Newton School of Technology

17

Indexing & Query Optimisation

By -

C101: DBMS

Quick Recap

Newton School of Technology

So far we have learnt:

- Transaction
- ACID properties

Dataset

Data Retrieval in Large Databases

Newton School of Technology

Imagine you want to find a piece of information from a large database. How would you retrieve?

Data Retrieval in Large Databases

Newton School of Technology

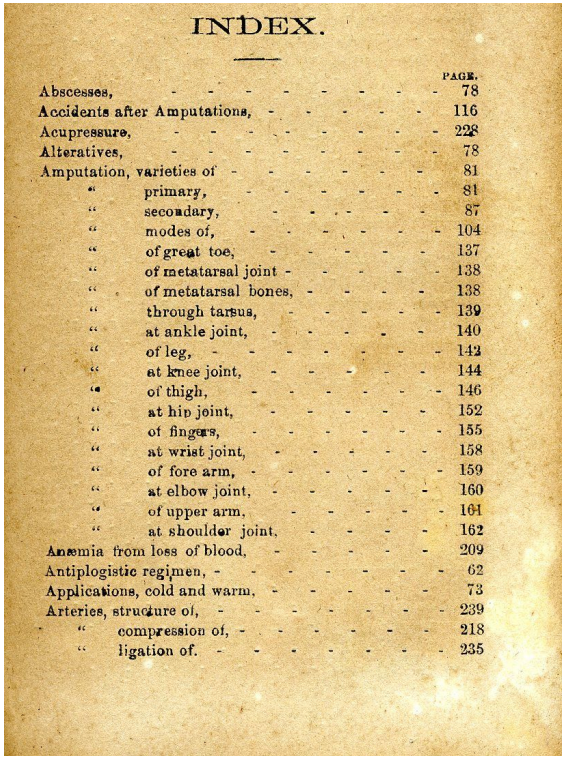
If the table was ordered alphabetically, searching for a name could happen a lot faster because we could skip looking for the data in certain rows.

Efficient Data Retrieval

Newton School of Technology

Indexing

An index in SQL is similar to an index in a book. It helps the database find the exact rows that match your query quickly, without scanning the entire table.



INDEX.

Use Case of Indexing

Newton School of Technology

Index

First_name	Pointer
Amit	104
Mira	102
Mohit	105
Rahul	101
Rajat	103

Table

Customer_id	First_name	Memory address
1	Rahul	101
2	Mira	102
3	Rajat	103
4	Amit	104
5	Mohit	105

Use Case of Indexing

Newton School of Technology

Before creating an index
Query:-
EXPLAIN SELECT * FROM customers WHERE first_name='priya';

Output:-
id|select_type|table|partitions|type|possible_keys|key|key_len|ref|rows|filtered|Extra
1|SIMPLE|customers|NULL|ALL|NULL|NULL|NULL|NULL|7|14.29|Using where

Use Case of Indexing

Newton School of Technology

After creating an index
Query:-
CREATE INDEX a ON customers (first_name);

Explain select * from customers
where first_name="Priya";

Output:-
id|select_type|table|partitions|type|possible_keys|key|key_len|ref|rows|filtered|Extra

1|SIMPLE|customers|NULL|ref|a|a|203|const|1|100.00|NULL

Explain command

Newton School of Technology

Explain select * from customers
where first_name="Priya"

Explain

Newton School of Technology

The EXPLAIN keyword provides detailed information about how a query will be executed. It is used to obtain the following information about the query execution plan:

id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows

- The ID of the query
- The type of your SELECT (if you are running a SELECT)
- The table on which your query was running
- Partitions accessed by your query
- Types of JOINs or scan used
- Indexes from which MySQL could choose
- Indexes MySQL actually used
- The length of the index chosen by MySQL i.e no. of bytes used
- The number of rows accessed by the query

Internal Implementation

Types of Indexing

Clustered Index

Newton School of Technology

A clustered index is a type of database index where the physical order of rows in a table matches the order of the indexed column.

