INDEXES IN DATABASE

TO IMPROVE PERFORMANCE

Indexes

📘1. Indexing Overview

- 🧬 Origin of indexes
- 🟦 Structured Format ?

🧱 2. Types of Indexes

- 🌳 B-Tree Index
- 🌳 B+ Tree Index (Enhanced version of B- Tree)
  1. Clustered
  2. No-Clustered index
  3. Composite (multi-column) index
  4. Unique index
- 🧮 Bitmap Index

🗂️3. How Data is Stored Using Indexes

📊4. How do `EXPLAIN` and `EXPLAIN ANALYZE` help in query analysis?

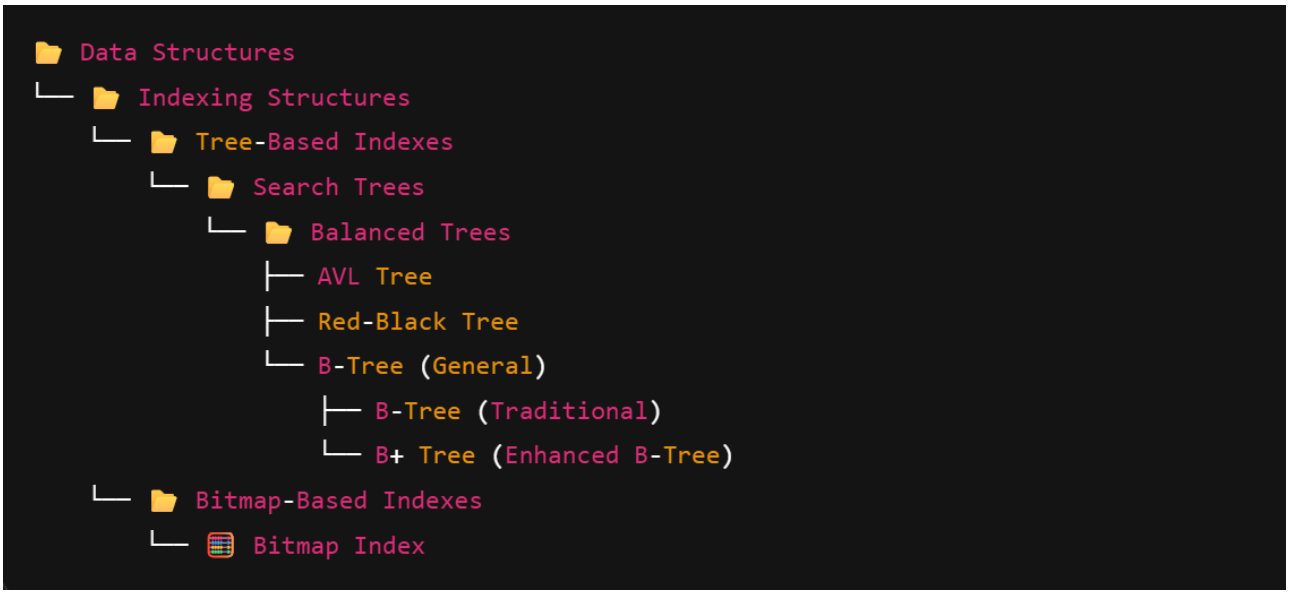
- ⚙️CPU cost
- 💽 I/O cost
- 🔢 Number of rows read
- 🔁 Execution steps (loops, join types)
- 🔍 Scan types (full table scan, index scan, index seek)

⚠️ 5. What are common indexing mistakes and how to avoid them?

Note: I used MySql database - workbench

Created By Ashokachari P, Good Luck👍

Origin of the indexes ?



📂 Data Structures

This is the top-level category in computer science that refers to ways we organize and store data for efficient access and modification.
Examples include arrays, linked lists, stacks, queues, and trees.

└── 📂 Indexing Structures

A category of data structures used to speed up the retrieval of records in large datasets—commonly used in database systems.
Indexes reduce the need for full table scans, improving performance of queries.

└── 📂 Tree-Based Indexes

These are indexes that rely on tree-shaped hierarchical structures.
They provide fast lookup, insert, delete, and range operations, typically in **O(log n)** time.

└── 📂 Search Trees

Search trees are specialized for storing **sorted data** and allow efficient searching.
Their design ensures values are positioned in a way that supports **binary-style decisions** at each level of the tree.

