Indexing

Contents

[1. Introduction 3](#_Toc186833337)

[Database Concepts: 3](#_Toc186833338)

[How Computers Work with Arrays and Binary Search 5](#_Toc186833339)

[2. How Indexes Work In Databases 7](#_Toc186833340)

[Fundamentals Of Indexing 9](#_Toc186833341)

[3. Index Creation And Maintenance 14](#_Toc186833342)

[1. Initial Creation of Indexes 14](#_Toc186833343)

[2. Maintenance and Updates of Indexes 15](#_Toc186833344)

[Indexes - Practice 17](#_Toc186833345)

[Creating Custom Indexes 18](#_Toc186833346)

# Introduction

## Database Concepts:

1. **Purpose of Databases**:
   * Store and retrieve data persistently.
   * Data is stored on persistent storage like HDD or SSD, not in RAM.
2. **Data Retrieval Process**:
   * Queries access data stored on disk.
   * Disk data is loaded into RAM because CPU cannot process data directly from disk.
   * CPU processes data in RAM for computations and checks.
3. **Speed Disparity**:
   * CPU is significantly faster than disk storage.
   * Direct interaction between CPU and disk would cause a bottleneck.
4. **Step-by-Step Query Execution**:
   * A query (e.g., SELECT \* FROM STUDENTS WHERE PSP = 80) causes the database to fetch data from disk.
   * Data is loaded into RAM in manageable chunks (due to limited RAM size).
   * CPU processes each chunk, applying conditions, and results are returned.
5. **Memory Hierarchy**:
   * Disk → RAM → CPU workflow ensures efficient data handling.
   * Entire databases can't fit into RAM due to size limitations.
6. **Hardware Dynamics**:
   * Disk data is stored in concentric circles or sections.
   * Hardware reads sections sequentially, which is inherently slower than RAM access.
7. **Optimizations Needed**:
   * Direct processing of all rows for every query is slow, especially for large datasets.
   * Fetching data section by section into RAM helps manage memory constraints but still isn't optimal.

### **How data is stored and retrieved in a database**

* Think of a **disk** as a large array, say Disk[], where each element in the array represents a row in a database table.
* If you write a query like SELECT \* FROM STUDENTS WHERE PSP = 80, you need to scan through each element of Disk[] to find rows where the condition PSP = 80 is true.
* However, the **CPU** cannot directly work with the Disk[] array because it’s too slow. Instead, data is **copied into a smaller array in memory (RAM)** for processing.

### **Why data is fetched in chunks**

* The size of the Disk[] array could be very large (e.g., 1 TB), but the size of the **RAM array** (say, RAM[]) is much smaller (e.g., 16 GB). Hence, you cannot copy the entire Disk[] into RAM[] at once.
* Instead, the database divides the Disk[] into smaller **sections** or **blocks**, and these blocks are fetched into RAM[] one by one.
* Example: If Disk[] has 1,000,000 rows, and RAM[] can hold 100,000 rows, you will process rows in **batches of 100,000**.

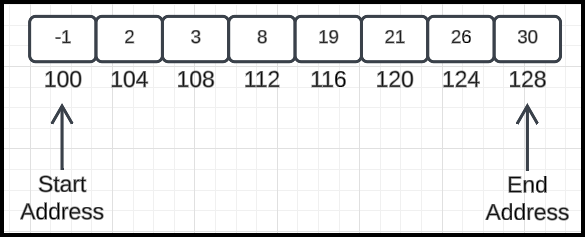
### **Sequential Scanning Without Indexes**

* Without an index, the query SELECT \* FROM STUDENTS WHERE PSP = 80 requires scanning the entire Disk[] array. This is called a sequential scan:
  + Start at Disk[0], check the PSP value.
  + If it matches 80, copy the row to a Results[] array.
  + If it doesn’t match, move to the next element.
  + Repeat this until the end of Disk[].
* This process is slow because every element must be checked, even if only a few rows match the condition.

## **How Computers Work with Arrays and Binary Search**

### **1. Sequential Access in Arrays**

* Computers store arrays in **contiguous memory locations**.
* The **address of the first element** is stored (e.g., 100).
  + For integers, subsequent elements are spaced by 4 bytes:
  + Address of the first element = 100
  + Second element = 104
  + Third element = 108
  + Fourth element = 112
  + And so on...



