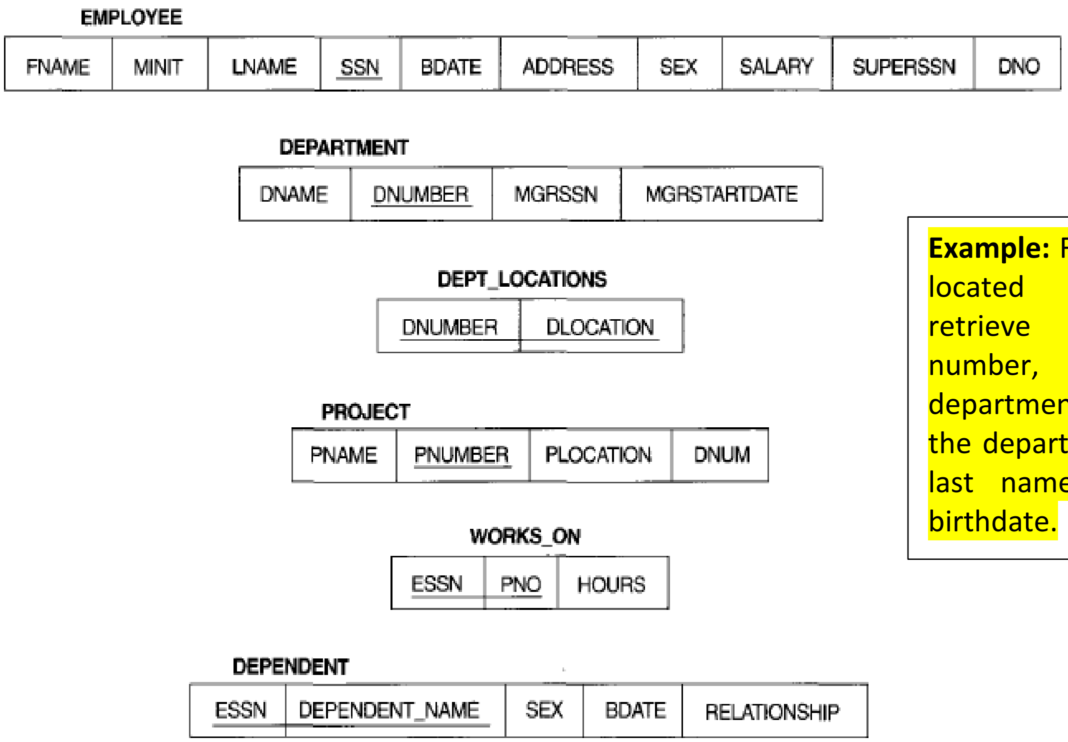

Query in SQL
Scanning, Parsing and Validation
Relational Algebra Expression
Intermediate form of Query
Query Optimizer
Statistics about data
Execution Plan
Query Code Generator
• Code can be executed directly (interpreter mode)
• Store and executed whenever needed (compiled mode)
Code to Execute the Query
Runtime DB Processor
Result of Query
Data

Notation for Query Trees and Query Graphs

A query tree is a tree data structure to represent a relational algebra expression with following characteristics:

- leaf nodes of the tree represents the input relations of the query
- internal nodes represent the relational algebra operations
- An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation.
- The execution terminates when the root node is executed and produces the result relation for the query.

To understand the construction of query tree consider following example query to be written based on following company schema.



EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATIONS

DNUMBER	DLOCATION

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORKS_ON

ESSN	PNO	HOURS

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP

Schema diagram for the COMPANY relational database schema.

Example: For every project located in 'Stafford', retrieve the project number, the controlling department number and the department manager's last name, address and birthdate.

We can write the above query in RA as shown below:

Project

Select

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

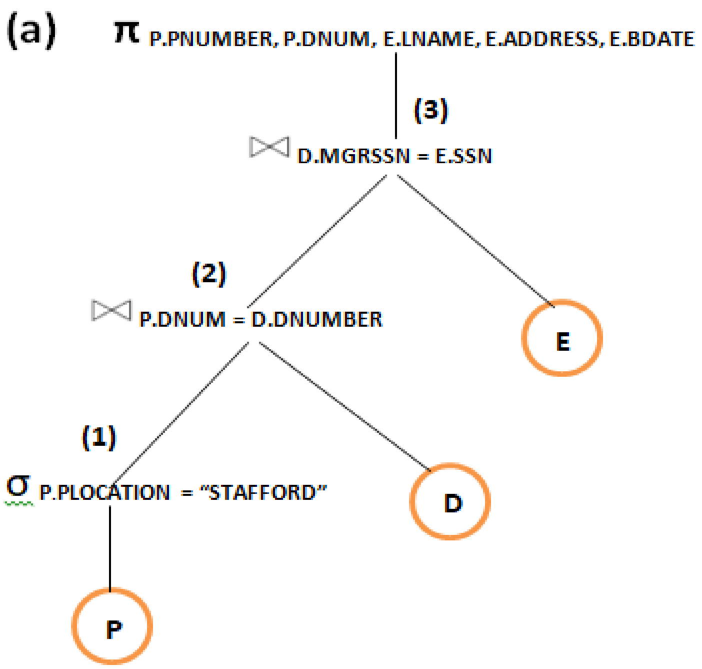
Join

Corresponding SQL query :

```
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.MGRSSN=E.SSN AND  P.PLOCATION='STAFFORD';
```

- **SELECT** clause would be mapping to projection,
- **WHERE** clause will be mapped to selection,
- **FROM** clause will be mapped to cross-product.
- And we assume **GROUP BY** and **HAVING** clauses as the extended operators in RA.

The WHERE clause should be in conjuctive normal form (CNF), that is, a collection of conjuncts connected by AND logical operator. A conjunct contains one or more terms connected by OR logical operator.