Clustered Index

Newton School of Technology

Clustered index organizes the actual rows of data on disk based on a single column (like a genre label). It’s the physical arrangement of data, so only one clustered index is allowed per table.

Clustered Index

Newton School of Technology

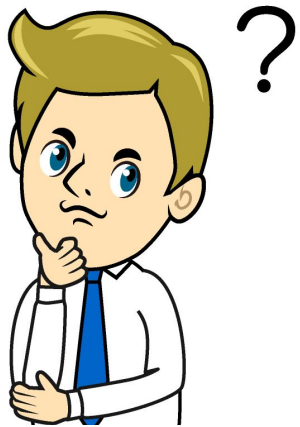
-- Creating a clustered index in MySQL

Each table can have only one clustered index, and by default, DB (MySQL’s default storage engine) creates a clustered index on the primary key.

Clustered Index

Newton School of Technology

But how do we check that index has been created on primary key?



Clustered Index

Newton School of Technology

By using INFORMATION_SCHEMA.STATISTICS table.

`INFORMATION_SCHEMA.STATISTICS` is a system table that provides details about indexes on all tables in the database. It includes important metadata for each index, such as its name, type, associated columns, and how the index is organized.

Clustered Index

Newton School of Technology

Query:-

```
SELECT INDEX_SCHEMA, INDEX_NAME, COLUMN_NAME, INDEX_TYPE, IS_VISIBLE
FROM information_schema.statistics
WHERE TABLE_NAME = 'customers';
```

Output:-

```
INDEX_SCHEMA|INDEX_NAME|COLUMN_NAME|INDEX_TYPE|IS_VISIBLE
test_db|PRIMARY|customer_id|BTREE|YES
```

Multiple Indexing

Newton School of Technology

Non-Clustered Index

Newton School of Technology

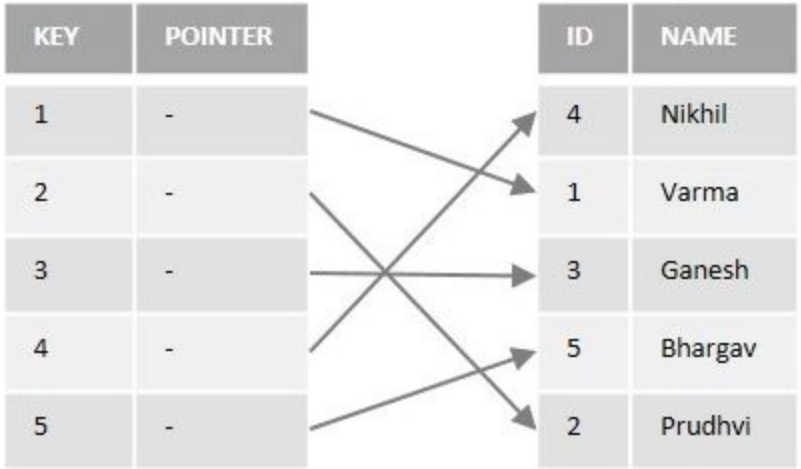
A non-clustered index creates a separate data structure (usually a B-Tree or B+ Tree) that references the actual data stored in the table.

A table can have many non-clustered indexes, providing flexibility for multiple search patterns.

Non-Clustered Index

Newton School of Technology

A non-clustered index is separate from the data; it's like a list that has pointers (addresses) to data rows.



KEY	POINTER
1	-
2	-
3	-
4	-
5	-

Non-Clustered Index

ID	NAME
4	Nikhil
1	Varma
3	Ganesh
5	Bhargav
2	Prudhvi

Table

Creating Non Clustered Index

Newton School of Technology

-- Syntax for creating a non-clustered index

```
CREATE INDEX <index_name>
ON <table_name> (column1, column2, ...);
```

-- Creating a non-clustered index on first_name,state column

```
CREATE INDEX cfs
ON customers (first_name,state);
```

Creating Non Clustered Index

Newton School of Technology

-- Checking the index columns

```
SELECT INDEX_SCHEMA, INDEX_NAME, COLUMN_NAME, INDEX_TYPE, IS_VISIBLE
FROM information_schema.statistics
WHERE TABLE_NAME = 'customers';
```

