Marwadi University

Department of CE/IT

DBMS:Database Management System

Data Storage and Query Processing

Unit no : 5
DBMS (01CE3201)

Prof. Mitesh Patel

- File Structure
- Indexing
- Query Processing and Query Optimization

Marwadi University

Department of CE/IT

File Structure

- Relative data and information is stored collectively in file formats.
- A file is a sequence of records stored in binary format.
- A disk drive is formatted into several blocks that can store records. File records are mapped onto those disk blocks.
- Types of File Organization to organize file records:
  - Heap File Organization
  - Sequential File Organization
  - Hash File Organization
  - Clustered File Organization

File Structure

- **Heap File Organization**

  - When a file is created using Heap File Organization, the Operating System allocates memory area to that file without any further accounting details.
  - File records can be placed anywhere in that memory area. It is the responsibility of the software to manage the records.
  - Heap File does not support any ordering, sequencing, or indexing on its own.

File Structure

➢ Sequential File Organization

- Every file record contains a data field (attribute) to uniquely identify that record.
- In sequential file organization, records are placed in the file in some sequential order based on the unique key field or search key.
- Practically, it is not possible to store all the records sequentially in physical form.

File Structure

- **Hash File Organization**

  - Hash File Organization uses Hash function computation on some fields of the records.
  - The output of the hash function determines the location of disk block where the records are to be placed.

File Structure

- **Clustered File Organization**
  - Clustered file organization is not considered good for large databases.
  - In this mechanism, related records from one or more relations are kept in the same disk block, that is, the ordering of records is not based on primary key or search key.

Indexing

- Indexes are the lookup tables that the database search engine can use to speed up retrieval of data.
- A database index is a data structure that improves the speed of data retrieval operations on a database table.
- An index in a database is similar to an index in provided in any book.
- Indexing can be used with only **Select** Operation.
- Syntax to create an index:

  - create index index_name on Table(col1, Col2, ..., Col N);

  => create index index_acc on Account(acc_no, name, city);
- Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is executed.
- It is a data structure technique which is used to quickly find and access the data in a database.

Indexing

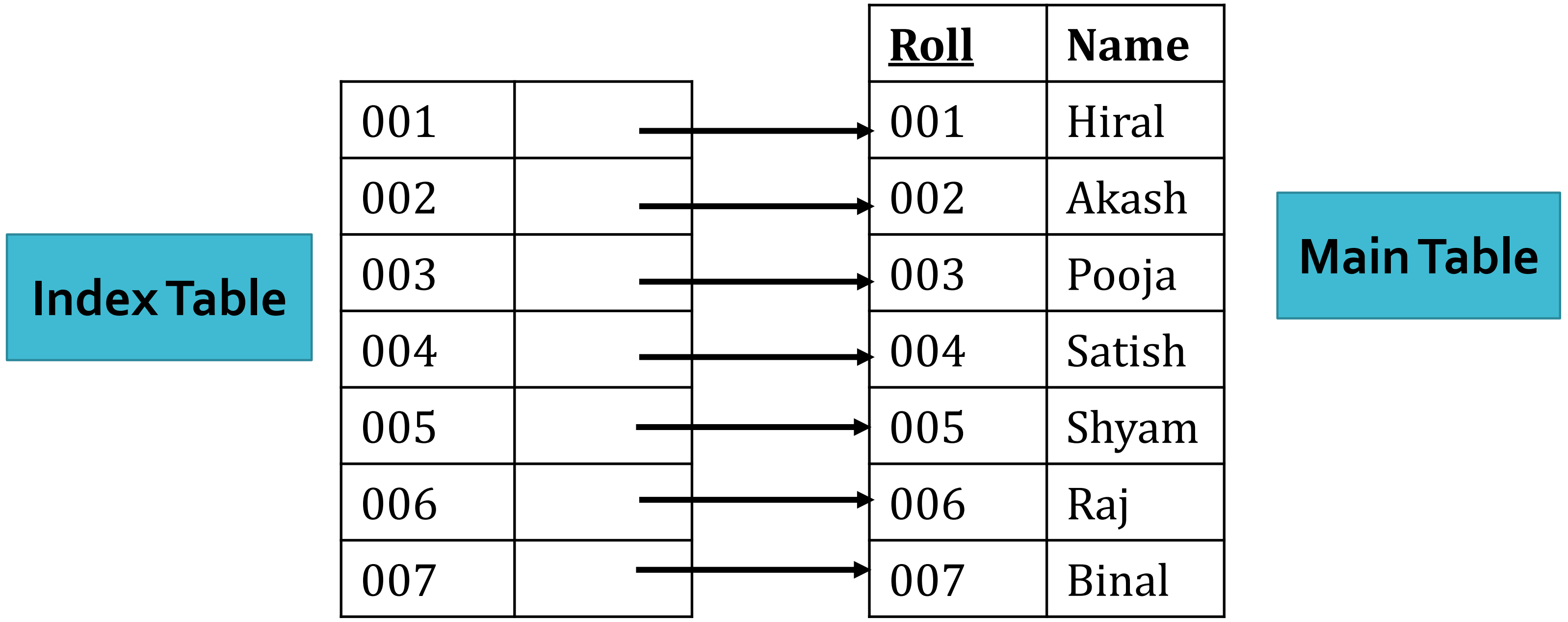
- Indexes are created using 2 database columns:
  - ✓ **Search-Key**: Generally it has the values of Primary Key column of the table stored in the sorted form for faster access.
  - ✓ **Pointer**: Contains the address of the disk where the values can be found for a given request.