└── 📂 Balanced Trees

Balanced trees ensure that no branch of the tree becomes too deep, which could degrade performance.
These trees automatically balance themselves when data is inserted or deleted.

Balanced trees ensure:

- Optimal performance: O(log n) time for search/insert/delete.
- Better memory and I/O efficiency.

├── 🌿 AVL Tree

- Named after inventors **Adelson-Velsky and Landis**.
- First self-balancing binary search tree.
- Rebalances the tree using **rotations** after insertions or deletions.
- Maintains a **strict balance factor** (height difference between left and right subtrees is at most 1).
- Suitable for **in-memory applications** but less for disk-based systems due to rebalancing overhead.

├── 🌿 Red-Black Tree

- Another self-balancing binary search tree, but less strict than AVL.
- Balances the tree using **coloring rules** instead of strict height checking.
- Allows faster insertion/deletion than AVL (because it rebalances less often).
- Widely used in:
    - **Java TreeMap**
    - **Linux kernel**
    - **C++ STL map/set**

└── 🌳 B-Tree (General)

- A **multi-way search tree**, where nodes can have more than two children.
- Designed specifically for **disk-based systems** to reduce I/O operations.
- Keeps data sorted and allows searches, sequential access, insertions, and deletions in **logarithmic time**.
- Every node can contain multiple keys and children, unlike binary trees.

├── 🧱 B-Tree (Traditional)

- Both **internal nodes and leaf nodes store keys and values**.
- Searching can end at internal nodes.
- Supports point queries well but less efficient for **range queries**.
- Used in some **older file systems** and early DBMS implementations.

└── 🧱 **B+ Tree (Enhanced B-Tree)**

- Most widely used tree structure in modern RDBMS (e.g., **MySQL InnoDB**, **PostgreSQL**, **SQL Server**).
- **Internal nodes contain only keys**, no actual data.
- **Leaf nodes store all data** and are linked together for fast sequential/range access.
- Advantages:
    - Efficient **range scans**
    - Great for **disk-based storage**
    - Supports **index-only scans** (since data is in leaves)

└── 📂 Bitmap-Based Indexes

Bitmap indexing is a technique used especially in **data warehouses** and **OLAP systems**.

- Unlike tree indexes, these use **bitmaps (arrays of bits)** to represent the presence of values in a column.
- Best suited for **columns with low cardinality** (e.g., gender, status, yes/no).

└── 🧮 **Bitmap Index**

- For each distinct value in a column, a **bit vector** is created.
  Example: A `gender` column with values `Male` and `Female` will have two bitmaps.
- A 1 in a bitmap indicates the row contains that value; `0` means it does not.
- Efficient for:
    - **Complex Boolean conditions** (`AND`, `OR`, `NOT`)
    - **Aggregation**
    - **Filtering large datasets**
- Great for analytical queries, but:
    - **Not ideal for frequent updates**
    - **Not suited for high-cardinality columns**

Don't Confuse:

*Note: ✅ **B+ Tree indexes are an enhanced version of B- Tree indexes** — with important structural improvements that make them **faster and more efficient for database indexing**.*
And clustered, non-clustered, unique and composite indexes are the part of the B+/B- indexes.

🔍 What Are Indexes?

Indexes are **data structures** used by relational databases to **quickly locate rows** without scanning the entire table=.
They work like a book's index — pointing you to the exact page (row) for a specific keyword (column value).
Indexes dramatically speed up **SELECT** queries but can add overhead to **INSERT**, **UPDATE**, and **DELETE** operations.

1. Clustered index:

- Data rows themselves are stored in the leaf nodes of the B-Tree.
- The table's physical order on disk matches the index order.

🔹 Page

- A **physical block** of storage on disk or in memory.
- Typically fixed size (e.g., 8 KB in MySQL/InnoDB, SQL Server, PostgreSQL)
- It is the **smallest unit of I/O** — the database reads/writes pages, not individual rows.
- **All B-Tree nodes** (root, internal, leaf) are stored in **pages**.

👉 Think of a **page** as a *container* of rows or keys.