OUTPUT:

INDEX_SCHEMA	INDEX_NAME	COLUMN_NAME	INDEX_TYPE	IS_VISIBLE
test_db	cfs	first_name	BTREE	YES
test_db	cfs	state	BTREE	YES
test_db	PRIMARY	customer_id	BTREE	YES

Dropping Non Clustered Index

Newton School of Technology

--Syntax for dropping a non-clustered index

```
DROP INDEX index_name
ON table_name;
```

Example:

```
DROP INDEX cfs
ON customers;
```

Advantages of Indexing

Newton School of Technology

- Improved Query Performance
- Efficient Data Access
- Optimized Data Sorting
- Consistent Data Performance
- Data Integrity

Disadvantages of Indexing

Newton School of Technology

- Increased Storage Space:
- Increased Maintenance Overhead:
- Slower Insert/Update Operations:
- Complexity in Choosing the Right Index

Clustered vs Non-Clustered Index

Newton School of Technology

Clustered	Non-clustered
Stores actual data rows in the index.	Separate structure from the actual data rows.
One per table because it changes the physical order of the data.	A table can have many non-clustered indexes.
Clustered indexes organize the actual data in the index and are great for range queries.	Non-clustered indexes provide flexible, multiple access paths without altering the physical order of data.
Example:- E-commerce platform : Product records physically stored by ProductID in a clustered index enabling fast range queries (e.g., products priced between $100 and $500).	Example:- Inventory system : Frequently querying product records by Category, but the data is stored by ProductID. A non-clustered index using a B-Tree makes category lookups fast.

Why query Optimization?

Newton School of Technology

- Enhance performance
- Reduce execution time
- Enhance the efficiency

Best Practices #1

Newton School of Technology

Suppose your task is to retrieve just the product name and category. Which query among the following two is optimized and why?

```
SELECT * FROM products;
```

```
SELECT product_name, category FROM products;
```

Best Practices #1

Newton School of Technology

```
SELECT * FROM products;
```

→ Retrieves all columns, leading to inefficiency with large tables (time/resources).

```
SELECT product_name, category FROM products;
```

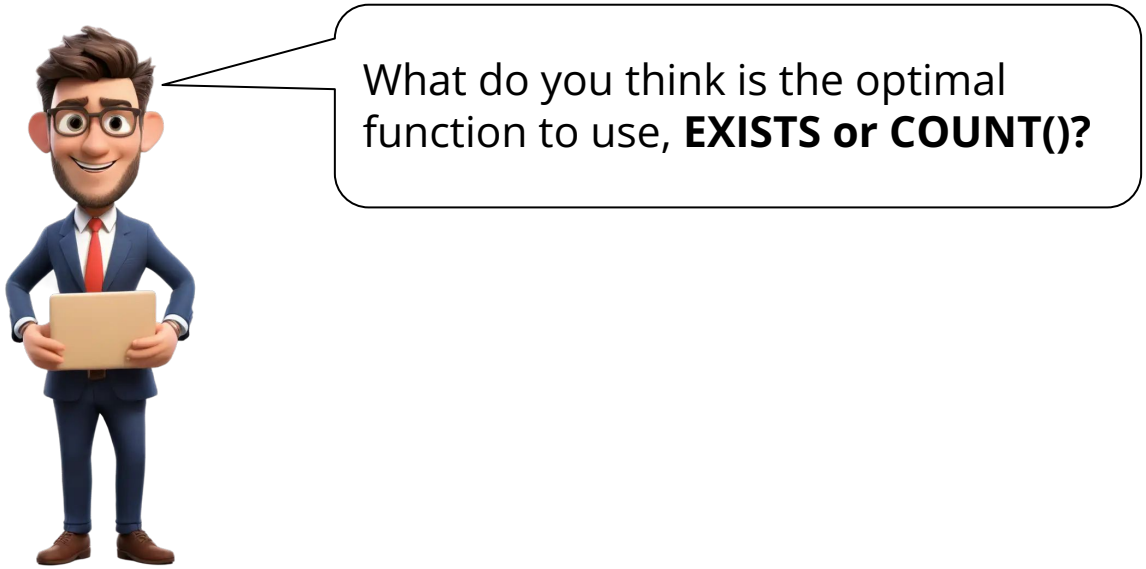
→ Optimized by selecting only needed columns, resulting in faster retrieval and reduced resource use.

