DATA WAREHOUSING AND DATA MINING (CE306)

-Prof. Pooja A. Jadhav
(poojabhise@mes.ac.in)
Assistant Professor (Dept. of Comp. Engg.)

Syllabus:

Course Code	Course Name	Scheme	Theory	Practical	Tutorial	Total
CE 306	Data Warehouse and Data Mining	Contact Hours	3	2	-	5
		Credits	3	1	-	4

Course Code	Course Name	Examination Scheme							
		Theory Marks				Term Work	Practical	Oral	Total
		Internal Assessment			End Sem Exam				
		IA 1	IA 2	Average					
CE 306	Data Warehouse and Data Mining	40	40	40	60	25	-	25	150

Course Objectives:

The course is aimed to:

1. To enhance query performance by enabling fast data access, efficient execution, and optimal resource usage.
2. To ensure reliable, secure, and consistent data access across distributed environments while maintaining transaction integrity and enforcing access controls.
3. To identify the scope and essentiality of Data Warehousing and ETL.
4. To analyze data, choose relevant models and algorithms for respective applications.
5. To study data mining techniques for various data mining algorithms.
6. To develop research interest towards advances in data mining.

Course Outcomes:

On successful completion of course learner/student will be able to:

1. Analyze the impact of improved query efficiency, reduced resource usage, and faster data retrieval on overall database performance.
2. Describe how distributed database systems ensure enhanced data security, consistency, and reliable performance.
3. Understand the fundamentals of Data Warehouse Architecture and ETL process, Explain the concepts of dimensional modeling and Online Analytical Processing (OLAP) operations.
4. Identify and Implement the appropriate data mining algorithms to solve real world problems.
5. Compare and evaluate different data mining techniques like classification, prediction, clustering.
6. Assess various data mining techniques for their suitability in association rule mining.

CO-PO Mapping (3 High , 2 Medium , 1 Low)												
CO/PO	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
CO1	2	2	2	1	2							
CO2	2	2	2	1	1	2	1					
CO3	2	2	2	2	1							2
CO4	2	2	2	2	2							
CO5	2	2	2	2	2							1
CO6	2	2	2	1	2							1

Module No	Module	Detailed Content	Hrs.
1	Advanced Data Management System	Indexing and Hashing, Indexing Techniques; Static and dynamic Types of Single-Level & Multilevel Indexes; Dynamic Multilevel Indexes Using B-Trees and B+-Trees; Indexes on Multiple Keys, Hashing Techniques. Overview of Query Processing and Optimization, SQL Queries into relational algebra approaches and cost-based optimization.	6
2	Advanced Transaction Management System	Transaction concept, Transaction states, ACID properties, Serializability, Recoverability, Concurrency Control: Lock-based, Time-stamp based, Deadlock handling, Recovery System: Failure Classification, Log based recovery, Checkpoints, Shadow Paging, ARIES Algorithm. Distributed Database Systems; Types, Architecture; Data Fragmentation, Replication and Allocation, 3PC locking protocol, Types of Access Control; DACB, MAC & RAC, Data Security in Distribution transaction.	7

Module No	Module	Detailed Content	Hrs.
3	Introduction to Data Warehousing and Dimensional Modelling, ETL and OLAP	**Introduction to Data Warehouse** : Components of Data warehouse Architecture, Data warehouse architecture, Data warehouse versus Data Marts, E-R Modeling versus Dimensional Modeling, Data Warehouse Schemas; Star Schema, Snowflake Schema, Fact Constellation Schema., **ETL and OLAP** : Major steps in ETL process, Data Extraction Methods, Data Transformation; Basic Tasks in Transformation, Data Loading Techniques, OLTP versus OLAP, OLAP operations: Slice, Dice, Rollup, Drilldown and Pivot.	8
4	Introduction to Data Mining, Data Exploration and Data Pre-processing	Data Mining Task Primitives, Architecture, KDD process, Issues in Data Mining, Applications of Data Mining, **Data Exploration:** Types of Attributes, Statistical Description of Data, Data Visualization; **Data Preprocessing:** Descriptive data summarization, Cleaning, Integration & transformation, Data reduction,Data Discretization & Concept hierarchy generation	5

Module No	Module	Detailed Content	Hrs.
5	Classification, Prediction and Cluster Analysis	**Classification:** Decision Tree: ID3, Bayesian Classification; **Prediction:** Logistic regression; **Clustering:** Partitioning Methods (*k*-Medoids), Hierarchical Methods (BIRCH), Density based Method, Grid based Method.	7
6	Mining frequent patterns and associations, Basics of Web mining	**Basic concepts & a Road map:** Market Basket Analysis, Frequent Item sets, Closed Item sets, and Association Rule, Frequent Pattern Mining; **Efficient and Scalable Frequent Itemset Mining Methods:** Apriori Algorithm, Association Rule Generation, Improving the Efficiency of Apriori: FP growth, Web mining, Page rank algorithm, Text mining	6

Module 1

Advanced Data Management System

Prerequisites

- Data
- Information
- DBMS, File System, Data Mart, Meta Data
- Reviewing basic concepts of a Relational database
- SQL concepts
- Record
- Block
- Data Pointer

Prerequisites

Parameter	DBMS	RDBMS
Storage	DBMS stores data as a file.	Data is stored in the form of tables.
Database structure	DBMS system, stores data in either a navigational or hierarchical form.	RDBMS uses a tabular structure where the headers are the column names, and the rows contain corresponding values
Number of Users	DBMS supports single user only.	It supports multiple users.
Data Fetching	Data fetching is slower for the complex and large amount of data.	Data fetching is rapid because of its relational approach.
Data Relationship	No relationship between data	Data is stored in the form of tables which are related to each other with the help of foreign keys.

Module 1

Advanced Data Management System

Indexing and Hashing, Indexing Techniques; Static and dynamic Types of Single-Level & Multilevel Indexes; Dynamic Multilevel Indexes Using B-Trees and B+-Trees;

Indexes on Multiple Keys,

Hashing Techniques.

Overview of Query Processing and Optimization,

SQL Queries into relational algebra approaches and cost-based optimization

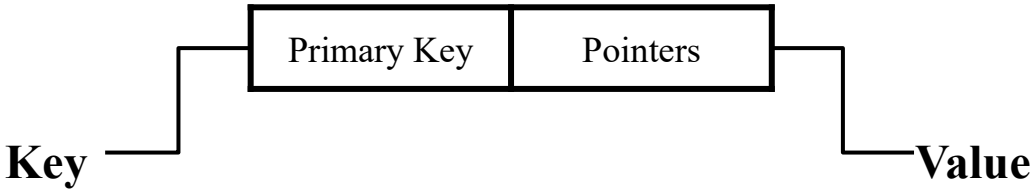
Indexing

Indexing is a data structure technique which allows you to quickly retrieve records from a database file.

An Index is a small table having only two columns.

The first column comprises a copy of the primary or candidate key of a table.