🔹 Node

- A **logical structure** inside the B-Tree: Root node, Internal node, Leaf node.
- Each node is **stored in a page**.
- Nodes contain:
  - **Keys** (index values)
  - **Pointers** to other nodes or to rows
  - Metadata (e.g., number of keys, sibling pointers, etc.)

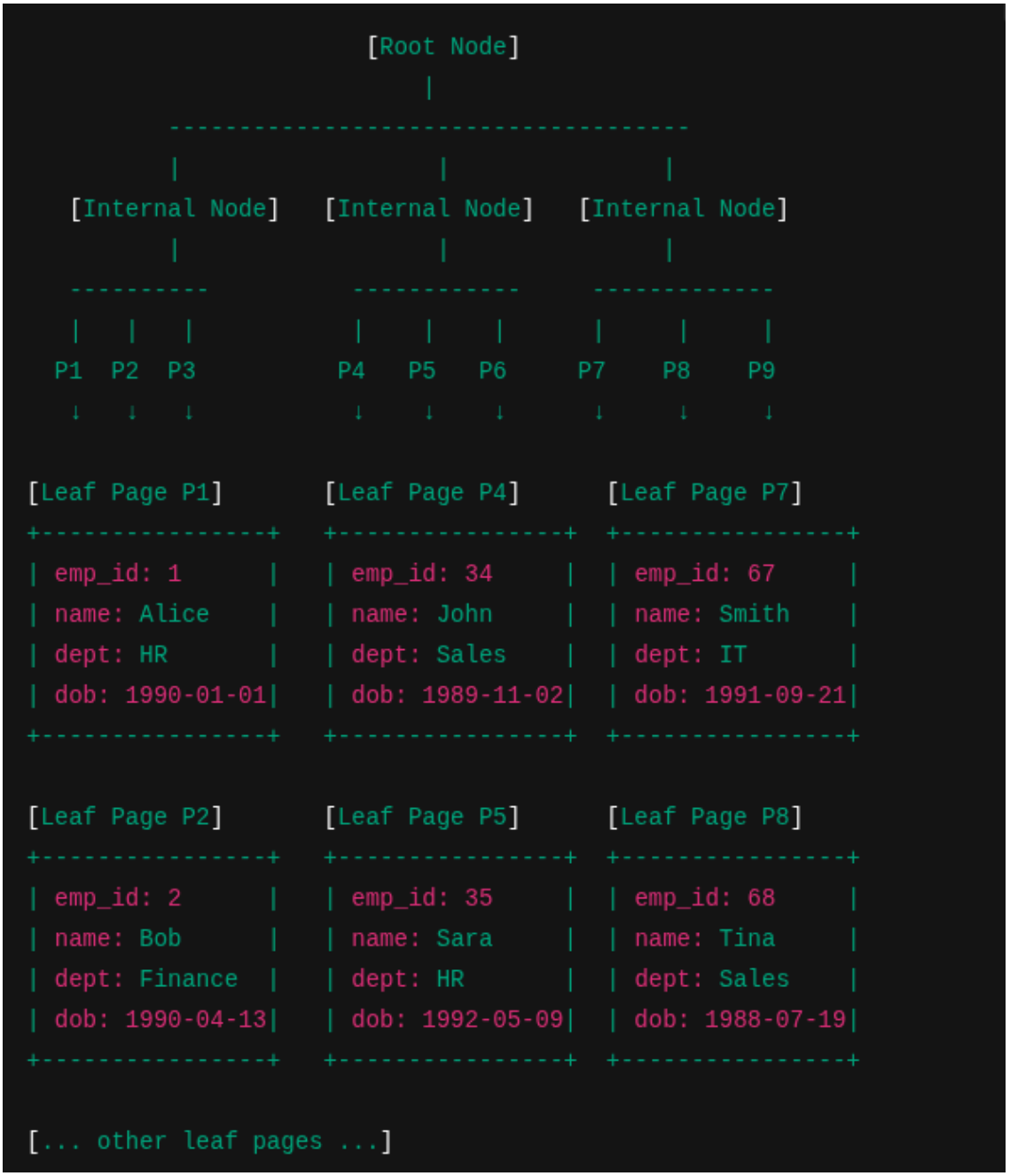
👉 A **node is the role** or structure, a **page is the container** that holds it.

