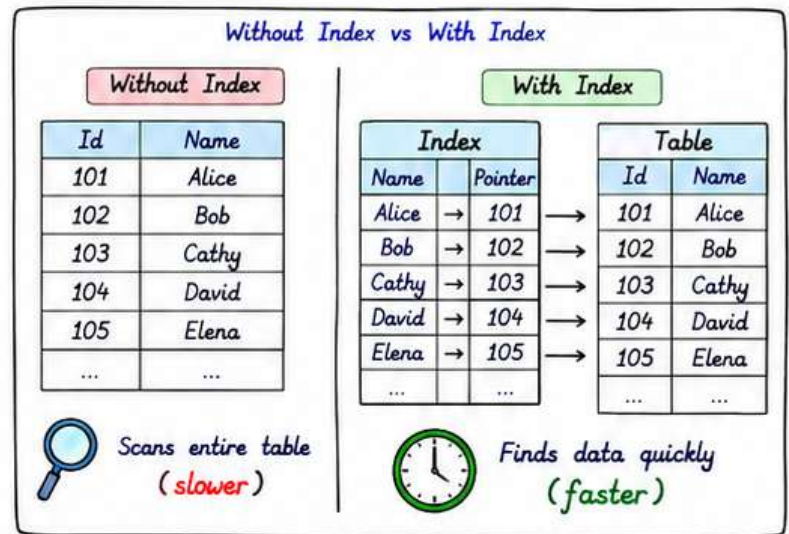


Chapter 17: MySQL Indexes

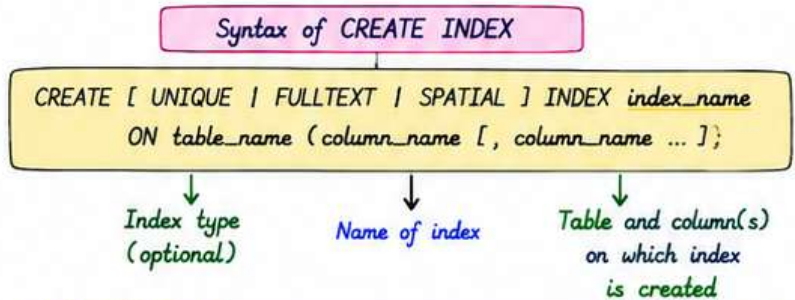
17.1 Introduction to Indexes

- * An index is a **special data structure** that improves the **speed of data retrieval** operations in a table.
- * Indexes work like a **book's index** — they help the database **find the data quickly** without scanning every row.
- * Indexes are created on **one or more columns** (fields) of a table.
- * An index **does not store the data itself**, it stores the values of the indexed columns and the address (pointer) to the actual data.
- * Indexes are very useful for **large tables** where **fast searching, filtering, and sorting** are required.



17.2 Creating Indexes

- * Indexes can be created using the **CREATE INDEX** statement.
- * An index can be created on a **single column** or **multiple columns**.
- * You can specify the type of index (e.g., **PRIMARY**, **UNIQUE**, **INDEX**, **FULLTEXT**, etc.).
- * Creating an index **does not create a new table**, it only creates the index data structure for the existing table.
- * Indexes can also be created during **table creation** using the **INDEX** keyword.



Key Points :

- ✓ Use meaningful names for indexes.
- ✓ Create indexes only on columns that are frequently used in **WHERE**, **JOIN** or **ORDER BY** clauses.
- ✓ Too many indexes can slow down **INSERT**, **UPDATE**, and **DELETE** operations.
- ✓ Indexes make **read operations faster** but may slow down **write operations**.

Example 1 : Create an index on a single column.

Query :

```
CREATE INDEX idx_name ON employees (name);
```

Result :

Index_name	Table_name
idx_name	employees

Example 2 : Create an index on multiple columns.

Query :

```
CREATE INDEX idx_emp_ename ON employees (dept_id, name);
```

Result :

Index_name	Table_name
idx_emp_ename	employees

Example 3 : Create a **UNIQUE** index on a column.

Query :

```
CREATE UNIQUE INDEX idx_email ON users (email);
```

Result :

Index_name	Table_name
idx_email	users

Example 4 : Create a **FULLTEXT** index on a column.