- The indexing can be mainly classified into:
  - ✓ Primary Index
    - Dense Index
    - Sparse Index
  - ✓ Secondary Index
  - ✓ Clustering Index

Primary Index

- If the index is created on the **primary key** of the table, then it is known as primary index.
- As primary keys are stored in sorted form, the performance of the data access is faster.

- **Dense Index**
  - ✓ There is an index record for every search key value in the database.
  - ✓ This makes searching faster. Number of records in Index table and main table are same.



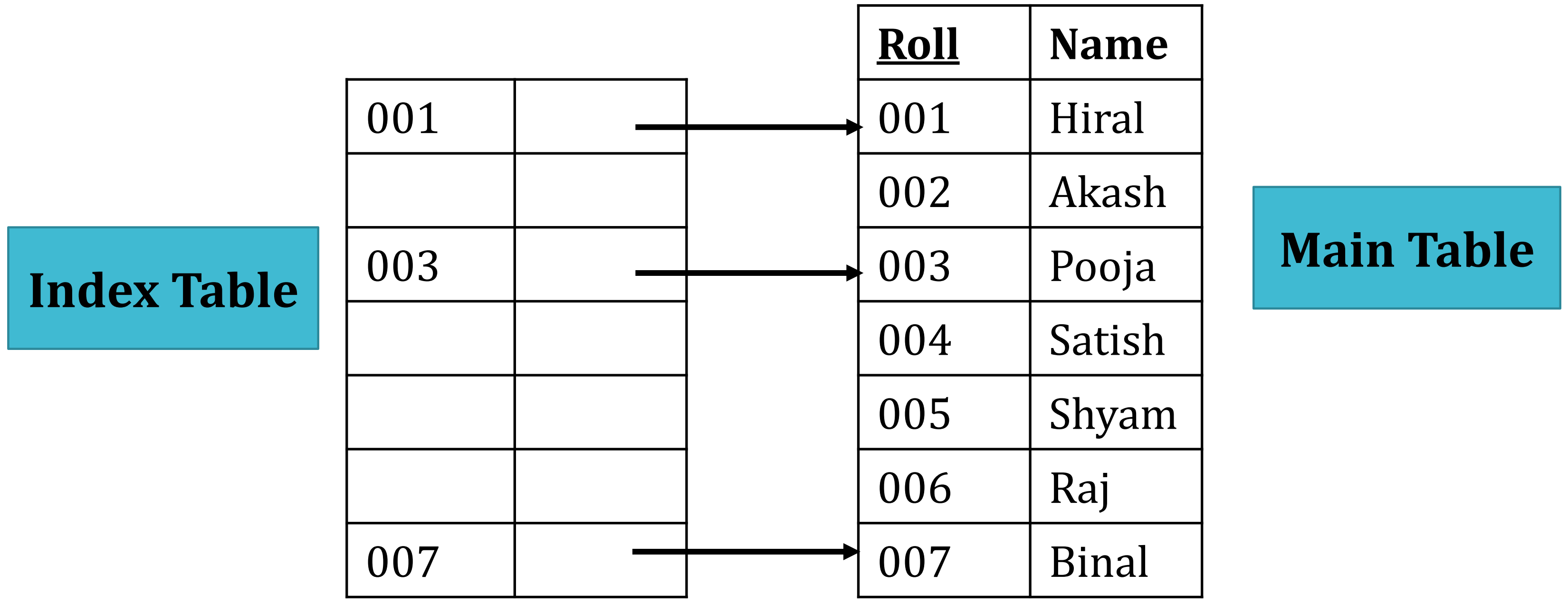
001	
002	
003	
004	
005	
006	
007	

<u>**Roll**</u>	**Name**
001	Hiral
002	Akash
003	Pooja
004	Satish
005	Shyam
006	Raj
007	Binal

Primary Index

➢ **Sparse Index**

- ✓ An index record for every search key value in the database is not created.
- ✓ The index remains for few items and we find the index value that is less than or equal to the value to be searched.
- ✓ Then we apply linear search to get the actual data from the database.