How to create clustered index:

✅ In MySQL

- **You CANNOT create a clustered index explicitly using** .
- The **clustered index is always tied to the** .
- You can only create it **implicitly** by declaring a `PRIMARY KEY`.

This is how the data internally stores:



Note: This picture is talking about the enhanced b- tree storage structure.

💡 Explanation of Diagram Structure:

- **Root Node**: Entry point, contains index keys and pointers to internal nodes.
- **Internal Nodes**: Help navigate to the right leaf page.
- **Leaf Pages**: Store **actual table rows** sorted by .

Each **leaf page** holds **multiple rows**, and the rows are **physically sorted** by .

2. Non-Clustered Index

✅ Key Idea:

- The **index is separate** from the table data.
- The leaf nodes **store keys + row locators** (not actual rows).

📦 How It Stores:

```
[B-Tree Node]
  - Key: dept_id
  - Value: row pointer (e.g., to clustered index or heap row)

[Leaf Nodes]
  dept_id=10 → pointer to row with dept_id=10
  dept_id=20 → pointer to row with dept_id=20
```

Example:
CREATE INDEX idx_dept_id ON employee(dept_id);

3. Unique Index

✅ Key Idea:

- A **Unique Index** ensures that **no two rows** in a table have the **same value** in the indexed column(s).
- It's a constraint **and** a performance feature.

🧠 Use Case:

- Email addresses
- Social Security Numbers (SSNs)
- Mobile numbers
- Any other business key that must be unique

📦 How It Stores:

- Same structure as clustered/non-clustered index
- Database rejects inserts/updates if duplicate keys are attempted.

🔍 **Example:** CREATE UNIQUE INDEX idx_email ON employee(email);

4. Composite (Multi-Column) Index

✅ Key Idea:

- A **composite index** (also called a **multi-column index**) is an index created on **two or more columns** of a table. It helps improve performance for queries that **filter, join, or sort using those columns together**.

Benefits of Composite Indexes:

- Faster filtering on **multiple columns**
- Optimized **JOIN**, **ORDER BY**, and **GROUP BY** when columns match the index
- Reduce the need for multiple single-column indexes

📌 Use Composite Index When:

- Your query often filters by **two or more specific columns**
- You want to avoid creating multiple separate indexes (which take more space and maintenance)

📦 How It Stores:

```
Leaf Node Entries:
  (dept_id=10, emp_name='Alice') → pointer
  (dept_id=10, emp_name='Bob')   → pointer
  (dept_id=20, emp_name='Eve')   → pointer
```

Example:
CREATE INDEX index_name ON table_name (column1, column2, ...);

Primary key and foreign keys from indexes perspective:

Primary key:

- when we create a table with primary key and unique index will be created (BTREE).
- the primary key index is also the clustered index, meaning:
  - Table data is physically stored in the order of the primary key.Foreign key:

Foreign key:

- when we create a table with foreign key and non-unique index will be created (BTREE).
- Other indexes (called secondary indexes) refer to the primary key to locate full rows.
  - You can see which is secondary and which is the primary index by “show indexes on table_name”.

