Exploring Database Indexes: Types, Implementations, and Performance Analysis

1. Introduction

If every system and application in the world only had to deal with hundreds or thousands of data, then we would not have to stress our brains with indexes. However, this is not the case. Facebook alone has billions of rows of data which makes us wonder, how does it retrieve our Facebook posts, likes, comments, photos, and videos in seconds? Of course, there are multiple technologies involved in such massive applications but one core component of these are indexes. Although there are several types of indexes like B-tree indexes, Hash indexes, full-text indexes, and a lot more, let's first focus on non-clustered indexes. A non-clustered index creates another column having references to the actual data in a table's actual column. This index sorts these references and because the data is sorted, we can perform a binary search on these data efficiently with a time complexity of just O (log n).

Let's take a deeper dive into this. Suppose we have a table of Google's employees with their respective positions and salaries.

Employee Table

Employee ID	Name	Position	Salary
1	Mary Therese Colina	Software Engineer	154000
2	Marc Lennard Colina	Software Engineer	145000
3	Rose Mabelle Seares	Accounting Manager	144000
4	Ken Angelo Ardiente	Android Developer	148000
5	Rey Seno	Product Owner	143000
6	Sean Ouano	UI Designer	149000
7	Matthew Cosico	Database Developer	146000

If we had to retrieve all employees with a salary greater than $145,000, we would have to go through each row and select all those that match the criteria.

If we create an index for the Salary column, we'd have a table of references that are sorted, and the actual searching happens here instead of the actual table. The DB

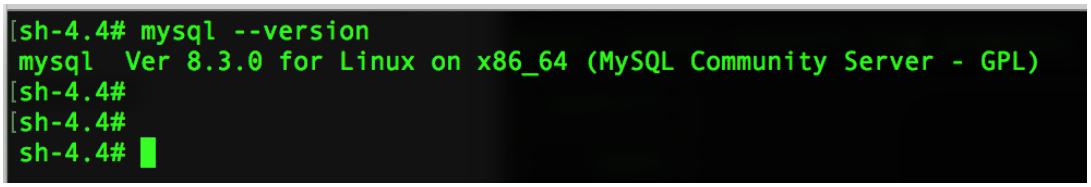
engine performs a binary search starting from the middle (Row 4). It will then continue searching down because it knows all the rows that fit the criteria won't be found by searching above since the index is sorted. Through this the number of rows we must visit drops significantly less thus improving performance.

Salary	Pointer
143000	Pointer to Row 5
144000	Pointer to Row 3
145000	Pointer to Row 2
146000	Pointer to Row 7
148000	Pointer to Row 4
149000	Pointer to Row 6
154000	Pointer to Row 1