### **2. Calculating Array Boundaries**

* If the array has **8 elements** and each element takes **4 bytes**, then:
  + Start address = 100
  + End address = 100 + (8 - 1) \* 4 = 128
* This allows the computer to determine the **total memory range** of the array (100 to 128).

### **3. Accessing Elements Mathematically**

* To find a specific element, the computer applies mathematics.
  + Example: To find the middle element in the array:
    - Middle index = (start\_index + end\_index) / 2
    - Address of the middle element = start\_address + (middle\_index \* element\_size)
    - For an array starting at 100 with 8 elements:
      * Middle element = at address 116.

### **4. Sequential Scanning vs. Binary Search**

#### **Sequential Scanning**

* Without applying any optimization, finding an element requires scanning **each element one by one**.
* This involves **more disk accesses**:
  + Example: To find the number 3 in the array, you may need **4 disk accesses**:
    1. Fetch data from Disk[0], check if it matches.
    2. Fetch data from Disk[1], check again.
    3. Repeat for the next addresses until the number is found.

#### **Binary Search**

* With **binary search**, the computer uses the sorted structure of the array to minimize the number of checks:
  + Start with the **middle element**.
  + If the target is smaller, search the **left half**.
  + If larger, search the **right half**.
  + Repeat until the target is found.

#### **Number of Disk Accesses in Binary Search**

* Binary search requires **log(n) disk accesses**, where n is the number of elements.
* Example: To find the number 3 in the array:
  + First disk access: Fetch the middle element (116), check if it matches.
  + Second disk access: Fetch the left or right half of the array, depending on the comparison result.

### **5. Role of Disk Accesses and Indexing**

* **Disk access** is a slow operation compared to accessing data in RAM.
* Transferring large amounts of data from disk to RAM for computation is inefficient.
* **Indexes** aim to reduce the number of disk accesses, making operations faster.
  + Example:
    - Without indexing: **4 disk accesses** to find an element.
    - With indexing and binary search: **2 disk accesses**.

### **Key Insight:**

Indexes improve efficiency by reducing the **number of disk accesses**, which directly speeds up data retrieval.

# How Indexes Work In Databases

**Disk Access is Slow**

* Databases store most of their data on disk by default. Disk access is inherently slower than accessing data from RAM. To improve query performance, the number of disk accesses must be minimized.