Statistics

SHOW INDEXES FROM customers;

```
10 •  SHOW INDEXES FROM customer;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
customer	0	PRIMARY	1	customer_id	A	49910	NULL	NULL		BTREE			YES	NULL
customer	1	city_id	1	city_id	A	521	NULL	NULL		BTREE			YES	NULL

- **`Table`:** The table name.
- **`Non_unique`:** Whether the index can have duplicates (`0` = unique, 1 = not unique).
- **`Key_name`:** The name of the index (e.g., `PRIMARY`, or a custom index name).
- **`Seq_in_index`:** The sequence of the column in the index (useful for multi-column indexes).
- **`Column_name`:** The name of the indexed column.
- **`Collation`**: How the column is sorted in the index (`A` for ascending).
- **`Cardinality`:** Estimate of the number of unique values.
- **`Index_type`:** The type of index (e.g., `BTREE`, `FULLTEXT`, `HASH`).

✅ *`EXPLAIN Plan Output :`*

- **`Column:`** Meaning
- **`Id:`** Step in the plan (1 = first)
- **`Select_type:`** Type of query (e.g., SIMPLE)
- **`Table:`** Table being accessed
- **`Type:`** Access type (const, ref, range, ALL, etc.)
- **`Possible_keys:`** Indexes that could be used
- **`Key:`** Index that is actually used
- **`Key_len`**: Length of the index used
- **`Ref:`** Column compared to the index
- **`Rows`**: Estimated number of rows scanned (1 is best for PK lookups)
- **`Filtered:`** Percentage of rows filtered (ideally 100.00)
- Extra Any additional info (e.g., Using index)
- Note : there are 5000 records in the customer table.

✅ *EXPLAIN ANALYZE Plan Output :*

- 🔍 **Iterator / Operation** → e.g., `Index scan`, `Table scan`, `Nested loop`
- 📋 **Table** → The table being accessed
- 📊 **Rows examined** → Actual rows read from table/index
- 📤 **Rows produced** → Actual rows returned by this step
- 🎯 **Estimated rows** → MySQL's estimate for rows
- ⏱️ **Actual time** → Real start and end time for each step
- 🔁 **Loops** → Number of times this operation was executed
- 🧭 **Access type** → e.g., `const`, `ref`, `eq_ref`, `ALL` (full scan)
- 🧮 **Index used** → Shows the index used for that step (if any)
- 🔗 **Join type** → e.g., `Nested Loop Join`, `Hash Join`
- 🎯 **Filtered %** → % of rows that passed the condition
- 💰 **Cost info** → Includes estimated:
    - 🧠 **CPU cost**
    - 📦 **I/O cost**
    - 📈 **Subtree cost**
- 🧠 **Execution order** → Step-by-step logical flow
- 💾 **Buffer info** → Indicates if data was read from memory (buffer pool) or disk
- 🧩 **Condition pushed down** → Whether filter conditions are pushed to storage engine

Explain plan for the Query With index :

```
17 •  EXPLAIN SELECT * FROM customer WHERE customer_id = 100;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customer	NULL	const	PRIMARY	PRIMARY	4	const	1	100.00	NULL

Explain Analyse plan for the QueryWith index:

```
23 •  EXPLAIN ANALYZE SELECT * FROM customer WHERE customer_id = 100;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

EXPLAIN
-> Rows fetched before execution (cost=0..0 ...

-> Rows fetched before execution (cost=0..0 rows=1) (actual time=700e-6..800e-6 rows=1 loops=1)

Explain plan for the Query Without index :

Created another table by copying the customer table to distinguish.

CREATE TABLE customer_copy AS SELECT * FROM customer;

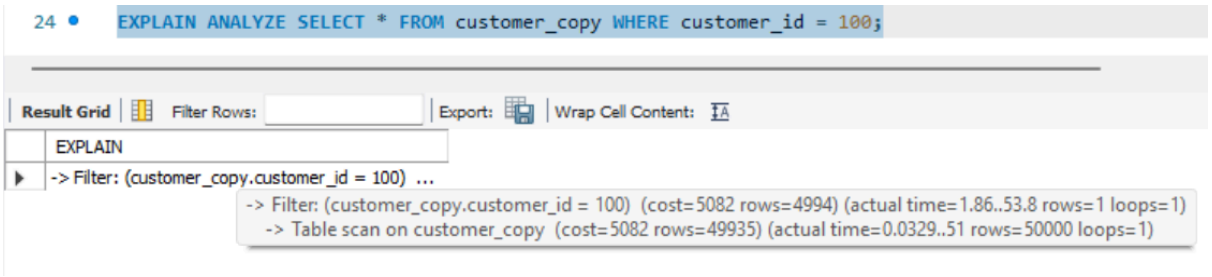
```
21 •  explain select * from customer_copy where customer_id=100;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customer_copy	NULL	ALL	NULL	NULL	NULL	NULL	49935	10.00	Using where

Note: we can see the attributes information in the above explain plan.

Explain Analyse plan for the Query Without index:



✅ What is Cost in MySQL?

Query cost is an estimated number that represents how "expensive" a query is to execute. The MySQL optimizer uses it to choose the fastest execution plan.

→ Think of cost as a score — **lower is better**.

💡 Factors That Affect Query Cost

- **Number of Rows Examined**
  The more rows the query scans, the higher the cost.

- **Table Size**
  Larger tables take more time to process, increasing the cost.

- **Index Usage**
  Using indexes reduces the cost. If no index is used, a full table scan increases cost significantly.