Query :

```
CREATE FULLTEXT INDEX idx_content ON articles (content);
```

Result :

Index_name	Table_name
idx_content	articles

Remember :

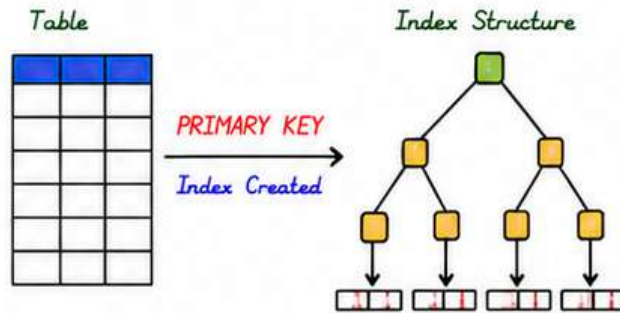
Indexes improve search performance, but they increase storage space and can slightly slow down data modification operations.



17.3 PRIMARY KEY Index

- * **PRIMARY KEY** is a column (or combination of columns) that **uniquely identifies** each row in a table.
- * MySQL automatically creates a **unique index** on the **PRIMARY KEY** column(s).
- * A table can have only one **PRIMARY KEY**.
- * **PRIMARY KEY** columns **cannot** contain **NULL** values.
- * It helps in fast data retrieval and maintains **data integrity**.

How PRIMARY KEY Index Works ?



Helps to locate rows quickly using Primary Key.

Example 1 : Creating PRIMARY KEY while creating the table.

Query :

```
CREATE TABLE students (  
  student_id INT PRIMARY KEY,  
  name VARCHAR(50),  
  department VARCHAR(30)  
);
```

Explanation :

PRIMARY KEY is defined on student_id. MySQL will automatically create a unique index on this column.

Result (students table) :

student_id	name	department
101	Alice	CSE
102	Bob	ECE
103	Cathy	IT
104	David	CSE

Example 2 : Adding PRIMARY KEY using ALTER TABLE.

Query :

```
ALTER TABLE students  
ADD PRIMARY KEY (student_id);
```

Explanation :

Adds PRIMARY KEY constraint on student_id column.

Key Points :

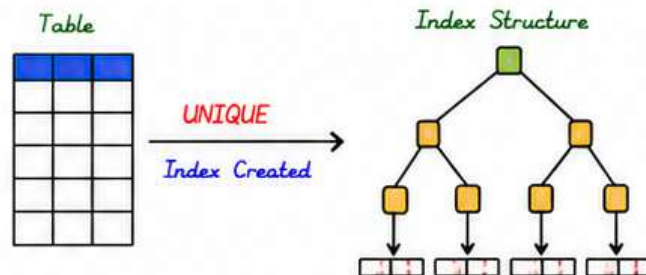
- ✓ PRIMARY KEY creates a unique index automatically.
- ✓ Ensures each row is uniquely identified.
- ✓ Does not allow NULL values.
- ✓ Only one PRIMARY KEY allowed per table.
- ✓ Improves performance of SELECT and JOIN operations.



17.4 UNIQUE Index

- * A **UNIQUE** index ensures that all values in the indexed column(s) are **unique**.
- * A table can have **multiple UNIQUE** indexes.
- * Unlike **PRIMARY KEY**, a **UNIQUE** column can allow **NULL** values.
- * It helps prevent **duplicate data** and improves query performance.

How UNIQUE Index Works ?



Ensures all values are unique and speeds up searching.

Example 1 : Creating UNIQUE index on a column.

Query :

```
CREATE UNIQUE INDEX idx_email  
ON students(email);
```

Explanation :

Creates a UNIQUE index named idx_email on the email column.

Example 2 : Adding UNIQUE constraint using ALTER TABLE.

Query :

```
ALTER TABLE students  
ADD UNIQUE (email);
```

Explanation :

Adds UNIQUE constraint on email column.

