DATA MODELLING - 5

Disk Structure:

A disk drive consists of several physical components and a logical structure to store and retrieve data.

Components:

1. **Platters**: Circular disks coated with magnetic material where data is stored.

2. **Tracks**: Concentric circles on the platter. Each track can store data.

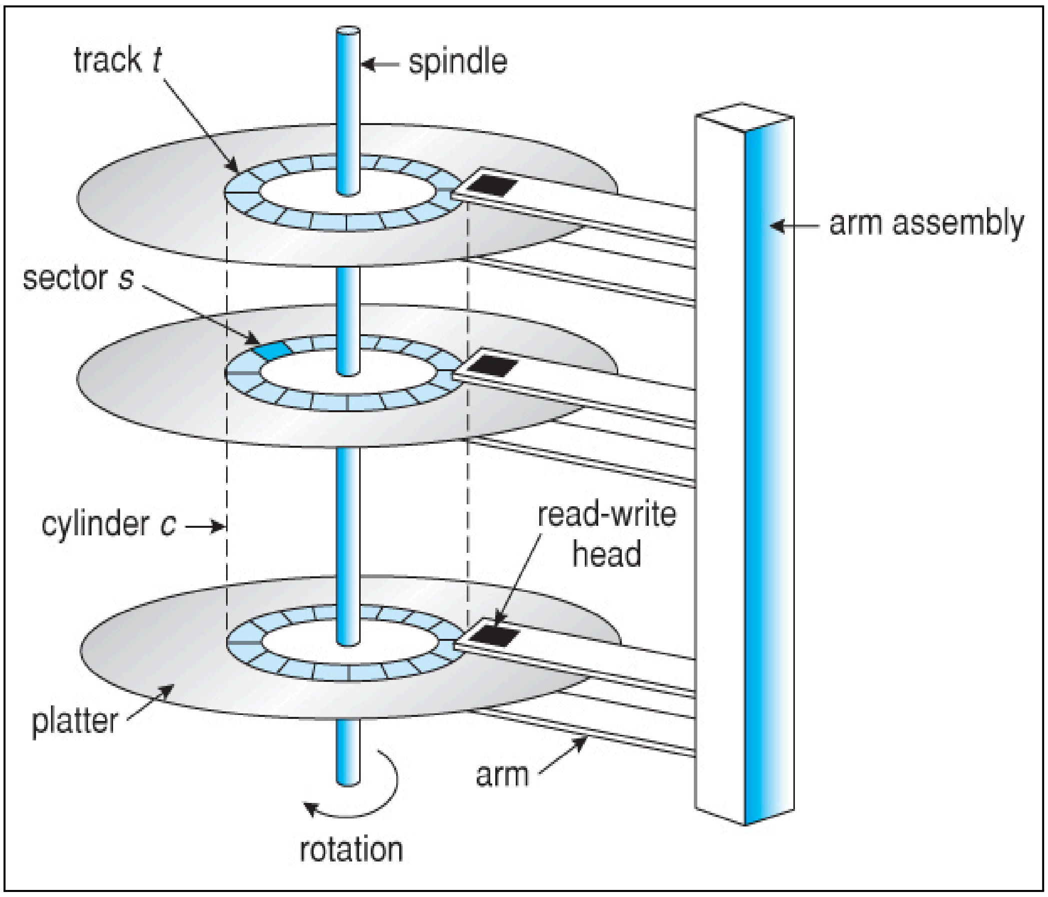
3. **Sectors**: Each track is divided into smaller units called sectors, typically 512 bytes or
4096 bytes in size.

4. **Blokcs:** Smallerst Unit of Storage is called as block, The Block is nothing but the space generated by dividing the disk into Tracks and Sectors, a block can be identified by (Track Number, Sector Number). Generic Size of Block is 512 Bytes.

5. **OffSet:** If Each Block is of 512 bytes, then the block will start from 0 to 511, each of this numbers are known as offsets. That Means a Block Size of 512 bytes will have 512 offsets to identify each and every bytes in the Block. In modern days in most of the HDDs the block size is of 4KB.

6. **Cylinders**: A collection of corresponding tracks across all platters.

7. **Read/Write Head**: A mechanical arm with a head that reads from and writes data to the platters.



How Data is Accessed:

1. **Seek Time**: Time taken for the head to move to the correct track.

2. **Rotational Latency**: Time for the platter to rotate and bring the correct sector under the head.

3. **Transfer Time**: Time taken to transfer data once the head is positioned correctly.

NOTE: Suppose Any Program is accessing some data, then data needs to be brough into the Main Memory otherwise data can't be modified, as the Disks are slow, we can't directly do operations on them, it will slow down the whole process, thats why the required data is brought into the RAM first then only its gets updates, once updated it will be written back into the disk. Organising the Data into the RAM is done via DATA STRUCTURE, Organising the data into the DISK is done via DBMS.