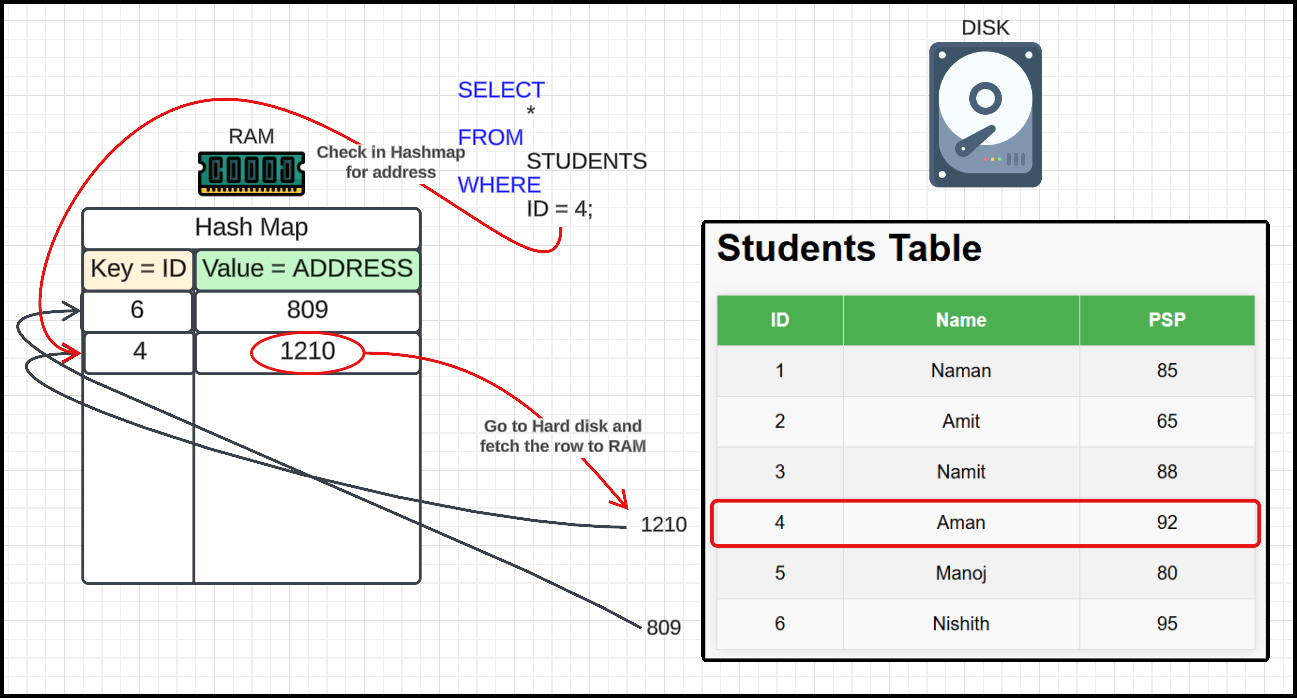
**Primary Keys and Sorted Storage**

* By default, databases often store data sorted by the **primary key**. This allows queries like SELECT \* FROM STUDENTS WHERE ID = X to be efficient even without additional optimizations. However, this alone isn't always enough.

**Hash Maps for Faster Access**

* A **hash map** allows for fast lookup (average-case O(1) time complexity) by mapping keys (e.g., student IDs) to values (e.g., addresses or data).
* However, hash maps stored **on disk** do not provide the same speed benefits because disk access remains the bottleneck.
* Hence, a hash map or similar structure must reside **in RAM** for effective use.

**Indexes as a Solution**

* **Indexes** are auxiliary data structures stored in memory to speed up data retrieval.
* They map keys (e.g., student IDs) to pointers or addresses where the actual data resides on disk.
* With an index, instead of scanning the entire table, the database can directly retrieve the address from the index and fetch the data in a **single disk access**.

**Trade-offs**

* While indexes significantly reduce query time, they come at the cost of increased storage and memory usage.
* Maintaining indexes also increases the time complexity of **write operations** (inserts, updates, and deletes) since the index must be updated alongside the actual data.

### **How This Works in Practice**

* **Scenario:**  
  A query like SELECT \* FROM STUDENTS WHERE ID = X needs to fetch data for a specific student.
* **Without Index:**
  + The database might perform a **linear search** through the table, leading to O(n) disk accesses.
* **With Index:**
  + The database uses an **in-memory index** to determine the disk address of the required student in O(1) or O(log n) time. Then it performs a **single disk access** to fetch the data.

## Fundamentals Of Indexing

1. Hash Length and Table Size:

The hash length corresponds to the number of distinct keys or rows. Handling large data for operations like DISTINCT requires all data to be loaded into RAM, posing potential memory limitations.

1. Limitations of Indexing:

Indexes do not eliminate the need for scanning rows entirely in certain queries, such as SUM. They merely optimize access by reducing the number of disk reads.

1. Trade-offs in Computer Science:

Everything in computer systems, including indexing, involves trade-offs. While indexes speed up certain queries, they introduce storage overhead and might slow down write operations.

1. Indexes and Data Organization:

Indexes store mappings (e.g., key to addresses) rather than every row individually. They optimize by grouping rows or storing pointers to disk locations, minimizing the need for multiple disk accesses.

1. Query Optimization via Hash Maps:

A hash map in RAM can link a key (e.g., PSP) to disk addresses where data resides. This avoids scanning all rows but does not guarantee that each section contains only relevant data, as disk sections may include unrelated rows.

1. Disk Access Patterns:

Disks fetch data in sections, not byte by byte, due to the high cost of individual disk accesses. Fetching a section ensures multiple rows are brought into RAM in a single operation.