- **Join Operations**
  Queries with multiple joins, especially on large tables, increase the cost.

- **Sorting and Grouping**
  Using `ORDER BY`, `GROUP BY`, or `DISTINCT` adds CPU and memory overhead.

- **Subqueries and Nested Queries**
  Complex subqueries may cause the optimizer to perform more work, increasing the cost.

- **Functions on Columns**
  Using functions in `WHERE` or `JOIN` clauses (e.g., `YEAR(date_column) = 2024`) can prevent index use, increasing cost.

- **Data Distribution (Selectivity)**
  Highly selective conditions (e.g., `customer_id = 1`) cost less than broad ones (e.g.,

`gender = 'M'` if 50% of rows match).

- **Temporary Tables / Disk Usage**
  If MySQL needs to create temp tables or use disk (e.g., due to a big sort or join), cost goes up.

- **Loops and Execution Steps**
  More operations (like multiple nested loops) make the query more expensive to execute.

1. Table Scans

🔍 1. Full Table Scan:

🔸 What is it?

- MySQL reads every row in the table.
- Happens when there is no usable index for the condition.

⚠️ Performance:

- Slowest, especially on large tables.
- Avoid for large datasets unless absolutely needed.

🧪 Example:

```
SELECT * FROM customer WHERE first_name = 'John';
```

Note: If first_name has no index → full table scan.

🔍 2. Full Index Scan (type = index)

🔸 What is it?

- MySQL reads every entry in an index instead of the full table.
- Rows are read in index order (efficient for ORDER BY).
- Still reads all rows, just through the index structure (e.g., B-Tree).

⚠️ Performance:

- Better than a full table scan because index data is smaller and already sorted.
- Still not ideal for filtering — it's just reading everything in index order.

🧪 Example:

```
SELECT first_name FROM customer;
```

Note: If first_name is indexed and the query doesn't need full rows.

🔍 3. Index Range Scan (type = range)

🔸 What is it?

- MySQL uses an index to read a range of rows.
- Common with conditions like >, <, BETWEEN, or LIKE 'abc%'.

✅ Performance:

- Very efficient if the range is selective.

🧪 Example:

```
SELECT * FROM customer WHERE customer_id BETWEEN 100 AND 200;
```

🔍 4. Index Lookup (Exact Match) (type = ref, eq_ref, or const)

🔸 What is it?

- MySQL uses an index to directly find matching rows.
- Most efficient — especially when using PRIMARY or UNIQUE keys.

✅✅ Best Performance

🧪 Example:

```
SELECT * FROM customer WHERE customer_id = 100;
```

💾2. I/O Cost (Input/Output Cost)

🔍 What is it?

- **I/O cost** is the estimated cost of reading data **from disk or memory**.
- MySQL uses **disk I/O** when it reads rows, index blocks, or temporary tables that are **not in the buffer pool** (memory cache).

📚 Examples of I/O operations:

- Reading table rows (full scan or range scan)
- Reading index pages (B-Tree nodes)

- Sorting large datasets into temporary files
- Creating temporary tables on disk for `GROUP BY`, `ORDER BY`, etc.

⚠️ High I/O cost = more disk access = slower query

🧪 Real-world analogy:

Imagine searching for a page in a **physical book** (disk) vs a **digital note** on your phone (memory). Disk is slower.

✅ How to reduce I/O cost:

- Use **indexes** to reduce row scans
- Increase **InnoDB buffer pool size** to cache more data in memory
- Use `COVERING INDEXES` so MySQL doesn't need to go back to the table
- Optimize queries to avoid temporary tables and filesorts

🧠3. CPU Cost

🔍 What is it?

- CPU cost is the estimated **processing time** required by MySQL to:
  - Evaluate expressions and conditions
  - Apply filters (`WHERE`)
  - Perform sorting, aggregation, grouping
  - Process joins and result sets

🧮 CPU cost is mainly affected by:

- Number of rows read
- Complexity of filtering and joining logic
- Functions or expressions (e.g., `LOWER(name)`)
- Sorting or grouping large datasets

🧪 Real-world analogy:

Disk = slow bookshelf (I/O), CPU = your brain processing the info.

