## Data on External Storage in DBMS

The storage system in a DBMS refers to the hierarchical arrangement of storage devices and media to store, manage, and retrieve data efficiently.

And to handle different storage capacities, access speeds, volatility, and costs.

### Storage System Hierarchy in DBMS

The storage hierarchy typically has multiple levels, from fastest (and usually most expensive per byte) to slowest (and usually least expensive per byte):

- **1. Registers**
  - Located within the CPU.
  - Smallest and fastest type of storage.
  - Used to hold data currently being processed.
- **2. Cache Memory (L1, L2, L3 caches)**
  - On or very close to the CPU.
  - Extremely fast but small in size.
  - Acts as a buffer for frequently used data.

- **3. Main Memory (RAM)**
  - Data that's actively being used or processed is loaded here.
  - Faster than secondary storage.
  - Volatile in nature (i.e., data is lost when power is turned off).
- **4. Flash Storage (Solid State Drives - SSD)**
  - No moving parts.
  - Faster than traditional hard drives.
- More durable and reliable

- **5. Magnetic Disks (Hard Disk Drives - HDD)**
  - Primary secondary storage medium.
  - Non-volatile, persistent storage.
  - Data is stored in tracks, sectors, and cylinders.
  - Slower than main memory but offers a large storage capacity at a lower cost.
- **6. Optical Disks (CD, DVD, Blu-Ray)**
  - Data is read using lasers.
  - Slower than magnetic disks and usually have less storage capacity.
  - Portable and commonly used for media and software distribution.

- **7. Magnetic Tapes**
  - Sequential access storage, unlike disks which are random access.
  - Often used for backups and archiving due to their high capacity and low cost.
  - Much slower access times compared to magnetic disks.
- **8. Remote Storage/Cloud Storage**
  - Data stored in remote servers and accessed over the internet.
  - Provides scalability, availability, and fault-tolerance.
  - Latency depends on network speed and distance to servers.

## • Types of Storage:

1. **Primary Storage:** Includes registers, cache memory, and main memory (RAM). It's the main memory where the operating system, application programs, and data in current use are kept for quick access by the computer's processor.
2. **Secondary Storage:** Encompasses data storage devices like HDDs, SSDs, CDs, and USB drives. It is non-volatile and retains data even when the computer is turned off.
3. **Tertiary Storage or Off-line Storage:** Often involves magnetic tape systems or optical disk archives. This is slower than secondary storage and is used for data archiving and backup.
4. **Quaternary Storage:** Refers to methods like cloud storage or other remote storage techniques where data is stored in remote servers and is fetched over the internet or other networks.

## File Organization

- File organization refers to the arrangement of data on storage devices.
- **Sequential (or Serial) File Organization**
- In sequential file organization, records are stored in sequence, one after the other, based on a key field.
- This key field is a unique identifier for records, ensuring that they have some order.
- The records are inserted at the end of the file, ensuring the sequence is maintained.

- **Features of Sequential File Organization**
  - **Ordered Records:** Records in a sequential file are stored based on a key field.
  - **Continuous Memory Allocation:** The records are stored in contiguous memory locations.
  - **No Direct Access**: To access a record, you have to traverse from the first record until you find the desired one.

- **Advantages Sequential File Organization**
  - Simplicity.
  - Efficient for Batch Processing
  - Less Overhead
- **Disadvantages Sequential File Organization**
  - Inefficient for Random Access
  - Insertion and Deletion.
- Redundancy Issues

| Roll No | Name |
|---|---|
| 1 | Madhu |
| 2 | Naveen |
| 4 | Shivaji |
| 5 | Durga |

## Direct (or Hashed) File Organization

- In hash file organization, a hash function is used to compute the address of a block (or bucket) where the record is stored.
- **Features of Hash File Organization**
  - **Hash Function:** A hash function converts a record's key value into an address.
  - **Buckets:** A bucket typically stores one or more records. A hash function might map multiple keys to the same bucket.
  - **No Ordering of Records:** Records are not stored in any specific logical order.

- **Advantages Hash File Organization**

  - **Rapid Access:** If the hash function is efficient and there's minimal collision, the retrieval of a record is very quick.
  - **Uniform Distribution:** A good hash function will distribute records uniformly across all buckets.
  - **Efficient Search:** Searching becomes efficient as only a specific bucket needs to be searched rather than the entire file.

- **Disadvantages Hash File Organization**

- **Collisions**:** A collision occurs when two different keys hash to the same bucket. Handling collisions can be tricky and might affect access time.
- **Dependency on Hash Function**:** The efficiency of a hash file organization heavily depends on the hash function used.
- A bad hash function can lead to clustering and inefficient utilization of space.
- **Dynamic Growth and Shrinking**:** If the number of records grows or shrinks significantly, rehashing might be needed which is an expensive operation.

- **Imagine a database that holds information about books**.
- Each book has a unique ISBN number.
- A hash function takes an ISBN and returns an address.
- When you want to find a particular book's details, you hash the ISBN, which directs you to a particular bucket.
- If two books' ISBNs hash to the same value, you handle that collision, maybe by placing the new record in a linked list associated with that bucket

## Indexed File Organization

- Indexed file organization is a method used to store and retrieve data in databases.
- It is designed to provide quick random access to records based on key values.
- In this organization, an index is created which helps in achieving faster search and access times.

- **Features Indexed File Organization:**

  - **Primary Data File:** The actual database file where records are stored.
  - **Index:** An auxiliary file that contains key values and pointers to the corresponding records in the data file.
  - **Multi-level Index:** Sometimes, if the index becomes large, a secondary (or even tertiary) index can be created on the primary index to expedite searching further.

## Advantages Indexed File Organization:

- **Quick Random Access:** Direct access to records is possible using the index.
- **Flexible Searches:** Since an index provides a mechanism to jump directly to records, different types of search operations (like range queries) can be efficiently supported.
- **Ordered Access:** If the primary file is ordered, then indexed file organization can support efficient sequential access too.

## Disadvantages Indexed File Organization:

- **Overhead of Maintaining Index:** Every time a record is added, deleted, or updated, the index also needs to be updated. This can introduce overhead.
- **Space Overhead:** Indexes consume additional storage space.
- **Complexity:** Maintaining multiple levels of indexes can introduce complexity in terms of design and implementation.

- **Consider a database that holds information about students.**
- where each student has a unique student ID. The main file would contain detailed records for each student.
- A separate index file would contain student IDs and pointers to the location of the detailed records in the main file.
- When you want to find a specific student's details, you first search the index to find the pointer and then use that pointer to fetch the record directly from the main file.

- **Indexed Sequential Access Method (ISAM)**
- ISAM is a popular method for indexed file organization. In ISAM:
- The primary file is stored in a sequential manner based on a primary key.
- There's a static primary index built on the primary key.
- Overflow areas are designated for insertion of new records, which keeps the main file in sequence.
- Periodically, the overflow area can be merged back into the main file.

## Indexing

- Indexing involves creating an auxiliary structure (an index) to improve data retrieval times.
- Just like the index in the back of a book, a database index provides pointers to the locations of records.

**Structure of Index**

We can create indices using some columns of the database.

```
|-----------------|----------------------|
|  Search Key  | Data Reference |
|-----------------|----------------------|
```

- The search key is the database’s first column, and it contains a duplicate or copy of the table’s candidate key or primary key.
- The primary key values are saved in sorted order so that the related data can be quickly accessible.
- The data reference is the database’s second column.
- It contains a group of pointers that point to the disk block where the value of a specific key can be found.

- **Types of Indexes:**

1. **Single-level Index:** A single index table that contains pointers to the actual data records.
2. **Multi-level Index:** An index of indexes. This hierarchical approach reduces the number of accesses (disk I/O operations) required to find an entry.
3. **Dense and Sparse Indexes:**
   - In a dense index, there's an index entry for every search key value in the database.
   - In a sparse index, there are fewer index entries. One entry might point to several records.
4. **Primary and Secondary Indexes:**
   - A primary index is an ordered file whose records are of fixed length with two fields. The first field is the same as the primary key, and the second field is a pointer to the data block
   - A secondary index provides a secondary means of accessing data. For each secondary key value, the index points to all the records with that key value.

**5.Clustered vs. Non-clustered Index:**

- In a clustered index, the rows of data in the table are stored on disk in the same order as the index. There can only be one clustered index per table.
- In a non-clustered index, the order of rows does not match the index's order. You can have multiple non-clustered indexes.