Result (students table) :

student_id	name	email	department
101	Alice	alice@gmail.com	CSE
102	Bob	bobe@gmail.com	ECE
103	Cathy	cathy@gmail.com	IT
104	David	david@gmail.com	CSE



Important Note :

- * UNIQUE columns can have NULL values.
- * But duplicate non-NULL values are not allowed.
- * Multiple UNIQUE indexes can exist on a table.



Example 3 : UNIQUE index on multiple columns.

Query :

```
CREATE UNIQUE INDEX idx_name_dept  
ON students(name, department);
```

Explanation :

Ensures that the combination of name and department must be unique.

Key Points :

- ✓ UNIQUE index ensures column values are unique.
- ✓ Allows NULL values (only one NULL per column).
- ✓ Multiple UNIQUE indexes can be created.
- ✓ Helps in improving performance and preventing duplicates.



17.5 Normal Index

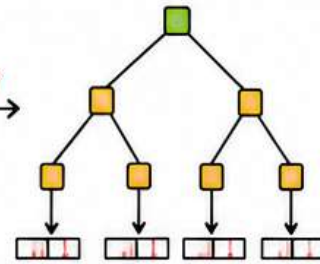
How NORMAL Index Works ?

- * A **NORMAL INDEX** is an index created using the **CREATE INDEX** statement on one or more columns.
- * It is also called as **SECONDARY INDEX**.
- * MySQL allows **multiple normal indexes** on a table.
- * It **does not enforce uniqueness**.
- * It helps in improving the performance of **SELECT, WHERE, JOIN, ORDER BY** and **GROUP BY** operations.

Table

Index Structure

NORMAL INDEX
Index Created



Speeds up searching but does not ensure unique values.

Example Table : employees

emp_id	name	department	salary
1	Alice	Sales	60000
2	Bob	HR	55000
3	Cathy	IT	70000
4	David	Sales	65000
5	Eva	HR	50000
6	Frank	IT	62000

Example 1 : Creating Normal Index on a column.

Query :

```
CREATE INDEX idx_dept
ON employees(department);
```

Explanation :

Creates a normal (non-unique) index named idx_dept on the department column.

Example 2 : Creating Normal Index on multiple columns.

Query :

```
CREATE INDEX idx_dept_salary
ON employees(department, salary);
```

Explanation :

Creates a normal index named idx_dept_salary on the combination of department and salary columns.

Key Points :

- ✓ Normal index improves query performance.
- ✓ It allows duplicate values.
- ✓ Multiple normal indexes can be created on a table.
- ✓ It is stored separately from the actual table data.
- ✓ It is automatically updated when table data changes.



17.6 Composite Index

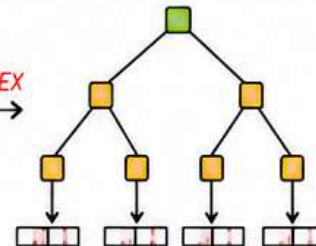
How COMPOSITE Index Works ?

- * A **COMPOSITE INDEX** (or Compound Index) is an index on two or more columns.
- * The **order** of columns in a composite index is **important**.
- * MySQL can use the **leftmost prefix** of the index. For example, if an index is on (A, B, C), it can be used for (A), (A, B) and (A, B, C) searches.
- * It improves performance when queries filter or sort on **multiple columns**.

Table

Index Structure

COMPOSITE INDEX
Index Created



Uses combination of columns to locate rows efficiently.

Example Table : employees

emp_id	name	department	salary
1	Alice	Sales	60000
2	Bob	HR	55000
3	Cathy	IT	70000
4	David	Sales	65000
5	Eva	HR	50000
6	Frank	IT	62000

Example 1 : Creating Composite Index on (department, salary).

Query :

```
CREATE INDEX idx_dept_salary
ON employees(department, salary);
```

Explanation :

Creates a composite index on the combination of department and salary.

Example 2 : Creating Composite Index on (department, name, salary).

Query :

```
CREATE INDEX idx_dept_name_salary
ON employees(department, name, salary);
```

Explanation :

Creates a composite index on department, name and salary columns.

Key Points :

- ✓ Composite index is created on two or more columns.
- ✓ Column order is important.
- ✓ MySQL uses leftmost prefix of the index.
- ✓ Reduces number of lookups and improves performance.
- ✓ Useful in WHERE clause, JOIN condition and ORDER BY.