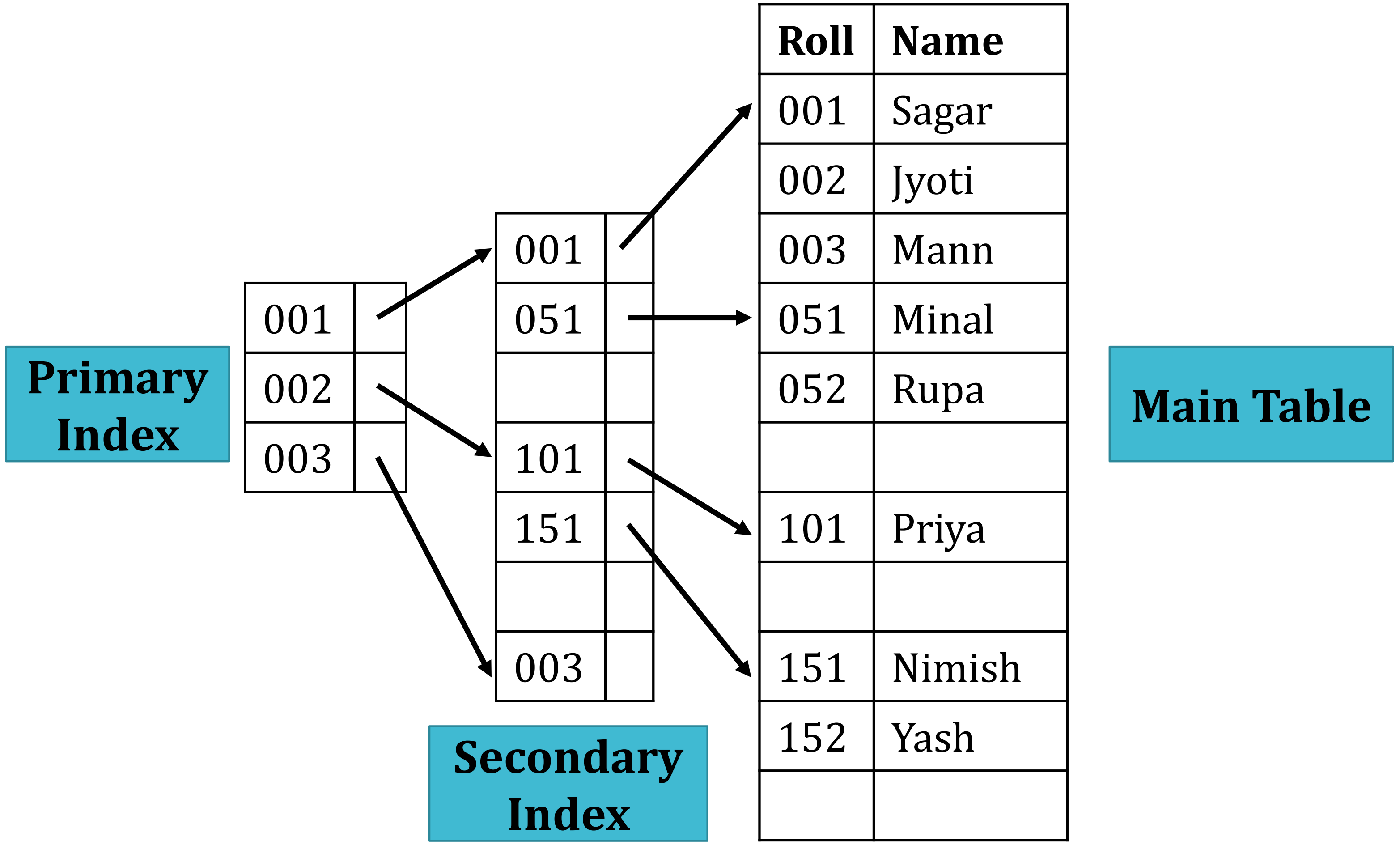


001	
003	
007	

Roll	Name
001	Hiral
002	Akash
003	Pooja
004	Satish
005	Shyam
006	Raj
007	Binal

Secondary Index

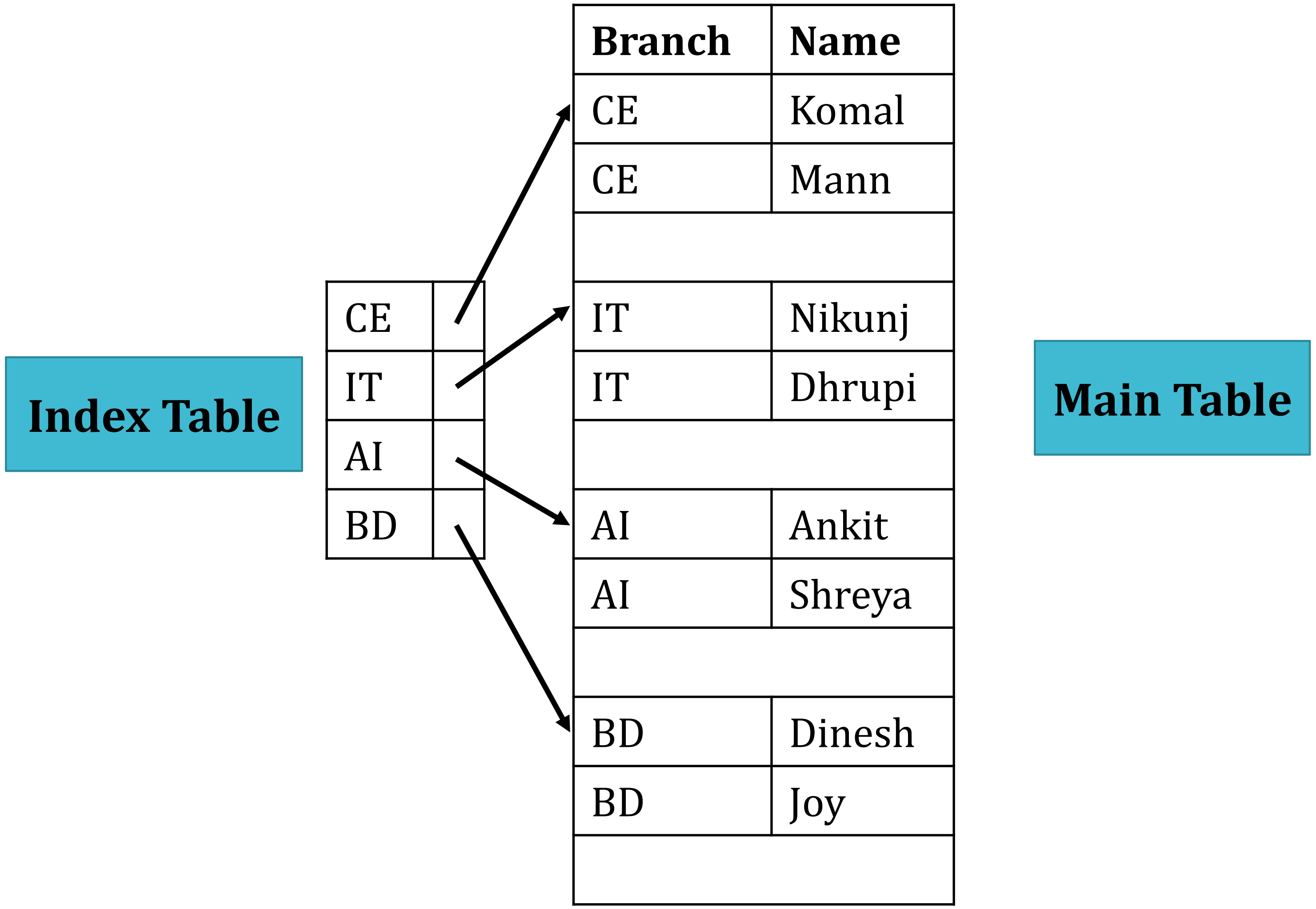
- ✓ To reduce the size of mapping, another level of indexing is introduced.
- ✓ Huge range of column values is selected to keep primary index of smaller size. The range is then divided into smaller ranges in Secondary Index for faster access.
- ✓ Linear Search can be used for data retrieval



Roll	Name
001	Sagar
002	Jyoti
003	Mann
051	Minal
052	Rupa
101	Priya
151	Nimish
152	Yash

Clustering Index

- ✓ index is created on non-primary key columns.
- ✓ The columns with similar properties are grouped to get the unique values in order to create the index.