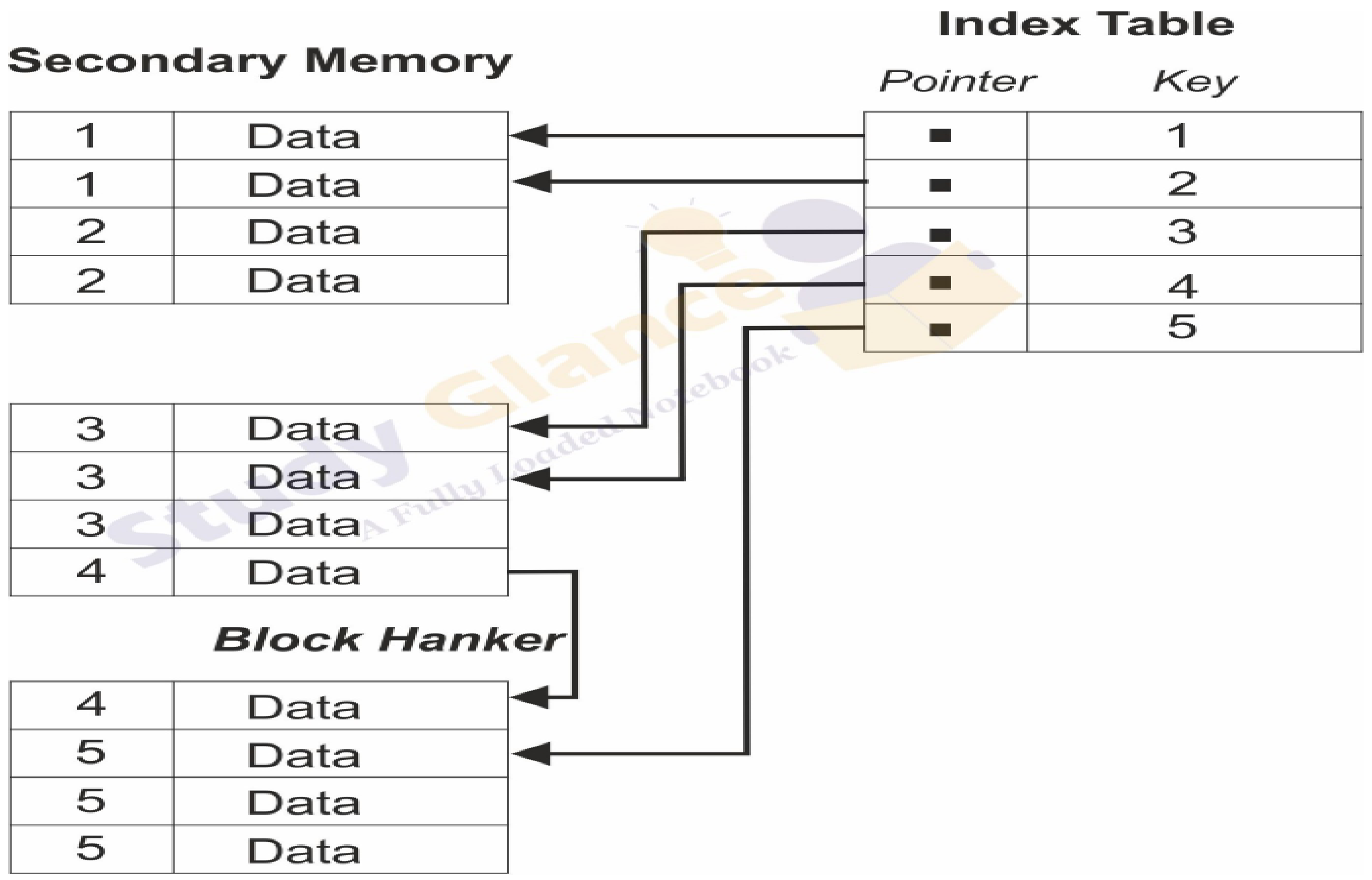
**6. Bitmap Index:** Used mainly for data warehousing setups, a bitmap index uses bit arrays (bitmaps) and usually involves columns that have a limited number of distinct values.

**7. B-trees and B+ trees:** Balanced tree structures that ensure logarithmic access time. B+ trees are particularly popular in DBMS for their efficiency in disk I/O operations.

- **Benefits of Indexing:**
  - Faster search and retrieval times for database operations.
- **Drawbacks of Indexing:**
  - Overhead for insert, update, and delete operations, as indexes need to be maintained.
  - Additional storage requirements for the index structures.

## Cluster Indexes in DBMS

- A clustered index determines the physical order of data in a table.
- In other words, the order in which the rows are stored on disk is the same as the order of the index key values.
- There can be only one clustered index on a table, but the table can have multiple non-clustered (or secondary) indexes.
-



Secondary Memory
Index Table
Pointer
Key
1
2
3
4
5
1 Data
1 Data
2 Data
2 Data
3 Data
3 Data
3 Data
4 Data
Block Hanker
4 Data
5 Data
5 Data
5 Data

- **Characteristics of a Clustered Index:**

1. It dictates the physical storage order of the data in the table.
2. There can **only be one clustered index** per table.
3. It can be created on columns with **non-unique values**, but if it's on a non-unique column, the DBMS will usually add a uniqueifier to make each entry unique.
4. clustered index is **fast** because the desired data is directly located without the need for additional lookups (unlike non-clustered indexes which require a second lookup to fetch the data).

- **Clustered Index Example**
- Imagine a `Books` table with the following records:
-

| BookID | Title | Genre |
|--------|-------|-------|
| 3 | A Tale of Two Cities | Fiction |
| 1 | Database Systems | Academic |
| 4 | Python Programming | Technical |
| 2 | The Great Gatsby | Fiction |

- If we create a clustered index on `BookID`, the physical order of records would be rearranged based on the ascending order of `BookID`.
-

| BookID | Title | Genre |
|---|---|---|
| 1 | Database Systems | Academic |
| 2 | The Great Gatsby | Fiction |
| 3 | A Tale of Two Cities | Fiction |
| 4 | Python Programming | Technical |

- Now, when you want to find a book based on its ID, the DBMS can quickly locate the data because the data is stored in the order of the BookID.

- **Benefits of a Clustered Index:**
  - **Fast Data Retrieval:** Because the data is stored sequentially in the order of the index key, range queries or ordered queries can be very efficient.
  - **Data Pages:** With the data being stored sequentially, the number of data pages that need to be read from the disk is minimized.
  - **No Additional Lookups:** Once the key is located using the index, there's no need for additional lookups to fetch the data, as it is stored right there.
- **Drawbacks of a Clustered Index:**
  - **Overhead on Inserts/Updates:** Because the data must be stored physically in the order of the index keys, inserts or updates can be slower since they might require data pages to be rearranged.
  - **Single Clustered Index:** You can have only one clustered index per table, so you have to choose wisely based on the most critical queries' requirements.

## • Primary Indexes in DBMS

- A primary index is an ordered file whose records are of fixed length with two fields.
- The first field is the same as the primary key of the data file,
- and the second field is a pointer to the data block where that specific key can be found.
- **The primary index can be classified into two types**

1. Dense Index: In this, an index entry appears for every search key value in the data file.
2. Sparse (or Non-Dense) Index: Here, index records are created only for some of the search key values. A sparse index reduces the size of the index file.