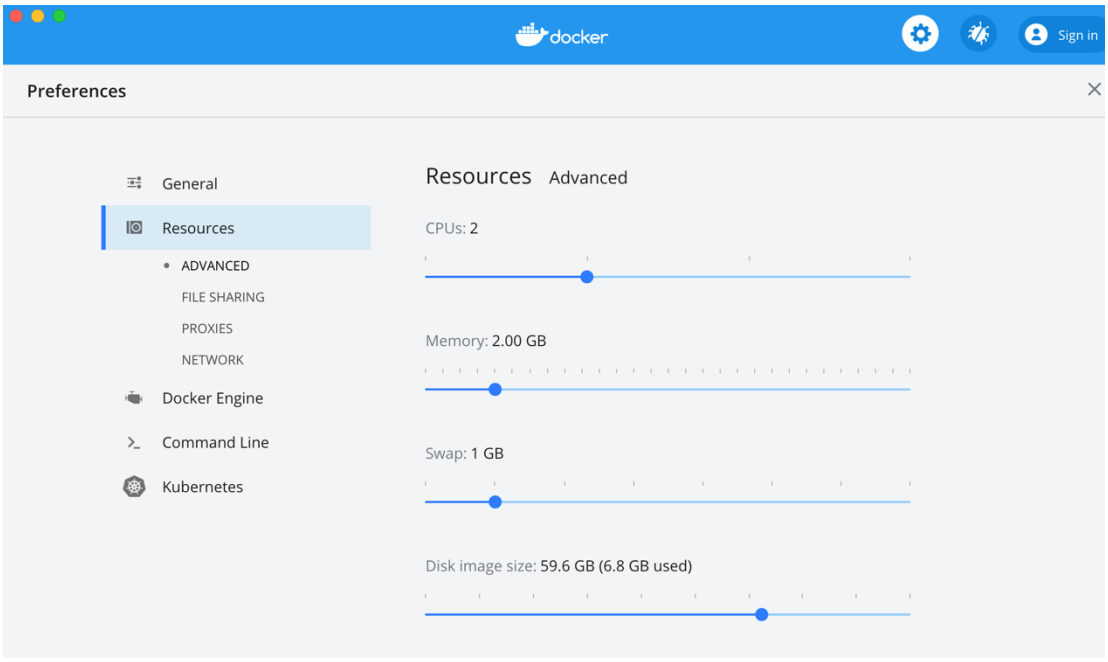
In transactional DB operations used in banks for example, multiple users could be accessing our system and through indexes, we are able to cater to these requests without service degradation. In analytical database operations, where we mostly work on large datasets, we can feel the significance of indexes because it does retrieval operations at a fraction of the time compared to doing a sequential retrieval.

2. Methodology

- Environment Setup:
    - MySQL specifications

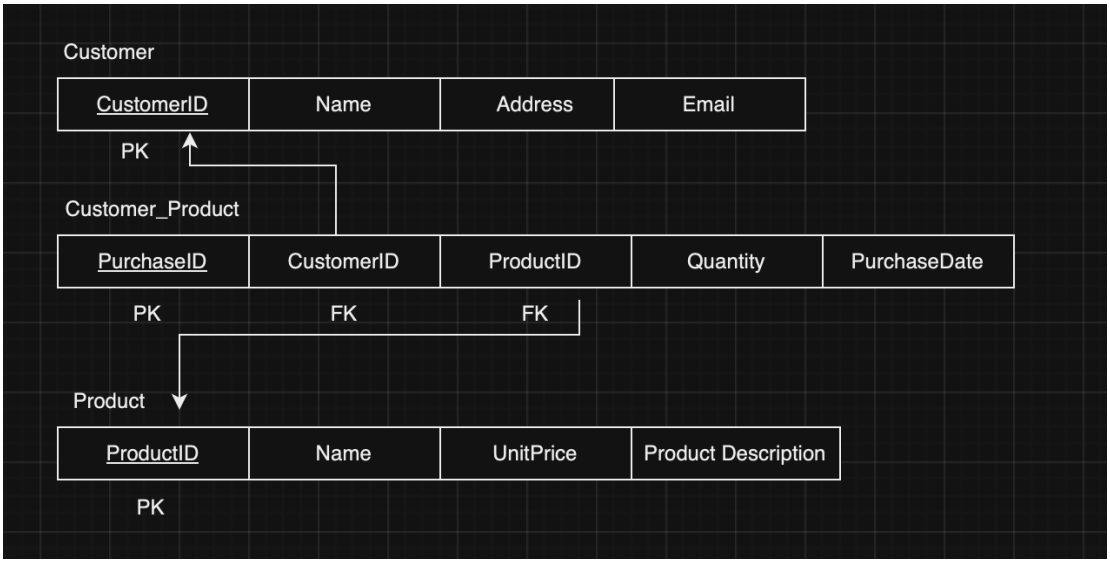


- Virtual environment specifications



Note: ***Docker Engine****, v19.03.13-beta2*

- Data Modeling:
  - DB Schema Design – The tables were designed to only contain the key columns because these were enough to work with the experiment on indexes.