(a)
π P.PNUMBER, P.DNUM, E.LNAME, E.ADDRESS, E.BDATE
(3)
⋈ D.MGRSSN = E.SSN
(2)
⋈ P.DNUM = D.DNUMBER
E
(1)
σ P.PLOCATION = "STAFFORD"
D
P

In Figure (a) we represent the above RA expression.

- three relations PROJECT, DEPARTMENT, and EMPLOYEE are represented by leaf nodes P, D, and E.
- RA operations of the expression are represented by internal tree nodes.
- When this query tree is executed, the node marked (1) in Figure must begin execution before node (2) because some resulting tuples of operation (l) must be available before we can begin executing operation (2) and so on.

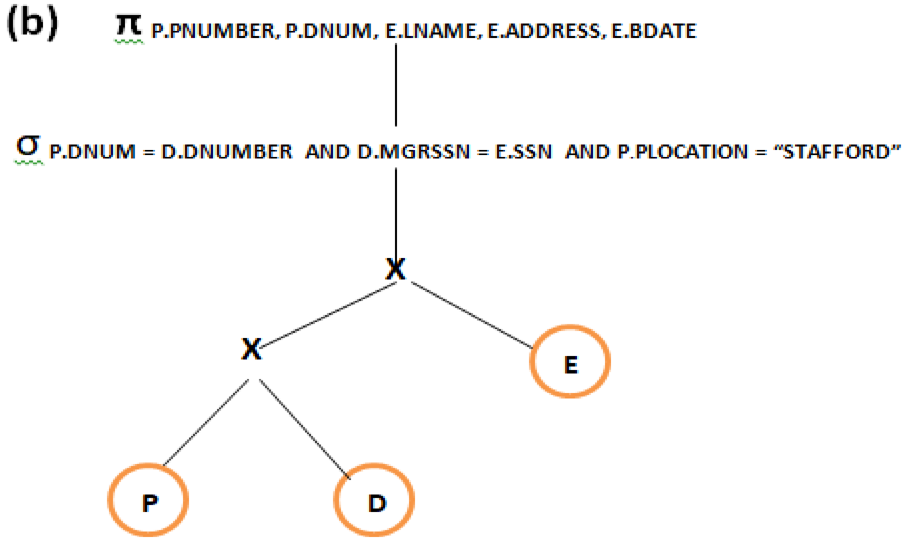
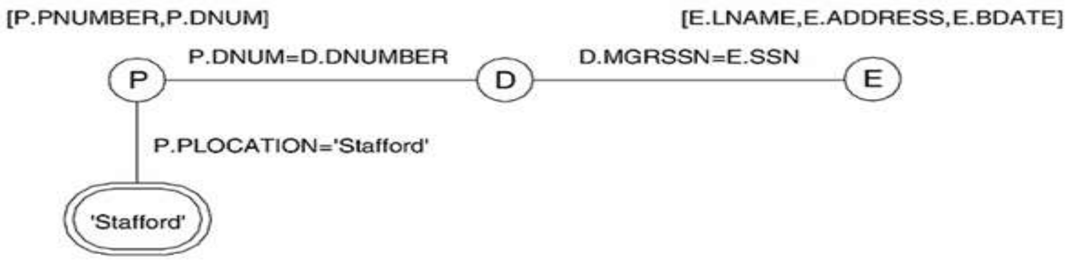

(b)
π P.PNUMBER, P.DNUM, E.LNAME, E.ADDRESS, E.BDATE
σ P.DNUM = D.DNUMBER AND D.MGRSSN = E.SSN AND P.PLOCATION = "STAFFORD"
X
X
E
P
D

FIGURE (b) represents Initial (canonical) query tree for SQL query.

As we can see, the query tree represents a specific order of operations for executing a query. A more neutral representation of a query is the query graph notation. Figure (c) shows the query graph for above query.



In Query Graph : • Nodes represent Relations. • Ovals represent constant nodes. Constant values, typically from the query selection conditions, are represented by constant nodes, which are displayed as double circles or ovals. • Edges represent Join & Selection conditions. • Attributes to be retrieved from relations represented in square brackets above each relation.	*Drawback* :- Does not indicate an order on which operations are performed. It is now generally accepted that query **trees are preferable** because, in practice, the query optimizer **needs to show the order of operations** for query execution, which is not possible in query graphs.

Query Optimizer Concepts

- User access DBMS using declarative language SQL without specifying which algorithm to use when writing the query but expect the performance optimization is guaranteed.
- As the choice of algorithm depends on the physical storage structure of relations it is also not feasible that the end-user specify the appropriate algorithm while writing the SQL. It would not be desirable to always use the same algorithm.

- Obviously, to optimize the performance for any query, RDBMS has to select the correct algorithm based on the query.

- The **query optimizer** (called simply the **optimizer**) is built-in database software that predicts which approach is the best without actually executing and comparing them.

- **Query Optimization:** A single query can be executed through different algorithms or re-written in different forms and structures. The query optimizer attempts to determine the most efficient way to execute a given query by considering the possible query plans.

- For making query processing more efficient the query optimizer performs following jobs :
  - selecting the appropriate indexes for acquiring data,
  - classifying predicates used in a query,
  - performing simple data reductions,
  - selecting access paths,
  - determining the order of a join,
  - performing predicate transformations and etc.

The performance of a query plan is determined largely by the order in which the tables are joined. For example, when joining 3 tables A, B, C of size 10 rows, 10,000 rows, and 1,000,000 rows, respectively, a query plan that joins B and C first can take several orders-of-magnitude more time to execute than one that joins A and C first. Most query optimizers determine **join** order via a **dynamic programming** algorithm.