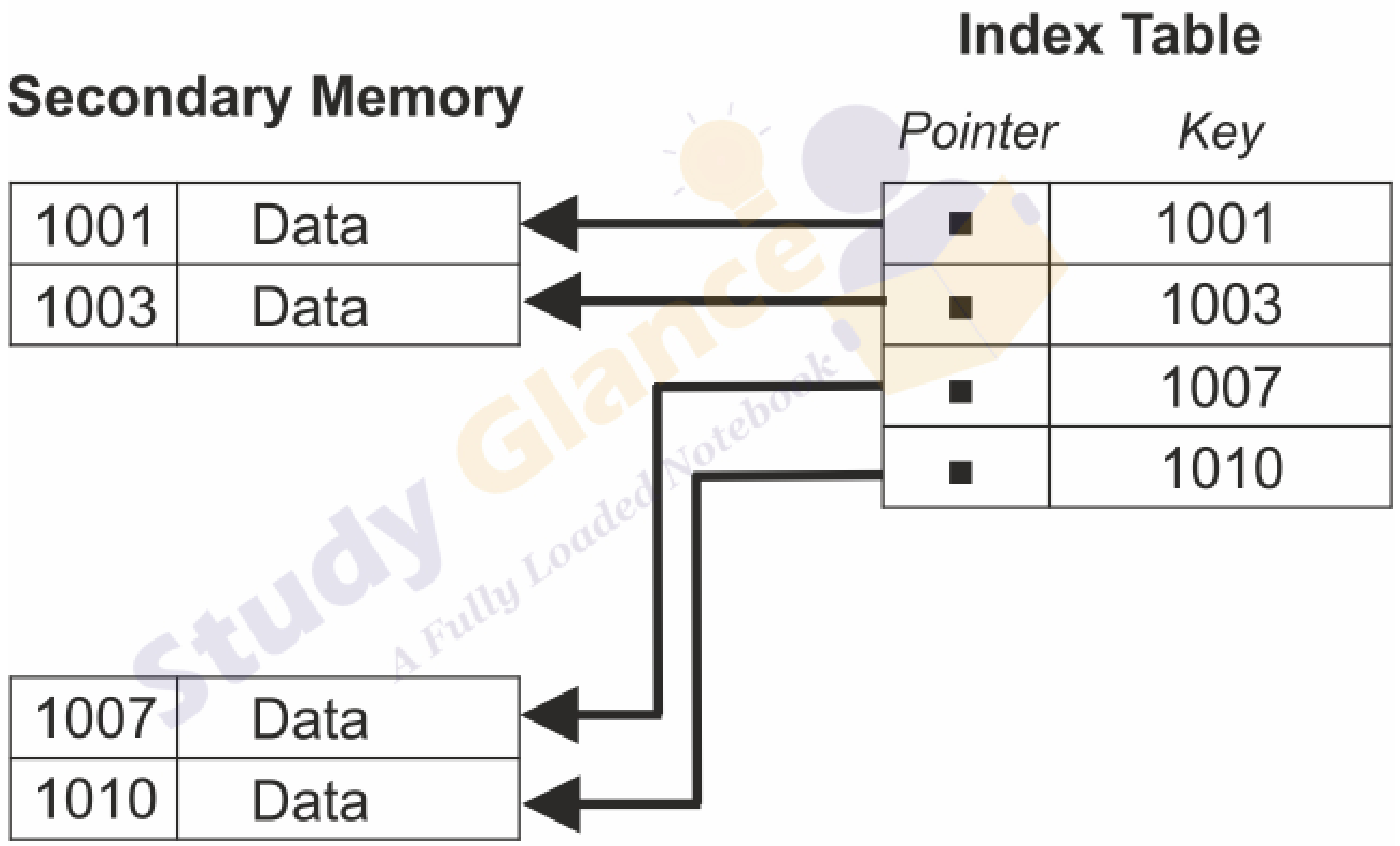
- **Characteristics of a Primary Index:**

1. It is based on the primary key of the table.
2. The primary key must have unique values.
3. The index entries are sorted in the same order as the data file (hence often a clustered index).
4. The index has one entry for each block in a data file, not for each record.

| RollNumber (Primary Key) | Name | Age |
|---|---|---|
| 1001 | Madhu | 20 |
| 1003 | Mahi | 22 |
| 1007 | Ramu | 21 |
| 1010 | Durga | 23 |

- Assuming each block of our storage can hold two records, our blocks will be:
  - Block 1: Contains records for RollNumbers 1001 and 1003.
  - Block 2: Contains records for RollNumbers 1007 and 1010.

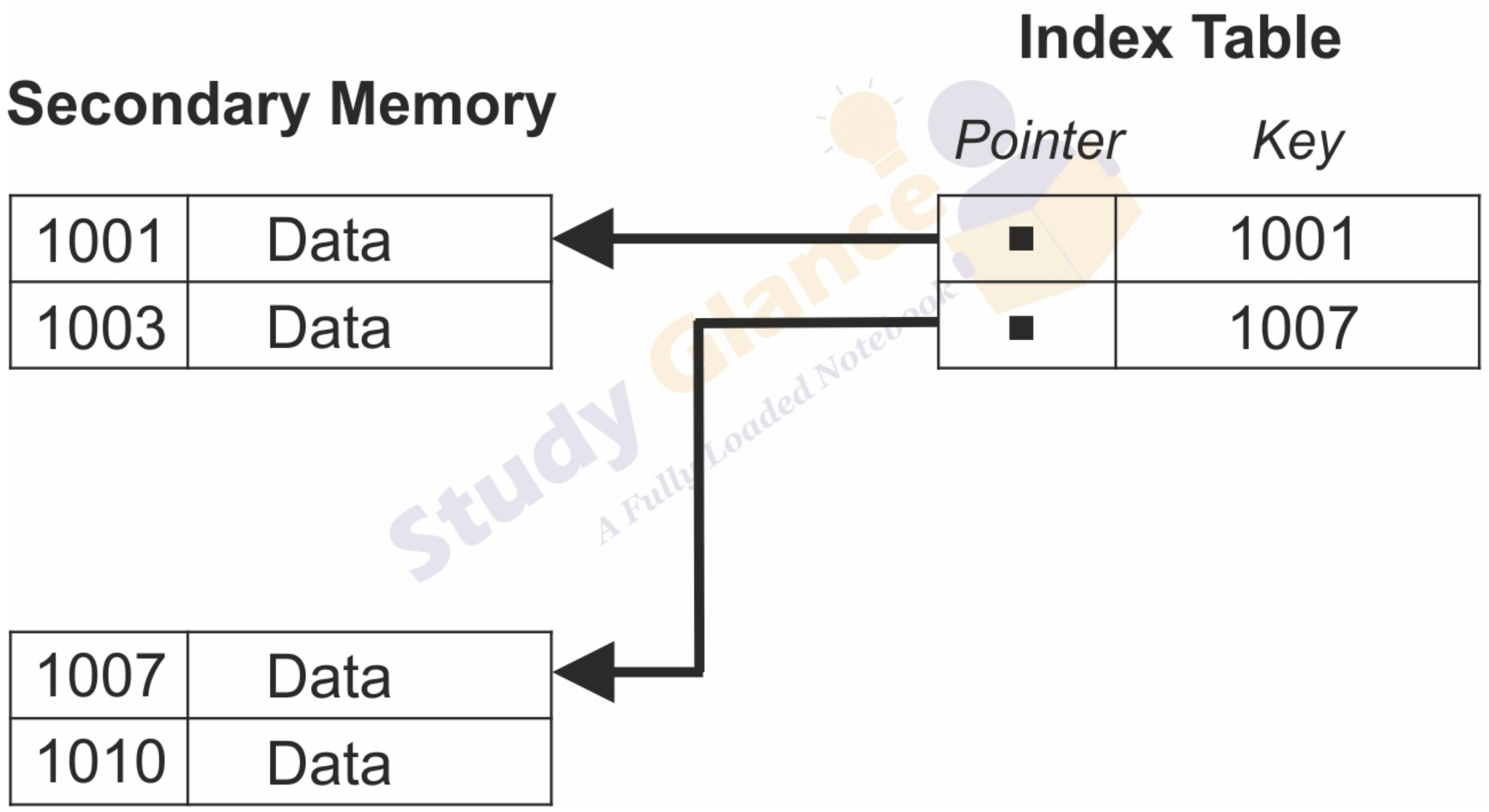
- **Dense Primary Index:**
- | Key (RollNumber) | Pointer to Block |
- |------------------|------------------|
- | 1001 | Block 1 |
- | 1003 | Block 1 |
- | 1007 | Block 2 |
- | 1010 | Block 2 |


Secondary Memory
Index Table
Pointer
Key
1001
Data
1003
Data
1007
Data
1010
Data
1001
1003
1007
1010

- **Sparse Primary Index:**

| Key (RollNumber) | Pointer to Block |
|---|---|
| 1001 | Block 1 |
| 1007 | Block 2 |

-



Index Table
Secondary Memory
Pointer
Key
1001
1007
1001
Data
1003
Data
1007
Data
1010
Data

- **Benefits of a Primary Index:**
  - Fast retrieval of records based on the primary key.
  - Efficient for range-based queries due to sorted order.
  - Can be used to enforce the uniqueness of the primary key.
- **Disadvantages**
- slower for insertions, updates, and deletions
- maintaining the sorted order of the index can be time-consuming

## Secondary Indexes in DBMS

- **Characteristics of a Secondary Indexes :**

1. Provides an alternative path to access the data.
2. Can be either dense or sparse.
3. Allows for non-unique values.
4. Does not guarantee the order of records in the data file.
5. secondary index is typically a non-clustered index. This means the physical order of rows in a table is not the same as the index order.

## Secondary Index Example

- Let's continue with the `Students` table:

| RollNumber (Primary Key) | Name | Age |
|---|---|---|
| 1001 | John | 20 |
| 1003 | Alice | 22 |
| 1007 | Bob | 21 |
| 1010 | Clara | 23 |

- Assuming we want to create a secondary index on the `**Age**` column:
- **Dense Secondary Index on Age:**
-  | Age | Pointer to Record |
- |-----|-------------------|
- | 20  | Record 1          |
- | 21  | Record 3          |
- | 22  | Record 2          |
- | 23  | Record 4          |
- Here, each age value has a direct pointer to the corresponding record.