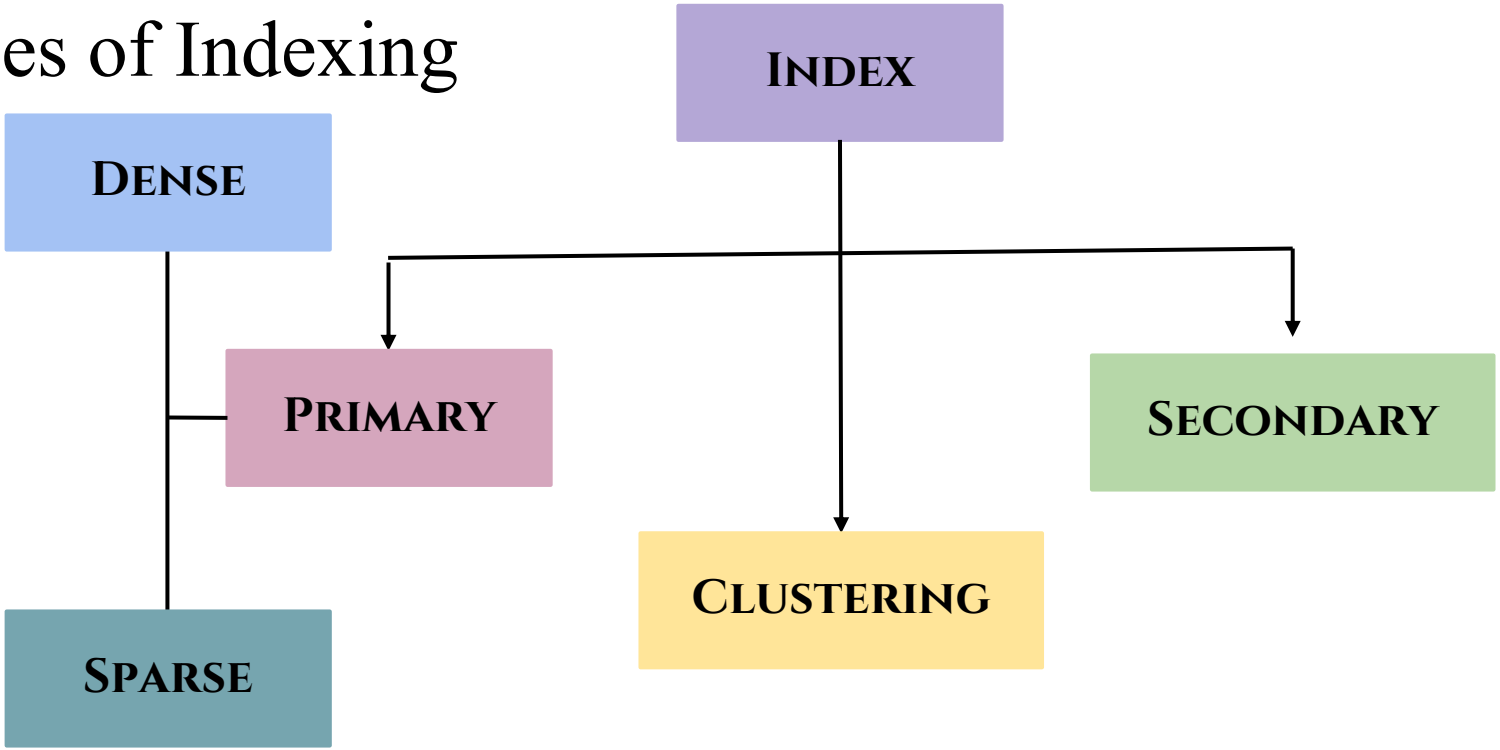
Its second column contains a set of pointers for holding the address of the disk block where that specific key value stored.



Various attributes

- Access Type
- Access Time
- Insertion Time
- Deletion Time
- Space Overhead - additional space required by the index

Types of Indexing



Primary Indexing

Primary Index is an ordered file which is fixed length size with two fields.

The first field is the same a primary key and second, filed is pointed to that specific data block.

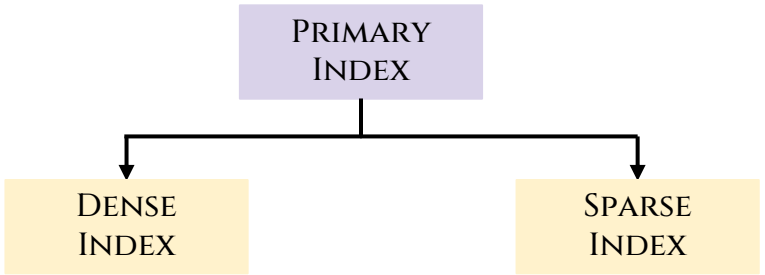
In the primary Index, there is always one to one relationship between the entries in the index table.

Index entry is created for the first record of each block called as block anchor.

Number of index entries is equal to the number of blocks.

Number of access required = $log_2^n + 1$

n means the number of blocks acquired by index file.

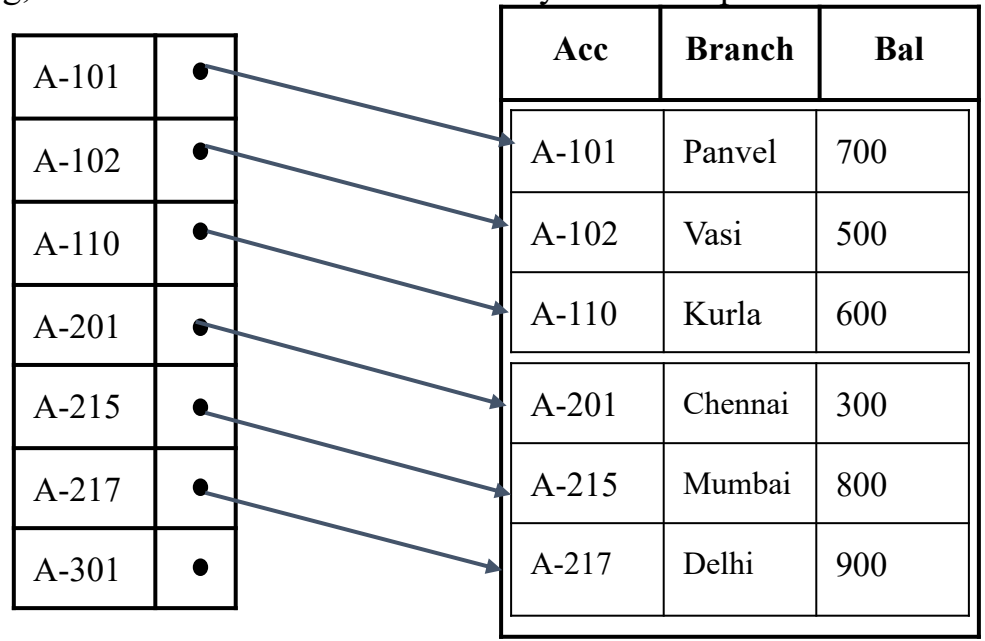


Dense Indexing

In a dense index, a record is created for every search key valued in the database.

This helps you to search faster but needs more space to store index records.

In this Indexing, method records contain search key value and points to the real record on the disk.



Sparse Index

It is an index record that appears for only some of the values in the file.

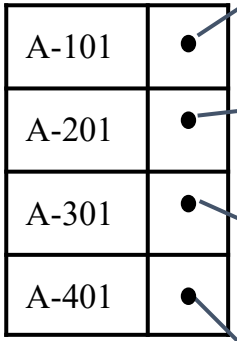
Sparse Index helps you to resolve the issues of dense Indexing in DBMS.

In this method of indexing technique, a range of index columns stores the same data block address, and when data needs to be retrieved, the block address will be fetched.

However, sparse Index stores index records for only some search-key values.

It needs less space, less maintenance overhead for insertion, and deletions but It is slower compared to the dense Index for locating records.

Sparse Index Example



Acc	Branch	Bal
A-101	Panvel	700
A-102	Vasi	500
A-110	Kurla	600
A-201	Chennai	300
A-215	Mumbai	800
A-217	Delhi	900
A-301	Chennai	300
A-302	Mumbai	800
A-310	Delhi	900
A-401	Chennai	300
A-415	Mumbai	800

