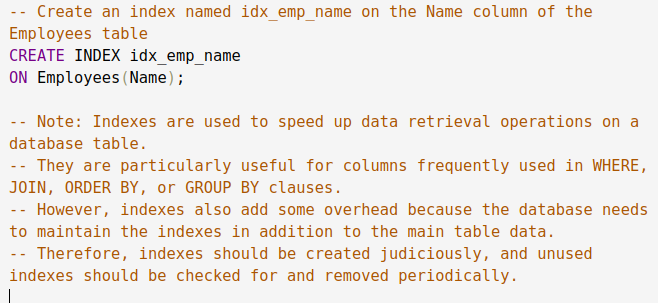
**Techniques for optimizing SQL queries**

Optimising SQL queries is essential to improve the performance and efficiency of database systems. It helps reduce response time, avoid significant lags, and allows servers to run efficiently with low power and memory consumption.

**1. Indexing**

An index is a schema object used by the server to speed up the retrieval of rows using a pointer. Indexes help locate data quickly, speeding up data retrieval operations on a database table. It's important to index columns that are frequently used in WHERE, JOIN, ORDER BY, or GROUP BY clauses.

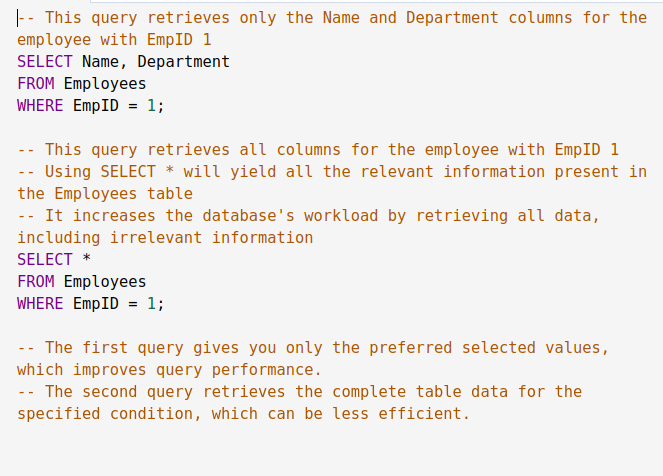
**Note:** While indexes speed up queries, they add overhead since the database must maintain the indexes along with the main table data. Create indexes carefully and periodically check for unused indexes.





**2. Avoiding SELECT \***

Retrieve only necessary columns by specifying the required conditions in a SELECT query instead of using SELECT \*. Using SELECT \* increases workload by retrieving all data, which can slow down performance.



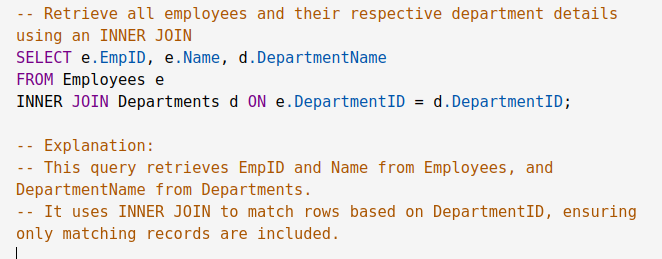
**3. Using Proper Joins**

Use the appropriate type of join to combine data from multiple tables based on a common field. This ensures optimal performance:

* **Inner Join:** Retrieves records with matching values in both tables.
* **Left Join:** Retrieves all records from the left table and matched records from the right.
* **Right Join:** Retrieves all records from the right table and matched records from the left.
* **Full Join:** Retrieves all records when there is a match in either table.

Having indexes on join columns can significantly improve lookup speed.

**Example:** To retrieve all employees and their respective department details, use an Inner Join.



**4. Using EXISTS Instead of IN or COUNT**

When searching for specific records, EXISTS is often more effective than IN or COUNT. EXISTS stops searching after finding the first matching entry, whereas IN and COUNT continue searching.