- If another student with an age of 22 is added:

| RollNumber | Name | Age |
|---|---|---|
| 1012 | David | 22 |

- **The dense secondary index would then be:**

| Age | Pointer to Record |
|---|---|
| 20 | Record 1 |
| 21 | Record 3 |
| 22 | Record 2, Record 5 |
| 23 | Record 4 |

- **Benefits of a Secondary Index:**
  - Provides additional query paths, potentially speeding up query performance.
  - Can be created on non-primary key columns, even on columns with non-unique values.
  - Useful for optimizing specific query patterns.
- **Drawbacks of a Secondary Index:**
  - Increases the overhead insert, update, or delete operations
  - Consumes additional storage space.
  - May increase the complexity of database maintenance.

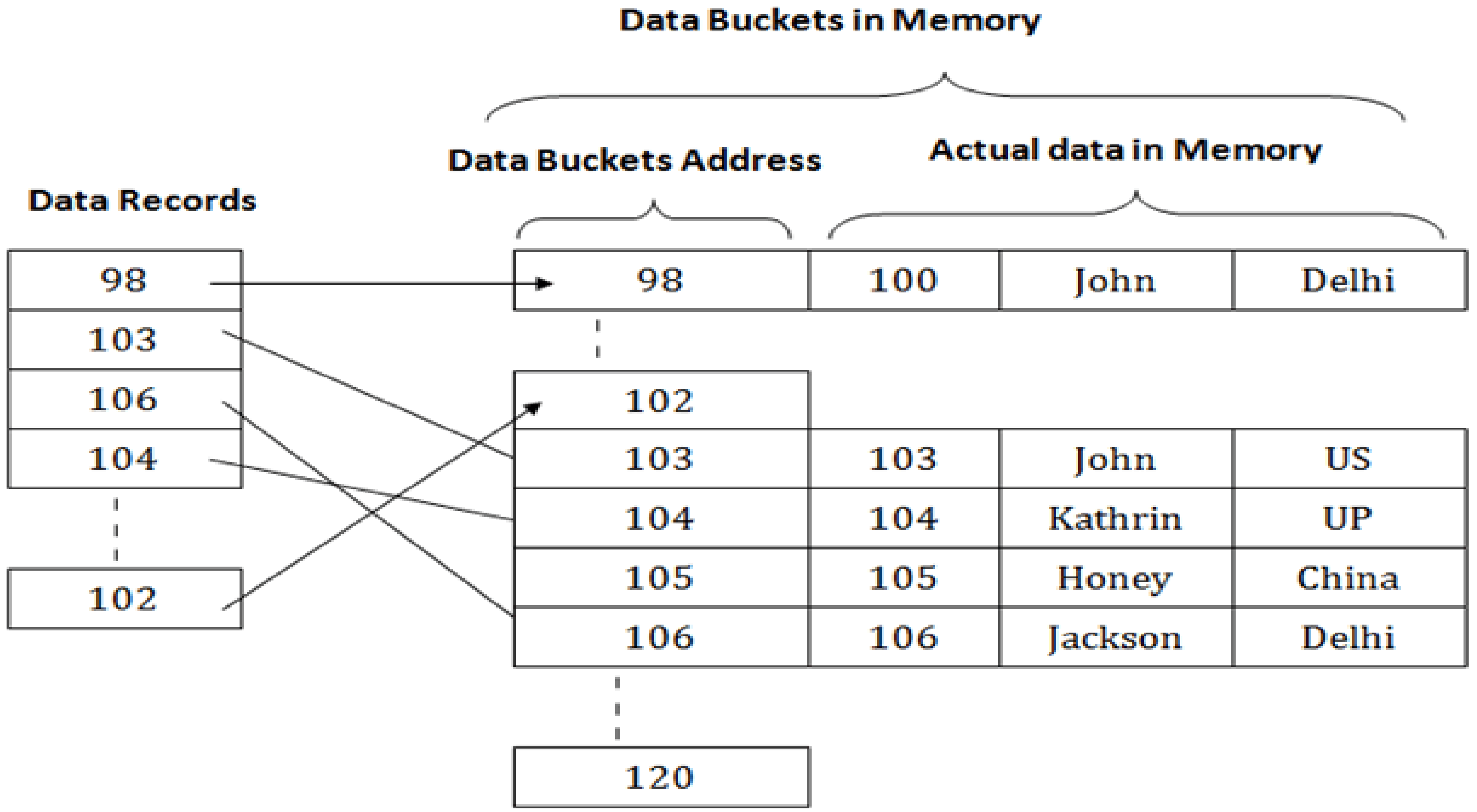
## Hash-Based Indexing

- In hash-based indexing, a hash function is used to convert a key into a hash code.
- This hash code serves as an index where the value associated with that key is stored.
- **Buckets**
- In hash-based indexing, the data space is divided into a fixed number of slots known as "buckets."
- A bucket usually contains a single page (also known as a block), but it may have additional pages linked in a chain if the primary page becomes full. This is known as overflow.

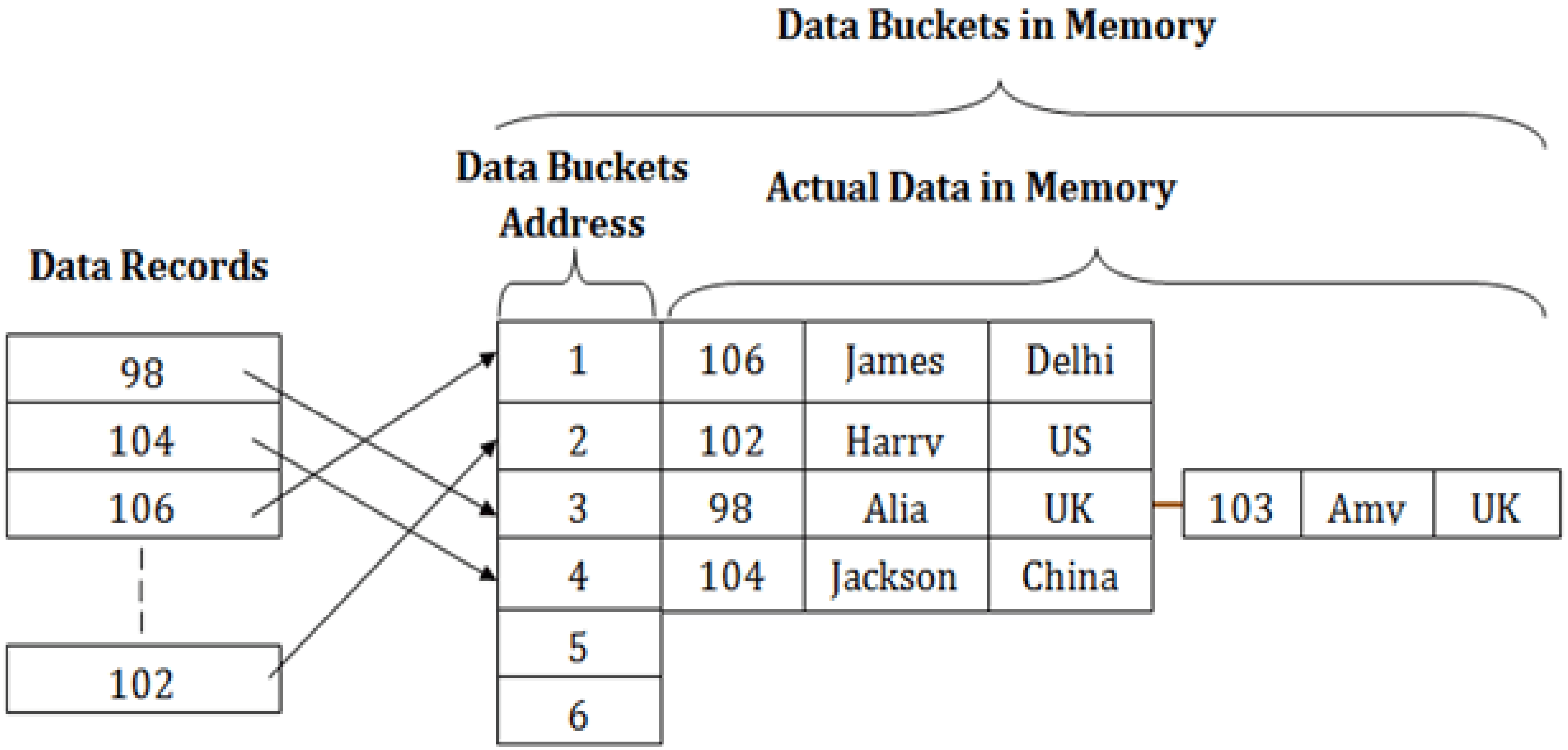
- **Hash Function**
- The hash function is a mapping function that takes the search key as an input and returns the bucket number where the record should be located.
- Hash functions aim to distribute records uniformly across buckets to minimize the number of collisions (two different keys hashing to the same bucket).