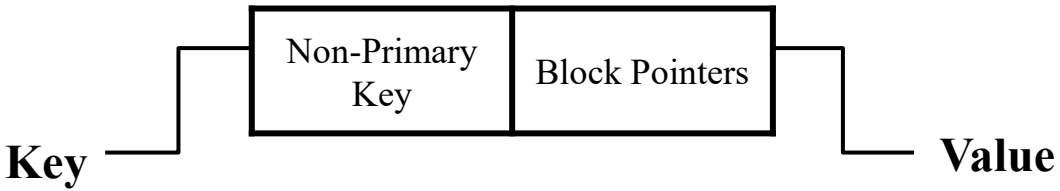
$$log_2^n + 1$$

Problem Solving

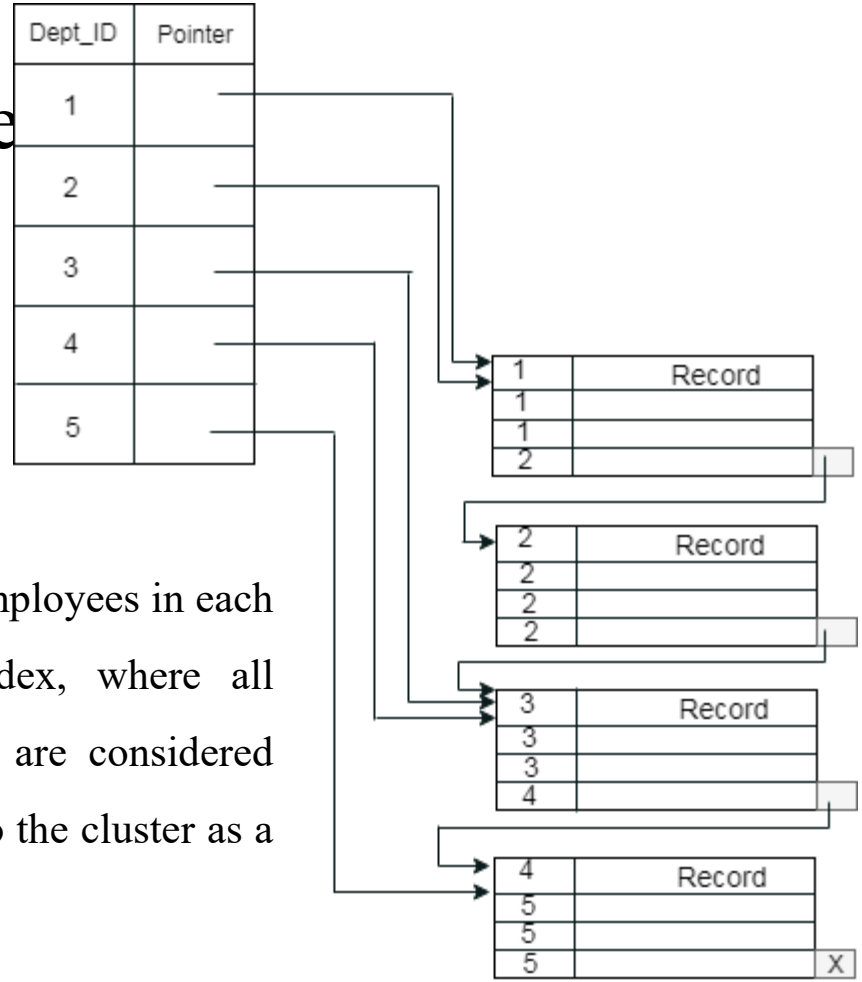
Number of records is 30000, block size is 1024 bytes the strategy is unplanned, the record size is 100 bytes, key size is 6 bytes and the pointer size is 9 bytes. Find the average number of block access with or without index.

Clustered Indexing

- A clustered index can be defined as an ordered data file. Sometimes the index is created on non-primary key columns which may not be unique for each record.
- In this case, to identify the record faster, we will group two or more columns to get the unique value and create index out of them. This method is called a clustering index.
- The records which have similar characteristics are grouped, and indexes are created for these group.

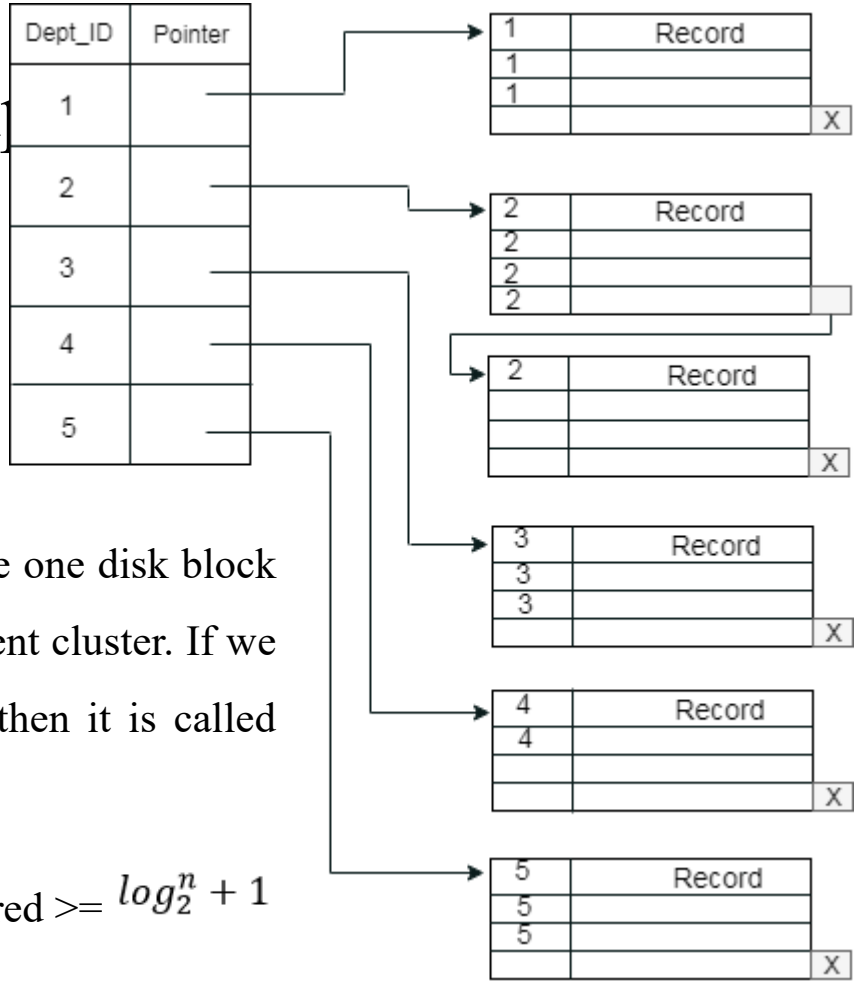


Clustered Indexing Example



Example: suppose a company contains several employees in each department. Suppose we use a clustering index, where all employees which belong to the same Dept_ID are considered within a single cluster, and index pointers point to the cluster as a whole. Here Dept_Id is a non-unique key.

Clustered Indexing Exam

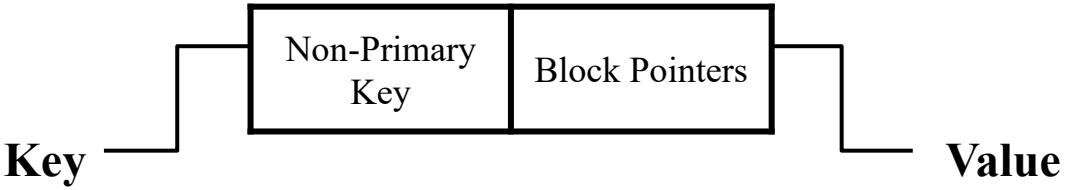


The previous schema is little confusing because one disk block is shared by records which belong to the different cluster. If we use separate disk block for separate clusters, then it is called better technique.

The average number of block access is required >= $log_2^n + 1$

Secondary Index

Secondary Index can be defined as an unordered data file and the index is created on non key or candidate key.

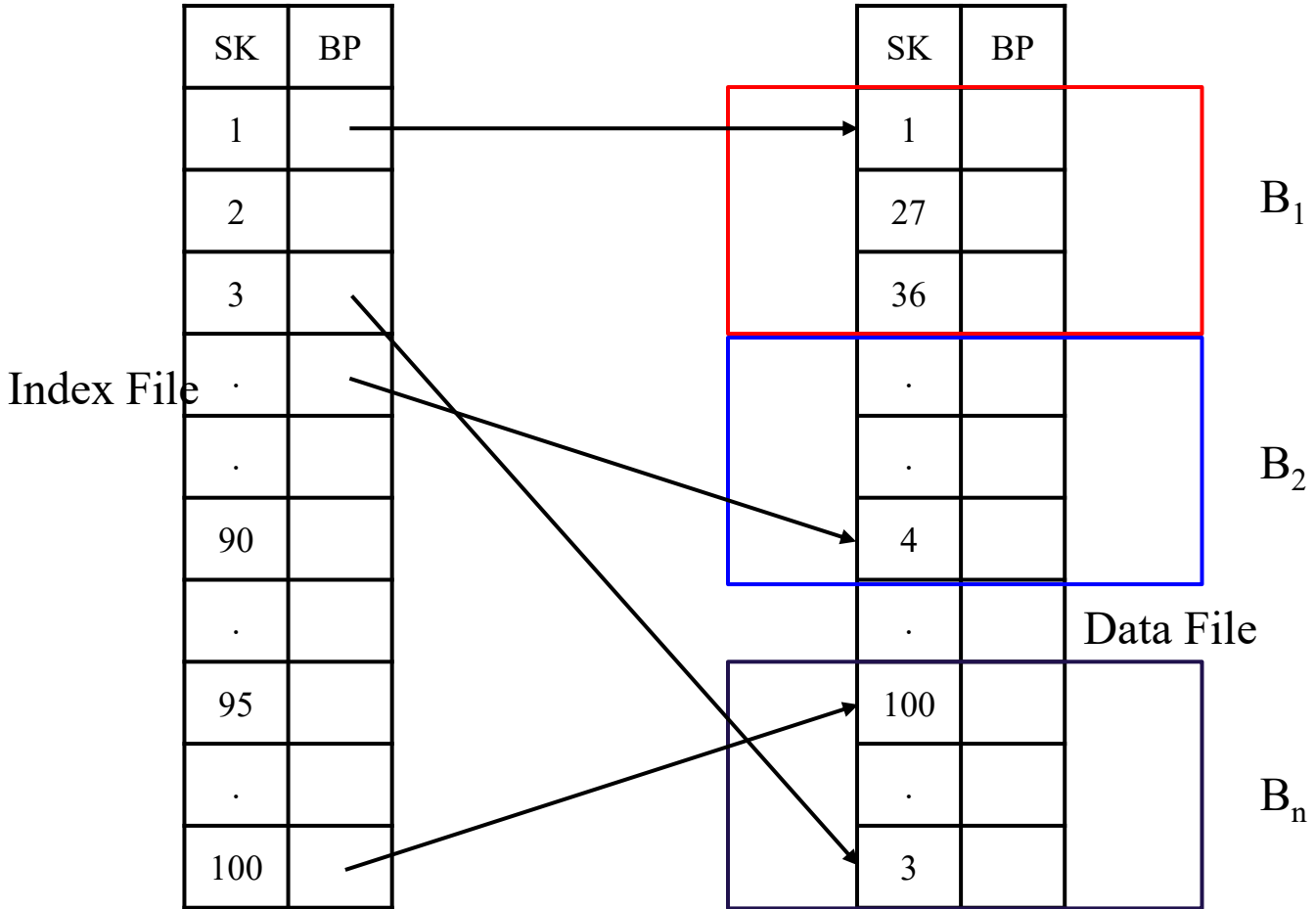


Number of entries in the index file should be equal to the number of entries in block file.