Data Storage and Block Access:

1. Table Structure and Record Size:

eid (10)	ename (10)	department (10)	section (5)	address (50)
1	Akash	CSE	A	Bangalore
. .	. .	. .	. .	. .
. .	. .	. .	. .	. .
. .	. .	. .	. .	. .
. .	. .	. .	. .	. .

. .	. .	. .	. .	. .
100	Ankit	IT	A	IND

- Tuple/Row size of the table = 128 bytes
- Block Size = 512 bytes
- Records per Block = Block Size / Row Size = 512 / 128 = 4 records

2. Number of Records to Store

To store 100 rows:

- 4 records = 1 block
- 1 record = ¼ block
- 100 records = 100 / 4 = 25 blocks

Thus, to store 100 rows, 25 blocks are required.

3. Block Access

To perform any operation on this table, at most 25 blocks need to be accessed.

Example:

To find `eid: 1`, you would need to traverse 25 blocks. To reduce this time, an index can be used.

Dense Indexing and Block Access with Indexes

1. Index Structure

eid (10)	Record Pointer (6)

1	address pointer
.	.
100	address pointer

- Index Row Size = 16 bytes

2. Index Block Size Calculation

- Block Size in Disk = 512 bytes
- A block can store: 512 / 16 = 32 index records

3. Number of Index Blocks to Store

To store 100 records:

- 32 index records are stored in 1 block
- 100 / 32 = 3.125 blocks ≈ 4 blocks

4. Block Access with Indexing

With indexing, to access the data, you need to:

- Access 4 blocks for the index
- From the index, get the target block, resulting in a total of 4 + 1 = 5 blocks accessed

Previously, 25 blocks were accessed, but now with indexing, the number is reduced to 5 blocks.

Multilevel Indexing

Example Scenario:

- Suppose the table size grows to 1000 records, then the Dense Index size will also increase.
- The table will require 250 blocks to store all the table data.

Index Size Calculation:

- The index will require: 100032=31.25≈32 blocks\frac{1000}{32} = 31.25 \approx 32 \text{ blocks}321000=31.25≈32 blocks
- This is because 32 records can be stored per block.

Introducing Multilevel Indexing:

- Now, as the search on the index becomes larger, we can introduce another index on top of this index to query faster.

Key Insight:

- We already know:
    - **32 index records** can be stored in **1 block**.

    - **1000 index records** can be stored in: 1000/32 = 31.2 ≈ 32 blocks
    - Thus, each block will have one entry. That means all the data can be stored in just **1 block**.

Data Retrieval:

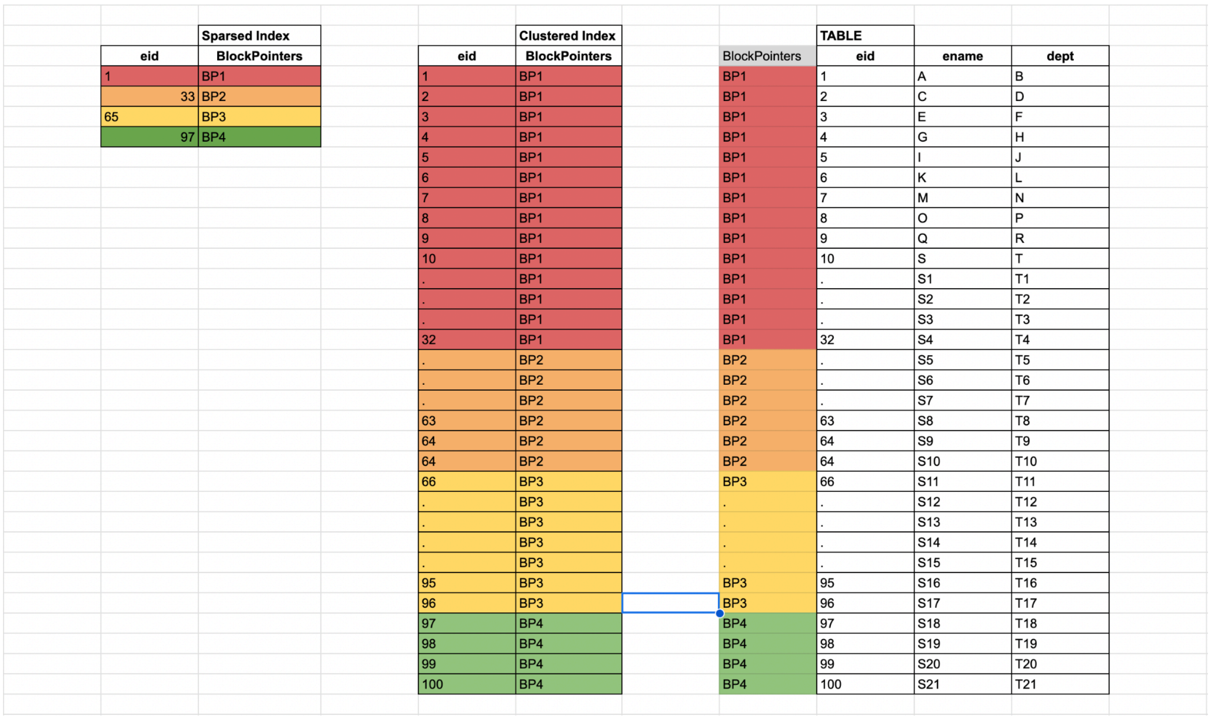
- Now, to read any data from the table, only **3 block reads** will be required:
    - This reduces the number of block accesses compared to a non-indexed search.