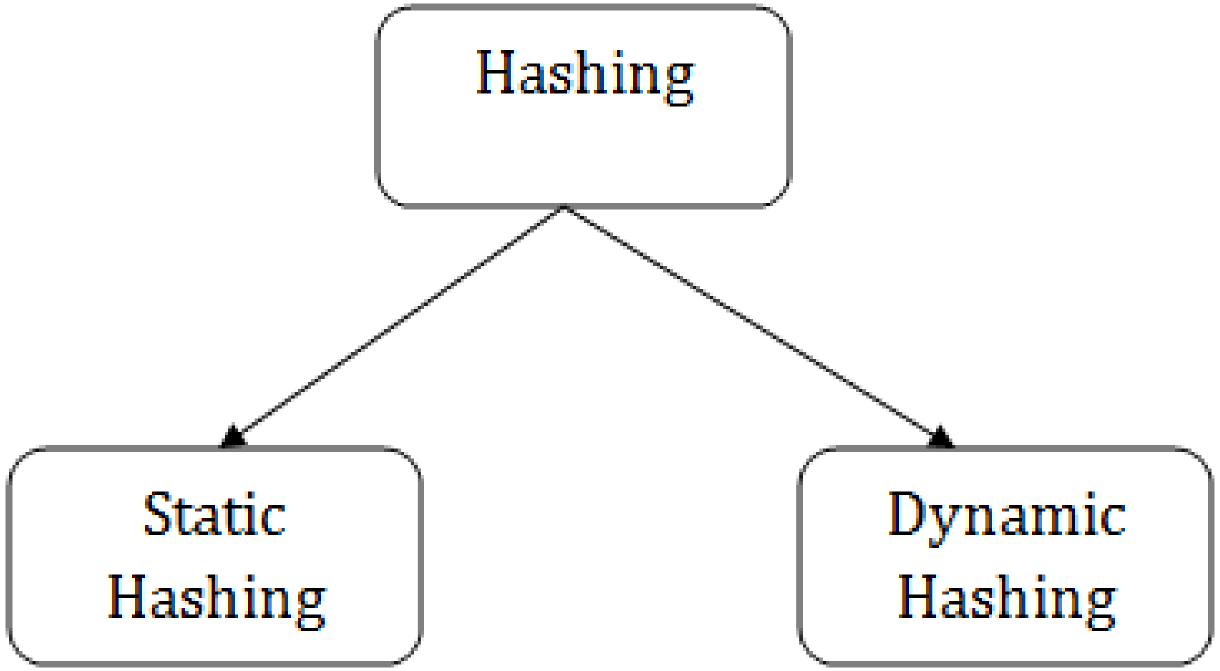
- **Disk I/O Efficiency**
- Hash-based indexing is particularly efficient when it comes to disk I/O operations. Given a search key, the hash function quickly identifies the bucket (and thereby the disk page) where the desired record is located. This often requires only one or two disk I/Os, making the retrieval process very fast.
- **Insert Operations**
- When a new record is inserted into the dataset, its search key is hashed to find the appropriate bucket. If the primary page of the bucket is full, an additional overflow page is allocated and linked to the primary page. The new record is then stored on this overflow page.

- **Search Operations**
- To find a record with a specific search key, the hash function is applied to the search key to identify the bucket. All pages (primary and overflow) in that bucket are then examined to find the desired record.
- **Limitations**
- Hash-based indexing is not suitable for range queries or when the search key is not known. In such cases, a full scan of all pages is required, which is resource-intensive.


Data Buckets in Memory
Data Buckets Address
Actual data in Memory
Data Records
98
103
106
104
102
98
100
John
Delhi
102
103
103
John
US
104
104
Kathrin
UP
105
105
Honey
China
106
106
Jackson
Delhi
120

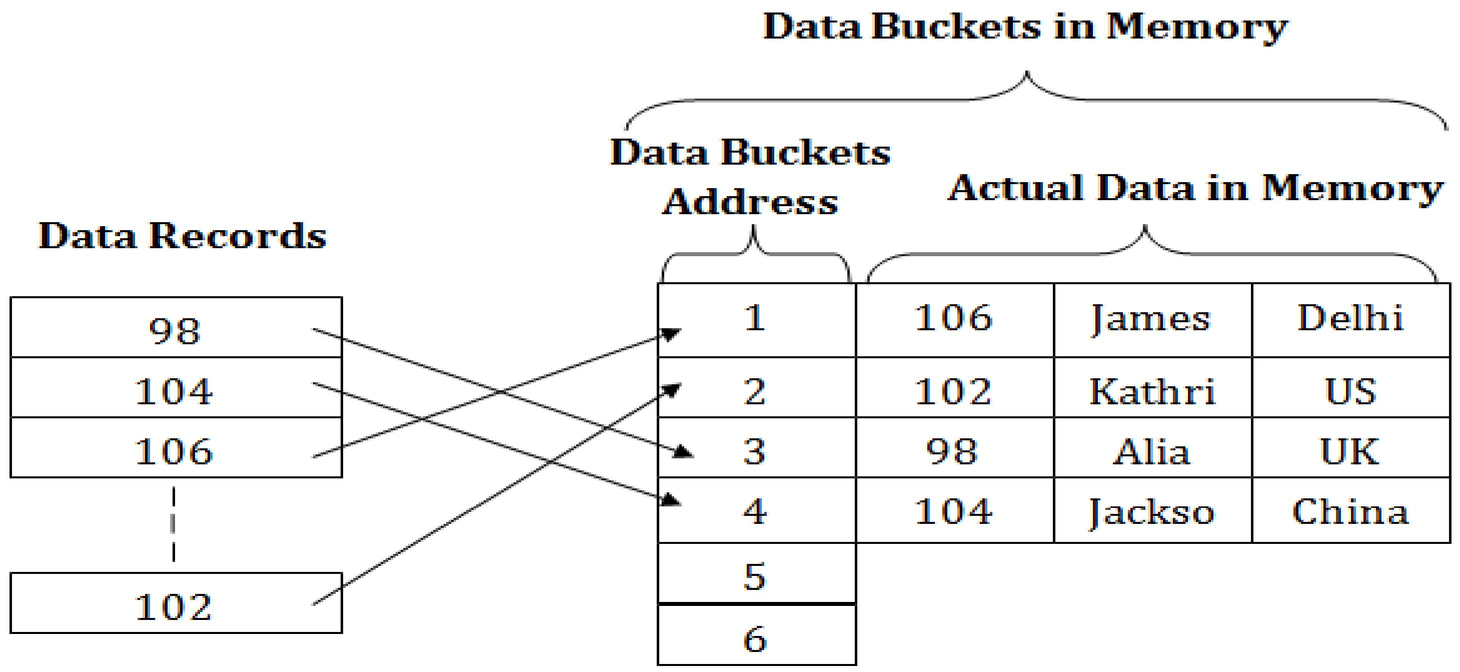
- In this case, it applies mod (5) hash function on the primary keys and generates 3, 3, 1, 4 and 2 respectively, and records are stored in those data block addresses.




