Indexing in Database (Detailed Explanation)

# What is Indexing in a Database?

Indexing is a database technique used to optimize the speed of data retrieval. It creates a data structure that allows the database engine to find records more efficiently, without scanning every row in a table.

# Why is Indexing Needed?

Imagine a table with millions of rows. If you run a query like:

SELECT \* FROM employees WHERE employee\_id = 12345;

Without an index, the database performs a full table scan — checking every row to find a match. An index creates a shortcut for the database, allowing it to go directly to the data.

# How Does Indexing Work?

Most relational databases use a B-Tree (Balanced Tree) or a Hash Table to store index data. An index stores the values of one or more columns in a sorted order, along with pointers (references) to the actual table rows. This allows for faster searching.

# Types of Indexes

1. Primary Index (Clustered Index): Automatically created on the primary key.

2. Secondary Index (Non-clustered): Separate from actual table data.

3. Unique Index: Ensures all values are unique.

4. Composite Index: Multi-column index.

5. Full-text Index: For text searches.

6. Spatial Index: For geographical data.

7. Bitmap Index: Efficient for low cardinality columns.

# Benefits of Indexing

✔ Fast Query Execution

✔ Reduces I/O

✔ Improves Sorting

✔ Enforces Uniqueness

✔ Better JOIN Performance

# Drawbacks of Indexing

✘ Slower Write Operations

✘ More Storage Usage

✘ Maintenance Overhead

# When to Use Indexing?

Use indexes when:

- You frequently filter/search by a column (WHERE).

- You JOIN tables using foreign keys.

- You sort/group by columns.

- Table has many rows.

- Column has high cardinality.

Avoid indexing:

- Columns with few values.

- Tables with frequent writes.

- Very small tables.

# SQL Examples

CREATE INDEX idx\_name ON students(name);

CREATE INDEX idx\_name\_age ON students(name, age);

DROP INDEX idx\_name ON students;

# B-Tree vs Hash Index

B-Tree Index:

- Range queries, sorting

- Maintains order

- Supports <, >, BETWEEN

Hash Index:

- Exact-match only

- No order

- Only = supported

Table: students

| **id** | **name** | **age** |
| --- | --- | --- |
| 1 | Alice | 22 |
| 2 | Bob | 25 |
| 3 | Charlie | 23 |

Index on 'name':

"Alice" → Row 1

"Bob" → Row 2

"Charlie" → Row 3

When you query WHERE name = 'Bob', it uses this index instead of scanning the entire table.

| **Type** | **Description** |
| --- | --- |
| **Primary Index** | Automatically created on a primary key. |
| **Unique Index** | Ensures all values in the indexed column are unique. |
| **Composite Index** | Index on multiple columns. |
| **Clustered Index** | Sorts actual data rows in the order of the index. |
| **Non-clustered Index** | Maintains a separate structure for the index. |
| **Full-text Index** | Used for searching large text fields. |