Index Table

CE	
IT	
AI	
BD	

Main Table

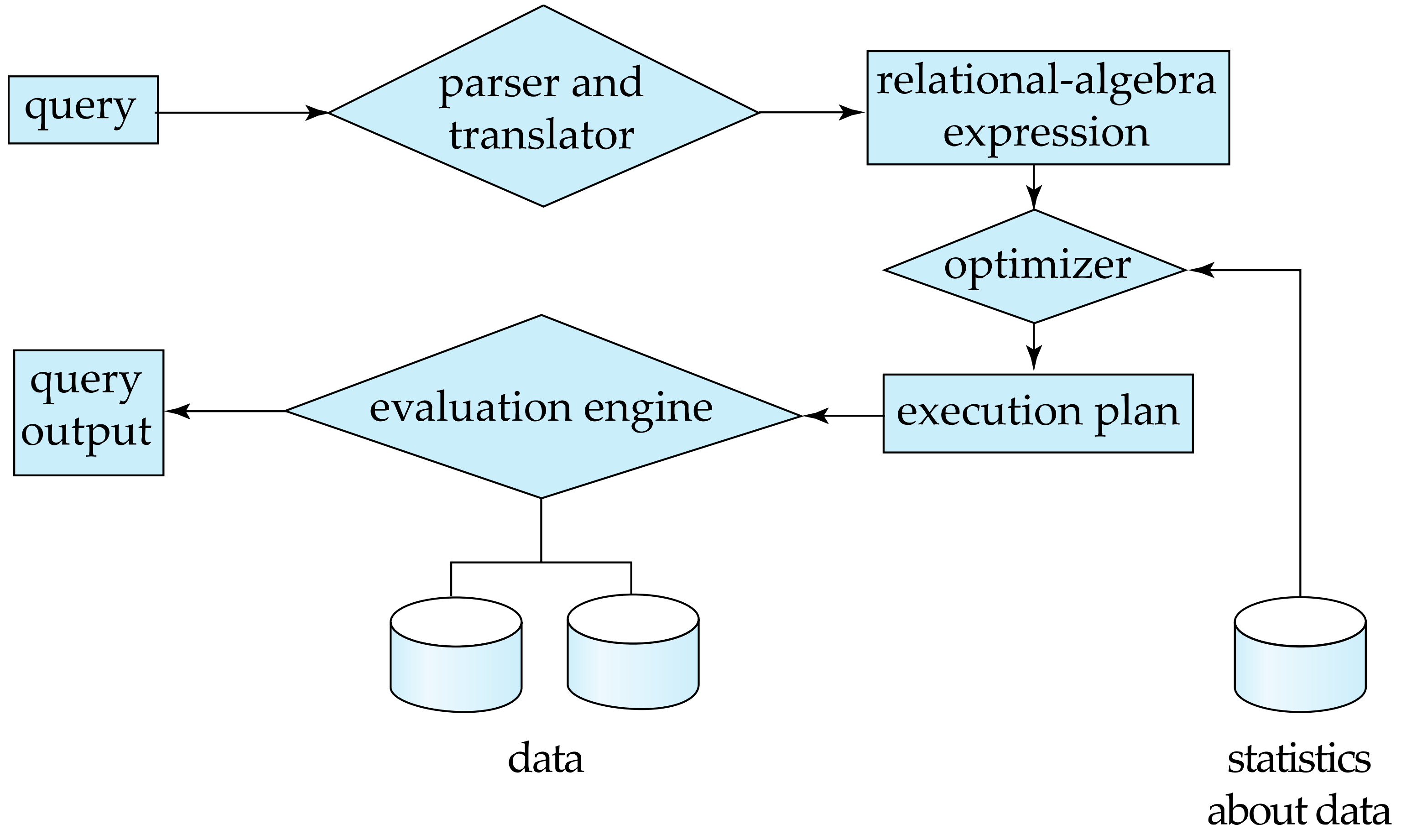
Branch	**Name**
CE	Komal
CE	Mann
IT	Nikunj
IT	Dhrupi
AI	Ankit
AI	Shreya
BD	Dinesh
BD	Joy

Query Processing

- **Query processing** refers to the range of activities involved in extracting data from a database.
- Basic steps of query processing are:

1. Parsing and translation
2. Optimization
3. Evaluation

Steps of Query processing:



Steps of Query processing:

1. Parsing and translation.

- The first action the system must take in query processing is to translate a given query into its internal form.
- This translation process is similar to the work performed by the **parser** of a compiler.
- In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on.
- The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression. If the query was expressed in terms of a view, the translation phase also replaces all uses of the view by the relational-algebra expression that defines the view.