Hashing
Static Hashing
Dynamic Hashing

## Static Hashing

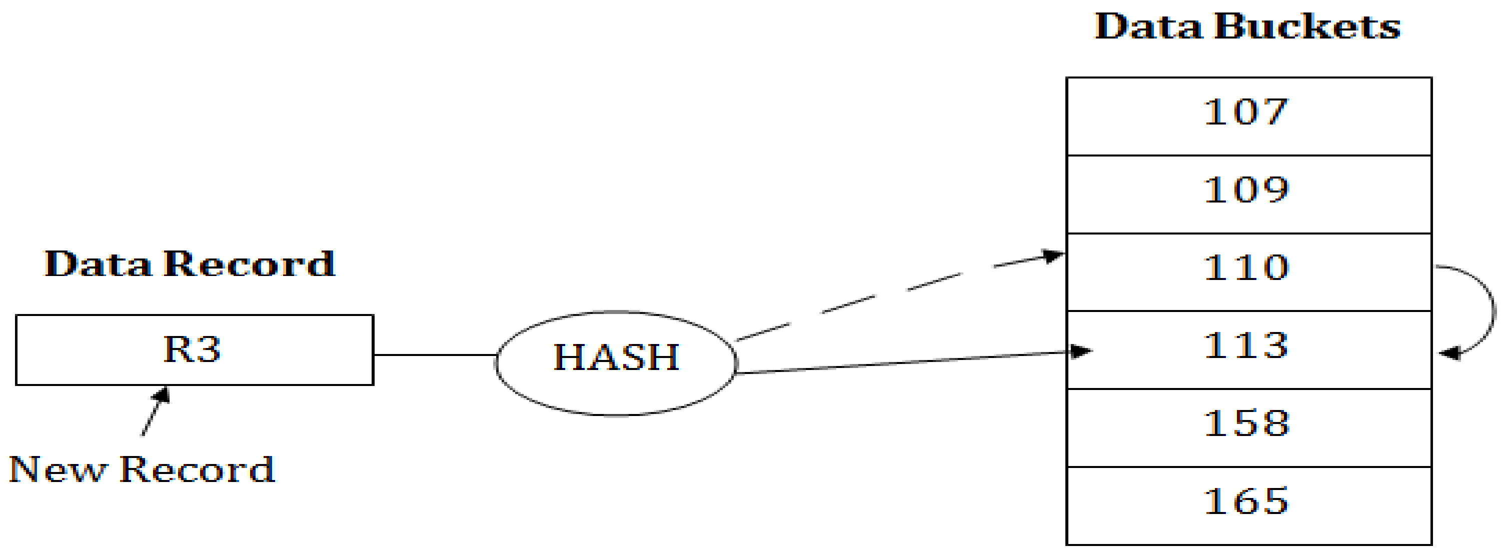
- this static hashing, the number of data buckets in memory remains constant throughout. In this example, we will have five data buckets in the memory used to store the data.
- That means if we generate an address for EMP_ID =103 using the hash function mod (5) then it will always result in same bucket address 3. Here, there will be no change in the bucket address.



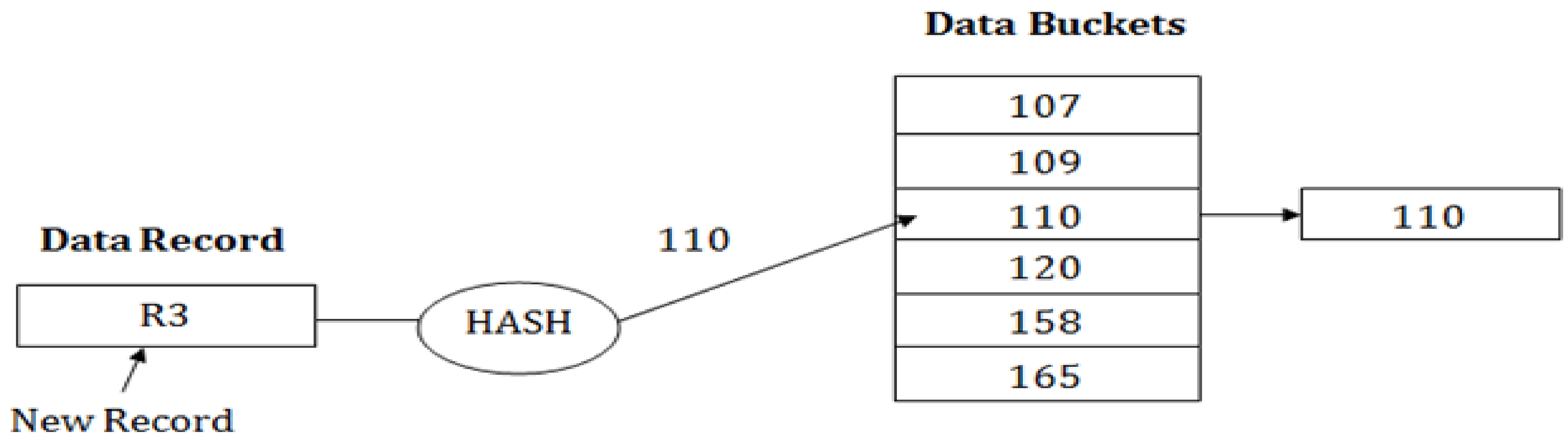
## Operations of Static Hashing

- **Searching a record**
- **Insert a Record**
- **Delete a Record**
- **Update a Record**
- If we want to insert some new record into the file but the address of a data bucket generated by the hash function is not empty, or data already exists in that address.
- This situation in the static hashing is known as **bucket overflow**. This is a critical situation in this method.
- To overcome this situation, there are various methods. Some commonly used methods are as follows:
- 1. Open Hashing
- 2. Close Hashing

- **1. Open Hashing**
- When a hash function generates an address at which data is already stored, then the next bucket will be allocated to it. This mechanism is called as **Linear Probing**.
- **For example:** suppose R3 is a new address which needs to be inserted, the hash function generates address as 112 for R3. But the generated address is already full. So the system searches next available data bucket, 113 and assigns R3 to it.



- **2. Close Hashing**
- When buckets are full, then a new data bucket is allocated for the same hash result and is linked after the previous one. This mechanism is known as **Overflow chaining**.
- **For example:** Suppose R3 is a new address which needs to be inserted into the table, the hash function generates address as 110 for it. But this bucket is full to store the new data. In this case, a new bucket is inserted at the end of 110 buckets and is linked to it.



## Dynamic Hashing

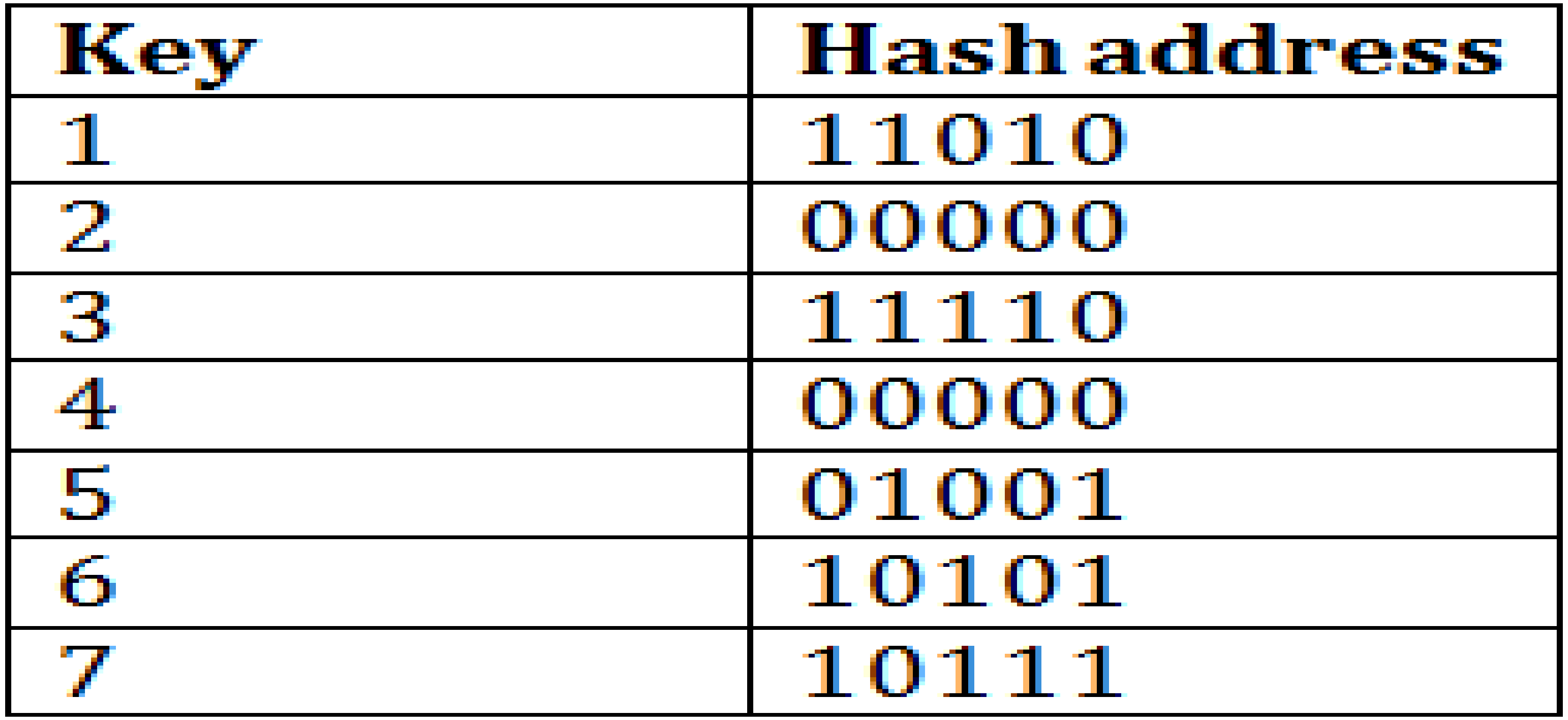
- The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
- In this method, data buckets grow or shrink as the records increases or decreases.
- This method is also known as **Extendable hashing method.**
- This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

- **How to insert a new record**

  - Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.
  - If there is still space in that bucket, then place the record in it.
  - If the bucket is full, then we will split the bucket and redistribute the records.