It’s better to select as few columns as possible.

Best Practices #2

Newton School of Technology

Suppose you want to check if any customer has placed an order in the last 30 days.



Best Practices #2

Newton School of Technology

COUNT():

- Counts all orders placed by each customer in the last 30 days.
- Can be slow with large datasets.

EXISTS:

- Checks if a customer placed any orders in the last 30 days.
- It's faster because it stops once it finds one order.

The best practice is to use EXISTS instead of COUNT() when you only need to know if a record exists. Optimise for the average case.

Best Practices #4

Newton School of Technology

LIMIT is a trap:

- LIMIT speeds up performance, but doesn’t reduce costs.
- The row restriction of the LIMIT clause is applied after SQL databases scan the full range of data.

Best Practices

Newton School of Technology

Suppose you are at an e-commerce company and need to find products ordered by more than 100 customers.

Which approach is more optimized: using **IN or EXISTS?**

Best Practices

Newton School of Technology

Query 1

```
SELECT product_id FROM products
WHERE product_id IN (
    SELECT product_id FROM orders
    GROUP BY product_id
    HAVING COUNT(DISTINCT customer_id) > 100
);
```

Query 2

```
SELECT product_id FROM products p
WHERE EXISTS (
    SELECT 1 FROM orders o
    WHERE o.product_id = p.product_id
    GROUP BY o.product_id
    HAVING COUNT(DISTINCT o.customer_id) > 100
);
```

Best Practices

Newton School of Technology

IN Method:

- Builds a list of product_ids before filtering, using more memory.
- Less efficient on large datasets as it loads all matches before processing.

EXISTS Method:

- Directly checks for matching rows without a list, saving memory.
- More efficient, especially on large datasets, as it stops at the first match.

Use EXISTS for better efficiency on large datasets as it stops at the first match. Again, this is average case scenario. No changes would occur for worst case scenarios.

Best Practices #3

Newton School of Technology

To efficiently retrieve the row count in a MySQL database, which version of **COUNT** is best?

```
SELECT COUNT(*) FROM table_name;
```

```
SELECT COUNT(1) FROM table_name;
```

```
SELECT COUNT(column_name) FROM table_name;
```

```
SELECT COUNT(DISTINCT column_name) FROM table_name;
```

Best Practices

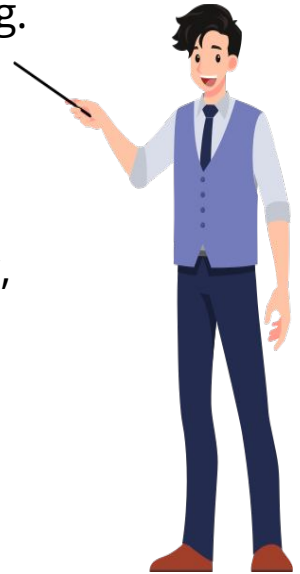
Newton School of Technology

SELECT **COUNT(*)** FROM table_name;

- Counts all rows, including NULLs.
- Efficient and generally recommended for counting all rows in a table, as it's handled at the database engine level without extra processing.

SELECT **COUNT(1)** FROM table_name;

- Similar in function to COUNT(*).
- Replaces all query results with the value 1, counting each row, including NULLs.
- Often thought to be faster, but performance is usually the same as COUNT(*).



Best Practices

Newton School of Technology

SELECT **COUNT(column_name)** FROM table_name;

- Counts only non-NULL values in the column.
- Useful when you need rows with data in a specific column.

SELECT **COUNT(DISTINCT column_name)** FROM table_name;

- Counts unique, non-NULL values in the column.
- Ideal for distinct counts but may be slower on large datasets due to extra processing.

