QUERY EXECUTION PERFORMANCE OF SQL AND NO-SQL DATABASES IN COMPLEX QUERIES

Gowreeshan Sachchithananthan
Department of Computer Engineering
University of Sri Jayewardenepura
Colombo, Srilanka
gowreeshan1@gmail.com

***Abstract*—In today's world, data is a very important thing. Databases are used to store and retrieve data. Performance of database is important for better management of databases. SQL and NoSQL databases are two main types. Each of them has different strengths and weaknesses. Previous studies compared how well these databases handle complex queries. But there is still uncertainty about NoSQL database's performance. This research focuses on comparing SQL and NoSQL databases, including document, graph, key-value, and column databases. An e-commerce system database is implemented on MySQL, MongoDB, Neo4j, Redis, and Cassandra. Complex queries are tested on these databases with and without indexing to evaluate the performance. Key findings reveal that MySQL demonstrates superior overall performance. Neo4j excels in executing indexed queries, particularly for data aggregation based on predefined keys. Redis emerges as an optimal choice for handling small queries with indexing. It leverages its in-memory data storage capabilities. Although MongoDB generally performs well with small queries, challenges arise in indexing efficiency. Cassandra encounters difficulties in single-node setups due to its architecture optimized for distributed deployments. These insights provide valuable guidance for businesses in selecting the most suitable database solution tailored to their specific requirements.**

***Index Terms*—SQL, No-SQL, Complex Queries, Database Performance, Indexing**

Abbreviations and Acronyms

SQL	Structured Query Language
No-SQL	Not Only SQL
GSQL	Graph Structured Query Language
AQL	ArangoDB Query Language
JSON	JavaScript Object Notation
BSON	Binary JSON

I. Introduction

In today's data-driven world, Databases are important for storing and retrieving data . Currently organizations face increasingly complex data structures and query requirements. So, it is very important to know about performance of SQL and No-SQL databases. Traditional SQL databases excel at retrieving structured data. However, as data formats become more varied, there is a need for flexible and scalable alternatives. NoSQL databases offer such alternatives. But their performance in handling complex queries is still unclear. Previous studies have focused on comparing query execution performance in complex queries with and without optimization techniques, primarily for relational databases and graph databases [1]. Other researchers have compared various types of databases. But They limited their analysis to insert, update, delete, and select operations [2]. Furthermore, some studies have compared database loading, response, and retrieval times between SQL databases and document databases [3]. So, This research is focused on evaluating the Query execution performance of SQL and NO-SQL databases (Document database, Graph Database, Key-value database, column database) in executing complex queries with and without indexing. and also provides insights about in handling complex query operations by evaluating query execution performance.

In this research an e-commerce system database is implemented across various database systems, including MySQL for SQL, MongoDB for Document DB, Neo4j for Graph DB, Redis for Key-value DB, and Cassandra for Column DB. Following database creation, complex queries are created to assess the execution performance of both SQL and NoSQL databases. Indexed queries are created and performance analysis is conducted to measure the query execution time of both SQL and NoSQL databases. Finally, Based on the findings obtained from the performance analysis, an evaluation and conclusion are drawn. Through this evaluation, Insights are provided into the comparative performance of different database systems in handling complex queries and indexed queries. The rest of the paper is organized as follows. Section 2 consists of an overview of database systems, indexing and related work about SQL and No-SQL database systems. Section 3 elaborates methodology of database comparison. Section 4 presents the results and discussion. Finally, insights about test results are presented in section5.

II. Background

There are two primary types of databases: SQL and NoSQL.

A. SQL databases

SQL database is a relational database management system. It uses tables to store data. In that table, each row represents an individual record. Each column represents a specific attribute. primary keys are used to uniquely identify each record in a table. Relationships between each table are implemented through foreign keys. These databases support one-to-many and many-to-many relationships.

SQL databases provide ACID (Atomicity, Consistency, Isolation, Durability) compliance. It ensures data integrity and reliability.

- Atomicity : A transaction should be fully performed, or none performed.
- Consistency: If a transaction is made, its value must be preserved before and after the transaction.
- Isolation : Transaction will not be affected by any other concurrent transaction.
- Durability : Once a transaction is completed and confirmed, its changes will continue permanently in the database.

SQL is used to query relational databases. It is used to perform database operations such as creating, updating, and deleting tables. It does not require much training to use it. It is a flexible database. Because Modifying a database using SQL is not much harder. It separates data and forms tables to avoid dependency. It is called normalization. Examples of SQL databases include MySQL, PostgreSQL, SQLite, Oracle, and Microsoft SQL Server.

B. No-SQL databases

NoSQL database uses unstructured or semi-structured data. It offers more flexibility and scalability compared to SQL databases. NoSQL databases are categorized into four types based on their data models and structures:

1) Graph databases: Graph databases are designed to store and manage data as graphs. It consists of nodes, edges, and properties. properties are key-value pairs. Nodes represent entities while Edges represent relationships between nodes. Nodes and edges have properties. Properties provide additional information about entity. Graph database allows nodes and edges to have different properties and relationships. It gives flexibility to adopt changing requirements. Graph databases support relationship-centric querying. It allows users to traverse the graph and navigate relationships between nodes efficiently. Graph traversal algorithms can be used with graph queries.