Clustered Index:

- A clustered index keeps the data records in sorted order based on the index.
- In the diagram:
    - The `cid` (Clustered Index) is associated with record pointers.
    - The blocks are numbered, and the records are sorted with the pointers pointing to their respective addresses.

Sparse Index:

- A sparse index is created by indexing only a subset of the records.
- In the provided example:
    - e-id and Pointer values are given for certain blocks.



	Sparsed Index
eid	BlockPointers
1	BP1
33	BP2
65	BP3
97	BP4

	Clustered Index
eid	BlockPointers
1	BP1
2	BP1
3	BP1
4	BP1
5	BP1
6	BP1
7	BP1
8	BP1
9	BP1
10	BP1
.	BP1
.	BP1
.	BP1
32	BP1
.	BP2
.	BP2
.	BP2
63	BP2
64	BP2
64	BP2
66	BP3
.	BP3
.	BP3
.	BP3
.	BP3
95	BP3
96	BP3
97	BP4
98	BP4
99	BP4
100	BP4

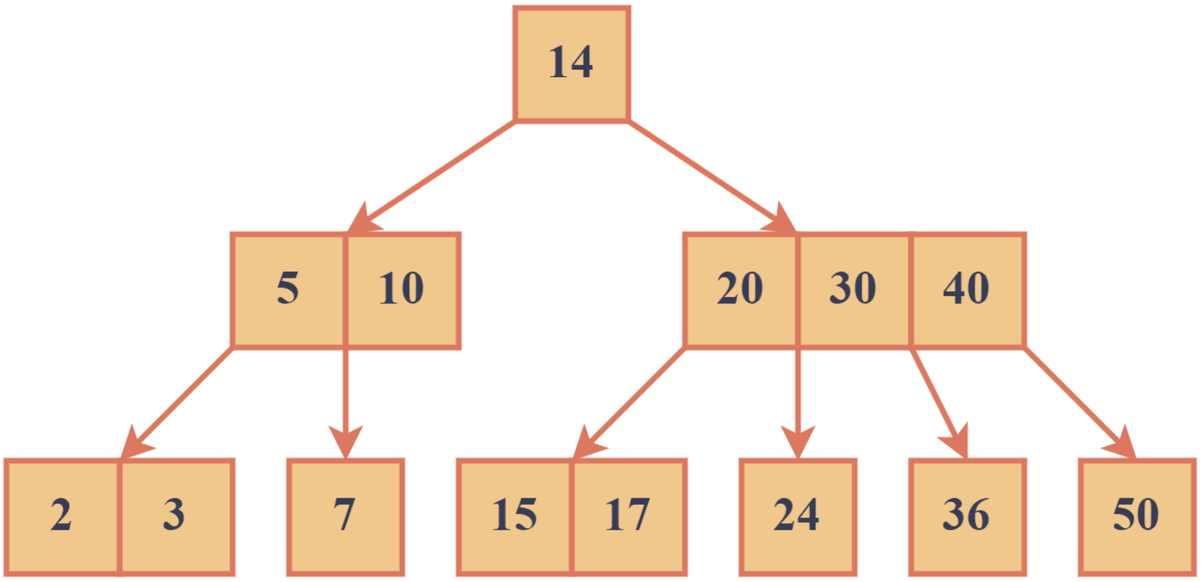
	TABLE		
BlockPointers	eid	ename	dept
BP1	1	A	B
BP1	2	C	D
BP1	3	E	F
BP1	4	G	H
BP1	5	I	J
BP1	6	K	L
BP1	7	M	N
BP1	8	O	P
BP1	9	Q	R
BP1	10	S	T
BP1	.	S1	T1
BP1	.	S2	T2
BP1	.	S3	T3
BP1	32	S4	T4
BP2	.	S5	T5
BP2	.	S6	T6
BP2	.	S7	T7
BP2	63	S8	T8
BP2	64	S9	T9
BP2	64	S10	T10
BP3	66	S11	T11
.	.	S12	T12
.	.	S13	T13
.	.	S14	T14
.	.	S15	T15
BP3	95	S16	T16
BP3	96	S17	T17
BP4	97	S18	T18
BP4	98	S19	T19
BP4	99	S20	T20
BP4	100	S21	T21

Problem:

The more multilevel indexes we have, the more the data search complexity will be reduced. We don't want to create these indexes manually; this should be an automated process.