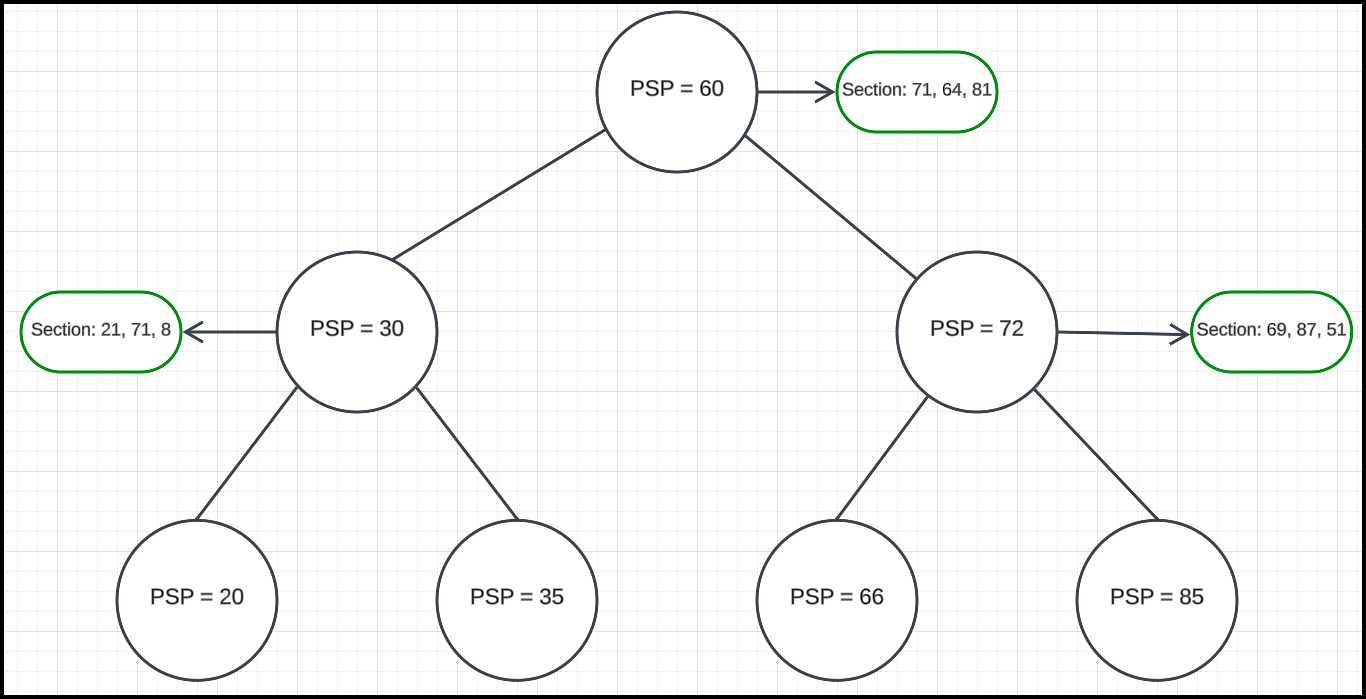
1. Analogy for Understanding Sections:

The relationship between disk sections and rows is likened to identifying which train bogie contains a specific seat. A bogie may include other unrelated seats, similar to a disk section containing rows with different attributes.