**Example:** Finding customers who have placed at least one order.

* COUNT: Counts all orders for each customer.
* IN: Checks if the customer ID is in the list of customer IDs returned by the subquery.
* EXISTS: Checks if at least one order exists for each customer.

**5. Using WHERE Clause Instead of HAVING**

Filter data as early as possible to reduce the number of rows processed. Use WHERE instead of HAVING since WHERE filters rows before grouping.

**Example:** To find the total sales amount for products where the sales amount is greater than $1000:

* HAVING: Groups all rows and then filters.
* WHERE: Filters rows before grouping, reducing data processed.

**6. Using JOIN Instead of Subqueries**

Subqueries can slow down query performance because they may return many rows. Joins are typically more efficient as they use indexing and allow merging tables in a single operation.

**Example:** Finding products ordered more than 10 times:

* Subquery: Counts matching rows for each product.
* JOIN: Combines products with order details efficiently.

**7. Using LIMIT or TOP to Sample Query Results**

Use LIMIT or TOP to restrict the number of rows returned, which is useful for large datasets and pagination.

**Example:**

* LIMIT: SELECT ProductID, ProductName, Price FROM Products ORDER BY ProductID LIMIT 10;
* TOP: SELECT TOP 10 ProductID, ProductName, Price FROM Products ORDER BY ProductID;

**8. Avoiding Wildcards at the Beginning of LIKE Patterns**

Using wildcards (like %) at the beginning of a pattern prevents the use of indexes, leading to full table scans.

**Example:**

* Inefficient: WHERE CustomerName LIKE '%Ram';
* Efficient: WHERE CustomerName LIKE 'Ram%';

**9. Using Proper Data Types**

Use the correct data type for each column to save space and improve performance. This helps avoid implicit type conversions.

**Example:**

CREATE TABLE Employees (EmployeeID INT PRIMARY KEY, Name VARCHAR(100), Salary DECIMAL(10, 2), DateOfJoining DATE);

**10. Avoiding Functions on Indexed Columns**

Applying functions to indexed columns can prevent the database from using indexes effectively, leading to slower performance.

**Example:**

* Inefficient: WHERE UPPER(Name) = 'RAJ';
* Optimised: WHERE Name = 'Raj';

**11. Monitor Query Performance**

Monitoring the runtime of queries is critical for identifying poor performance. Use query profiling to analyze statistics such as execution duration and the number of rows returned.

Database-specific monitoring tools like EXPLAIN, SHOW STATUS, and slow query logs in MySQL can help detect issues

**Indexing and its types**

Indexing is a **data structure technique** implemented over database columns that improves the speed of data retrieval operations on a database table by minimizing the number of disk accesses required when a query is processed. It provides a quick way to look up rows in a table based on the values of one or more columns.

Indexing, however, fastens the retrieval at the cost of additional writes and storage space to maintain the index data structure.

**Structure of an Index**



The search key is the first column in the database and it contains a duplicate or replica of the table's candidate key or primary key. The primary key values are saved in sorted order so that the relevant data is easily accessible.

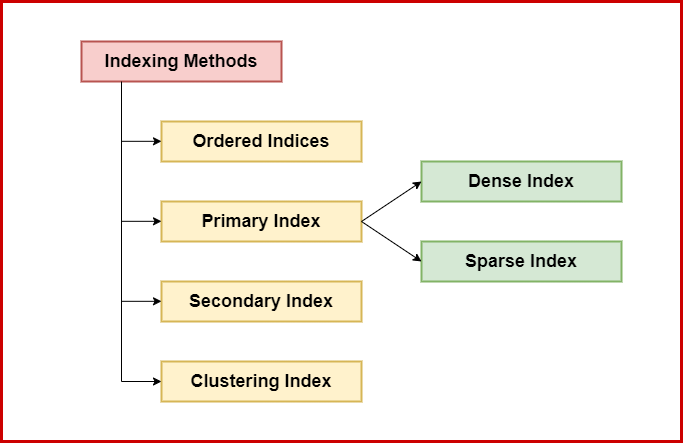
The data reference is the database’s second column. It comprises a collection of pointers that point to the disk block where the value of a certain key is stored.