In a bank account database, data is stored sequentially by acc_no; you may want to find all accounts in of a specific branch of ABC bank.

Here, you can have a secondary index in DBMS for every search-key. Index record is a record point to a bucket that contains pointers to all the records with their specific search-key value.

Example


SK
BP
1
2
3
.
.
90
.
95
.
100
Index File
SK
BP
1
27
36
.
.
4
.
100
.
3
Data File
B_1
B_2
B_n

Properties of Index Types

Types of Index	No. of Index entries	Dense or Sparse	Block Anchoring on the datafile
Primary	No. of blocks in datafile	Dense and Sparse	Yes
Clustering	No.of distinct index field value	Sparse	Yes/No
Secondary	No. of records in datafile	Dense	No

Multilevel Indexing

Multilevel indexing will communicate with root node, internal node, leaf node and data file.

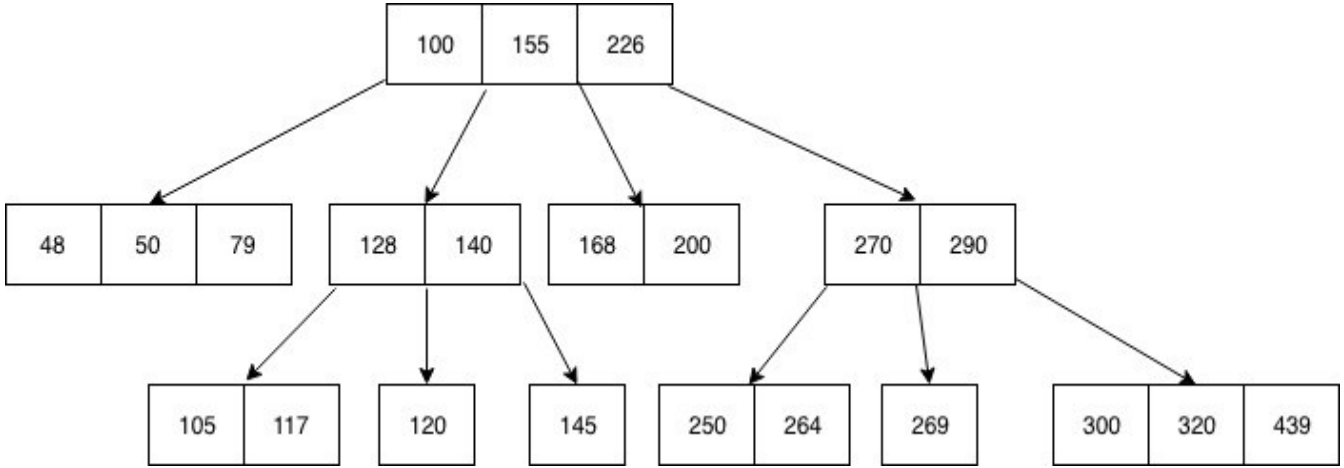
Two different techniques in multilevel indexing are B tree and B+ tree.

B-tree is a data structure that provides sorted data and allows searches, sequential access, attachments, and removals in sorted order.

The B-tree is highly capable of storage systems that write large blocks of data.

The B-tree simplifies the binary search tree by allowing nodes with more than two children.

Sample B tree



B-tree stores data such that each node contains keys in ascending order.

Each of these keys has two references to another two child nodes.

Te left side child node keys are less than the current keys and the right side child node keys are more than the current keys.

If a single node has “n” number of keys, then it can have maximum “n+1” child nodes.

Properties of B tree

At every level we use key and block pointer.

Block pointer pointed either block or record.

First it check the root node which contains two children (min and max).

The internal node is also consisting of children between p/2 and p.

The Key from internal node is between (p/2)-1 and p-1.

The leaf nodes all are at the same level.

The Key from leaf node is between (p/2)-1 and p-1.

B^+ Tree

- The B+ Tree stores the data pointers only at the leaf nodes of the Tree, which makes search more process more accurate and faster.
- All the leaf nodes remains at same height.
- The leaf nodes are linked using linked list.
- B+ tree occupy a little more space than B tree.
- This technique, greatly reduces the number of entries that can be packed into a node of a B-tree, thereby contributing to the increase in the number of levels in the B-tree, hence increasing the search time of a record.
- B+ tree eliminates the above drawback by storing data pointers only at the leaf nodes of the tree.

Comparison of B Tree and B+ Tree

B Tree	B^+ Tree
B Tree is used to sort the data with various facilities such as insertion, deletion, search, etc.	B+ Tree is the modified version of B Tree providing all the functions smoothly and quickly.
Both Internal nodes and leaf nodes are used to store Records and Keys.	Keys are stored in Internal node whereas records stored in leaf nodes.
No duplication is allowed.	Duplication may occur.
Linking of leaf nodes is not done with each other.	Linking of leaf nodes is done with each other.
Speed of searching is not as quick as compared to B+ Tree.	Speed of searching is very quick.
Sequential access is not possible.	Sequential access is possible.

Hashing Techniques & its need Hashing

- For a huge database structure, it’s tough to search all the index values through all its level and then you need to reach the destination data block to get the desired data.
- Hashing method is used to index and retrieve items in a database as it is faster to search that specific item using the shorter hashed key instead of using its original value.
- Data is stored in the form of data blocks whose address is generated by applying a hash function in the memory location where these records are stored known as a data block or data bucket.
- Hashing is an ideal method to calculate the direct location of a data record on the disk without using index structure.
- It is also a helpful technique for implementing dictionaries.

Important Terminologies in Hashing

- **Data bucket** – Data buckets are memory locations where the records are stored. It is also known as Unit Of Storage.
- **Key:** A DBMS key is an attribute or set of an attribute which helps you to identify a row(tuple) in a relation(table). This allows you to find the relationship between two tables.
- **Hash function:** A hash function, is a mapping function which maps all the set of search keys to the address where actual records are placed.
- **Linear Probing** – Linear probing is a fixed interval between probes. In this method, the next available data block is used to enter the new record, instead of overwriting on the older record.
- **Quadratic probing**– It helps you to determine the new bucket address. It helps you to add Interval between probes by adding the consecutive output of quadratic polynomial to starting value given by the original computation.
- **Hash index** – It is an address of the data block. A hash function could be a simple mathematical function to even a complex mathematical function.
- **Double Hashing** –Double hashing is a computer programming method used in hash tables to resolve the issues of has a collision.
- **Bucket Overflow:** The condition of bucket-overflow is called collision. This is a fatal stage for any static has to function.

A **data bucket** is a memory location where records are stored.
Example:
•Suppose our hash table has **5 buckets**: B0, B1, B2, B3, B4.
•If a record Student(ID=23) hashes to bucket B3, it's stored there.
•All records mapped to the same hash value go into that bucket.

Key
A DBMS key uniquely identifies a row in a table.

- **Hash Function**
- A **hash function** maps keys to addresses in the hash table.
- Example:
- Hash function: **h(Key) = Key mod 5**
- For Student IDs:
  - h(101) = 101 mod 5 = **1** → Bucket B1
  - h(102) = 102 mod 5 = **2** → Bucket B2
  - h(107) = 107 mod 5 = **2** → Bucket B2 (collision occurs).

Linear Probing

Used when there's a **collision**. Search the next available bucket sequentially.

Example:

•Hash table size = 5, h(Key) = Key mod 5.

•Insert 102 → h(102) = 2 → place in bucket B2.

•Insert 107 → h(107) = 2 → B2 is full, so check B3 → empty → store in B3.

•Insert 112 → h(112) = 2 → B2 full → B3 full → B4 empty → store in B4.