- Data Generation:

A stored procedure was used to generate 100,000 customers, 100,000 products and 100,000 purchases.

```
--Stored procedure to insert customers
BEGIN
  -- Declare variables
  DECLARE i INT DEFAULT 1;
  -- Start the loop
  WHILE i < 100000 DO
    INSERT INTO customers (customerID,name, email)
    VALUES(i,CONCAT('Customer', i), CONCAT('Email',i,'@gmail.com'));
    SET i = i + 1;
  END WHILE;
END
```

```
--Store procedure to insert products
BEGIN
  -- Declare variables
  DECLARE i INT DEFAULT 1;
  -- Start the loop
  WHILE i < 100000 DO
    INSERT INTO products (productID,name, price,description)
    VALUES(i,CONCAT('Product', i), i*2.5, CONCAT('Description',i));
    SET i = i + 1;
END WHILE;
END
```

```
--Store procedure to insert purchase
BEGIN
  -- Declare variables
  DECLARE i INT DEFAULT 1;
  -- Start the loop
  WHILE i < 100000 DO
    INSERT INTO purchases (purchaseID,customerID,productID, quantity,purchaseDate)
    VALUES(i, i, i, 3,'2024-10-10');
    SET i = i + 1;
END WHILE;
END
```

3. Practical Implementation

- Index Creation:
  - SQL Commands

```
-- Indexes on Purchase table
CREATE INDEX idx_purchase_customer ON Purchase(CustomerID);
CREATE INDEX idx_purchase_product ON Purchase(ProductID);

-- Indexes on Customer table
CREATE INDEX idx_customer_name ON Customer(Name);

-- Indexes on Product table
CREATE INDEX idx_product_name ON Product(Name);
```

Note:
The indexes created were for the columns added in the WHERE clause of the queries used in the experiment. These were also based on real-life scenarios because in an e-commerce app, it's common to search by customers and products. These indexes were also created in addition to the clustered indexes automatically created by the MySQL engine.

- Query Execution:
  - Select Query#1

```
SELECT
  p.purchaseID,
  c.customerID,
  pr.productID,
  p.quantity,
  p.purchaseDate
FROM
  purchases p
    JOIN
  customers c ON p.customerID = c.customerID
    JOIN
  products pr ON p.productID = pr.productID
WHERE c.name='Customer99999';
```

- Select Query#2

```
SELECT
  p.purchaseID,
  c.customerID,
  pr.productID,
  p.quantity,
  p.purchaseDate
FROM
  purchases p
    JOIN
  customers c ON p.customerID = c.customerID
    JOIN
  products pr ON p.productID = pr.productID
WHERE c.name='Customer99999' and pr.name='Product99999';
```

- Select Query#3

```
SELECT
  p.purchaseID,
  c.customerID,
  pr.productID,
  p.quantity,
  p.purchaseDate
FROM
  purchases p
    JOIN
  customers c ON p.customerID = c.customerID
    JOIN
  products pr ON p.productID = pr.productID
WHERE pr.name='Product99999';
```

- Update Query#4

```
UPDATE customers SET name='Alice Bob' WHERE name='Alice';
```

- Delete Query#5

```
DELETE FROM customers WHERE name='Alice Bob';
```

- Insert Query#6

```
INSERT INTO customers(customerID,name,email)
    -> values(100001,'Alice','alice@gmail.com');
```

4. Performance Analysis

- Metrics
  Query Response Times

  - Select Query#1

Without Index

```
+------------+------------+-----------+----------+--------------+
| purchaseID | customerID | productID | quantity | purchaseDate |
+------------+------------+-----------+----------+--------------+
|      99999 |      99999 |     99999 |        3 | 2024-10-10   |
+------------+------------+-----------+----------+--------------+
1 row in set (0.19 sec)
```

With Index on Customer Name

```
+------------+------------+-----------+----------+--------------+
| purchaseID | customerID | productID | quantity | purchaseDate |
+------------+------------+-----------+----------+--------------+
|      99999 |      99999 |     99999 |        3 | 2024-10-10   |
+------------+------------+-----------+----------+--------------+
1 row in set (0.01 sec)
```