Steps of Query processing:

2. Evaluation Plan.

- A relational algebra operation annotated with instructions on how to evaluate it is called an evaluation primitive. A sequence of primitive operations that can be used to evaluate a query is a query-execution plan or **query-evaluation** plan.
- The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

Steps of Query processing:

3. Optimization.

- It is the responsibility of the system to construct a query evaluation plan that minimizes the cost of query evaluation; this task is called **query optimization**.
- Once the query plan is chosen, the query is evaluated with that plan, and the result of the query is output.
- In order to optimize a query, a query optimizer must know the cost of each operation.

Measures of Query cost

- There are multiple possible evaluation plans for a query, and it is important to be able to compare the alternatives in terms of their (estimated) cost, and choose the best plan.
- We must estimate the cost of individual operations, and combine them to get the cost of a query evaluation plan.
- The cost of query evaluation can be measured in terms of a number of different resources, including disk accesses, CPU time to execute a query,
- For simplicity we just use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures

  t_T – time to transfer one block

  t_S – time for one seek

  => Cost for b block transfers plus S seeks $\boldsymbol{b * t_T + S * t_S}$

Measures of Query cost

- The costs of all the algorithms that we consider depend on the size of the buffer in main memory.
- The **response time** for a query-evaluation plan, assuming no other activity is going on in the computer, would account for all these costs, and could be used as a measure of the cost of the plan.
- The response time of a plan is very hard to estimate without actually executing the plan.

File Scan

- In query processing, the file scan is the lowest-level operator to access data.
- **File scans** are search algorithms that locate and retrieve records that fulfill a selection condition.
- In relational systems, a file scan allows an entire relation to be read in those cases where the relation is stored in a single, dedicated file.
- There are two scan algorithms to implement in selection operation.

1. Linear search
2. Binary search

1. Linear search (brute force algorithm)

- In a linear search(A1 Algoritm), the system scans each file block and tests all records to see whether they satisfy the selection condition. An initial seek is required to access the first block of the file.

- Since the records are grouped into disk blocks, each disk block is read into a main memory buffer, and then a search through the records within the disk block is conducted in main memory.

- Suppose t_S is the seek time (number of Seek is usually one – to reach the beginning of the file) , t_T is the number of traversal time for one block, and B is the number of blocks to be transferred, then the cost is calculated as:

tS + (B*tT)

2. Binary search

- If the selection condition involves an equality comparison on a key attribute on which the file is ordered, binary search which is more efficient than linear search can be used.

- This method of selection is applicable only when the records are sorted based on the search key value and we have equal condition. i.e.; this method is not suitable for range operation or any other kind. The filter condition should be 'search key column = value', like we had City= 'Ahmedabad''.

- Suppose blocks of records are stored continuously in the memory than the cost of the query to fetch first record is calculated as **log (B)* (ts+ tT).**

Join operations

JOIN operation is one of the most time-consuming operations in query processing.

Equi-join is a join of the form $r \bowtie_{r.A=s.B} s$, where *A* and *B* are attributes or sets of attributes of relations *r* and *s*, respectively.

- **Nested loop join:**

Relation *r* is called the **outer relation** and relation *s* the **inner relation** of the join, since the loop for *r* encloses the loop for *s*. The algorithm uses the notation $tr \cdot ts$, where *tr* and *ts* are tuples; $tr \cdot ts$ denotes the tuple constructed by concatenating the attribute values of tuples *tr* and *ts* .

```
for each tuple tr in r do begin
    for each tuple ts in s do begin
        test pair (tr , ts ) to see if they satisfy the join condition
        if they do, add tr · ts to the result;
    end
end
```