Quadratic Probing

Instead of moving one by one, move in **1^2, 2^2, 3^2…** steps.

Example:

•h(Key) = Key mod 5, table size = 5.

•Insert 102 → h(102) = 2 → B2.

•Insert 107 → h(107) = 2 → collision.

•Check B2 + 1^2 = B3 → empty → store in B3.

Hash Index

A **hash index** is the address where data is stored.

Example:

•Hash table has 5 buckets.

•h(101) = 1 → data is stored in **memory location of B1** (say at address 0xAB12).

•This 0xAB12 is the **hash index**.

Double Hashing

Uses a second hash function when the first causes a collision.

Example:

•h1(Key) = Key mod 5

•h2(Key) = 1 + (Key mod 4)

•Table size = 5.

•Insert 102: h1(102) = 2 → B2 (empty).

•Insert 107: h1(107) = 2 → B2 full.

- •Use h2(107) = 1 + (107 mod 4) = 1 + 3 = 4.
- •New position = (2 + 4) mod 5 = 6 mod 5 = **B1** → store in B1.

Types of Hashing Techniques

1. Static Hashing
2. Dynamic Hashing

Static Hashing

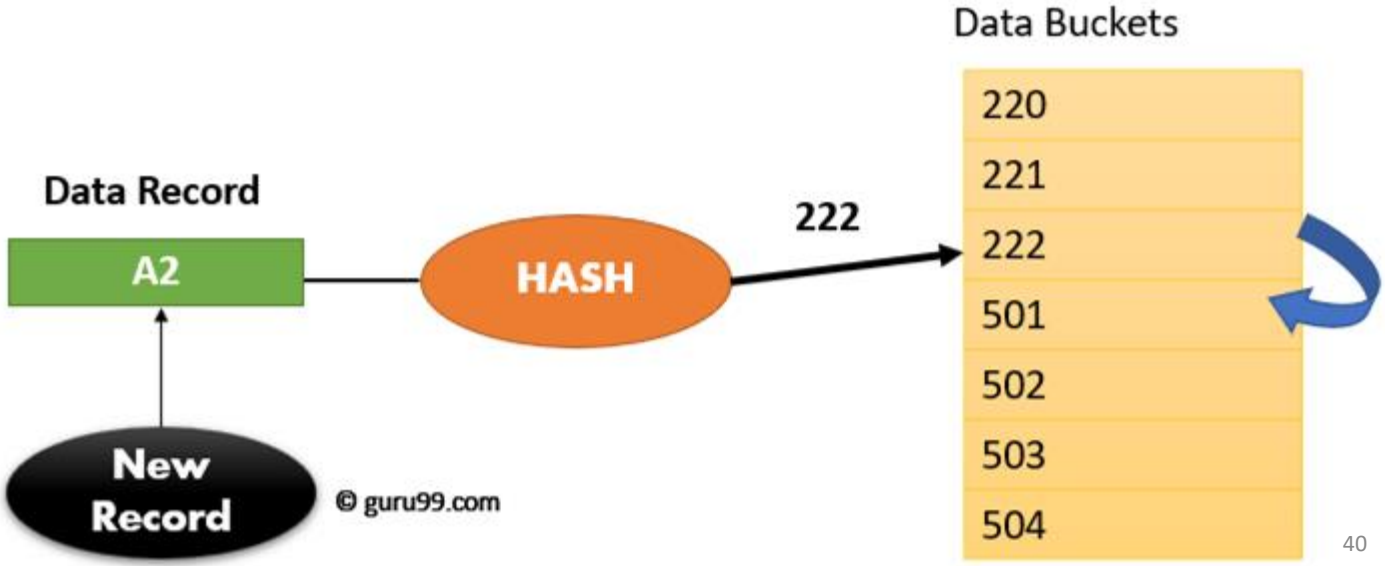
- In the static hashing, the resultant data bucket address will always remain the same.
- Therefore, if you generate an address for say Student_ID = 10 using hashing function mod(3), the resultant bucket address will always be 1. So, you will not see any change in the bucket address.

Static Hash Functions

- **Inserting a record:** When a new record requires to be inserted into the table, you can generate an address for the new record using its hash key. When the address is generated, the record is automatically stored in that location.
- **Searching:** When you need to retrieve the record, the same hash function should be helpful to retrieve the address of the bucket where data should be stored.
- **Delete a record:** Using the hash function, you can first fetch the record which is you wants to delete. Then you can remove the records for that address in memory.
- Static hashing is further divided into
    - ***Open hashing***
    - ***Close hashing.***

Open Hashing

- In Open hashing method, Instead of overwriting older one the next available data block is used to enter the new record, This method is also known as linear probing.
- For example, A2 is a new record which you wants to insert. The hash function generates address as 222. But it is already occupied by some other value. That's why the system looks for the next data bucket 501 and assigns A2 to it.



Close Hashing

In the close hashing method, when buckets are full, a new bucket is allocated for the same hash and result are linked after the previous one.

Closed Hashing (also called **Open Addressing**) is a hashing technique where all elements are stored **inside the hash table itself**—no separate linked lists or overflow structures are used.

•When a key is hashed, its index is calculated using a hash function h(key).
•If that slot is **empty**, insert the key there.
•If that slot is **already occupied** (collision occurs), we probe (search) for the next available slot in the table using a defined sequence.

Example

Suppose:
•Table size = 7
•Hash function: h(key) = key % 7
•Keys to insert: 10, 20, 15, 7

Insertion:
1.10 → 10 % 7 = 3 → place at index 3
2.20 → 20 % 7 = 6 → place at index 6
3.15 → 15 % 7 = 1 → place at index 1
4.7 → 7 % 7 = 0 → place at index 0

If we now insert 17:
•17 % 7 = 3 → index 3 is full (10)
•Linear probe: check 4 → empty → place 17 at index 4.

- **Advantages**

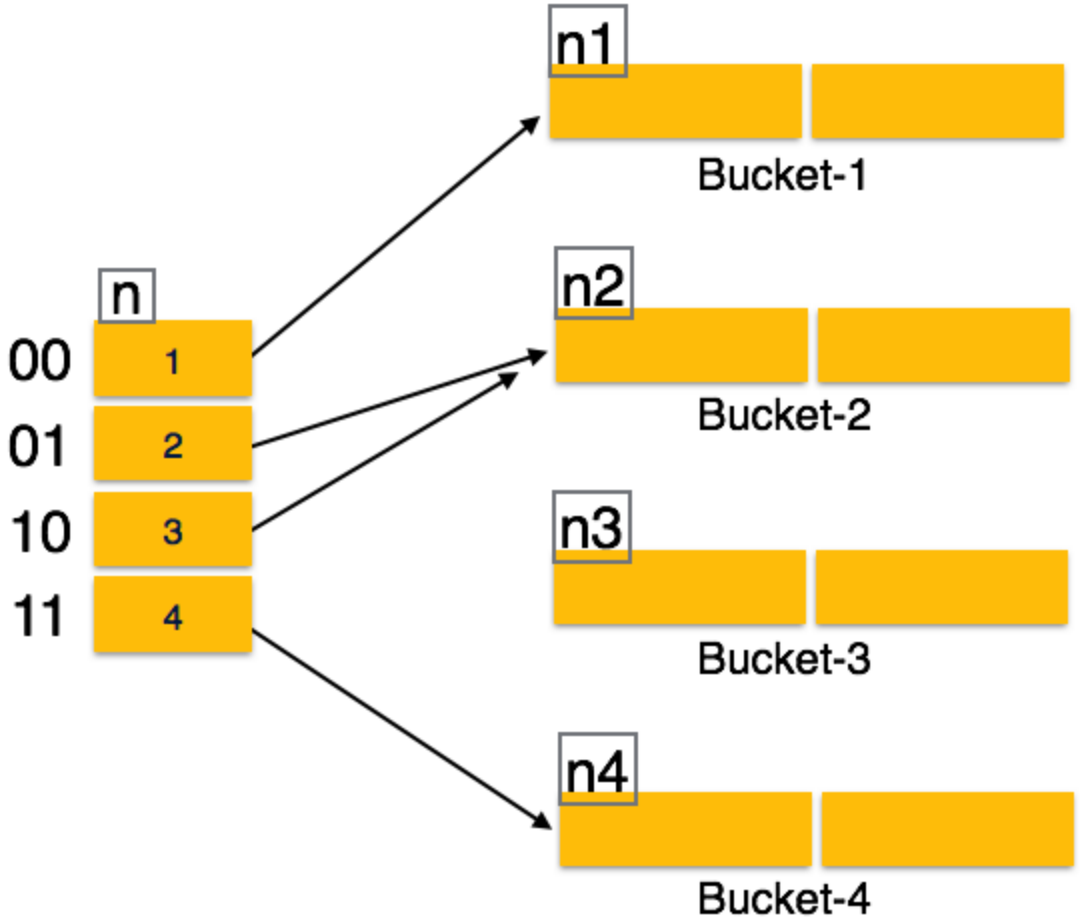
No separate memory allocation for linked lists.
Can be faster due to better cache performance.