**Purpose of Indexing**

The main purpose of indexing is to improve the speed of data retrieval operations by reducing the number of disk I/O operations. It helps in:

* **Faster Search Operations**: Indexes allow quick location of data without scanning the entire table.
* **Efficient Query Processing**: Enhances the performance of queries involving SELECT, JOIN, and WHERE clauses.
* **Improved Sorting**: Helps in sorting operations, as indexes maintain the order of the indexed columns.

**Types of Indexes**



1. **Primary Index**

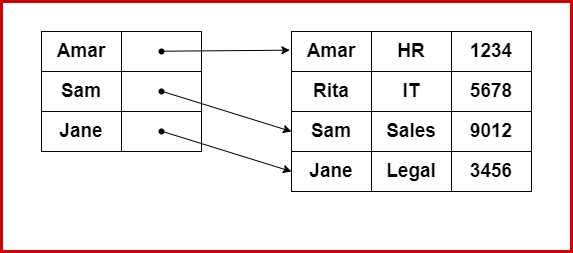
A primary index is created on the primary key of a table. It uniquely identifies each record in the table. The searching operation is very efficient since primary keys are stored in sorted order.

There are two types of primary indexes:

**Sparse Index**

Contains index records for only some of the values. A sparse index points to each block in the data file.

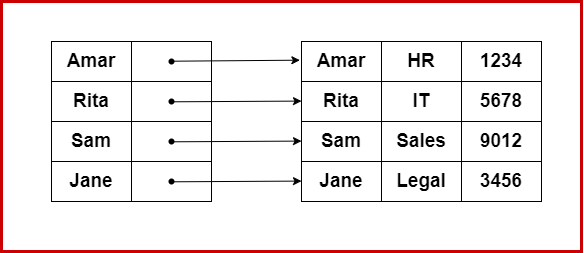
**Example:** If a table has 1000 records, a sparse index might have entries for every 100th record, pointing to the block where the record starts.



**Dense Index**

Contains an index record for every search key value in the data file, pointing to the actual data record. It needs more space to store the index record itself.

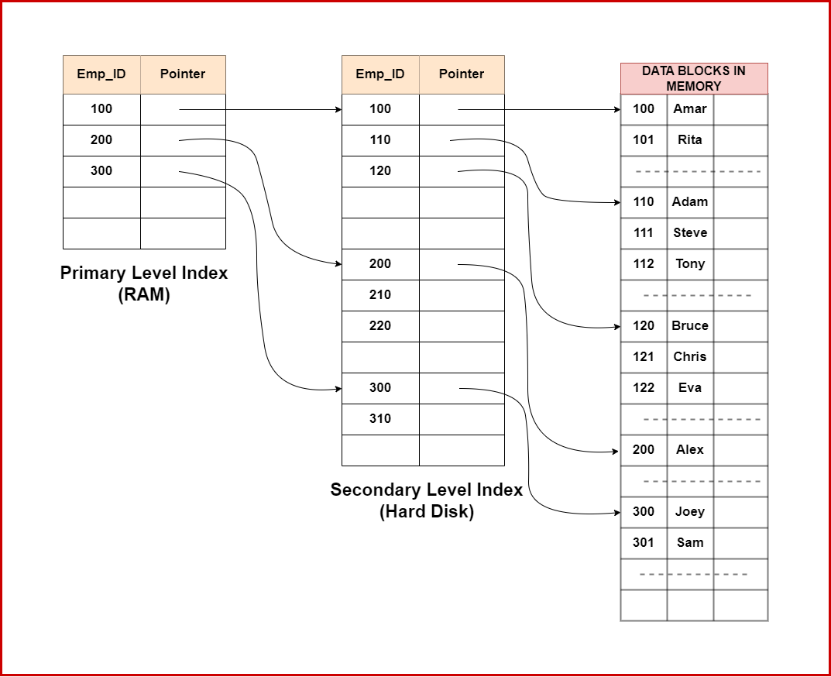
**Example:** For a table with 1000 records, a dense index will have 1000 entries, each pointing to the corresponding data record.