M-way Search Tree:

- In an M-way search tree, each node can have at most M−1M-1M−1 keys and MMM child nodes.
- A key property is that the keys in each node are arranged in ascending order, and the child nodes represent the range between keys.
- For example, in a 3-way search tree (where M=3).
  - The parent node can have up to 2 (M-1) keys and 3 (M) child nodes.
  - Keys in the child nodes must be within the range defined by the parent keys.



M-Way Search Trees, while efficient for certain applications, have some limitations that necessitate the introduction of B-Trees, particularly in the context of database systems and large-scale storage systems. Here are the key issues with M-Way Search Trees that B-Trees address:

1. Unbalanced Tree Structure

- **M-Way Search Trees** do not enforce any balancing mechanism. This means that during insertions and deletions, the tree can become unbalanced, leading to performance degradation.

- An unbalanced tree can result in very deep paths for certain nodes, making searches, insertions, and deletions inefficient (potentially linear in time complexity).
- **B-Trees** solve this by ensuring that the tree remains balanced after every insertion and deletion. All leaf nodes in a B-Tree are at the same level, providing a consistent depth for operations.

2. Overflow of Nodes

- In an M-Way Search Tree, there can be a situation where a node gets too many children, leading to overflow conditions.
- Handling node overflow in an M-Way tree can be complicated, often requiring restructuring of large portions of the tree.
- **B-Trees** address this by splitting nodes that overflow into two nodes, redistributing the keys and maintaining balance. This split operation ensures that no node exceeds its maximum allowed number of keys and children.

3. Underflow of Nodes

- M-Way trees can also suffer from **underflow** (i.e., when a node has fewer children than the required minimum). Managing underflows often involves complex node merging and key redistribution operations.
- **B-Trees** handle underflows more systematically, merging nodes or redistributing keys across adjacent nodes when necessary, ensuring the tree remains balanced.

4. Complexity in Search Operations

- In M-Way Search Trees, since the tree can become unbalanced, the depth of the tree can grow unevenly, resulting in varying search times depending on where a particular key is located.
- In **B-Trees**, because the tree is balanced and the nodes contain multiple keys, the height of the tree is minimized, and search operations are efficient (logarithmic in time complexity). A B-Tree can store more keys per node, reducing the number of disk accesses or comparisons needed.

5. Inconsistent Node Sizes

- M-Way Search Trees do not have a strict rule on the number of children or keys per node, which can lead to inefficiencies, especially when dealing with disk-based storage (as disk pages are generally of fixed size).
- **B-Trees** are designed to be optimized for disk-based storage. Each node in a B-Tree is designed to fit perfectly into a disk page, reducing the number of disk reads and writes. This makes B-Trees particularly useful for database indexing.

6. Inefficient Disk Access

- In M-Way Search Trees, as the number of levels increases (due to lack of balancing), more disk accesses are required to search, insert, or delete data. This is inefficient for large databases or storage systems that rely on disk-based access.
- **B-Trees** are optimized for disk access, as they ensure that nodes are stored in contiguous memory locations (or disk blocks), which minimizes the number of disk accesses needed for any operation.