Join operations

- **Block Nested-loop join**
- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```
for each block $B_r$ of $r$ do begin
for each block $B_s$ of $s$ do begin
    for each tuple $t_r$ in $B_r$ do begin
        for each tuple $t_s$ in $B_s$ do begin
            Check if $(t_r, t_s)$ satisfy the join condition
            if they do, add $t_r \bullet t_s$ to the result.
        end
    end
end
end
```

Join operations

Indexed nested-loop join

- In a nested-loop join, if an index is available on the inner loop's join attribute, index lookups can replace file scans. For each tuple *tr* in the outer relation *r*, the index is used to look up tuples in *s* that will satisfy the join condition with tuple *tr* .

- This join method is called an **indexed nested-loop join**; Looking up tuples in *s* that will satisfy the join conditions with a given tuple *tr* is essentially a selection on *s*.

- For example, consider *student* $\bowtie$ *takes*. Suppose that we have a *student* tuple with *ID* "00128". Then, the relevant tuples in *takes* are those that satisfy the selection "*ID* = 00128".

Join operations

Merge Join

- The **merge-join** algorithm (also called the **sort-merge-join** algorithm) can be used to compute natural joins and equi-joins.
- If the records of *R* and *S* are *physically sorted* (ordered) by value of the join attributes *A* and *B*, respectively, we can implement the join in the most efficient way possible.
- Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for *A* and *B*. If the files are not sorted, they may be sorted first by using external sorting.

Join operations

Hash join:

- Hash join is one type of joining techniques that are used to process a join query. Hash join is proposed for performing joins that are Natural joins or Equi-joins.
- The idea behind the hash-join algorithm is this: Suppose that an *r* tuple and an *s* tuple satisfy the join condition; then, they have the same value for the join attributes.
- If that value is hashed to some value *i*, the *r* tuple has to be in *ri* and the *s* tuple in *si* .
- Therefore, *r* tuples in *ri* need only to be compared with *s* tuples in *si* ; they do not need to be compared with *s* tuples in any other partition.

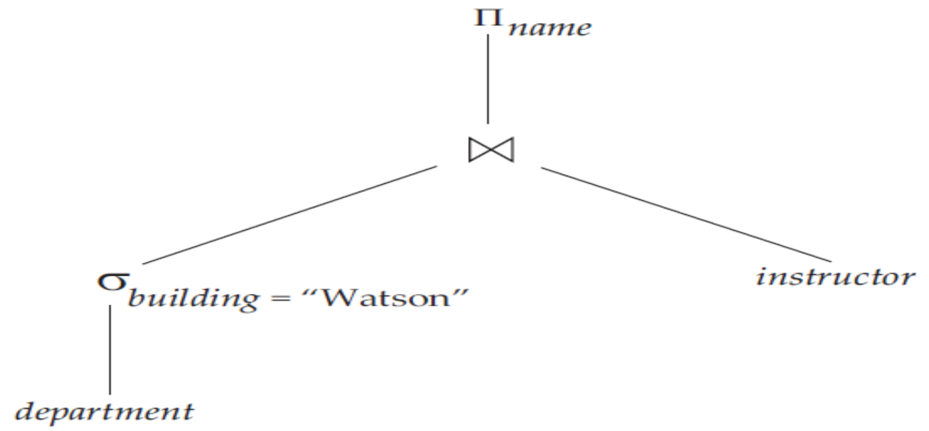
Evaluation of Expressions

- Now we consider how to evaluate an expression containing multiple operations.
- The obvious way to evaluate an expression is simply to evaluate one operation at a time, in an appropriate order.
- There are two approaches of query optimization:-

1. Materialization: Generate results of an expression whose inputs are relations or are already computed, materialize (store) it on disk.
2. Pipeline: Pass on tuples to parent operations even as an operation is being executed.

Evaluation of Expressions

- 1. Materialized:
- It is easiest to understand how to evaluate an expression by looking at a pictorial representation of the expression in an **operator tree**. Consider the expression:
- $\pi name(\sigma building$ ="Watson"$(department) \bowtie instructor)$



- By repeating the process, we will eventually evaluate the operation at the root of the tree, giving the final result of the expression. In our example, we get the final result by executing the projection operation at the root of the tree, using as input the temporary relation created by the join.
- Evaluation as just described is called **materialized evaluation**, since the results of each intermediate operation are created (materialized) and then are used for evaluation of the next-level operations.

Evaluation of Expressions

- **Pipelined evaluation :** evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{building="\text{Watson}"}(department)$$

  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.

Query optimization

- **Query optimization** is the process of selecting the most efficient query-evaluation plan from among the many strategies usually possible for processing a given query, especially if the query is complex.
- One aspect of optimization occurs at the relational-algebra level, where the system attempts to find an expression that is equivalent to the given expression, but more efficient to execute.
- When any query constructs the large intermediate relation and we want the same result but with small intermediate representation we convert that query by equivalent

$$\Pi_{name, title}\left(\sigma_{dept\_name = \text{"Music"}}\left(instructor \bowtie \left(teaches \bowtie \Pi_{course\_id, title}(course)\right)\right)\right)$$

Equivalence rule:

- ***Transformation of relational expression to equivalent relational expression.***
- A query can be expressed in several different ways, with different costs of evaluation.
- Two relational-algebra expressions are said to be **equivalent** if, on every legal database instance, the two expressions generate the same set of tuples.
- An **equivalence rule** says that expressions of two forms are equivalent. We can replace an expression of the first form by an expression of the second form, or vice versa since the two expressions generate the same result on any valid database.
- We use θ, $\theta 1$, $\theta 2$, and so on to denote predicates,
- $L1$, $L2$, $L3$, and so on to denote lists of attributes,
- E, $E1$, $E2$, and so on to denote relational-algebra expressions.

Equivalence rule:

Rule:1

Conjunctive selection operations can be deconstructed into a sequence of individual selections. This transformation is referred to as a cascade of σ.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