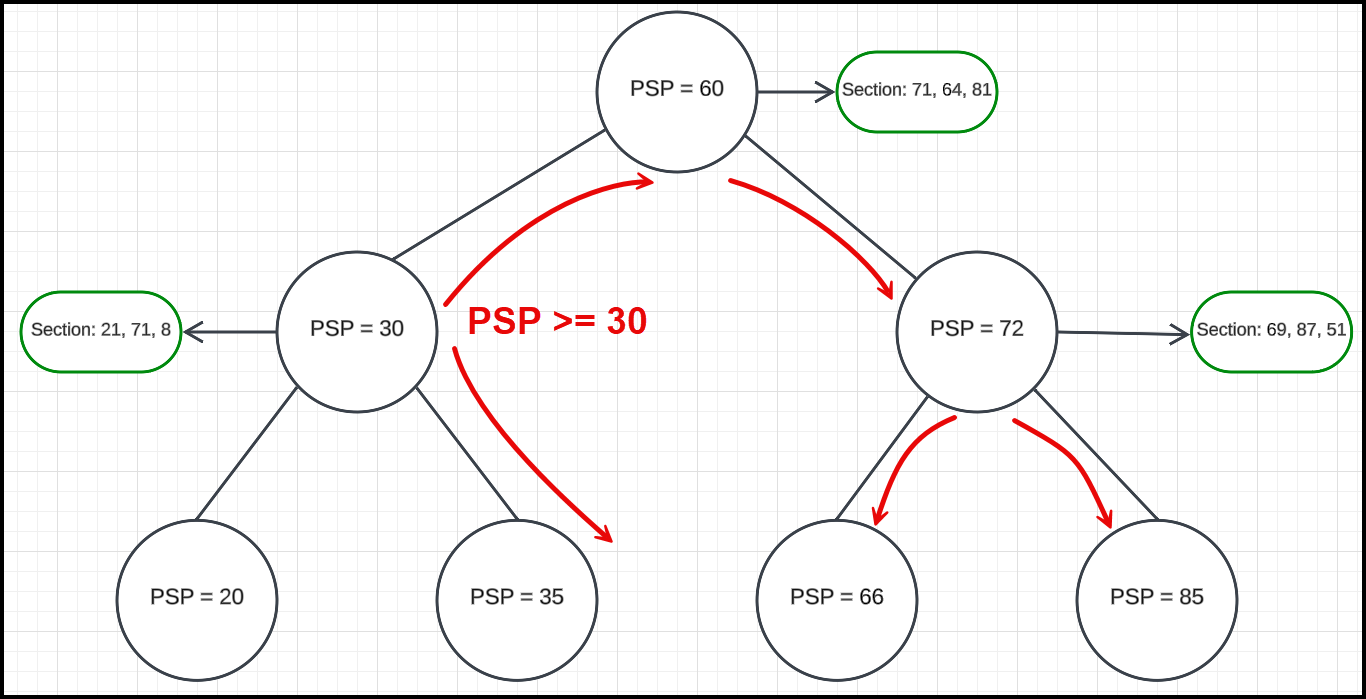
1. Challenges of Handling Large Data:

Operations like DISTINCT or SUM require careful consideration of memory and storage constraints. Efficient indexing and data partitioning strategies help mitigate these challenges.

* What happens when column values are updated but the index is not updated?
  + If the index is not updated when column values change, the index becomes inconsistent. This inconsistency leads to incorrect or inefficient query results. For example, if a phone number in a database changes but its index isn't updated, querying the phone number using the index will fail.
  + In databases:
    - **Reads become faster** because the index provides a direct or near-direct path to the data.
    - **Write, update, and delete operations become slower** because they involve updating the index as well, leading to additional computational overhead.
* Why do hash maps fail for range queries?
  + Hash maps are efficient for exact matches, but they struggle with range queries (e.g., PSP >= 70 AND PSP <= 90). This is because:
    - Hash maps store data based on hash values derived from keys, and these hash values are distributed arbitrarily.
    - Finding values in a range requires traversing the entire hash map, resulting in O(n) time complexity for range queries.
* What data structure can efficiently handle range queries?
  + A **binary search tree (BST)** is a suitable data structure for range queries because:



* + - It is inherently sorted, allowing efficient traversal of elements in a specific range.
    - The time complexity to find a specific value is O(log n) in a balanced BST.
    - Moving to the next greater value (successor) is also efficient, allowing quick iteration through a range.



* Example: Processing a range query with a binary search tree
  + For a query like:

SELECT

    \*

FROM

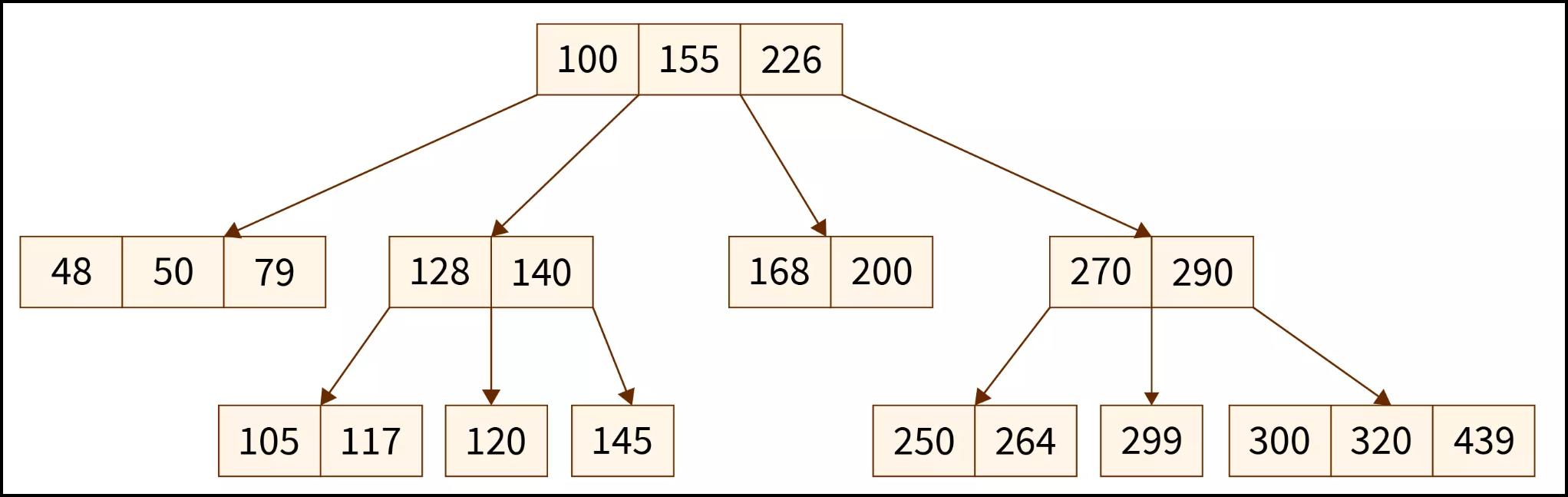
    STUDENTS

WHERE

    PSP >= 30

    AND PSP <= 90;

* + Steps:
    - Start at the root and find the smallest value (≥30).
    - Traverse the tree, visiting nodes in sorted order.
    - Stop once a node with a value >90 is encountered.
  + The above process efficiently retrieves all matching records.
  + By combining indexing strategies and appropriate data structures, databases achieve a balance between read and write performance, catering to diverse query requirements.
* Real-world usage of a tree-based index
  + Databases use tree-based structures like **B-trees** or **B+ trees** to implement indexing:



* + - These structures are optimized for disk-based storage and allow efficient range queries and sequential access.
    - Each node in the tree may store key-value pairs and pointers to child nodes.
    - This setup enables efficient insertion, deletion, and traversal operations, making them ideal for managing large-scale database indexes.

# Index Creation And Maintenance

Indexes in databases are data structures that help improve the performance of queries by allowing faster data retrieval. However, the creation and maintenance of indexes come with trade-offs. Here’s a detailed explanation of how indexes are created and updated, addressing the two main aspects: **initial creation** and **maintenance/updates**.

## **Initial Creation of Indexes**

### **Approach 1: In-Memory Index**

* The index is stored only in the **RAM (volatile memory)**.
* When the database starts:
  + The index is recomputed from the rows stored on the disk.
  + The system traverses every row in the database and reconstructs the index structure (e.g., a B-Tree or a Hash Map).
* **Advantages:**
  + **Write operations** are faster because updates only affect the in-memory index.
  + **Storage requirements** are lower since the index is not stored persistently on the disk.
* **Disadvantages:**
  + If the machine restarts or crashes, the index is lost because RAM is volatile.
  + Upon restart, the system has to **recompute the index**, leading to slower **startup times**.

### **Approach 2: Persistent Index**

* The index is stored both in **RAM** and on the **disk** (persistent storage).
* When the database starts:
  + The index is directly loaded from the disk into memory.
  + This eliminates the need to recompute the index from the rows on the disk.
* **Advantages:**
  + Faster **startup times**, as the index is readily available from the disk.
  + Once loaded, queries can execute immediately without delay.
* **Disadvantages:**
  + **Write operations** become slower because updates need to be made in two places:
    - The in-memory index.
    - The on-disk index.
* Requires more **storage space** since the index is stored persistently.