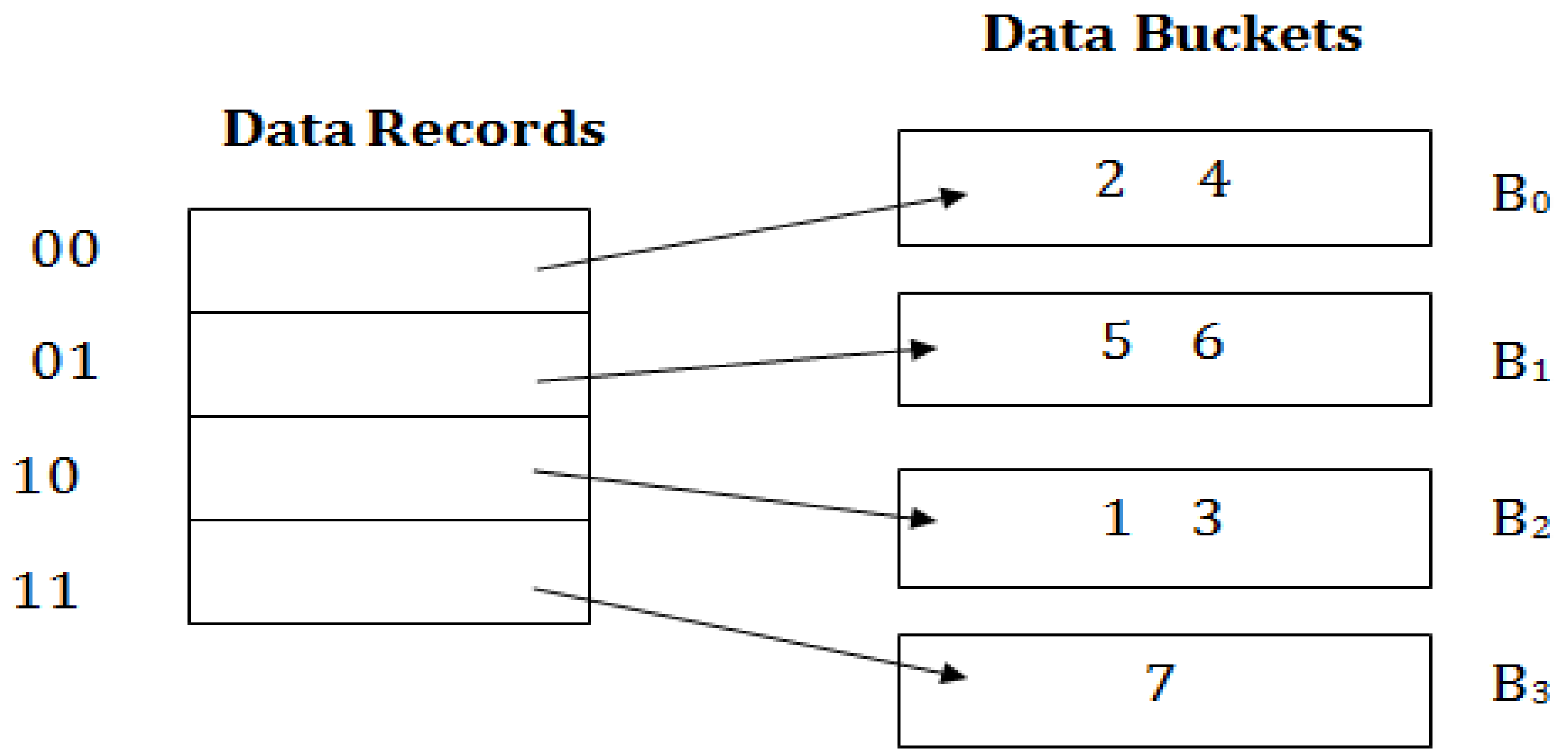
## For example:

Consider the following grouping of keys into buckets, depending on the prefix of their hash address:

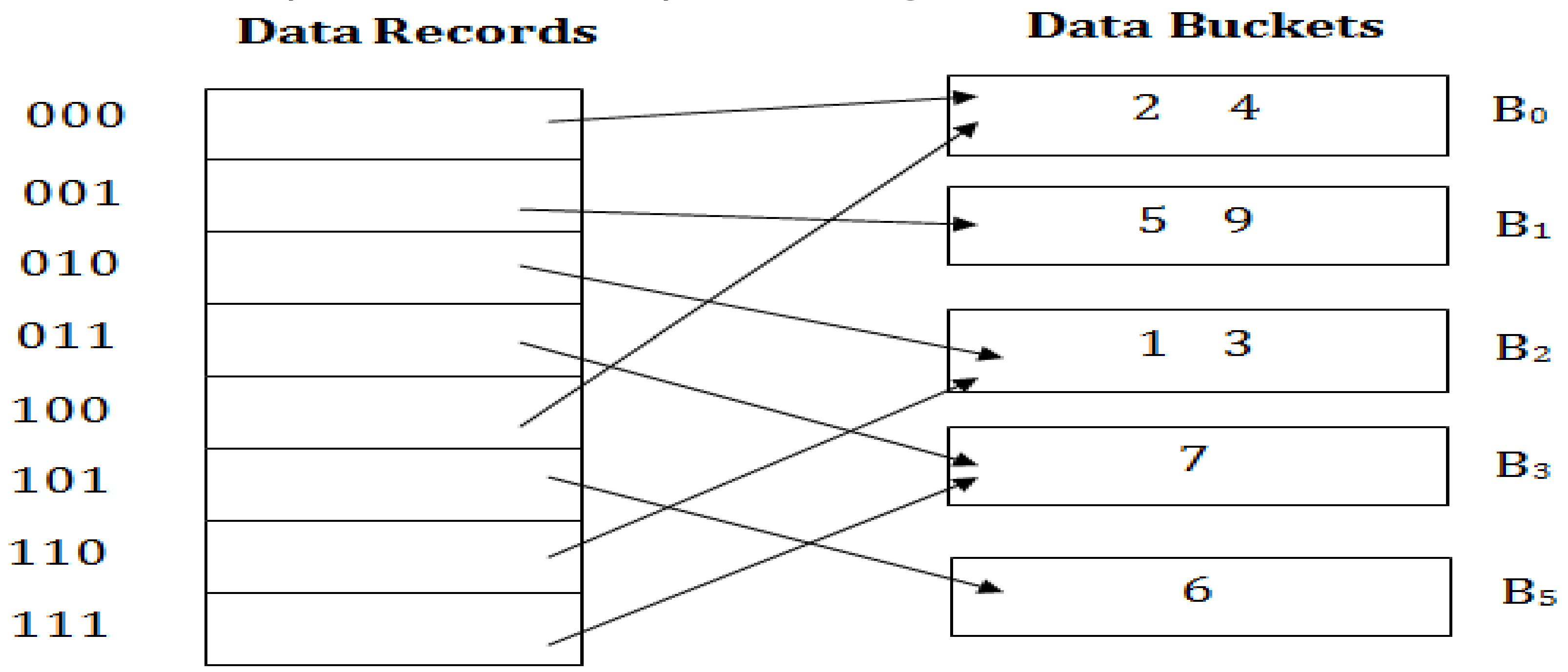


| Key | Hash address |
|---|---|
| 1 | 11010 |
| 2 | 00000 |
| 3 | 11110 |
| 4 | 00000 |
| 5 | 01001 |
| 6 | 10101 |
| 7 | 10111 |

- The last two bits of 2 and 4 are 00. So it will go into bucket B0.
- The last two bits of 5 and 6 are 01, so it will go into bucket B1.
- The last two bits of 1 and 3 are 10, so it will go into bucket B2.
- The last two bits of 7 are 11, so it will go into B3.



- **Insert key 9 with hash address 10001 into the above structure:**
    - Since key 9 has hash address 10001, it must go into the first bucket. But bucket B1 is full, so it will get split.
    - The splitting will separate 5, 9 from 6 since last three bits of 5, 9 are 001, so it will go into bucket B1, and the last three bits of 6 are 101, so it will go into bucket B5.
    - Keys 2 and 4 are still in B0. The record in B0 pointed by the 000 and 100 entry because last two bits of both the entry are 00. similarly remaining entries also.