- **Disadvantages**

Performance degrades as the table fills up.
Needs careful handling of deletion (special "deleted" markers are used).
Clustering problems can occur, especially with linear probing.

Dynamic Hashing

- The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as extended hashing.
- Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially.


n1
Bucket-1
n2
Bucket-2
n3
Bucket-3
n4
Bucket-4
n
00
01
10
11
1
2
3
4

Operation

- Querying − Look at the depth value of the hash index and use those bits to compute the bucket address.
- Update − Perform a query as above and update the data.
- Deletion − Perform a query to locate the desired data and delete the same.
- Insertion − Compute the address of the bucket
    - If the bucket is already full.
        - Add more buckets.
        - Add additional bits to the hash value.
        - Re-compute the hash function.
    - Else
        - Add data to the bucket,
    - If all the buckets are full, perform the remedies of static hashing.

Hashing Cont….

Hashing is not favorable when the data is organized in some ordering and the queries require a range of data. When data is discrete and random, hash performs the best.

Hashing algorithms have high complexity than indexing. All hash operations are done in constant time.

Query Processing & Optimization

Query processing:

The activities include translation of queries from high-level database languages into expressions that can be used at the physical level of the file system, a variety of query-optimizing transformations, and actual evaluation of queries.

steps involved in processing a query:

1. Parsing and translation.

2. Optimization.

3. Evaluation.

1. Parsing and translation

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.

The translation phase also replaces all uses of the view by the relational-algebra expression that defines the view.

As an illustration, consider the query:

select salary from instructor

where salary < 75000;

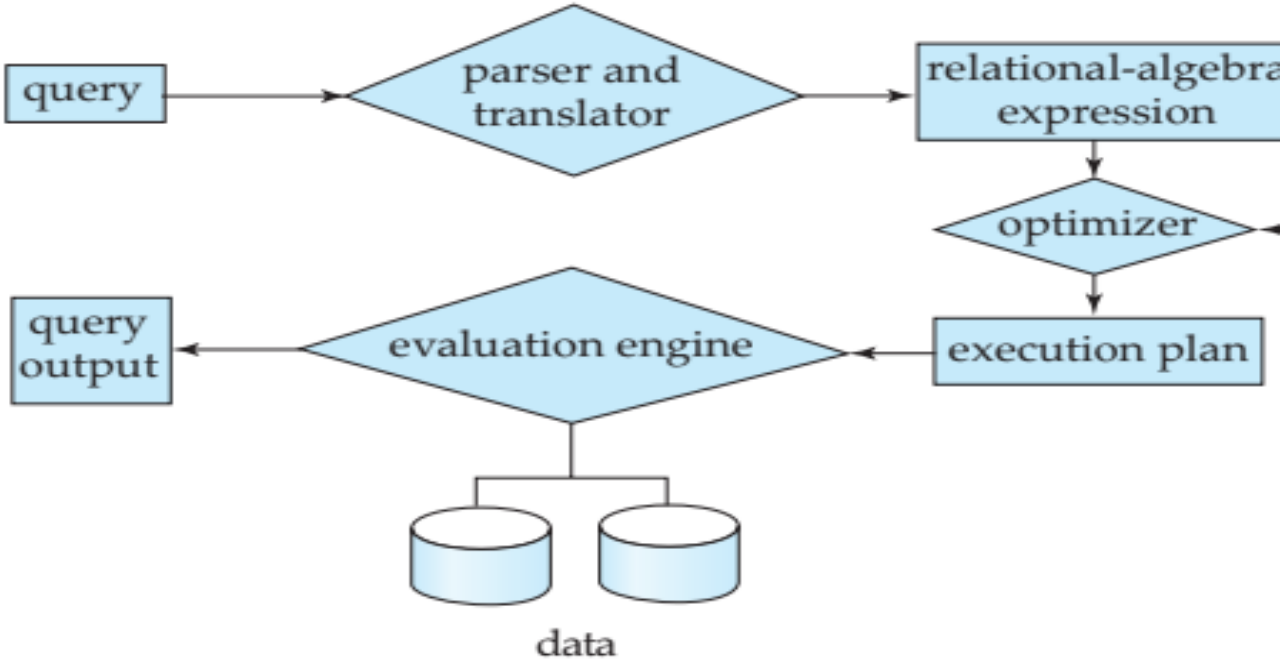


Figure 12.1 Steps in query processing.

This query can be translated into either of the following relational-algebra expressions:

- $\sigma_{salary\ <75000}\ (\Pi_{salary}\ (instructor))$
- $\Pi_{salary}\ (\sigma_{salary<75000}\ (instructor))$

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.

Query Optimization

The different evaluation plans for a given query can have different costs. We do not expect users to write their queries in a way that suggests the most efficient evaluation plan. Rather, 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.

In order to optimize a query, a query optimizer must know the cost of each operation.

Several operations may be grouped together into a pipeline, in which each of the operations starts working on its input tuples even as they are being generated by another operation. We examine how to coordinate the execution of multiple operations in a query evaluation plan, in particular, how to use pipelined operations to avoid writing intermediate results to disk.

Query Optimization:

The process of selecting the most efficient query-evaluation plan.

The relational-algebra level, where the system attempts to find an expression that is equivalent to the given expression, but more efficient to execute.

Choosing the algorithm to use for executing an operation, choosing the specific indices to use, and so on.

The difference in cost (in terms of evaluation time) between a good strategy and a bad strategy is often substantial, and may be several orders of magnitude.

Topics covered in Query Optimization are,

- Query Optimization Overview
- Transformation of Relational Expression
- Estimating Statistics of Expression Results.
- Choice of Evaluation Plan.

Database Schema:
•Student(SID, Name, Age, DeptID)
•Department(DeptID, DeptName)

Query:
SELECT Name FROM Student
WHERE DeptID = 10 AND Age > 18;

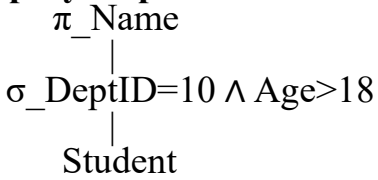
2. Relational Algebra Expression
•σ = **Selection**
•π = **Projection**
•∧ = AND (logical conjunction)

Relational Algebra:

π_Name (σ_DeptID=10 ∧ Age>18 (Student))

- **3. Parse Tree**

Step-by-step Parse Tree:



This tree tells us:

1.First apply selection on Student to filter DeptID = 10 AND Age > 18

2.Then apply projection to get only the Name attribute

Cost-Based Query Optimization

- Student table has 10,000 tuples/records.
- Each tuple/record is 100 bytes.
- Page size/ Block size = 1,000 bytes → 10 tuples/records per page → 1,000 pages
- DeptID=10 returns 1,000 students
- Age>18 returns 5,000 students
- Combined DeptID=10 ∧ Age>18 returns 800 students

Unoptimized Evaluation Order (as in parse tree)

1.Full table scan on Student: read 1,000 pages

2.Apply selection DeptID=10 ∧ Age>18 → filter to 800 tuples

3.Project Name

4.Cost: 1,000 I/Os

Optimized Evaluation Using Selection Pushdown and Index (if available)

If there are **indexes**:
•Suppose we have an **index on DeptID**
•Use index to directly retrieve matching rows for DeptID=10
Let's assume:
•DeptID=10 → 100 pages (based on index)
•Now within those, apply Age>18 → ~800 tuples
•Then project Name
Optimized Cost: ~100 pages I/O (much better than 1,000)

- Relational algebra allows splitting:

π_Name (σ_DeptID=10 ∧ Age>18 (Student))

≡ π_Name (σ_Age>18 (σ_DeptID=10 (Student)))

This lets the optimizer use indexes or filters early, reducing intermediate result size.