  - Select Query#2

Without Index

```
+------------+------------+-----------+----------+--------------+
| purchaseID | customerID | productID | quantity | purchaseDate |
+------------+------------+-----------+----------+--------------+
|      99999 |      99999 |     99999 |        3 | 2024-10-10   |
+------------+------------+-----------+----------+--------------+
1 row in set (0.18 sec)
```

With Index on Customer Name and Product Name

```
+------------+------------+-----------+----------+--------------+
| purchaseID | customerID | productID | quantity | purchaseDate |
+------------+------------+-----------+----------+--------------+
|      99999 |      99999 |     99999 |        3 | 2024-10-10   |
+------------+------------+-----------+----------+--------------+
1 row in set (0.01 sec)
```

- Select Query#3

Without Index

```
+------------+------------+-----------+----------+--------------+
| purchaseID | customerID | productID | quantity | purchaseDate |
+------------+------------+-----------+----------+--------------+
|      99999 |      99999 |     99999 |        3 | 2024-10-10   |
+------------+------------+-----------+----------+--------------+
1 row in set (0.32 sec)
```

With Index on Product Name

```
+------------+------------+-----------+----------+--------------+
| purchaseID | customerID | productID | quantity | purchaseDate |
+------------+------------+-----------+----------+--------------+
|      99999 |      99999 |     99999 |        3 | 2024-10-10   |
+------------+------------+-----------+----------+--------------+
1 row in set (0.06 sec)
```

- Update Query#4

Without Index

```
mysql> UPDATE customers SET name='Alice Bob' WHERE name='Alice';

Query OK, 0 rows affected (0.12 sec)
Rows matched: 0  Changed: 0  Warnings: 0
```

With Index on Customer Name

```
mysql> UPDATE customers SET name='Alice Bob' WHERE name='Alice';
Query OK, 0 rows affected (0.01 sec)
Rows matched: 0  Changed: 0  Warnings: 0
```

- Delete Query#5

Without Index

```
mysql> DELETE FROM customers WHERE name='Alice Bob';

Query OK, 1 row affected (0.10 sec)
```

With Index on Customer Name

```
mysql> DELETE FROM customers WHERE name='Customer4444';
```

```
Query OK, 1 row affected (0.02 sec)
```

- Insert Query#6

Without Index

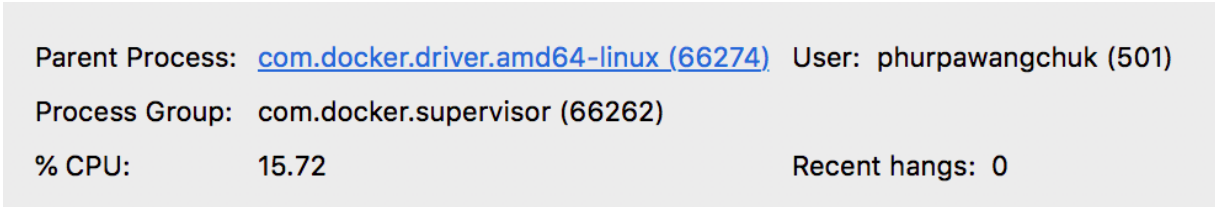
```
mysql> INSERT INTO customers(customerID,name,email)
    -> values(100001,'Alice','alice@gmail.com');
Query OK, 1 row affected (0.01 sec)
```

With Index on Customer Name

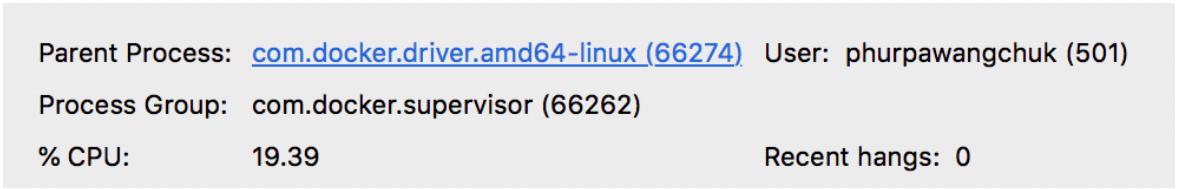
```
mysql> INSERT INTO customers(customerID,name,email)
-> values(100002,'Bob','bob@gmail.com');
Query OK, 1 row affected (0.03 sec)
```