Purpose of the Query Optimizer

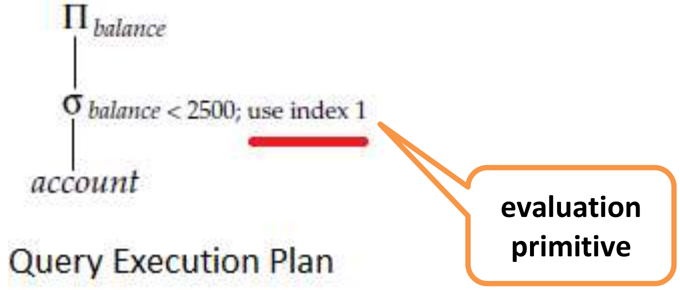
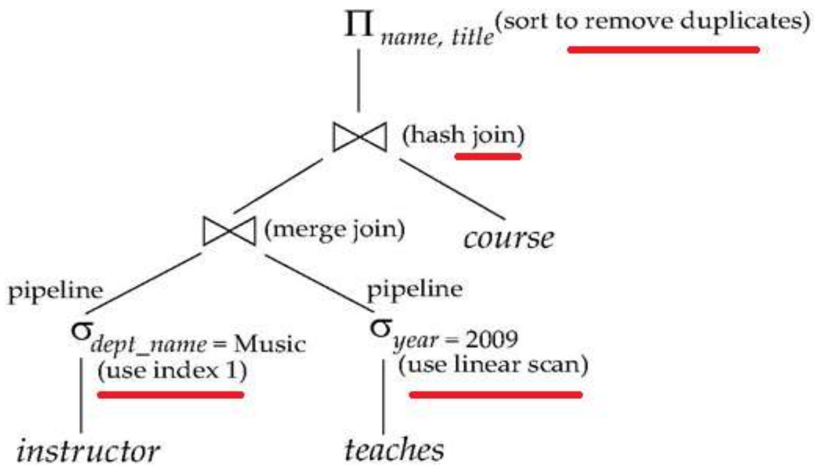
- Attempts to generate the most optimal execution plan for a SQL statement.
- The optimizer chooses the plan with the lowest cost among all considered candidate plans. The optimizer uses available statistics to calculate cost.
- A solid understanding of the optimizer is essential for SQL tuning.

There are **two approaches to optimization**. They are as follows:

- **Cost based**: This was developed by IBM. The optimizer estimates the cost of each processing method of the query and chooses the one with the lowest estimate. Presently, most systems use this.
- **Heuristic**: Rules are based on the form of the query. Oracle used this at one point. Presently, no system uses this.

Execution Plans (Query Execution Plan)

The relational-algebra representation of a query specifies only **partially** how to evaluate a query; there are usually several ways to evaluate relational-algebra expressions. As an illustration, consider the query

select *balance* **from** *account* **where** *balance* < 2500 This query can be translated into either of the following relational-algebra expressions: $\sigma_{balance<2500}(\Pi_{balance}(account))$ $\Pi_{balance}(\sigma_{balance<2500}(account))$	 $\Pi_{balance}$ $\sigma_{balance < 2500; \text{ use index 1}}$ account evaluation primitive Query Execution Plan
 $\Pi_{name, title}$ (sort to remove duplicates) (hash join) (merge join) course pipeline pipeline $\sigma_{dept\_name = \text{Music}}$ (use index 1) $\sigma_{year = 2009}$ (use linear scan) instructor teaches	Each relational-algebra operation can be executed by one of several different algorithms. For example we can search every tuple in *account* to find tuples with balance less than 2500 (previous example). If a B+-tree index is available on the attribute *balance*, we can use the index instead to locate the tuples.

To specify **fully** how to evaluate a query,

- We need to provide the relational algebra expression as well as **annotate it with instructions** (**algorithm/index** to be used) specifying how to evaluate each operation.
- A relational-algebra operation annotated with instructions on how to evaluate it is called an **evaluation primitive**.

A sequence of evaluation primitive operations that can be used to evaluate a query is a **query execution plan** or **query-evaluation plan**. Above figure illustrates an evaluation plan for our example query, in which a particular index (denoted in the figure as "index 1") is specified for the selection operation.

An execution plan for a relational algebra expression represented as a query tree includes information about the access methods available for each relation as well as the algorithms to be used in computing the relational operators represented in the tree.

The **query-execution engine** takes that query-evaluation plan, executes that plan, and returns the answers to the query.

Measures of Query Cost

In DBMS, the cost involved in executing a query can be measured by considering the number of different resources that are listed below.

- **I/O cost** : the cost of accessing index and data pages from disk.
- **Processing cost** : CPU time to execute the query
- **Cost of communication** : in a distributed or parallel database system

In large database systems, however, **disk accesses** (which we measure as the number of transfers of blocks from disk) are usually the most important cost, since disk accesses are slow compared to in-memory operations. Therefore, most people consider the disk-access cost a reasonable measure of the cost of a query-evaluation plan.

Real-life query optimizers also take CPU costs into account when computing the cost of an operation. For simplicity we ignore these details.

In the cost estimates we ignore the cost of writing the final result of an operation back to disk. These are taken into account separately where required.

The costs of all the algorithms depend on the size of the buffer in main memory. In the best case, all data can be read into the buffers, and the disk does not need to be accessed again. In the worst case, we assume that the buffer can hold only a few blocks of data—approximately one block per relation. When presenting cost estimates, we generally assume the worst case.

Cost of query depends on various factors like Query Algorithms used, Approach of evaluation used (Materialization, Pipelining) etc. We will now briefly discuss those factors:

- **QUERY ALGORITHMS**

Queries are ultimately reduced to a number of file scan operations on the underlying physical file structures. For each relational operation, there exist several different access paths to the particular records needed. The query execution engine can have a multitude of specialized algorithms designed to process particular relational operation and access path combinations. Here we will list some of the algorithm without discussing the details.