Final Optimized Relational Algebra (after transformations)

text

π_Name (σ_Age>18 (σ_DeptID=10 (Student)))

Summary Table

Step	Operation	Cost Estimate	Optimization Technique
Original Query	π_Name (σ_DeptID=10 ∧ Age>18 (Student))	1,000 pages	Full scan, no optimization
Optimized with Index	σ_DeptID=10 first, then Age>18	~100 pages	Use index + selective filtering
Logical Optimization	Split σ into σ_DeptID then σ_Age	~100 pages	Selection pushdown + reordering

Solved Example 2

Query :

SELECT Name, DeptName

FROM Student S

JOIN Department D ON S.DeptID = D.DeptID

WHERE D.DeptName = 'Computer Science' AND S.Age > 18;

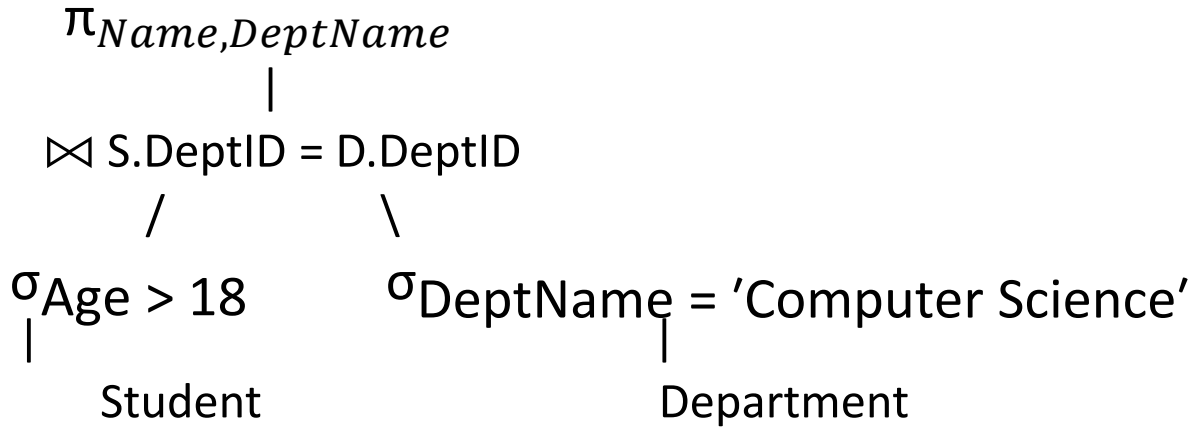
Schema

•Student(SID, Name, Age, DeptID)
•Department(DeptID, DeptName)

Relational Algebra Expression (with selection pushdown)

We apply **selection** to both relations **before the join** (i.e., *selection pushdown*):
text
π_Name, DeptName ((σ_Age > 18 (Student)) ⋈ S.DeptID = D.DeptID (σ_DeptName = 'Computer Science' (Department)))

- Final Optimized Parse Tree



- **π (Projection):** Selects only the columns Name and DeptName.
- **⋈ (Join):** Performs the join on S.DeptID = D.DeptID.
- **σ (Selection):**
  - Filters:
  - Student tuples where Age > 18
  - Department tuples where DeptName = 'Computer Science'
- **Leaves:** Base tables Student and Department.

Table	Rows	Row Size	Pages (approx)
Student	10,000	100 B	1,000 pages
Department	100	50 B	10 pages

•Age > 18 matches ~50% → 5,000 rows (500 pages)
•DeptName = 'Computer Science' matches ~1 row (1 page)
•Assume DeptID is foreign key (many-to-one)

1. Nested Loop Join (Naive)
Outer: 500 pages (filtered Student)
Inner: 1 page (filtered Department)
Total Cost = 500 (outer scan) + (500 × 1) = **1,000 I/Os**

2. Block Nested Loop Join
Same as above: scan Student blocks, reuse Department page
Similar I/O: **~1,000 I/Os**

3. Index Nested Loop Join

If **index on D.DeptID**, then:

Read 500 pages (Student)

For each tuple, use index to get matching dept

Lookup cost: 5000 × 1 I/O (assuming clustered index) = 5,000 I/Os

Total: 500 (Student) + 5,000 = **5,500 I/Os**

4. Hash Join (Best for Equality Joins)

Build hash table on smaller table (Department)

Probe with 500 pages of Student

Cost = 1 (build) + 500 (probe) = **~501 I/Os**

Final Optimized Physical Plan

```
π_Name, DeptName (
  HashJoin on S.DeptID = D.DeptID
    ├── σ_S.Age > 18 (Student)   → 500 pages
    └── σ_D.DeptName = 'Computer Science' (Department) → 1 page
)
```

Total I/O ≈ **501 I/Os**

Summary

Plan	Technique	I/O Estimate	Comments
Unoptimized	Join → Selection → Projection	~10,000+	Full scan, very inefficient
Nested Loop Join	Selection + join	~1,000	Okay, but scales poorly
Index Nested Loop	Needs index	~5,500	Worse than nested loop in this case
Hash Join (Best)	Selection + Hash Join	**~501**	Best for equality join, small input

Query Optimization

Query Optimization:

The process of selecting the most efficient query-evaluation plan.

The relational-algebra level, where the system attempts to find an expression that is equivalent to the given expression, but more efficient to execute.

Choosing the algorithm to use for executing an operation, choosing the specific indices to use, and so on.

The difference in cost (in terms of evaluation time) between a good strategy and a bad strategy is often substantial, and may be several orders of magnitude.

Query Optimization Cont...

Topics covered in Query Optimization are,

- ❑ Query Optimization Overview
- ❑ Transformation of Relational Expression
- ❑ Estimating Statistics of Expression Results.
- ❑ Choice of Evaluation Plan.

Query Optimization Cont...

Topics covered in Query Optimization are,

- Query Optimization Overview

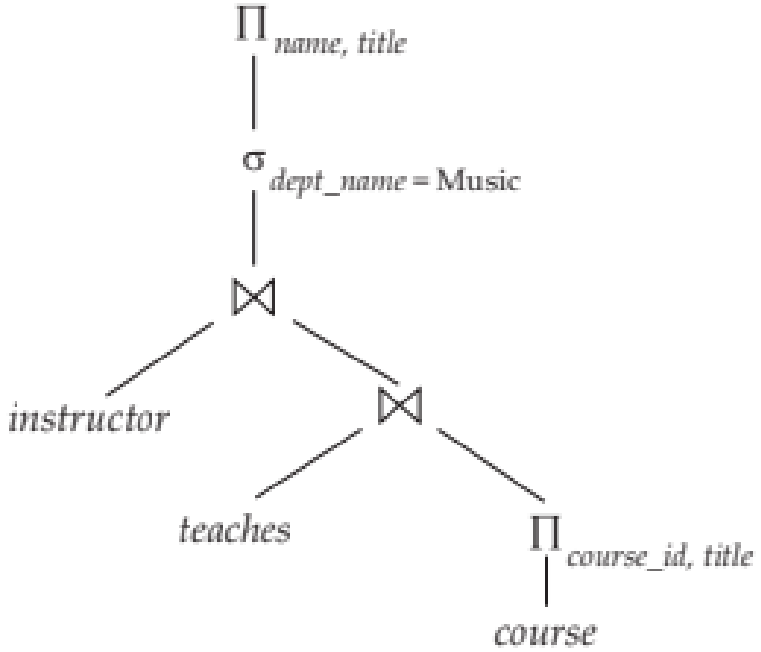
Consider the following relational-algebra expression, for the query "Find the names of all instructors in the Music department together with the course title of all the courses that the instructors teach."

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

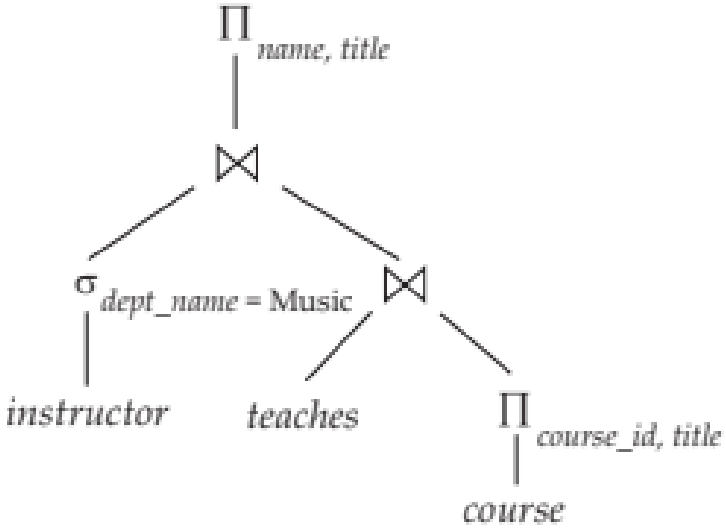
Note that the projection of course on (course id,title) is required since course shares an attribute dept name with instructor; if we did not remove this attribute using the projection, the above expression using natural joins would return only courses from the Music department, even if some Music department instructors

Query Optimization Cont...

The above expression constructs a large intermediate relation



(a) Initial expression tree



(b) Transformed expression tree

Query Optimization Cont...