Rule:2

Selection operations are **commutative**.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

Rule:3

Only the final operations in a sequence of projection operations are needed; the others can be omitted. This transformation can also be referred to as a cascade of π.

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) = \Pi_{L_1}(E)$$

Equivalence rule:

Rule:1

Conjunctive selection operations can be deconstructed into a sequence of individual selections. This transformation is referred to as a cascade of σ.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

Rule:2

Selection operations are **commutative**.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

Rule:3

Only the final operations in a sequence of projection operations are needed; the others can be omitted. This transformation can also be referred to as a cascade of π.

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) = \Pi_{L_1}(E)$$

Equivalence rule:

Rule: 4

Selections can be combined with Cartesian products and theta joins.

$$\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$$

$$\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$$

Rule: 5

Theta-join operations are commutative.

$$E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$$

Rule:6

A) Natural-join operations are **associative**.

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

B) Theta joins are associative in the following manner

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

Equivalence rule:

Rule-7

The selection operation distributes over the theta-join operation under the following two conditions:

A. It distributes when all the attributes in selection condition θ 0 involve only the attributes of one of the expressions (say, *E*1) being joined.

$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$

B. It distributes when selection condition θ 1 involves only the attributes of *E*1 and θ 2 involves only the attributes of *E*2.

$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$

Rule-8

The set operations union and intersection are commutative.

$E1 \cup E2 = E2 \cup E1$

$E1 \cap E2 = E2 \cap E1$

Set difference is not commutative.

Equivalence rule:

- Rule-9
- Set union and intersection are associative.

  $(E1 \cup E2) \cup E3 = E1 \cup (E2 \cup E3)$

  $(E1 \cap E2) \cap E3 = E1 \cap (E2 \cap E3)$

- Rule-10
- The projection operation distributes over the union operation.
- $\pi L\ (E1 \cup E2) = (\pi L\ (E1)) \cup (\pi L\ (E2))$

Cost based query optimization

- A cost based optimizer will look at all of the possible ways or scenarios in which a query can be executed – and each scenario will be assigned a 'cost', which indicates how efficiently that query can be run.
- The database 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 plans, the optimizer chooses the plan with the lowest cost estimate.

Heuristic optimization

- Heuristics are used by optimizers to reduce the cost of optimization.
- Heuristic optimization transforms the query-tree by using a set of rules (Heuristics) that typically (but not in all cases) improve execution performance.
- An example of a heuristic rule is the following rule for transforming relational algebra queries:
- Perform selection operations as early as possible. (reduces the number of tuples)
- Perform projection early (reduces the number of attributes)
- Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations.
- Query-optimization approaches that apply heuristic plan choices for some parts of the query, with cost-based choice based on generation of alternative access plans on other parts of the query, have been adopted in several systems.

Heuristic optimization

- Most optimizers allow a cost budget to be specified for query optimization.
- The search for the optimal plan is terminated when the optimization cost budget is exceeded, and the best plan found up to that point is returned.
- **Process for heuristics optimization**
- The parser of a high-level query generates an *initial internal representation*;
- Apply heuristics rules to optimize the internal representation.
- A query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.
- The **main heuristic** is to apply first the operations that reduce the size of intermediate results.
- E.g., Apply SELECT and PROJECT operations before applying the JOIN or other binary operations.

Materialized view

- When a view is defined, normally the database stores only the query defining the view.
- A **materialized view** is a view whose contents are computed and stored.
- Materialized views constitute redundant data, in that their contents can be inferred from the view definition and the rest of the database contents.
- Materialized views are important for improving performance in some applications.
- Consider this view, which gives the total salary in each department:

create view *department* **as**
select *dept name*, **sum** (*salary*)
from *instructor*
group by *dept name*;

Materialized view

- Suppose the total salary amount at a department is required frequently.
- Computing the view requires reading every *instructor* tuple pertaining to a department, and summing up the salary amounts, which can be time-consuming.
- In contrast, if the view definition of the total salary amount were materialized, the total salary amount could be found by looking up a single tuple in the materialized view.

Thanks

Marwadi University