Leftmost Prefix Rule (Example) :

If index is on (A, B, C)

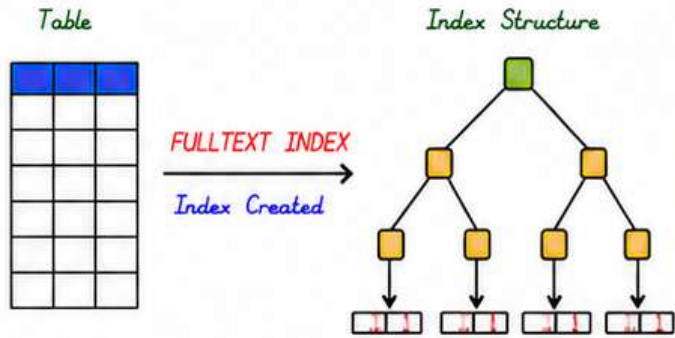
- Can be used for (A)
- Can be used for (A, B)
- Can be used for (A, B, C)
- Cannot be used for (B) or (B, C) only.



17.7 FULLTEXT Index

- * A **FULLTEXT INDEX** is used to perform fast text searches on large text data.
- * It works on **CHAR**, **VARCHAR** and **TEXT** columns.
- * It is used with the **MATCH() ... AGAINST()** clause.
- * It breaks the text into words and creates an index of them.
- * It is very useful for search engines, document search, blog search, etc.

How FULLTEXT Index Works ?



Example 1 : Creating FULLTEXT Index.

Query :

```
CREATE FULLTEXT INDEX idx_article
ON articles(content);
```

Explanation :

Creates a FULLTEXT index named idx_article on the content column.

Example Table : articles

id	title	content
1	MySQL Tutorial	Learn MySQL database indexing.
2	Indexing in DB	FULLTEXT index helps in text search.
3	Search Engines	Search engines use inverted index.
4	Database Guide	Understand indexes in database.

Example 2 : Searching using FULLTEXT Index.

Query :

```
SELECT * FROM articles
WHERE MATCH(content)
AGAINST('database index');
```

Explanation :

Searches for articles containing the words 'database' and 'index' in the content column.

Key Points :

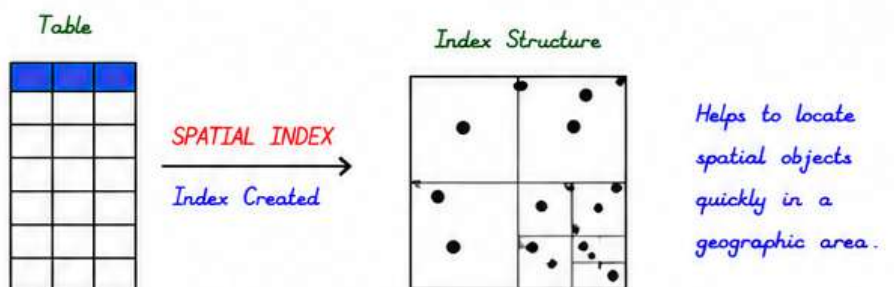
- ✓ FULLTEXT index is designed for text-based searches.
- ✓ It is case-insensitive.
- ✓ Stopwords (common words) are usually ignored.
- ✓ Improves performance of MATCH() ... AGAINST() queries.



17.8 SPATIAL Index

- * A **SPATIAL INDEX** is used for indexing spatial data types such as **POINT**, **LINESTRING**, **POLYGON**, etc.
- * It is used to perform fast spatial searches like "find all locations within a region".
- * It works with data types like **GEOMETRY**, **POINT**, **LINESTRING**, **POLYGON**, **MULTIPOINT**, etc.
- * It is useful in GIS applications, maps, tracking systems, location-based services, etc.

How SPATIAL Index Works ?



Example 1 : Creating SPATIAL Index.

Query :

```
CREATE SPATIAL INDEX idx_location
ON places(location);
```

Explanation :

Creates a SPATIAL index named idx_location on the location column.