A. Selection Algorithms	The *Select* operation must search through the data files for records meeting the selection criteria. Following are some examples of simple (one attribute) selection algorithms :
• Linear search	Every record from the file is read and compared to the selection criteria. The execution cost for searching on a non-key attribute = b_r, where b_r = no. of blocks in the file representing relation *r*. On a key attribute, the average cost = $b_r/2$, with a worst case of b_r.
• Binary search	A binary search, on equality, performed on a primary key attribute has a worst-case cost of $\lceil \log (b_r) \rceil$. This can be considerably more efficient than the linear search, for a large number of records.
• Search using a primary index on equality (With a B+tree index)	An equality comparison on a key attribute will have a worst -case cost of the height of the tree plus one to retrieve the record from the data file. An equality comparison on a non-key attribute will be the same except that multiple records may meet the condition, in which case, we add the number of blocks containing the records to the cost.
• Search using a primary index on comparison	When the comparison operators (<, ≤, >, ≥) are used to retrieve multiple records from a file sorted by the search attribute, the first record satisfying the condition is located and the total blocks before (<, ≤) or after (>, ≥) is added to the cost of locating the first record.
• Search using a secondary index on equality	Retrieve one record with an equality comparison on a key attribute; or retrieve a set of records on a non-key attribute. For a single record, the cost will be equal to the cost of locating the search key in the index file plus one for retrieving the data record. For multiple records, the cost will be equal to the cost of locating the search key in the index file plus one block access for each data record retrieval, since the data file is not ordered on the search attribute.
B. Join Algorithms	The join algorithm can be implemented in a different ways. In terms of disk accesses, the join operations can be very expensive, so implementing and utilizing efficient join algorithms is important in minimizing a query's execution time. The following are 4 well - known types of join algorithms:

- Nested-Loop Join**:** It consists of a inner for loop nested within an outer for loop.
- Index Nested-Loop Join**:** This algorithm is the same as the Nested-Loop Join, except an index file on the inner relation's join attribute is used versus a data-file scan on each index lookup in the inner loop is essentially an equality selection for utilizing one of the selection algorithms Let *c* be the cost for the lookup, then the worst -case cost for joining r and s is *br* + *nr* * *c*.
- Sort –Merge Join**:** This algorithm can be used to perform natural joins and equi-joins and requires that each relation be sorted by the common attributes between them.
- Hash Join: The hash join algorithm can be used to perform natural joins and equi-joins. The hash join utilizes two hash table file structures (one for each relation) to partition each relation's records into sets containing identical hash values on the join attributes. Each relation is scanned and its corresponding hash table on the join attribute values is built.

C. Indexes Role	The execution time of various operations such as select and join can be reduced by using indexes. There are various types of index such as Dense , Sparse, Secondary, Multi-Level, Custering, B+tree are used in DBMS. Each has its role to play in reducing execution time and overhead.

- **MATERIALIZATION**

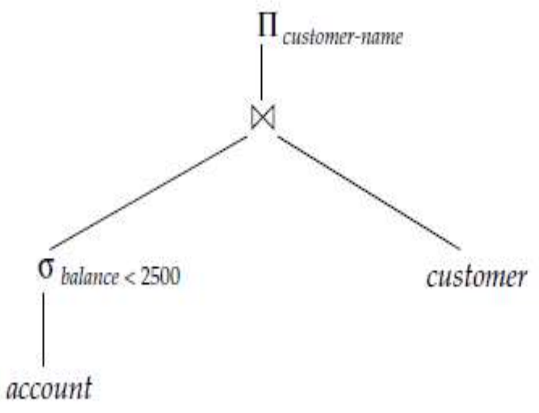
Consider the expression and the pictorial representation of expression as shown in figure.

In **materialized evaluation** the results of each intermediate operation are created (materialized) and then are used for evaluation of the next-level operations.

If we apply the materialization approach, we start from the lowest-level operations in the expression (at the bottom of the tree). The inputs to the lowest-level operations are relations in the database. We execute these operations by the appropriate algorithms, and we store the results in temporary relations. We can use these temporary relations to execute the operations at the next level up in the tree, where the inputs now are either temporary relations or relations stored in the database.

By repeating the process, we will eventually evaluate the operation at the root of the tree, giving the final result of the expression.

$$\Pi_{customer\text{-}name}\,(\sigma_{balance<2500}\,(account) \bowtie customer)$$



Pictorial representation of an expression.

The cost of a materialized evaluation = costs of all the operations + the cost of writing the intermediate results to disk.

We assume that the records of the result accumulate in a buffer, and, when the buffer is full, they are written to disk.

- **PIPELINING**

To improve query-evaluation efficiency, we can reduce the number of temporary files by combining several relational operations into a pipeline of operations. Here the results of one operation are passed along to the next operation in the pipeline and this type of evaluation is called **pipelined evaluation**. Combining operations into a pipeline eliminates the cost of reading and writing temporary relations.

For example, consider the expression $(\Pi_{a1,a2}(r \bowtie s))$.

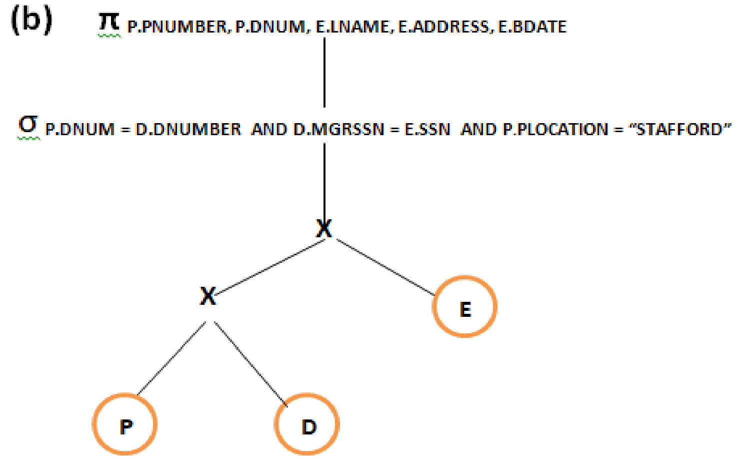
When the join operation generates a tuple of its result, it passes that tuple immediately to the project operation for processing. So we avoid creating the intermediate result.