7. Poor Support for Range Queries

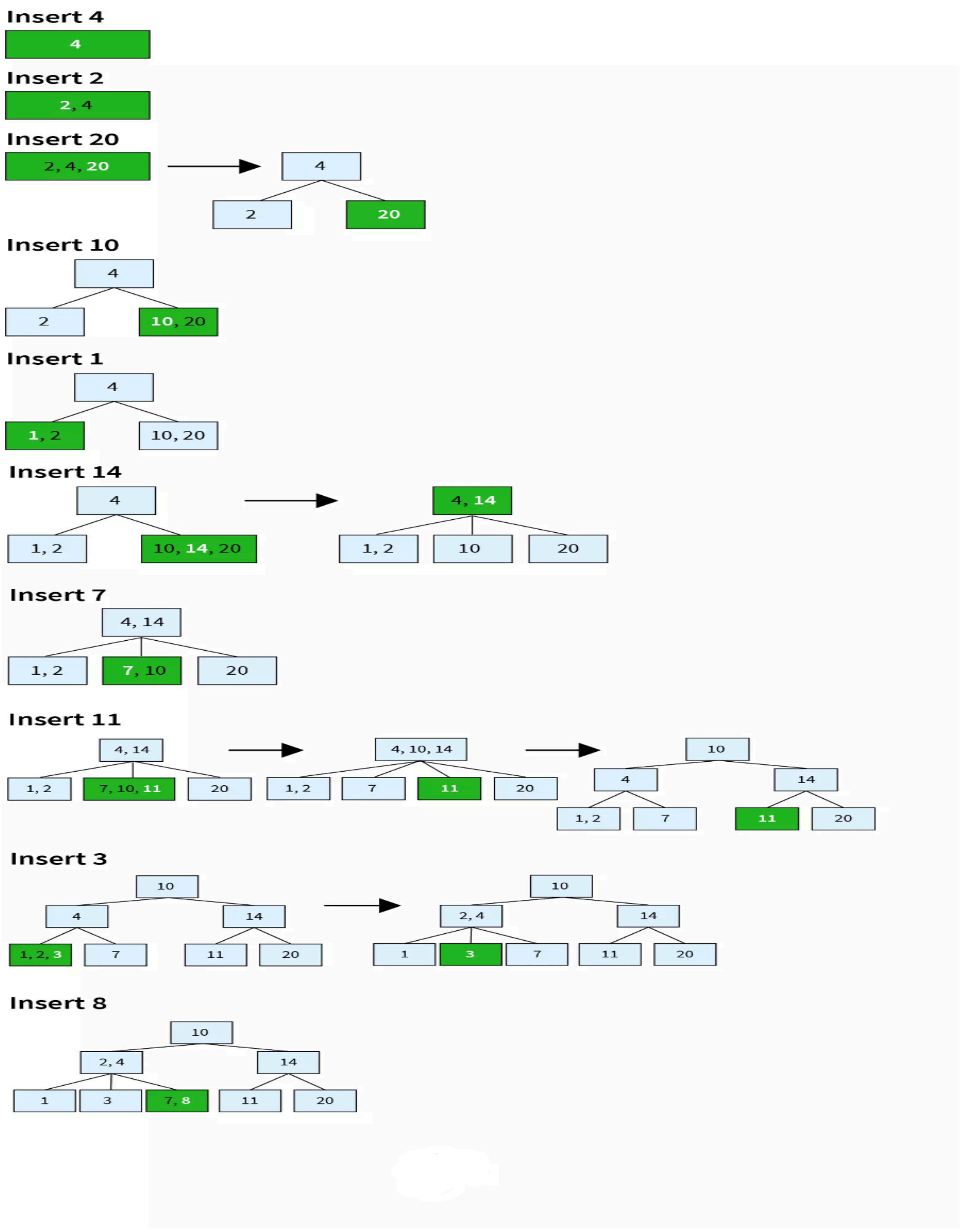
- M-Way trees are not designed to handle efficient range queries, which require finding all keys between two values.
- **B+ Trees** (a variant of B-Trees) address this problem by linking all the leaf nodes together in a linked list, making range queries efficient. Traversing between nodes for a range query in a B+ Tree is much faster and more structured.

8. Dynamic Growth and Shrinking

- M-Way trees can become inefficient as the tree grows and shrinks due to insertions and deletions, leading to uneven distribution of keys and varying levels of depth.
- **B-Trees** grow and shrink dynamically while maintaining balance, making them more robust for dynamic datasets. The tree gracefully expands or contracts without losing balance, ensuring consistently good performance.

B-Trees

- **M-way search trees** with some rules:
    1. ⌊M/2⌋ children should be present in the node.
    2. The root can have a minimum of 2 children.
    3. All leaf nodes should be at the same level.
    4. The creation process is **Bottom-up**.



While **B-Trees** offer significant advantages for efficient storage and retrieval in large datasets, especially in databases and filesystems, there are some limitations that led to the development of **B+ Trees**. The main issues with B-Trees that are addressed by B+ Trees are as follows:

1. Redundancy in Internal Nodes

- In **B-Trees**, both internal and leaf nodes store data keys. This means that internal nodes contain actual data pointers, which can be redundant since internal nodes are primarily used for navigation purposes, not for storing data.
- In **B+ Trees**, only the leaf nodes store actual data pointers, while internal nodes store only keys for routing. This separation reduces redundancy and makes the structure more space-efficient. It also simplifies the structure, as internal nodes do not hold data pointers.

2. Inefficient Sequential Access

- B-Trees are not optimized for **sequential access** (i.e., range queries or scans). If you need to access a sequence of records that fall within a range of keys, a B-Tree requires traversing the tree for each key individually, which can be inefficient.
- **B+ Trees** are designed to optimize sequential access. In B+ Trees, the leaf nodes are linked together in a **linked list**, allowing efficient traversal of all leaf nodes in **sorted order**. This makes B+ Trees highly efficient for range queries, as you can move from one leaf to the next without traversing the internal nodes again.