CPU Utilization *(Relatively the same across all queries)*

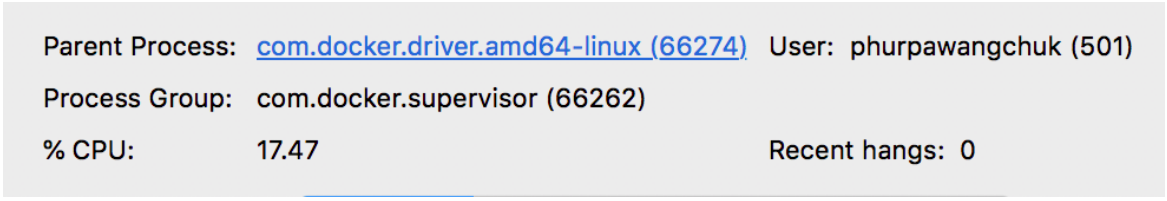
- *Before querying*



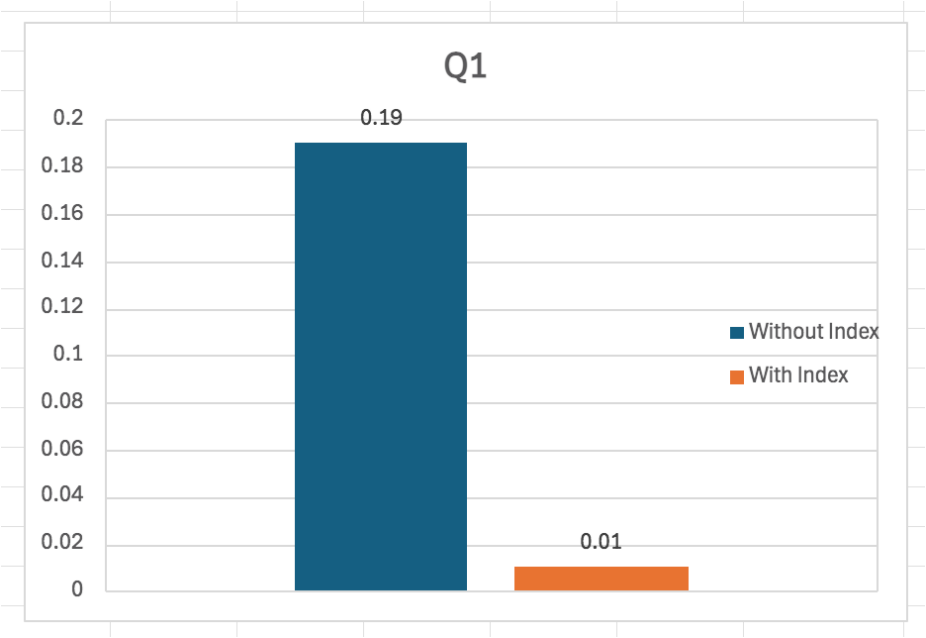
- *After querying without index;*