Best Practices

- Use **COUNT(*)** for counting all rows efficiently, including NULLs.
- Avoid **COUNT(1)** as it offers no performance advantage over COUNT(*).
- Use **COUNT(column_name)** when you need to count non-NULL entries in a specific column.
- Reserve **COUNT(DISTINCT column_name)** for counting unique values, keeping in mind it may be slower on large datasets.

Best Practices #4

Newton School of Technology

Which is more optimal: using just a **SELECT** clause or combining **SELECT with a WHERE clause**?

```
SELECT * FROM orders;
```

```
SELECT order_id, customer_id FROM orders
WHERE order_date >= '2024-01-01';
```

Best Practices

Newton School of Technology

Query 1:
Retrieves all columns and rows, resulting in unnecessary data processing and higher memory usage.

Query 2:

- Filters for relevant columns and rows, enhancing performance.
- Less data transferred, saving bandwidth and time.
- Faster query execution with less processed data.

Filter and limit your data early in the query to optimize performance and resource usage.

Best Practices #11

Newton School of Technology

A retail chain operates multiple warehouses in various cities and needs to identify product quantities and types at each New York warehouse.

Which is more optimized?

–joins larger table to smaller table

```
SELECT w.warehouse_id,
w.location, i.product_id,
i.quantity
FROM warehouses AS w
JOIN inventory AS i
ON w.warehouse_id =
i.warehouse_id
WHERE w.city = 'New York';
```

–joins smaller table to larger table

```
SELECT i.warehouse_id,
i.product_id, i.quantity, w.location
FROM inventory AS i
JOIN warehouses AS w
ON i.warehouse_id =
w.warehouse_id
WHERE w.city = 'New York';
```

Best Practices #11

Newton School of Technology

Query 1

```
SELECT w.warehouse_id, w.location, i.product_id, i.quantity
FROM warehouses AS w
JOIN inventory AS i
ON w.warehouse_id = i.warehouse_id
WHERE w.city = 'New York';
```

- Joins the **warehouses table (larger) with the inventory table (smaller)**, leading to potentially slower performance.
- Slower due to early joining on the smaller table.

Best Practices #11

Newton School of Technology

Improved Query:

```
SELECT i.warehouse_id, i.product_id, i.quantity, w.location
FROM inventory AS i
JOIN warehouses AS w
ON i.warehouse_id = w.warehouse_id
WHERE w.city = 'New York';
```

- Starting with the larger inventory table reduces rows earlier, optimizing performance.
- Faster due to early reduction in data.

Ordering joins from larger to smaller tables improves query efficiency for large datasets.

Best Practices #12

Newton School of Technology

A company wants to filter customer transactions by specific customer ID and transaction details.

Does WHERE sequence matters?

```
SELECT customer_id, transaction_date
FROM transactions
WHERE
customer_name LIKE '%Smith%'
AND transaction_type LIKE '%Online%'
AND customer_id = '123456789';
```

```
SELECT customer_id, transaction_date
FROM transactions
WHERE
customer_id = '123456789'
AND customer_name LIKE '%Smith%'
AND transaction_type LIKE '%Online%';
```

Best Practices #12

Newton School of Technology

Query 1

```
SELECT customer_id, transaction_date
FROM transactions
WHERE
customer_name LIKE '%Smith%'
AND transaction_type LIKE '%Online%'
AND customer_id = '123456789';
```

Placing LIKE filters before customer_id may initially check more rows, slowing performance.

Best Practices #12

Newton School of Technology

Query 2:

```
SELECT customer_id, transaction_date
FROM transactions
WHERE
customer_id = '123456789'
AND customer_name LIKE '%Smith%'
AND transaction_type LIKE '%Online%';
```

Placing customer_id first reduces rows early, making execution faster

Best Practices #12

Although MySQL's optimizer manages condition order, placing the most selective condition (e.g., customer_id) first can help with large datasets.

Best Practices #13

Newton School of Technology

Should We Push ORDER BY to the End of the Query?