Heuristic Optimization of Query Trees

In general, many different relational algebra expressions - and hence many different query trees-can be equivalent; that is, they can correspond to the same query.

The **query parser** will typically generate a standard initial query tree to correspond to an SQL query, without doing any optimization.

An initial tree is shown in the Figure.


(b)
π P.PNUMBER, P.DNUM, E.LNAME, E.ADDRESS, E.BDATE
σ P.DNUM = D.DNUMBER AND D.MGRSSN = E.SSN AND P.PLOCATION = “STAFFORD”
X
X
E
P
D

- The CARTESIAN PRODUCT of the relations specified in the FROM clause is first applied; then the selection and join conditions of the WHERE clause are applied, followed by the projection on the SELECT clause attributes.
- Such a **canonical query tree** represents a relational algebra expression that is *very inefficient if executed directly,* because of the CARTESIAN PRODUCT (X) operations. For example, if the PROJECT, DEPARTMENT, and EMPLOYEE relations had record sizes of 100, 50, and 150 bytes and contained 100, 20, and 5000 tuples, respectively, the result of the CARTESIAN PRODUCT would contain 10 million tuples of record size 300 bytes each.
- It is now the job of the **heuristic query optimizer** to transform this initial query tree into a final query tree that is efficient to execute.

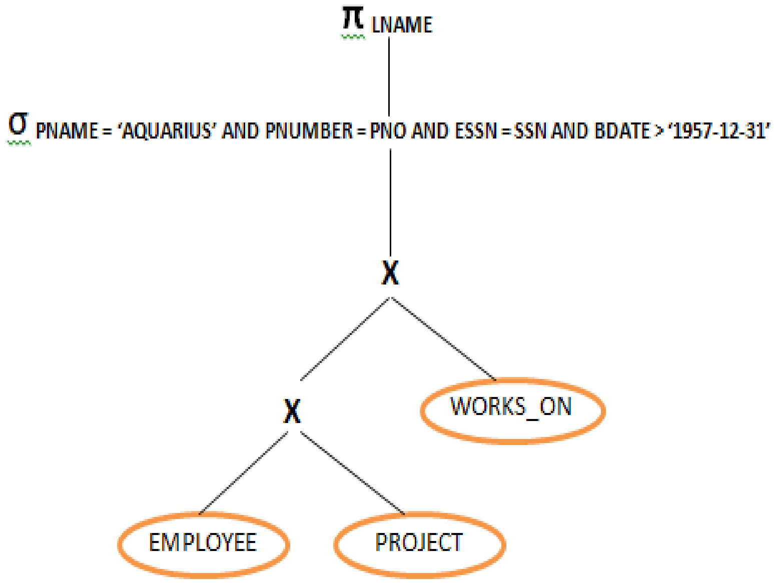
- The heuristic query optimization utilizes **equivalence rules** among relational algebra to transform the initial tree into the final, optimized query tree.

We first discuss informally how a query tree is transformed by using heuristics. Then we discuss general transformation rules and show how they may be used in an algebraic heuristic optimizer.