3. Larger Tree Height

- In B-Trees, since both internal and leaf nodes store data, the number of keys that can be stored in a node is limited. This can result in a taller tree, meaning more disk accesses are needed to search for a key or perform an update.
- **B+ Trees** pack more keys into each internal node because they only store keys for navigation (and not the actual data). As a result, the height of a B+ Tree is typically smaller than that of a B-Tree for

the same amount of data, reducing the number of disk accesses and making B+ Trees faster for search operations.

4. Complexity in Deletions

- Deleting a key in a B-Tree can be more complex because the key may exist in both internal and leaf nodes. When a key is deleted, it might have to be removed from both levels, which complicates the deletion process.
- In **B+ Trees**, deletions are simpler because keys are only stored in the leaf nodes. Internal nodes are used only for routing, and there is no need to update internal nodes when data is deleted from the leaf level, streamlining the deletion process.

5. Non-Uniform Access Times for Data

- In a B-Tree, since data can be stored at both internal and leaf nodes, access times can vary. If a key exists in an internal node, it can be accessed quickly, but if it's in a leaf node, more traversal may be needed.
- **B+ Trees** provide uniform access times because all data is stored in the leaf nodes. Since the structure guarantees that you only retrieve data from the leaf nodes, the access pattern is predictable and uniform across all queries.

6. Less Efficient Disk Utilization

- B-Trees may result in inefficient disk utilization, especially when there are fewer keys per node. Since data is stored in both internal and leaf nodes, more disk space may be wasted, especially if internal nodes are under-utilized.
- **B+ Trees** are more efficient in disk utilization. Internal nodes can hold more routing keys (as they don't store data), and leaf nodes can be packed with actual data pointers. This better disk usage translates to fewer I/O operations and better overall performance, particularly in disk-based storage systems.

7. Slower Range Queries

- Performing range queries in B-Trees is slower because B-Trees are not optimized for continuous scanning across a range of values. After finding the first key in the range, the next key may require additional traversals of the tree, resulting in multiple disk accesses.
- **B+ Trees** are ideal for range queries. Since leaf nodes are linked in sequential order, once the first key in the range is found, the subsequent keys can be accessed simply by following the linked list of leaf nodes, without retraversing the tree. This makes B+ Trees faster and more efficient for range queries, which are common in database applications.

8. Difficulty in Full Table Scans

- Full table scans (e.g., when reading all records in a dataset) are less efficient in B-Trees because the tree structure requires traversing both internal and leaf nodes to visit all records.
- In **B+ Trees**, full table scans are much faster. Since all the records are stored at the leaf level and the leaf nodes are connected via a linked list, it is possible to traverse all records in order without having to visit internal nodes, which simplifies and speeds up full scans.