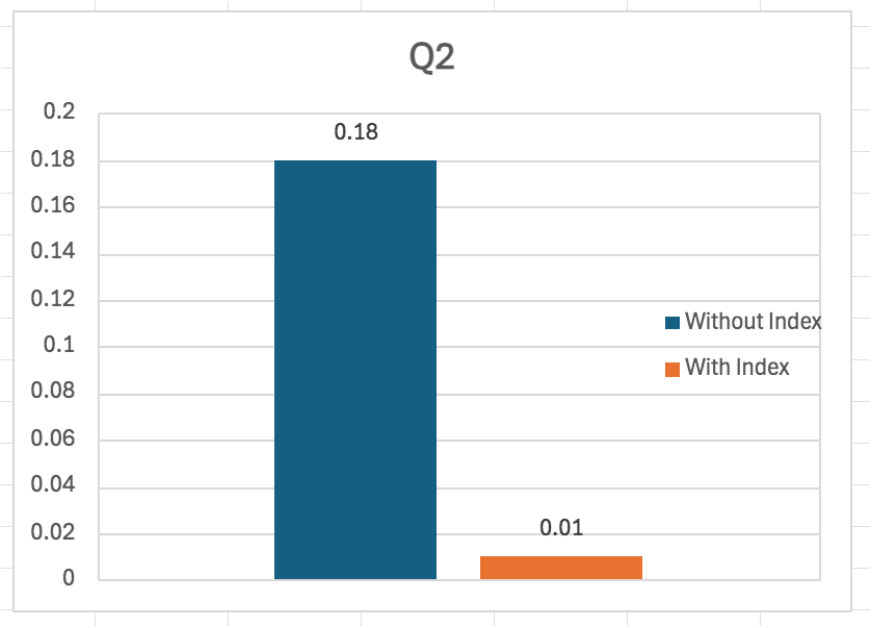


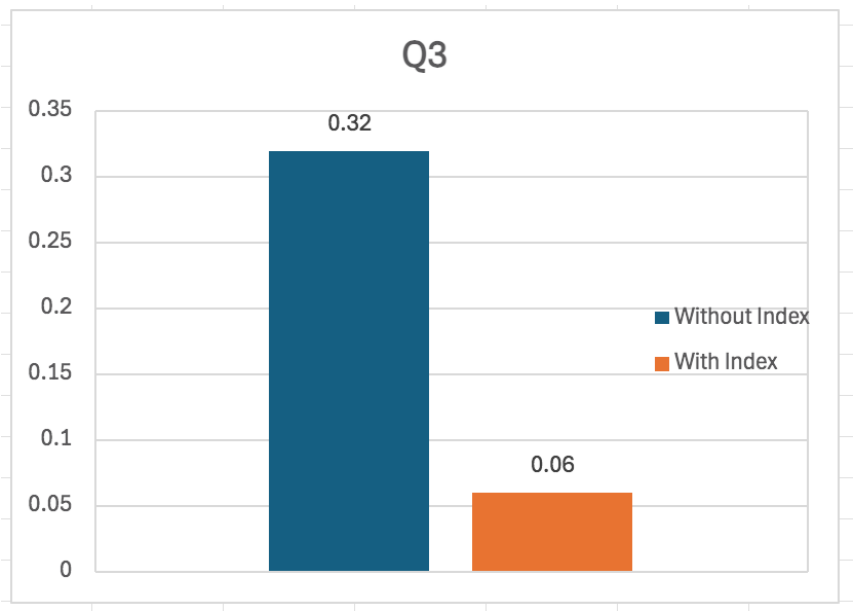
- *After querying with index;*

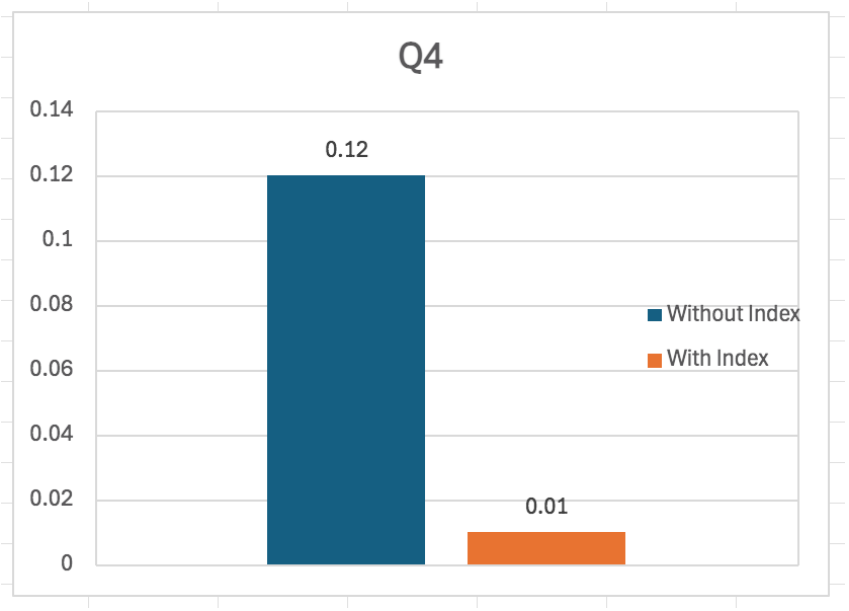


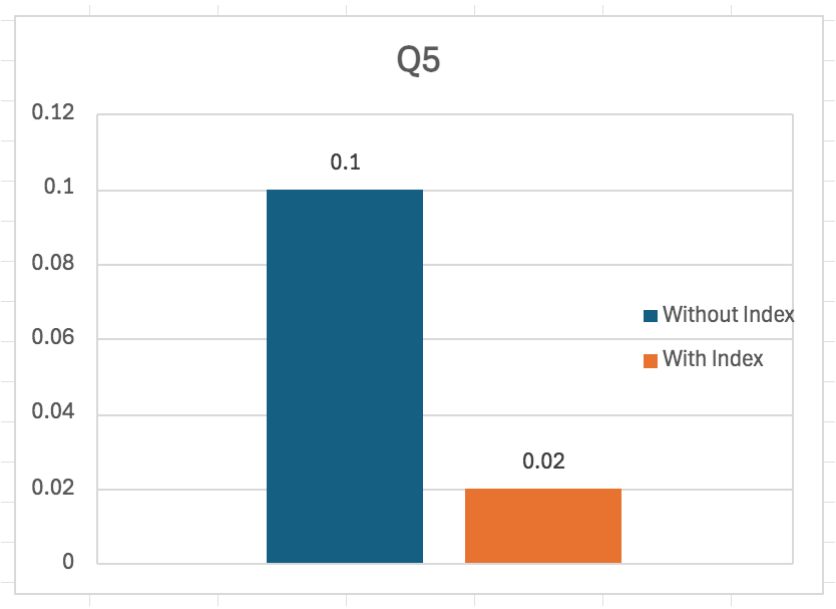
- Comparison and Visualizations:
  These graphs outline the performance differences in each query performed.

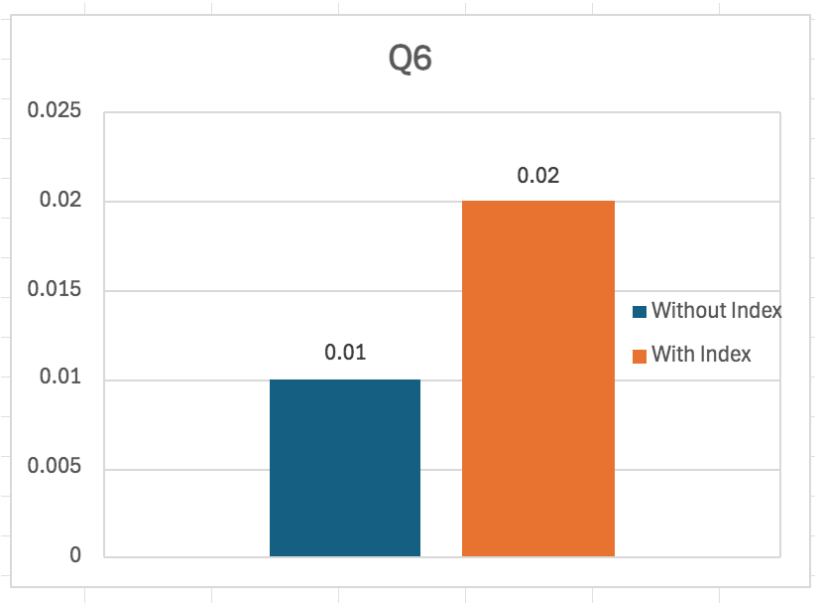





Q3
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0.32
0.06
Without Index
With Index