## **Maintenance and Updates of Indexes**

* Indexes must be kept up-to-date to remain effective. This happens during **write operations** (e.g., INSERT, UPDATE, or DELETE queries). Here's how the updates are handled:

### **Impact of Write Operations:**

* When a value in the indexed column changes, the database must:
  + Locate the old value in the index structure and remove it.
  + Insert the new value into the appropriate position in the index.

### **In-Memory Index Maintenance:**

* Updates are made only to the in-memory copy of the index.
* **Advantages:**
  + Write operations are faster since there is no need to sync changes to the disk.
  + Ideal for scenarios where write performance is critical, and occasional downtime for index re-computation is acceptable.
* **Disadvantages:**
  + The index is volatile. If the system crashes, the updates are lost, and the index must be rebuilt during startup.

### **Persistent Index Maintenance:**

* Updates are made to both the in-memory and on-disk copies of the index.
* **Advantages:**
  + The index remains consistent and reliable across restarts or crashes.
  + Ideal for scenarios where immediate availability after a restart is critical.
* **Disadvantages:**
  + Write operations are slower since every update must be written to both RAM and disk.

### **Trade-Offs Between the Two Approaches**

| **Approach** | **Startup Time** | **Write Speed** | **Read Speed** | **Reliability** |
| --- | --- | --- | --- | --- |
| **In-Memory Index** | Slow (needs re-computation) | Fast | Fast | Low (index lost on restart) |
| **Persistent Index** | Fast (loaded from disk) | Slow (dual updates) | Fast | High (index persists) |

### **Example Scenario:**

* Consider a database storing student information, indexed by their PSP (Problem-Solving Percentage).
* **Initial Creation:**
  + **In-Memory Index:** The database scans all student records and constructs the index in RAM.
  + **Persistent Index:** The database fetches the pre-existing index from disk and loads it into RAM.
* **Maintenance:**
  + When a student's PSP changes (e.g., from 60 to 81), the database must:
    - **In-Memory Index:** Update the in-memory structure only.
    - **Persistent Index:** Update both the in-memory and on-disk structures.

### **Conclusion**

* Choosing between **in-memory** and **persistent** indexes depends on the application's requirements:
  + **In-Memory Indexes** prioritize fast write operations but involve slower startup times and potential data loss during crashes.
  + **Persistent Indexes** ensure reliability and faster startups but make write operations slower due to additional overhead.

## **Indexes - Practice**

* An index is a data structure that improves the speed of data retrieval operations on a database table.
* Types of indexes:
  + **Primary Index**: Automatically created on the **primary key** of a table.
  + **Foreign Key Index**: Created for **foreign key** columns to maintain referential integrity.
  + **Custom Indexes**: Can be created on specific columns to optimize query performance.

### **Default Index on Primary Key**

* Every table automatically has an index on its primary key.
* Example: In the PRODUCTS table, an index is automatically created on the PRODUCT\_ID column.
* The type of index used is typically **B-Tree**.

### **How Indexes Work**

* Without an index:
  + The query must scan all rows in the table (full table scan).
  + Example:

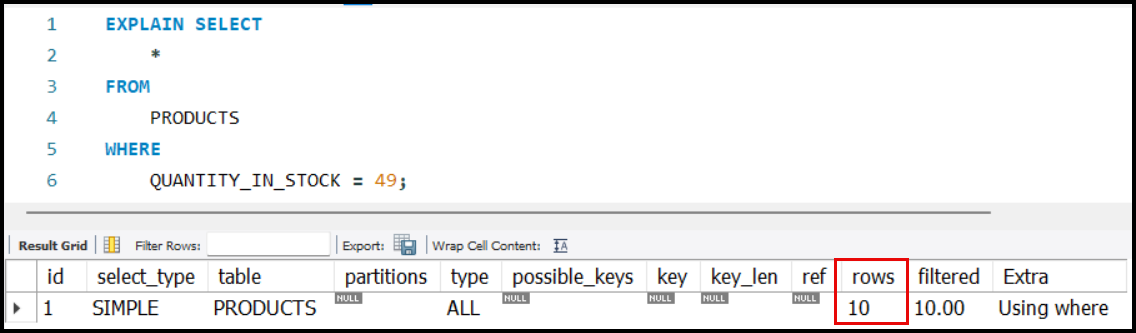
SELECT \* FROM

    PRODUCTS

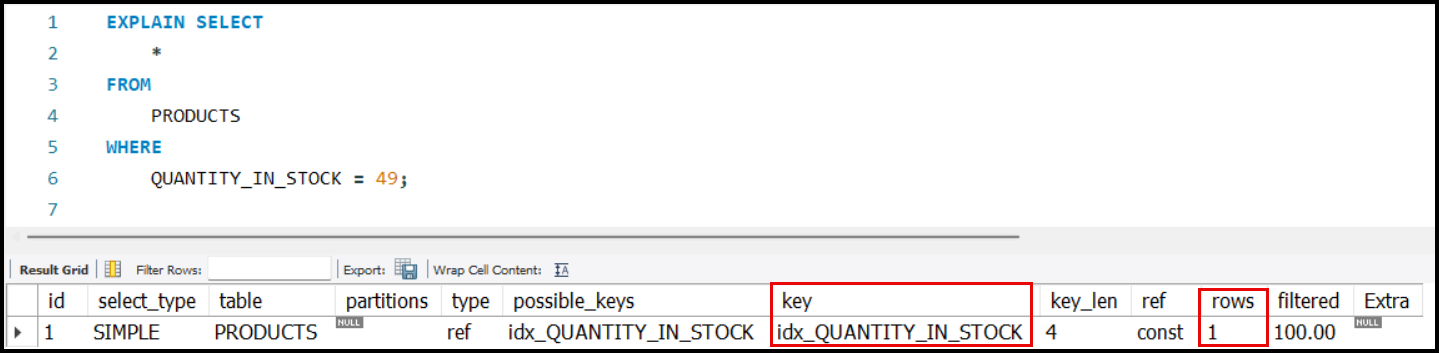
WHERE

    QUANTITY\_IN\_STOCK = 49;

* If there are 10 rows in the table, the query will scan all 10 rows.



* With an index:
  + The query can locate rows directly, reducing the number of rows scanned.
  + Example:

CREATE INDEX IDX\_QUANTITY\_IN\_STOCK ON PRODUCTS(QUANTITY\_IN\_STOCK);

* + - The query scans only 1 row because it uses the newly created index on QUANTITY\_IN\_STOCK.

## **Creating Custom Indexes**

* **Syntax**:
  + Example:

CREATE INDEX

    IDX\_<TABLE\_NAME>\_<COLUMN\_NAME>

ON

    <TABLE\_NAME>(<COLUMN\_NAME>);

CREATE INDEX

    IDX\_QUANTITY\_IN\_STOCK

ON

    PRODUCTS(QUANTITY\_IN\_STOCK);

* Naming Convention:
  + Start with IDX\_.
  + Include the table name and the column name in the index name for clarity.

### **Understanding** EXPLAIN **in SQL**

* The EXPLAIN keyword provides insights into how a query is executed.
  + **Possible Keys**:
    - Lists all the indexes that might optimize the query.
    - Example: If querying on QUANTITY\_IN\_STOCK, an index on PRODUCT\_ID is not a "possible key."
  + **Key**:
    - The actual index used by the query.
    - If multiple indexes are possible, the database optimizer chooses the most efficient one.

### **Effect of Indexes on Query Performance**

* Indexes make read operations faster:
  + For example, searching for a specific value in an indexed column requires scanning fewer rows.
* Indexes may increase the time taken for write operations (e.g., INSERT, UPDATE, DELETE), as the index structure needs to be updated.

### **Key Differences: Possible Keys vs. Key**

* **Possible Keys**:
  + All indexes that might optimize a query.
  + Example: An index on QUANTITY\_IN\_STOCK is a possible key for queries filtering by QUANTITY\_IN\_STOCK.
* **Key**:
  + The actual index used by the database to execute the query efficiently.

### **Important Points**

* Primary key indexes are automatically created for primary key columns.
* Custom indexes must be created explicitly for other columns to optimize specific queries.
* Use EXPLAIN to analyze how queries interact with indexes.
* While indexes improve read performance, they can slightly degrade write performance due to the need to maintain the index structure.
* Use indexes for frequently queried columns.
* Avoid excessive indexing, as it can negatively impact write performance.