Example of Transforming a 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';
```

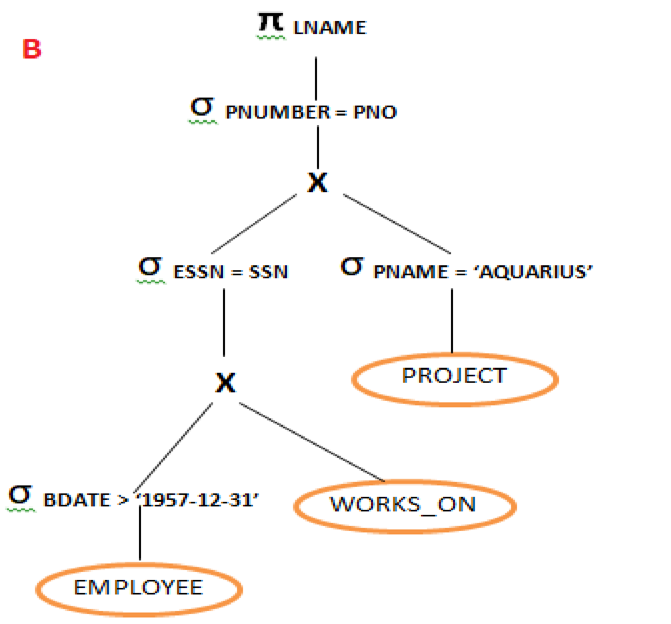

π LNAME
σ PNAME = 'AQUARIUS' AND PNUMBER = PNO AND ESSN = SSN AND BDATE > '1957-12-31'
X
X
WORKS_ON
EMPLOYEE
PROJECT

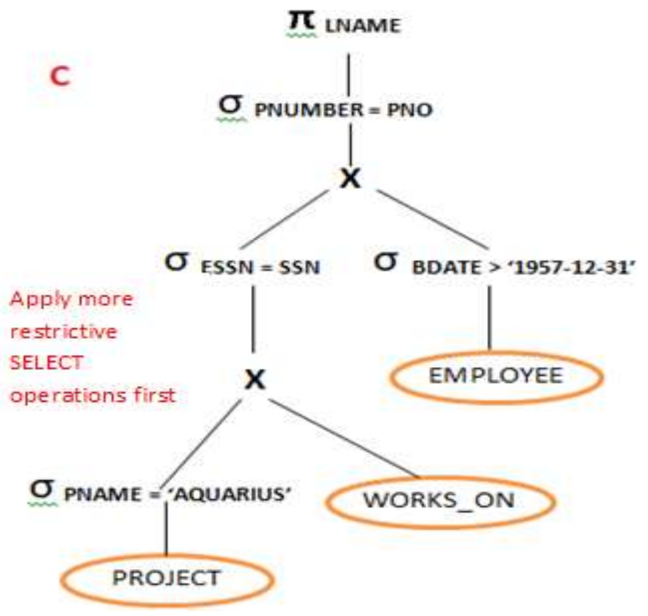
After analyzing the initial query tree as shown we gathered following points :

- First creates a very large file containing the CARTESIAN PRODUCT of the entire EMPLOYEE, WORKS_ON, and PROJECT files.
- Only one record from the PROJECT relation for the 'Aquarius' project is needed.
- Only the EMPLOYEE records whose date of birth is after '1957-12-31' is needed.
- So we need improvement.

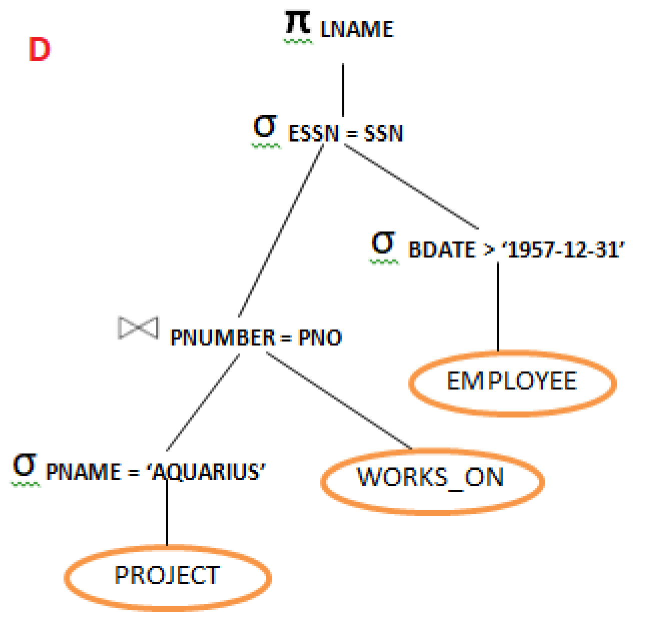
A initial query tree can be transformed step by step into another query tree that is more efficient to execute. To optimize any query the query optimizer follows the transformation rules that preserve this equivalence. We are examining these transformation below (Fig B-E)

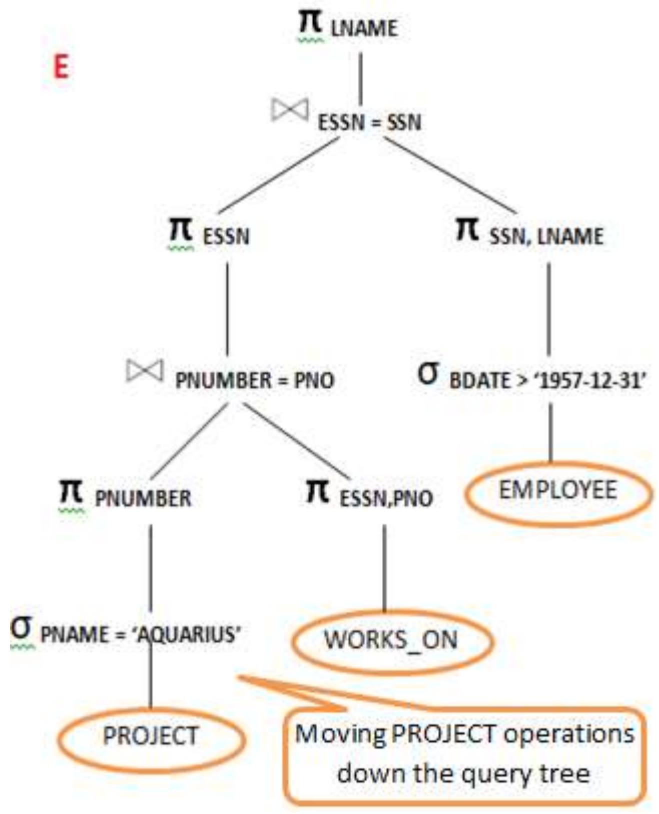
In figure B below we apply the SELECT operations to reduce the number of tuples that appear in the CARTESIAN PRODUCT.	In figure C below, a further improvement is achieved by switching the positions of the EMPLOYEE and PROJECT relations in the tree. This uses the information that **PNUMBER is a key attribute** of the project relation, and hence the SELECT operation on the PROJECT relation will retrieve a single record only.





Apply more restrictive SELECT operations first





Moving PROJECT operations down the query tree

- In figure D, we can further improve the query tree by replacing any CARTESIAN PRODUCT operation that is followed by a join condition with a JOIN operation.

- In figure E, we improve further by keeping only the attributes needed by subsequent operations in the intermediate relations, by including PROJECT (π) operations as early as possible in the query tree.

	• This reduces the attributes (columns) of the intermediate relations, whereas the SELECT operations reduce the number of tuples (records).

As shown in the above example a query tree can be transformed step by step into another query tree that is more efficient to execute. However, we must make sure that the transformation steps always lead to an equivalent query tree.

General Transformation Rules for Relational Algebra Operations.

There are **many rules** for transforming relational algebra operations into equivalent ones. We are not going to discuss those rules. Some simple rules are shown in following table. In next section we will outline a Heuristic Algebraic Optimization Algorithm which will follow those rules during optimization.

Some simple Equivalence Rules are

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.	$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
2. Selection operations are commutative.	$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
3. Only the last in a sequence of projection operations is needed, the others can be omitted.	$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{Ln}(E))\ldots)) = \Pi_{L_1}(E)$
4. Selections can be combined with Cartesian products and theta joins.	a. $\sigma_{\theta}(E_1 \text{ X } E_2) = E_1 \quad {}_{\theta} E_2$ **b.** $\sigma_{\theta 1}(E_1 \quad {}_{\theta 2} E_2) = E_1 {}_{\theta 1 \wedge \theta 2} E_2$
5. Theta-join operations (and natural joins) are commutative	$E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$
6. (a) Natural join operations are associative: (b) Theta joins are associative:	$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$ $(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$ **where θ_2 involves attributes from only E_2 and E_3.**

A **theta join** is a join that links tables based on a relationship other than equality between two columns. A theta join could use any operator other than the “equal” operator.

Outline of a Heuristic Algebraic Optimization Algorithm.

The algorithm will lead to transformations similar to those discussed in our example above. The steps of the algorithm are as follows:

1. Break up any SELECT operations with conjunctive conditions (OR conjuncts connected by AND) 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 selection operations down the query tree for the earliest possible execution as is permitted by the attributes involved in the select condition.
3. Execute first those selections and join operations that will produce the smallest relations.
4. Replace Cartesian product operations that are followed by a selection condition by join operations. Note that a Cartesian product is acceptable in some cases-for example, if each relation has Only a single tuple.
5. 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 means
     - either the ones that produce a relation with the fewest tuples or with the smallest absolute size.
     - Or one with the smallest selectivity (lesser % of records satisfy the predicate); this is more practical because estimates of selectivity are often available in the DBMS catalog.

   - Second, make sure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operations;
     for example, if the two relations with the most restrictive SELECT do not have a direct join condition between them, it may be desirable to change the order of leaf nodes to avoid Cartesian products.

5. 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.

Note on Selectivity

Predicate : A predicate is a filtering conditions that selects a certain number of values from a set of values from a value set. Example of predicate may be age=25 or country = India.