Q4
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0.12
0.01
Without Index
With Index


Q5
0.12
0.1
0.08
0.06
0.04
0.02
0
0.1
0.02
Without Index
With Index


Q6
0.025
0.02
0.015
0.01
0.005
0
0.01
0.02
Without Index
With Index

5. Discussion

There weren't many anomalies or unexpected results during the experiment. For the most part, the query response times, and CPU utilization results were expected. As outlined in the metrics, there is a significant performance difference when indexes were used in specific columns added to the WHERE clauses. These significant differences were consistent no matter the type of query be it SELECT, UPDATE, or DELETE. The differences might only be in milliseconds but if we scale our data up to the level of Facebook or Instagram, this is already very huge.

However, like everything else in life, Indexes also have their tradeoffs. Let's shift our focus to the purchases table and the purchasesindex table in the image below. Both tables have the same structure and have approximately the same data, but the purchases table only had the default clustered index while the purchaseindex had additional indexes like the one for the Name column. If we take a look at the size, **purchases** table occupies **4.5 MB** while **purchaseindex** occupies **10.1 MB**. This is more than double the size of the table without additional indexes.

	Table	Action							Rows	Type	Collation	Size	Overhead
☐	customers	☆	Browse	Structure	Search	Insert	Empty	Drop	~99,773	InnoDB	utf8mb4_0900_ai_ci	5.5 MiB	-
☐	products	☆	Browse	Structure	Search	Insert	Empty	Drop	~100,072	InnoDB	utf8mb4_0900_ai_ci	5.5 MiB	-
☐	purchases	☆	Browse	Structure	Search	Insert	Empty	Drop	~100,311	InnoDB	utf8mb4_0900_ai_ci	4.5 MiB	-
☐	purchasesIndex	☆	Browse	Structure	Search	Insert	Empty	Drop	~100,076	InnoDB	utf8mb4_0900_ai_ci	10.1 MiB	-
	4 tables	Sum							~400,232	InnoDB	utf8mb4_0900_ai_ci	25.6 MiB	0 B

Another tradeoff we must deal with is the performance overhead when doing INSERT operations on tables with columns that have indexes. If we shift our focus to our Insert Query#6, there is a minimal difference between Insert queries. However, the difference is not that much compared to the benefits we gain when doing SELECT queries.

- Insert Query#6

Without Index

```
mysql> INSERT INTO customers(customerID,name,email)
    -> values(100001,'Alice','alice@gmail.com');
Query OK, 1 row affected (0.01 sec)
```

With Index on Customer Name

```
mysql> INSERT INTO customers(customerID,name,email)
-> values(100002,'Bob','bob@gmail.com');
Query OK, 1 row affected (0.03 sec)
```

6. Conclusion

To clearly see the difference between the SELECT queries with and without indexes, let's scale up our operations to a billion queries. Applying this to the table WITHOUT indexes will take **2.2 days.** While applying this to the table WITH indexes will only take **2.78 hours**.

However, as mentioned in the discussion above, having indexes also has their tradeoffs. But if we take a closer look at the INSERT query, the performance penalty was only 0.02 seconds while the SELECT query#3 made a performance efficiency difference of 0.26 seconds. Based on this, when deciding to put or not put indexes, the tradeoff between performance efficiency in select queries should be given more importance than the overhead costs of insert queries. A DBMS professor named Umur Inan said it best (non-verbatim): *"I have never seen a system where indexes were not created just because the developers were afraid of the Insert cost overhead"*.

The difference we must focus more on is the memory. As mentioned above, the size of the table with indexes doubled from **4.5 MB to 10.1 MB.** This is now where we should carefully discern which we want to prioritize: memory consumption or efficient query performance.

To summarize, indexes have their advantages and disadvantages and when deciding to use it, where to use it, or how to use it would largely **depend on the requirements and specifications** of the system or application we are trying to build.