Best Practices #13

Newton School of Technology

Query 1:

```
WITH recent_sales AS (
    SELECT * FROM sales
    WHERE sale_date BETWEEN '2023-01-01' AND '2023-01-31'
    ORDER BY transaction_id, sale_amount
),
customer_details AS (
    SELECT * FROM customers
    WHERE status = 'active' ORDER BY customer_id
)
SELECT s.transaction_id, s.sale_amount, c.customer_name FROM recent_sales s
JOIN customer_details c ON s.customer_id = c.customer_id
ORDER BY sale_amount;
```

Best Practices #13

Newton School of Technology

Query 2:

```
WITH recent_sales AS (
    SELECT * FROM sales
    WHERE sale_date BETWEEN '2023-01-01' AND '2023-01-31'
),
customer_details AS (
    SELECT * FROM customers WHERE status = 'active'
)
SELECT s.transaction_id, s.sale_amount, c.customer_name
FROM recent_sales s
JOIN customer_details c ON s.customer_id = c.customer_id
ORDER BY sale_amount;
```

Best Practices #13

Newton School of Technology

Query 1:
Slower due to sorting in CTEs before filtering and joining.

Query 2:
Faster as sorting happens only after filtering and joining.

Sorting after filtering and joining is more efficient, with fewer rows to sort.

Best Practices

Should we avoid Subqueries?

Query 1:

```
SELECT *
FROM employees
WHERE department_id IN (
    SELECT id
    FROM departments
    WHERE name = 'Sales'
);
```

Query 2:

```
SELECT e.*
FROM employees e
JOIN departments d ON
e.department_id = d.id
WHERE d.name = 'Sales';
```

Best Practices

Newton School of Technology

Query 1:

```
SELECT *
FROM employees
WHERE department_id IN (
    SELECT id
    FROM departments
    WHERE name = 'Sales'
);
```

Slower due to the inner query execution.

Best Practices

Newton School of Technology

Query 2:

```
SELECT e.*
FROM employees e
JOIN departments d ON e.department_id = d.id
WHERE d.name = 'Sales';
```

Faster, as the join avoids extra nesting.

Best Practices

- Use joins or temporary tables instead of subqueries.
- Joins often enhance performance by streamlining the query structure.

Best Practices

Newton School of Technology

Should we prefer UNION ALL over UNION?

Best Practices

Newton School of Technology

Query 1:

```
SELECT name
FROM employees
WHERE status = 'active'
UNION
SELECT name
FROM contractors
WHERE status = 'active';
```

Slower because it sorts data to remove duplicates.

Best Practices

Newton School of Technology

Query 2:

```
SELECT name
FROM employees
WHERE status = 'active'
UNION ALL
SELECT name
FROM contractors
WHERE status = 'active';
```

Faster because it skips sorting to check for duplicates.

Best Practices #15

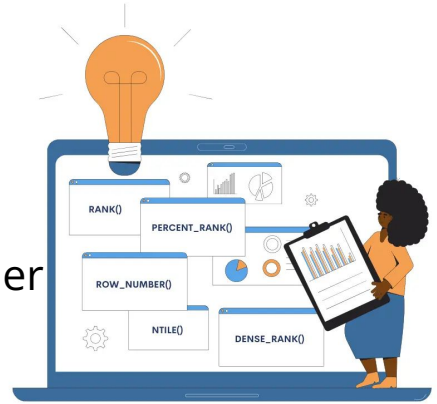
Newton School of Technology

- Use UNION ALL if duplicate removal isn’t needed.
- UNION ALL is faster as it skips sorting to eliminate duplicates.

Newton School of Technology

Summary

- Select only needed columns to reduce data load.
- Efficient WHERE clause to filter data early.
- Limit rows with LIMIT for large datasets.
- Use UNION ALL Instead of UNION if duplicates aren't needed.
- Use joins instead of subqueries when possible, and order joins from largest to smallest tables.



Summary

Newton School of Technology

So far we have learnt:

- Indexing
- Clustered Index
- Non Clustered Index
- Ways to Optimize queries

Newton School of Technology

Any Questions

Newton School of Technology

Please fill the feedback form.

Newton School of Technology

Thanks for watching!