1. **Secondary Index**

When sparse indexing is used, the mapping expands in parallel with the table's size. Secondary indexing adds a new level of indexing to reduce the size of the mapping.

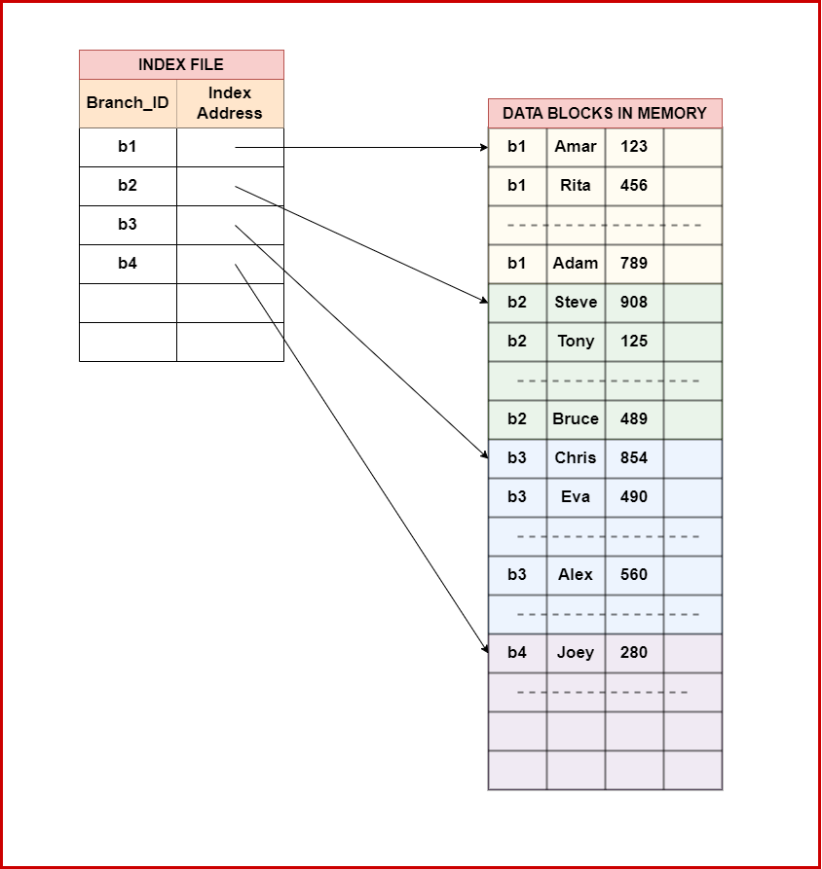
This method begins with selecting the huge range for the columns, resulting in a minimal mapping size at the first level. The ranges are then broken into smaller groups.



1. **Clustered Index (Clustering Index)**

A clustered index can be defined as an ordered data file. To identify the record faster, we will group two or more columns to get the unique value and create an index out of them.

**Example:** For a university database, a clustered index can be created to identify groups of records of students in the same semester or branch. Here Branch\_ID would be a non-unique key.



1. **Ordered Index**

Ordered indices are indices that have been sorted. To make searching easier and faster, the indices are frequently arranged/sorted.

**Example:** In the case of a university database with thousands of student records, if we need to retrieve the record of the student with ID 378, the DBMS would read the record after it reads 378\*2 = 756 bytes using an ordered index, which is significantly less than searching through the entire database.

**Considerations in Indexing**

The most important columns for indexing are selected based on how frequently they are used and the sorts of queries they are subjected to. Regular index maintenance, such as defragmentation and reorganization, can decrease fragmentation and ensure efficiency.