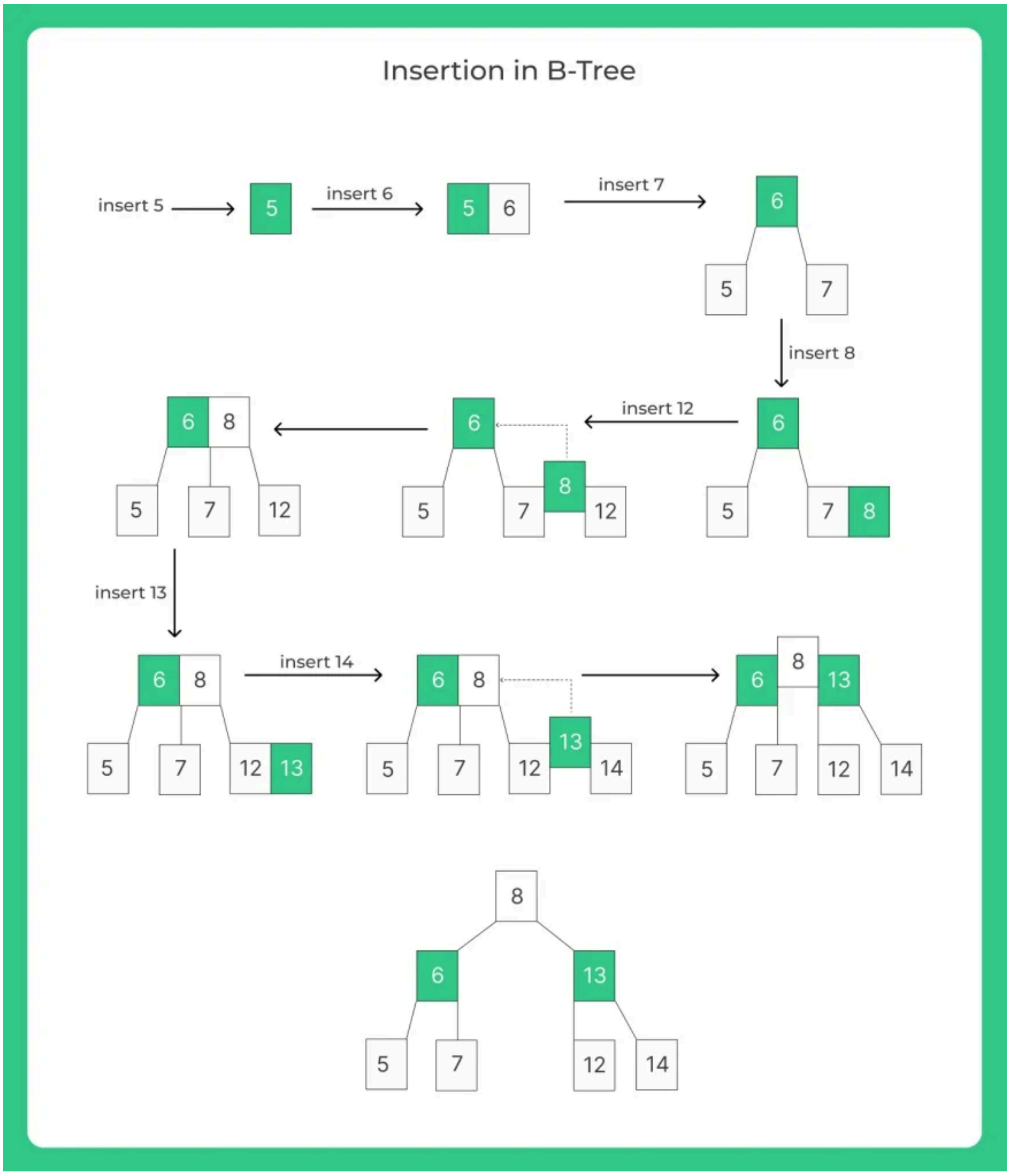
9. Higher Maintenance Overhead

- In a B-Tree, maintaining the balance of the tree is more challenging since both internal and leaf nodes store data. Insertions and deletions may lead to rebalancing at multiple levels, causing more complexity in keeping the tree balanced.
- **B+ Trees** have less overhead for maintaining balance because internal nodes only serve as navigational guides. Insertions and deletions mainly affect the leaf nodes, simplifying tree maintenance and rebalancing operations.

B+ Trees:

In addition to B Tree Properties. B+ trees introduces some more properties to make it more efficient.

- In B+ Trees, **only leaf nodes will have record pointers**.
- Each leaf node will have the **parent nodes**.
- All leaf nodes will be **connected to each other** like a linked list and will be in **sorted order**.



MySql Data Storage:

1. InnoDB's Internal Storage Structures:

- **Tablespaces:**
  - A tablespace is a logical storage unit in InnoDB, which can be a single file (`.ibd` file per table) or shared by multiple tables (`ibdata1`).
  - It organizes data into **pages** and **extents**.

- **Pages and Extents:**
  - **Pages**: The basic unit of storage in InnoDB, typically 16 KB in size. Pages store actual table data, index entries, or other information.
  - **Extents**: A collection of 64 consecutive pages, totaling 1 MB, used for efficient space management and allocation.

- **Logical Offset in Tablespace:**
  - An offset in InnoDB is essentially a byte position within the tablespace file. For example, if a page starts at byte 16,384, this would correspond to an offset of 16 KB from the beginning of the file.
  - InnoDB uses these offsets to determine where each page (or other data structure) resides within its tablespace.

- **Information Held by Offsets:**
  - The offset tells InnoDB the exact location of a specific page within the tablespace. This is crucial for reading and writing data because each page contains a specific piece of information, like rows or index entries.

2. Interaction with the OS File System:

a. File System's Role:

- The OS and its file system (e.g., ext4, NTFS) deal with files in terms of **file blocks** and **byte offsets** within these files.
- When InnoDB wants to read or write to a page, it calculates the byte offset within the tablespace file and passes this request to the file system.

b. How File System Understands Offsets:

- **InnoDB's Offset Calculation:**
    - InnoDB calculates the offset within its own logical structure (e.g., "I need to write to page 10, which is at byte 163,840").
    - It uses this offset to determine the byte position within the `.ibd` file (or shared `ibdata1` file).

- **File System's Interpretation:**
    - InnoDB's calculated offset is passed to the file system as part of a standard read or write operation.
    - The file system interprets this offset in terms of the specific file (e.g., `my_table.ibd`).
    - For example, a write operation might say: "Write 16 KB of data starting at byte position 163,840 in `my_table.ibd`."
    - The file system, using its internal structures (like inodes and block maps), translates this byte offset to physical disk blocks.