✅ How to reduce CPU cost:

- Avoid complex expressions and functions in `WHERE`, `JOIN`
- Fetch only needed columns
- Use **indexed filters** to reduce rows early
- Avoid sorting/grouping large sets unless needed

🔁4. Loop Count (Nested Loop Iterations)

🔍 What is it?

- Shows **how many times a part of the plan is repeated**, usually in a **nested loop join**.
- If you're joining two tables, and the inner loop runs for each row in the outer loop, this count can grow large.

📌 Example:

Suppose you join `customer (1000 rows)` with `orders (10000 rows)`:

Example:

```
SELECT * FROM customer

JOIN orders ON customer.customer_id = orders.customer_id;
```

- MySQL might loop over **1000 customers**, and for each one, scan matching orders.
- If poorly indexed, it might check **orders** 1000 times = high loop count.

⚠️ High loop count = performance bottleneck

✅ How to reduce loops:

- Add indexes on join keys
- Avoid nested subqueries that loop per row
- Flatten queries when possible (use joins instead of correlated subqueries)

🔹 5. Index Maintenance & Storage Cost

While indexes **improve read performance**, they come with **costs** that affect **writes, storage, and maintenance**. Understanding these trade-offs is key.

⚙️ A. Write Performance Impact

Whenever you do:

- `INSERT`
- `UPDATE` (on indexed column)
- `DELETE`

MySQL has to:

- Update **each index** associated with the table.
- Re-balance **B-Trees** (used by most indexes, including BTREE).
- Possibly **lock** resources longer if concurrency is high.

📌 **Example:**
If you have 5 indexes on a table, each `INSERT` triggers 5 separate index updates.

🧱 B. Storage Overhead

Each index:

- Requires disk space.
- Grows with table size.
- May consume RAM in buffer pool.

🧮 **Estimate**:

- A table with 10 million rows and a 3-column composite index could consume **hundreds of MBs** or more just for that one index.

🛠️ C. Extra Maintenance Requirements

When your data changes often (OLTP systems):

- You may need to **regularly optimize or rebuild** indexes.
- Use:

```
ANALYZE TABLE table_name;  -- updates index statistics

OPTIMIZE TABLE table_name; -- reorganizes storage and index pages
```

📈 D. Too Many Indexes = Slower Writes + Larger Storage

📌 **Rule of thumb**:

> Don’t index everything. Index only what helps your queries.

🔹 6. Indexing Pitfalls (Common Mistakes to Avoid)

❌ A. Indexing Low Cardinality Columns

Low cardinality = few unique values
E.g., `gender`, `status`, `country_code`

👉 Indexes are ineffective here:

- The optimizer knows filtering won't reduce row access significantly.
- Full or large index scans are nearly as bad as full table scans.

📌 **Use only in composite indexes** where other columns are selective.

❌ B. Over-Indexing

Common mistake:

- Indexing every column "just in case"

Results in:

- Slower writes
- Confusing query optimizer (chooses suboptimal plans)
- High storage cost
- Maintenance burden

👉 **Audit your indexes**:

```
SHOW INDEX FROM table_name;
```

And drop unused ones.

❌ C. Wrong Column Order in Composite Index

For index `(col1, col2)`:

- Works for `WHERE col1 = ?` or `WHERE col1 = ? AND col2 = ?`
- ❌ Doesn't work for `WHERE col2 = ?` only

🧠 Remember: MySQL uses **leftmost prefix** rule.

❌ D. Functions Prevent Index Use

If you use a function on an indexed column:

```
WHERE LOWER(email) = 'abc@gmail.com'
```

👉 MySQL **won't use the index** on `email`.

✅ Rewrite as:

```
WHERE email = 'abc@gmail.com'
```

❌ E. LIKE with Wildcard at the Start

```
WHERE name LIKE '%john'
```

- Index **not used**
- Full scan happens

✅ Use:

```
WHERE name LIKE 'john%'
```

- Index **can** be used

❌ F. Multiple Indexes Matching a Query

If you have:

- Index A on (first_name)
- Index B on (last_name)

Query:

```
SELECT * FROM users WHERE first_name = 'Ashok' AND last_name =
'Punugoti'
```

⚠️ MySQL may pick just **one** index and not use both effectively.

✅ Solution:
Create a **composite index**: `-INDEX (first_name, last_name`