- **Advantages of dynamic hashing**
  - In this method, the performance does not decrease as the data grows in the system. It simply increases the size of memory to accommodate the data.
  - In this method, memory is well utilized as it grows and shrinks with the data. There will not be any unused memory lying.
  - This method is good for the dynamic database where data grows and shrinks frequently.
- **Disadvantages of dynamic hashing**
  - If there is a huge increase in data, maintaining the bucket address table becomes tedious.
  - In this case, the bucket overflow situation will also occur. But it might take little time to reach this situation than static hashing.

## Tree-based Indexing

- The most commonly used tree-based index structure is the B-Tree, and its variations like B+ Trees and B* Trees.
- In tree-based indexing, data is organized into a tree-like structure.
- Each node represents a range of key values, and leaf nodes contain the actual data or pointers to the data.
- Tree-based indexes like B-Trees offer a number of advantages:
  - **Sorted Data**: They maintain data in sorted order, making it easier to perform range queries.
  - **Balanced Tree:**. This balancing ensures that data retrieval times are consistent.
  - **Multi-level Index:** Tree-based indexes can be multi-level, which helps to minimize the number of disk I/Os required to find an item.
  - **Dynamic Nature:** B-Trees are dynamic, meaning they're good at inserting and deleting records without requiring full reorganization.
  - **Versatility:** They are useful for both exact-match and range queries.

- Tree-based Indexing Example
- Continuing with the "Students" table:

| ID | Name |
|---|---|
| 1 | Abhi |
| 2 | Bharath |
| 3 | Chinni |
| 4 | Devid |

- A simplified B-Tree index could look like this:
-
```
        [1, 3]
       /       \
   [1]           [3, 4]
  /   \         /       \
1      2    3            4
```
- In the tree, navigating from the root to the leaf nodes will lead us to the desired data record.

- **Pros of Tree-based Indexing:**
  - Efficient for range queries.
  - Good for both exact and partial matches.
  - Keeps data sorted.
- **Cons of Tree-based Indexing:**
  - Slower than hash-based indexing for exact queries.
  - More complex to implement and maintain.

## Dynamic Index Structure - B+ Tree

- A dynamic index structure is an index that automatically adjusts its structure as the underlying data grows, shrinks, or otherwise changes.

- **$B^+$ Trees**

- A $B^+$ Tree is a type of self-balancing tree structure commonly used in databases and file systems to maintain sorted data in a way that allows for efficient insertion, deletion, and search operations.

- $B^+$ Trees are an extension of B-Trees but differ mainly in the way they handle leaf nodes, which contain all the key values and point to the actual records.

- **B+ Tree of order \`n\` has the following properties:**

1. Every node has a maximum of \`n\` children.

2. Every node (except the root) has a minimum of \`n/2\` children.

3. The tree is perfectly balanced, meaning that all leaf nodes are at the same level.

4. All keys are stored in the leaf nodes, and the internal nodes act as 'guides' to locate the leaf nodes faster.

- **operations on $B^+$ Trees:**

1. Search: Starts at the root and traverses down the tree, guided by the key values in each node, until it reaches the appropriate leaf node.
2. Insert: Inserts a new key-value pair and then reorganizes the tree as needed to maintain its properties.
3. Delete: Removes a key-value pair and then reorganizes the tree, again to maintain its properties.

- Let's say we have a B$^{+}$ Tree of order 4, and we want to insert the keys `[10, 20, 5, 6, 12, 30, 7, 17]` into an initially empty tree.
- --------------------
- ||    ||     ||     ||
- --------------------
- **Insertion**
- **1. Insert 10:**
  - The tree is empty, so 10 becomes the root.
- 
- `[10]`

- **2. Insert 20:**
  - There's room in the leaf node for 20.
-
- `    [10, 20]`
- **3. Insert 5:**
  - Still room in the leaf node for 5.
-
- `    [5, 10, 20]`
- **4. Insert 6:**
  - The leaf node is full; split it and promote the smallest key in the right node to be the new root.
-
- `         [10]`
- `        /    \`
- `      [5, 6]  [10, 20]`

- **5. Insert 12:**
  - Insert into the appropriate leaf node.
-
- `         [10,  ,  ]`
- `        /    \`
- `     [5, 6]  [10, 12, 20]`
- **6. Insert 30:**
  - Need to split the right leaf node, promote 20.
-
- `         [10 , 20    , ]`
- `        /    |        \`
- `     [5, 6] [10, 12] [20, 30]`

- **7. Insert 7:**
  - Insert into the appropriate leaf node.
-
- `     [10    , 20   ,  ]`
- `    /       |       \`
- `  [5, 6, 7] [10, 12] [20, 30]`
- **8. Insert 17:**
  - Insert into the appropriate leaf node and split.
-
- `        [10 , 20    ,  ]`
- `      /     |        \`
- `  [5, 6, 7] [10, 12] [17, 20, 30]`

- Here, the middle internal node gets split, and 17 is promoted.

```
•
•           [10  , 17     , 20  ]
•          /     |        |     \
•      [5, 6, 7] [10, 12] [17]  [20, 30]
```

- **Search (for 12):**
  - Start at the root, go down the second child because 12 > 10 and 12 < 17, and find 12 in the corresponding leaf node.

- **Deletion (of 10):**
- 1. Remove 10 from the leaf node.
-

```
               [10  , 17 , 20  ]
              /     |    |     \
          [5, 6, 7] [12] [17]  [20, 30]
```

- 2. Since the key 10 is also present in the internal node, we replace it with its in-order predecessor (or successor based on design), which is 7.
-

```
               [7 , 17    , 20  ]
              /   |       |     \
          [5, 6]  [7, 12] [17]  [20, 30]
```

## Comparison of File Organizations, Indexes and Performance Tuning

- **What is database performance tuning?**

- DBMS performance tuning describes a process – on the server side – that will properly configure the DBMS environment to respond to clients’ requests in the fastest way possible, while making optimum use of existing resources.

- Most performance-tuning activities focus on minimizing the number of I/O operations, because the I/O operations are much slower than reading data from the data cache.

- **File Organizations**
- **1. Objective:** To physically store records on storage media in an organized manner.
- **2. Methodologies:** Includes sequential, random (or direct), and hashed file organizations, among others.
- **3. Implications:**
  - Sequential organization is suitable for batch processing but inefficient for random access.
  - Direct or random organization allows fast access but can be inefficient in terms of storage space.
  - Hashed file organization is excellent for equality searches but not for range-based queries.
- **4. Real-world Examples:** Ledger systems, log files, archival systems.

- **Indexing**
- **1. Objective:** To create a data structure that improves the speed of data retrieval operations.
- **2. Methodologies:** Includes clustered, non-clustered, primary, secondary, composite, bitmap, and hash indexes, among others.
- **3. Implications:**
  - Clustered indexes are excellent for range-based queries but slow down insert/update operations.
  - Non-clustered indexes improve data retrieval speed but can take up additional storage.
  - Bitmap indexes are useful for low-cardinality columns.
- **4. Real-world Examples:** Search engines, e-commerce websites, any application that requires fast data retrieval.

- **Performance Tuning**
- **1. Objective:** To optimize the resources used by the database for efficient transaction processing.
- **2. Methodologies:** Query optimization, index tuning, denormalization, database sharding, caching, partitioning, etc.
- **3. Implications:**
  - Query optimization can dramatically reduce the resources needed for query processing.
  - Proper indexing can mitigate the need for full-table scans.
  - Denormalization and caching can improve read operations but may compromise data integrity or consistency.
- **4. Real-world Examples:** Financial trading systems, real-time analytics, high-performance computing.

- **Points of Comparison on File organization, indexing, and performance tuning**

- **1. Granularity:**
  - File organization is about how data is stored at the file level.
  - Indexing is about improving data access at the table or even column level.
  - Performance tuning is a broad set of activities that can encompass both file organization and indexing among many other techniques.

- **2. Resource Usage:**
  - File organization techniques aim to use disk space efficiently.
  - Indexing aims to use both disk space and memory for fast data retrieval.
  - Performance tuning aims to optimize all system resources including CPU, memory, disk, and network bandwidth.

- **3. Query Efficiency:**
  - File organization generally impacts how efficiently data can be read or written to disk.
  - Indexing significantly impacts how efficiently queries can retrieve data.
  - Performance tuning seeks to optimize both read and write operations through a variety of methods.
- **4. Complexity:**
  - File organization is relatively straightforward.
  - Indexing can become complex depending on the types of indexes and the nature of the queries.
  - Performance tuning is usually the most complex as it involves a holistic understanding of hardware, software, data, and queries.