- **Physical Disk Mapping:**
    - The file system knows where each block of the file is physically located on the disk (in terms of sectors and tracks). It uses this information to perform the actual read or write operation.

NOTE:

- InnoDB’s offsets are logical positions within its own data files. The OS file system doesn’t understand the structure of these files; it only sees them as a series of bytes.
- The file system is unaware of concepts like pages or extents. It simply handles data based on byte offsets and file blocks.

3. Workflow of a Write Operation:

1. **InnoDB’s Request:**
   - InnoDB decides to write a new row to page 10 of a table.
   - It calculates the offset of this page within the tablespace file (e.g., `page 10 * 16 KB = 160 KB`).

2. **System Call to File System:**
   - InnoDB issues a system call like `write()` to the OS, specifying the file (`my_table.ibd`), the offset (160 KB), and the data to be written.

3. **File System Handling:**
   - The OS receives the write request and interprets it based on the file’s internal structure.
   - It looks up its internal mapping to see which physical disk blocks correspond to byte range `160 KB to 176 KB` of `my_table.ibd`.

4. **Physical Disk Write:**
   - The file system instructs the disk controller to write the data to the appropriate physical blocks.

4. Key Points to Understand:

- **InnoDB’s Offsets are Logical:** These offsets only make sense within the context of InnoDB and its tablespace structure. They help InnoDB manage data efficiently without needing to know about the actual physical layout on the disk.

- **File System's View is Different:** The OS file system doesn't understand InnoDB's internal organization. It only sees files as a series of bytes and manages them based on its own block allocation and mapping mechanisms.

- **Translation Between Layers:**
    - InnoDB translates its internal page-based structure into byte offsets for read and write operations.
    - The file system translates these byte offsets into physical disk locations.

SQL QUERIES:

```
– Creating the Database

CREATE DATABASE cohort;

– Drop Tables if exists

DROP TABLE IF EXISTS Product;


– Creating Product Table

CREATE TABLE Product (

    product_id INT,

    user_id INT,

    product_name VARCHAR(255),

    price DECIMAL(10, 2)

);
```

– Creating Procedure to insert data in bulk in table

```
CREATE PROCEDURE InsertProducts()

BEGIN

    DECLARE i INT DEFAULT 1;

    WHILE i <= 100000 DO

        INSERT INTO Product (product_id, user_id,
   product_name, price)

        VALUES (i, FLOOR(RAND() * 1000) + 1,
   CONCAT('Product-', i), ROUND(RAND() * 1000, 2));

        SET i = i + 1;

    END WHILE;

END
```

– Call procedure to insert data

```
CALL InsertProducts();
```

– see the table size

```
SELECT table_name AS 'Table', ROUND(SUM(data_length +
   index_length) / 1024 / 1024, 2) AS 'Size (MB)'

FROM information_schema.TABLES

WHERE table_schema = 'cohort'

  AND table_name = 'Product';
```

– select all the records

```
SELECT * FROM Product;
```

– Describe Product

```
DESCRIBE Product;
```

– Query on non indexed column

```
SELECT * FROM Product WHERE product_id = 50000;
```

– create index on product_id

```
CREATE INDEX idx_product_id ON Product(product_id);
```

– Query on indexed column

```
SELECT * FROM Product WHERE product_id = 50000;
```

– Show index size

```
SELECT database_name, table_name, index_name,

       ROUND(stat_value * @@innodb_page_size / 1024 / 1024,
  2) AS 'Size (MB)'

FROM mysql.innodb_index_stats

WHERE stat_name = 'size'

  AND database_name = 'cohort'

  AND table_name = 'Product'

  AND index_name = 'idx_product_id';
```

– Drop index

```
DROP INDEX idx_product_id ON Product;
```

– Query on multiple non indexed columns

```
SELECT * FROM Product WHERE user_id = 799 AND product_id =
  50000;
```

– Create index on multiple columns

```
CREATE INDEX idx_user_product ON Product(user_id,
  product_id);
```

– Query on composite index columns

```
SELECT * FROM Product WHERE user_id = 799 AND product_id =
  50000;
```

– See the size of the composite index

```
SELECT database_name, table_name, index_name,

       ROUND(stat_value * @@innodb_page_size / 1024 / 1024,
  2) AS 'Size (MB)'

FROM mysql.innodb_index_stats

WHERE stat_name = 'size'

  AND database_name = 'cohort'

  AND table_name = 'Product'

  AND index_name = 'idx_user_product';
```

– Drop index idx_user_product

```
DROP INDEX idx_user_product ON Product;
```

– Use Explain Query to find out how your query is performing

```
EXPLAIN SELECT * FROM Product WHERE user_id = 799 AND
  product_id = 50000;
```

– count records based on user_id

```
SELECT COUNT(*) FROM Product WHERE user_id = 799;
```