Example Table : places

id	name	location (GEOMETRY)
1	Hyderabad	POINT(17.3850 78.4867)
2	Bangalore	POINT(12.9716 77.5946)
3	Mumbai	POINT(19.0760 72.8777)
4	Chennai	POINT(13.0827 80.2707)

Example 2 : Searching using SPATIAL Index.

Query :

```
SELECT * FROM places
WHERE location ST_Within(
GeomFromText('POLYGON((0 0,
10 0, 10 10, 0 10, 0 0))'));
```

Explanation :

Finds all places whose location lies within the given polygon area.

Key Points :

- ✓ SPATIAL index is used for geographic / spatial data.
- ✓ It works with data types like POINT, LINESTRING, POLYGON, etc.
- ✓ Speeds up spatial operations like ST_Within, ST_Distance, ST_Intersects, etc.
- ✓ Important for GIS and location-based applications.



17.9 Dropping Indexes

- * Sometimes an index is no longer needed.
- * Dropping an index **free's up space** and **reduces overhead** on **INSERT, UPDATE** and **DELETE** operations.
- * Use **DROP INDEX** statement to remove an index from a table.
- * It does not affect the data in the table, only the index structure.

Example Table : students

student_id	name	email	department
101	Alice	alice@gmail.com	CSE
102	Bob	bob@gmail.com	ECE
103	Cathy	cathy@gmail.com	IT
104	David	david@gmail.com	CSE

Why Drop Index ?

- ✓ Index is no longer required.
- ✓ Index is not used by queries.
- ✓ Too many indexes slow down write operations.
- ✓ To save storage space.

Example 1 : Dropping an index.

Query :

```
DROP INDEX idx_email  
ON students;
```

Explanation :

Drops the index named `idx_email` from the `students` table.



Example 2 : Dropping an index created on multiple columns.

Query :

```
DROP INDEX idx_name_dept  
ON students;
```

Explanation :

Drops the composite index `idx_name_dept` from the `students` table.

Key Points :

- ✓ `DROP INDEX` removes the index structure, not the table data.
- ✓ After dropping an index, related queries may run slower.
- ✓ Always test before dropping an index in production.
- ✓ If the index is used in constraints (`PRIMARY/UNIQUE`), you cannot drop it directly.



17.10 Index Selection

- * Creating every possible index is not a good idea.
- * Too many indexes **increase storage** and **slow down write operations**.
- * Choose indexes wisely based on query patterns and performance needs.
- * The goal is to improve read performance without degrading write performance.

Example Table : orders

order_id	customer_id	order_date	status	amount
1	101	2024-01-10	Shipped	1500
2	102	2024-01-11	Pending	800
3	101	2024-01-12	Shipped	1200
4	103	2024-01-13	Cancelled	600

Factors to Consider

- ✓ Columns used in `WHERE` clause.
- ✓ Columns used in `JOIN` conditions.
- ✓ Columns used in `ORDER BY` or `GROUP BY`.
- ✓ Columns used frequently in queries.
- ✓ Selectivity of the column (highly selective columns are better).
- ✓ Avoid indexes on columns with very low selectivity.
- ✓ Consider the impact on `INSERT, UPDATE, DELETE`.



Which columns should have index ?

- `customer_id` → used in `WHERE` and `JOIN`.
- `order_date` → used in range queries and reporting.
- `status` → used in filtering.
- `amount` → maybe, if used in range filters.



Good Indexing Practices

- ✓ Create index on columns used in `WHERE`, `JOIN`, `ORDER BY` and `GROUP BY`.
- ✓ Use composite index for multiple column searches.
- ✓ Prefer indexes on columns with high cardinality.
- ✓ Avoid indexing columns with low selectivity (like `gender`, `status` with few values).
- ✓ Monitor and remove unused indexes.
- ✓ Always test query performance.



Bad Indexing Practices

- * Creating too many indexes.
- * Indexing columns that are rarely used.
- * Indexing columns with very low selectivity.
- * Ignoring the impact on write operations.
- * Not reviewing and cleaning unused indexes.



Summary

- Indexing improves `SELECT` performance but may slow down `INSERT, UPDATE` and `DELETE`.
- Choose indexes based on actual query patterns and business needs.
- Regularly review and optimize indexes for best performance.