Selectivity represents a ratio of number of elements from a set that qualify a selection predicate. Speaking reverse, the selectivity of predicate indicates how many rows from a row set will qualify the predicate.

Selectivity is specified in % or in value range 0.0 to 1.

- Selectivity of 0 - means no rows are qualified
- Selectivity of 1.0 – means all rows are qualified.

Assume the total number of people = 200, out of which 20 people are Indian. So we can say the selectivity of (nationality = ‘Indian’) = 10% or 0.1

Selectivity plays a major role in cost estimation logic. When a query has many filtering conditions, the selectivity of individual condition has to be combined to get the resultant selectivity. This decided the resultant number of result rows.

The selectivity is what goes with the cardinality concept.

The "cardinality" refers to the number of "distinct" values, for a column "SEX" (assuming only two values "male" and "female") cardinality = 2, no matter how many rows you have in that table.
The selectivity is the "number of rows" / "cardinality", so if you have 10K customers, and search for all "female", you have to consider that the search would return 10K/2 = 5K rows, so a very "bad" selectivity.

The column for the primary key on the other side is "unique", and hence the cardinality is equal to the number of rows, by definition. So, the selectivity for searching a value in that column will be 1, by definition, which is the best selectivity possible

Statistical Information for Cost Estimation

The statistics mentioned here are optimizer statistics, which are created for the purposes of query optimization and are stored in the data dictionary. These statistics should not be confused with performance statistics.

DBMS statistics are a collection of data that describe more details about the database and the objects in the database. It tells us the size of the tables, the distribution of values within columns, and other important information so that SQL statements will always generate the best execution plans. Some of the typical statistics generated by DBMS are shown below.

• Table statistics	• Number of rows • Number of blocks • Average row length	• n_r: number of tuples in a relation *r.* • b_r: number of blocks containing tuples of *r.* • l_r: size of a tuple of *r.* • f_r: blocking factor of *r* — i.e., the number of tuples of *r* that fit into one block. • If tuples of *r* are stored together physically in a file, then: $b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$
• Column statistics	• Number of distinct values (NDV) in column • No. of nulls in column • Data distribution (histogram)	• *V(A, r):* number of distinct values that appear in *r* for attribute *A;* same as the size of $\prod_A(r)$.
• Index statistics	• Number of leaf blocks • Levels • Clustering factor	
• System statistics	• I/O performance and utilization • CPU performance and utilization	

- If new objects are created, or the amount of data in the database changes the statistics will no longer represent the real state of the database. As cost based optimization uses these statistics may not able to suggest the optimal execution plan.

- DBA's job is to make sure the numbers are good enough for that optimizer to work properly.
- The optimizer statistics are stored in the data dictionary. They can be viewed using data dictionary views. Only statistics stored in the dictionary itself have an impact on the cost-based optimizer.
- The new statistics take effect the next time the SQL statement is parsed.
- Because the objects in a database can be constantly changing, statistics must be regularly updated so that they accurately describe these database objects. Statistics are maintained automatically by DBMS or we can maintain the optimizer statistics manually using DBMS supplied package.

- The recommended approach to gathering statistics is to allow DBMS to automatically gather the statistics. Oracle gathers statistics on all database objects automatically and maintains those statistics in a regularly-scheduled maintenance job. This job is created automatically at database creation time and is managed by the Scheduler. The Scheduler runs this job when the maintenance window is opened. By default, the maintenance window opens every night from 10 P.M. to 6 A.M. and all day on weekends.

- Automated statistics collection eliminates many of the manual tasks associated with managing the query optimizer, and significantly reduces the chances of getting poor execution plans because of missing or stale statistics.

- Automatic statistics gathering should be sufficient for most database objects which are being modified at a moderate speed. However, there are cases where automatic statistics gathering may not be adequate. Because the automatic statistics gathering runs during an overnight batch window, the statistics on tables which are significantly modified during the day may become stale. For example : For tables which are being bulk-loaded, the statistics-gathering procedures should be run on those tables immediately following the load process, preferably as part of the same script or job that is running the bulk load. In this case manual statics gathering is needed.
- Another area in which statistics need to be manually gathered is the system statistics. These statistics are not automatically gathered.

Cost-Based Optimization

In cost based optimization approach the DBMS optimizes each SQL statement based on statistics collected about the accessed data. The optimizer determines the optimal plan for a SQL statement by examining multiple access methods, such as full table scan or index scans, different join methods such as nested loops and hash joins, different join orders, and possible transformations.