Transformed expression tree will take less time to find or project the output, because it directly select dept_name =”Music” from instructor.

Now the query is represented by relation algebra expression.

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

An evaluation plan defines exactly what algorithm should be used for each operation, and how the execution of the operation should be coordinated.

- Different Operation such as, Hash Join, Merge Join/Sorted Join etc..
- ID attributes helps to merge / sort the relation where edges are marked as pipelined, the output of the producer is pipelined directly to the consumer, without being written out to disk.
- Given a relational-algebra expression, it is the job of the query

Query Optimization Cont...

- To find the least-costly query-evaluation plan, the optimizer needs to generate alternative plans that produce the same result as the given expression, and to choose the least-costly one.

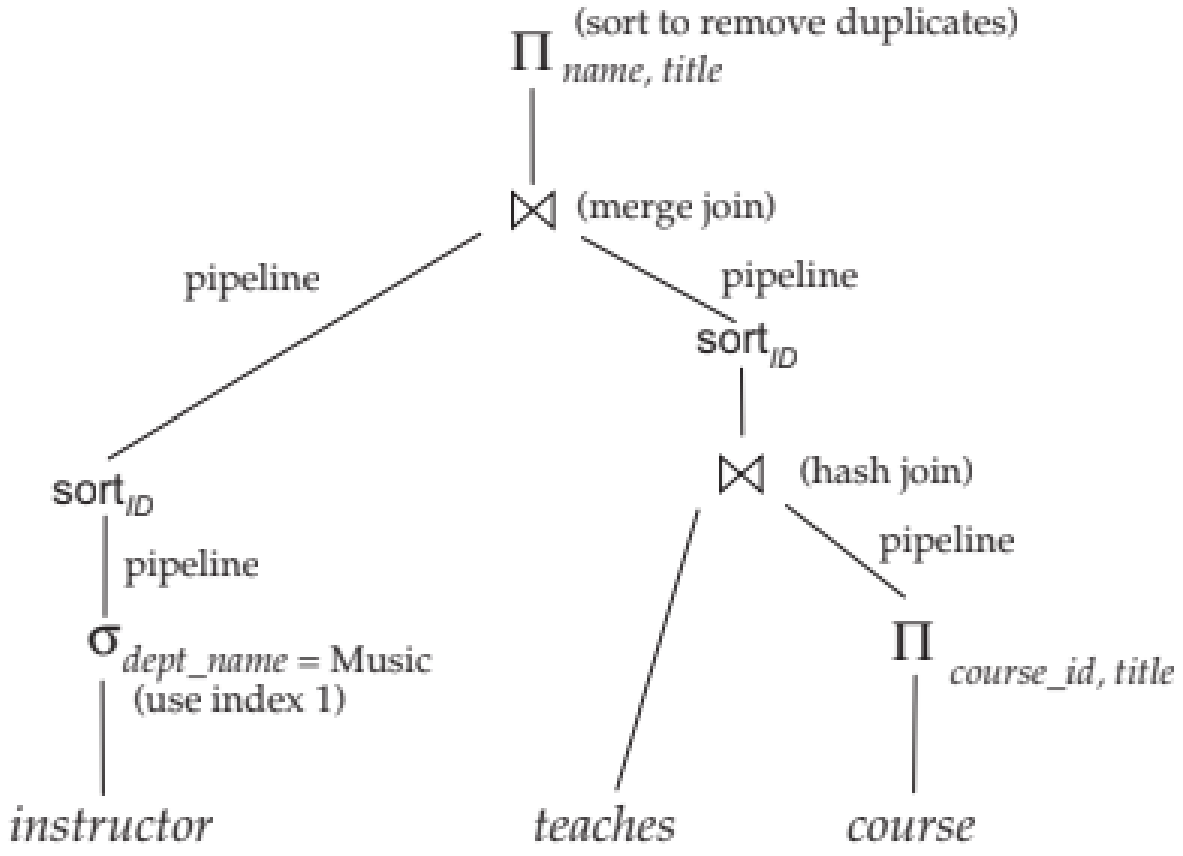
Generation of query-evaluation plans involves three

steps:

1. Generating expressions that are logically equivalent to the given expression,
2. Annotating the resultant expressions in alternative ways to generate alternative query-evaluation plans, and
3. Estimating the cost of each evaluation plan, and choosing the one

Query Optimization Cont...

An query-evaluation plans



Query Optimization Cont...

Alternatives for evaluating an entire expression tree –

Materialization: Materialize (i.e., store into temporary relations in the disk) intermediate results from lower-level operations, and use them as inputs to upper-level operations.

Materialized evaluation: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations

Pipelining: In pipelining approach,several relational operations are combined into a pipeline of operations,in which the results of one operation are passed along to the next operation in the pipeline.

Much cheaper than materialization: no need to store a temporary relation to

Query Optimization Cont...

- Transformation of 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.

  The two expressions generate the same set of tuples.

  The order of the tuples is irrelevant.

  The two expressions may generate the tuples in different orders, but would be considered equivalent as long as the set of tuples is the same.

Query Optimization Cont...

Equivalence Rules:

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.

We can replace an expression of the second form by an expression of the first form.

The two expressions generate the same result on any valid database.

List a number of general equivalence rules on relational-algebra expressions.

and so on to denote predicates, $\theta, \theta_1, \theta_2,$

Query Optimization Cont...

Equivalence Rules:

A relation name r is simply a special case of a relational-algebra expression, and can be used wherever E appears.

List of Equivalence Rules:

From Textbook (Database System Concepts, Abraham Silberschatz, Henry F. Korth, S. Sudarshan, 6th Edition) Page No. 583 - 585

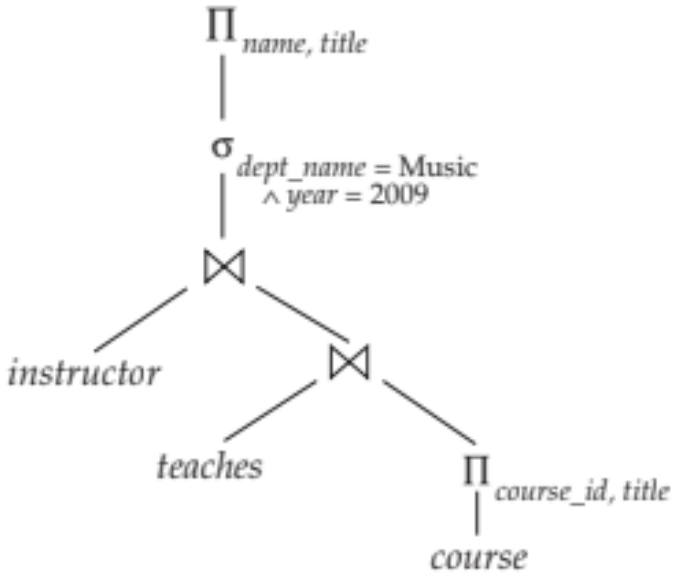
Example of Transformation:

Solve unive

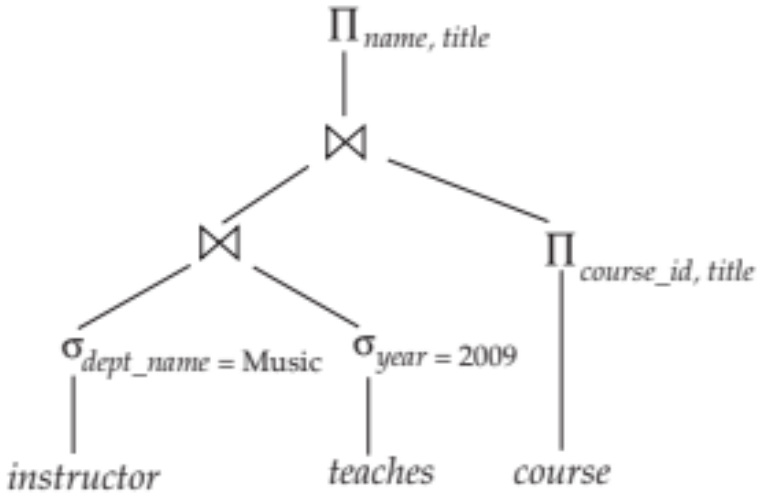
instructor(*ID*, *name*, *dept_name*, *salary*)
teaches(*ID*, *course_id*, *sec_id*, *semester*, *year*)
course(*course_id*, *title*, *dept_name*, *credits*)

Query Optimization Cont...

Multiple equivalence rules: Example from (Database System Concepts, Abraham Silberschatz, Henry F. Korth, S. Sudarshan, 6th Edition) Page No. 586 - 588



(a) Initial expression tree



(b) Tree after multiple transformations

Thank you….