**Advantages of Indexing**

* **Improved Query Performance**: Speeds up the retrieval of data by reducing the number of disk I/O operations.
* **Efficient Sorting**: Helps in sorting data quickly without needing a full table scan.
* **Optimized Searching**: Enhances the performance of search operations by providing quick access to records.
* **Data Integrity**: Ensures unique values are added to indexed columns, helping maintain data integrity.
* **Consistent Data Performance**: Ensures consistent database performance as data grows.

**Disadvantages of Indexing**

* **Increased Storage Space**: Indexes require additional storage space, which can be substantial for large tables.
* **Slower Write Operations**: Insertion, deletion, and update operations become slower due to the overhead of maintaining the indexes.
* **Complexity**: Managing and optimizing indexes can add complexity to database administration.
* **Potential for Fragmentation**: Frequent updates and deletions can lead to fragmented indexes, degrading performance

**B and B+ trees**

In Database Management Systems (DBMS), efficient data storage and retrieval are very important. B and B+ Trees are key data structures that help keep data sorted and allow for quick insertion, deletion, and searching operations.

**B Trees**

**B Trees** are balanced tree data structures that help in storing sorted data and allow operations like searching, adding, removing, and accessing data quickly.

**Properties:**

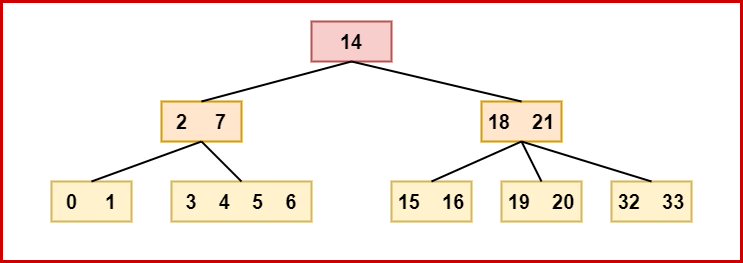
* Each node can have a maximum of **m** children and **(m-1)** keys, where **m** is the order of the tree.
* Non-root and non-leaf nodes can have at least **ceil(m/2)** children.
* All leaves are at the same level.
* Keys within nodes are sorted, guiding searches.

**Insertion:**

Nodes are added only at leaf nodes, allowing B Trees to grow and shrink from the root.

**Example:**

For instance, in a library database, a B Tree can efficiently find a book by its ISBN, add new books, or remove old ones.



**B+ Trees**

**B+ Trees** are an extension of B Trees, where all keys are stored at the leaf level, and internal nodes only store keys to guide searches.

**Properties:**

* Internal nodes act as guides, while actual data is in the leaf nodes.
* Leaf nodes are linked for efficient range queries and ordered traversals.
* Like B Trees, B+ Trees maintain balance with minimum and maximum keys per node.

**Example:**

An e-commerce platform can use a B+ Tree to manage product listings, allowing for quick lookups and handling range queries efficiently.

**B and B+ Trees vs. Conventional Data Structures**

* **Efficient Disk Access:** B Trees work well with slow storage like hard drives, unlike arrays and linked lists.
* **Self Balancing:** B Trees and B+ Trees keep paths from root to leaves of similar length for efficient searching.
* **Handling Large Volumes of Data:** They can manage big datasets that don’t fit in memory.
* **Dynamic Size Handling:** They adjust size easily without major reorganizations, unlike arrays.
* **Support for Range Queries:** B and B+ Trees allow efficient range queries while also enabling fast insertions and deletions.

**Applications of B and B+ Trees**

* **File Systems:** Used in systems like NTFS and ext4 for quick file operations.
* **Databases:** Databases like MySQL and PostgreSQL use B+ Trees for fast query performance.
* **Search Engines:** B+ Trees index web pages by keywords for quick retrieval.
* **Telecommunication Networks:** They manage subscriber information and call records effectively.
* **Geospatial Databases:** Systems like GIS use B+ Trees to index spatial data for efficient geographical queries.

B Trees and B+ Trees are vital in DBMS, ensuring efficient data management. Their balanced nature supports fast performance for adding, removing, and finding data, making them ideal for applications with large data needs.