For a given query and environment, the optimizer assigns a relative numerical cost to each step of a possible plan, and then factors these values together to generate an overall cost estimate for the plan.

After calculating the costs of alternative execution plans, the optimizer chooses the plan with the lowest cost estimate. For this reason, the optimizer is sometimes called the **cost-based optimizer (CBO)** to contrast it with the legacy rule-based optimizer (RBO).

Steps in cost-based query optimization

1. Generate logically equivalent expressions using **equivalence rules**
2. Annotate resultant expressions to get alternative query plans
3. Choose the cheapest plan based on **estimated cost**
4. Estimation of plan cost based on:
    - Statistical information about relations.
    - Statistical estimation for intermediate results

- Cost formulae for algorithms, computed using statistics

Generate logically equivalent expressions

This is an important step in any optimizer. We are briefly discussing only following basic approaches to choosing an evaluation plan:

1. **Use statistics on best join order** to reduce the examination of number of join order

A **cost-based optimizer** generates a range of query-evaluation plans from the given query by using the equivalence rules, and chooses the one with the least cost. For a complex query, the number of different query plans that are equivalent to a given plan can be large.

As an illustration, consider the expression $r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$
where the joins are expressed without any ordering. With n = 3, there are 12 different join orderings.

In general, with n relations, there are (2(n -1))!/(n-1)! different join orders. For joins involving small numbers of relations, this number is acceptable;
for example, n = 5, the number is 1680; n = 7, the number is 665280; n = 10, the number is greater than 17.6 billion! . As n increases, this number rises quickly.

Luckily, it is not necessary to generate all of the expressions equivalent to a given expression. For example, suppose we want to find the best join order of the form
$(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$ if we consider all possible join order we have to examine 144 join order. However, once we have found the **best join order** for the subset of relations {r1, r2, r3}, we can use that order for further joins with r4 and r5, and can ignore all costlier join orders of (r1,r2,r3). Thus, instead of 144 choices to examine, we need to examine only 12 + 12 choices.

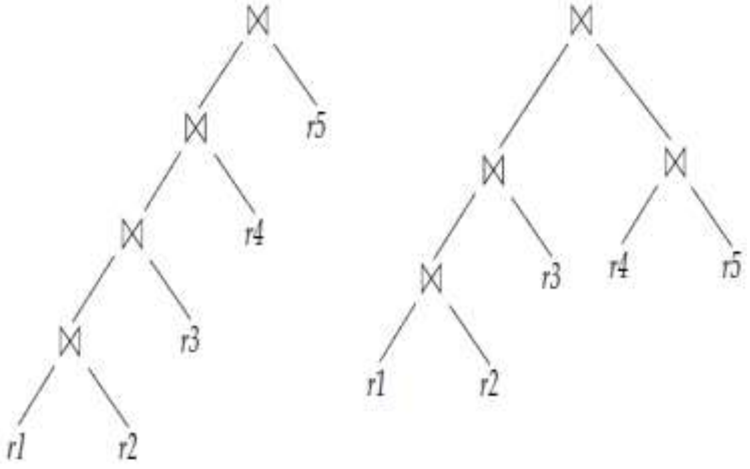
2. **Use Heuristics approach**

A drawback of cost-based optimization is the cost of optimization itself. Hence, many systems use heuristics to reduce the number of choices that must be made in a cost-based fashion. The same heuristic approach as discussed earlier in Heuristic Optimization is used. This approach is called heuristic because it usually, but not always, helps to reduce the cost. The steps in this approach can be summarized as

- an initial query-tree representation is transformed in such a way that the operations that reduce the size of intermediate results are applied first
- restructure the tree so that the system performs the most restrictive selection and join operations before other similar operations.

An evaluation plan includes not only the relational operations to be performed, but also the indices to be used, the order in which tuples are to be accessed, and the order in which the operations are to be performed. The access-plan –selection phase of a heuristic optimizer chooses the most efficient strategy for each operation.

Most practical query optimizers combine elements of both approaches. For example, certain query optimizers, such as the System R optimizer, do not consider all join orders, but rather restrict the search to particular kinds of join orders. The System R optimizer considers only those join orders where the right operand of each join is one of the initial relations r1,...,rn. Such join orders are called **left-deep join orders**. Left-deep join orders are particularly convenient for pipelined evaluation, since the right operand is a stored relation, and thus only one input to each join is pipelined.



(a) Left-deep join tree (b) Non-left-deep join tree

The time it takes to consider all left-deep join orders is O(n!),which is much less than the time to consider all join orders. With the use of dynamic programming optimizations, the System R optimizer can find the best join order in time $O(n2^n)$. Contrast this cost with the $O(3^n)$ time required to find the best overall join order. The System R optimizer uses heuristics to push selections and projections down the query tree.

Query optimization approaches that integrate heuristic selection and the generation of alternative access plans have been adopted in several systems. The cost-based optimization techniques described here are used for each block of the query separately.

Some systems even choose to use only heuristics, and do not use cost-based optimization at all.

- The System R/Starburst: dynamic programming on left-deep join orders. Also uses heuristics to push selections and projections down the query tree.
- DB2, SQLServer, Oracle are cost-based optimizers
  - SQLserver is transformation based, also uses dynamic programming.
- MySQL optimizer is heuristics-based (rather weak)