Different graph databases utilize distinct query languages: Neo4j employs Cypher, Amazon Neptune supports Gremlin and SPARQL, TigerGraph utilizes GSQL, JanusGraph relies on Gremlin, and ArangoDB employs AQL.

2) Document databases: Database data is stored in documents like JSON , BSON, or XML. Each document represents a record. Attributes of the entity are represented within the document using various data structures, including nested structures, arrays, and key-value pairs. In this database, the relationship between collections is mainly represented in two ways. They are reference document and embedded document. In reference document approach, a document in one collection may contain references such as id, to documents in another collection. In embedded document approach, documents may contain embedded sub-documents that represent related data. It increases query performance due to simplified data access. but it results redundancy and need for careful data synchronization to maintain consistency. To get the strengths of each strategy to optimize data access, query performance, and data consistency, hybrid approach is used to model database. In this hybrid approach, certain relationships may be represented using references while others are represented using embedded document. Examples of Document databases include MongoDB, Couchbase, and CouchDB.

3) Key-value databases: Key-value databases are designed to store and manage data using a simple key-value pair. It is a schema-less database. Each data item is represented as a unique key associated with a corresponding value. The value in a key-value pair can be of various data types, including strings, numbers, JSON objects, or binary data. key-value databases do not support relationships between data items. Developers need to implement relationships using strategies such as key references or embedded values. Querying in these databases revolves around key-based lookup operations. Typically this database does not have a standard query language. Examples of Key-value databases include Redis, Amazon DynamoDB and Couchbase.

4) column databases: In this Column database, data is stored in columns rather than rows. Each column represents a single attribute. Values for that attribute are stored contiguously. and rows consist of values corresponding to those attributes. Due to the support of relational structures, it also uses primary keys and composite keys to uniquely identify each record in the table. Foreign keys are also used to establish relationships.

Different databases use different query languages. Many databases support SQL as the primary query language. Examples of column-oriented databases include Apache Cassandra, Apache HBase, ClickHouse, Amazon Redshift, and Google Bigtable.

C. Indexing

Indexing is a technique used to increase performance of data retrieval operations. An index contains keys and pointers to rows or documents in the database. when a query condition matches with the indexed column, database engine retrieves the relevant rows using index without scanning entire database. There are various types of indexes based on their creation. Single-Column Indexes are created on a single column. Composite Indexes, are formed on multiple columns. Unique Indexes guarantee that the values within the indexed columns are unique. They are commonly used to ensure data integrity and prevent duplicate entries.

Different databases use different data structures and techniques for indexing. In MySQL and MongoDB, the predominant indexing method is B-trees, which organize data in a balanced tree structure. Each node contains keys and pointers to child nodes, with only actual data stored in leaf nodes. when a specific row or data needs to be retrieved, database engine follows the pointers down the branches until it reaches that data. This makes searches much faster than checking everything one by one.

Neo4j uses a custom indexing structure optimized for graph data storage and retrieval. That indexing mechanism is not publicly disclosed. Neo4j creates indexes on node properties and relationship properties.

Redis uses a sorted set to create indexes. Sorted sets can be created using the format ’entity name:property value’ as the key, with the relevant entity and entity keys stored as Key and value pairs within the sorted set.

Cassandra uses two types of indexing, primary indexing and secondary indexing. Primary indexing in Cassandra is based on the primary key of a table. The First part of primary key is used as partition key and other parts of primary key are used as culturing columns. The partition key determines the distribution of data across the nodes in the Cassandra cluster. Clustering columns dictate the order of data within each partition. Secondary indexing allows querying on non-primary key columns in a table. Secondary indexes are created based on query requirements.

D. Related work

Various studies have compared the performance of databases. Khan et al. (2023) conducted a comparative analysis of SQL (MySQL) and NoSQL (MongoDB) databases [3]. They tested loading, response, and retrieval times using PHP’s database connectivity. The study showed that SQL databases outperformed NoSQL databases significantly in terms of loading, responding, and retrieval times. It indicates SQL database’s efficiency and speed. Kotiranta et al. (2022) compared the performance of graph and relational databases for complex queries [1]. They used PostgreSQL as a relational database and Neo4j as a graph database. The study found that PostgreSQL outperformed Neo4j in complex and recursive query tasks. However, Neo4j was faster for queries that involved traversing relationships between nodes.

Sholichah et al. (2021) conducted a comparative analysis between MySQL and Neo4j databases. They focused on their performance using datasets and unstructured data [4]. The study utilized four queries, varying the datasets from 10 to 10,000 records. Results indicated that MySQL generally outperformed Neo4j in terms of speed and memory usage. This trend was observed as query complexity and record numbers increased. Stanescu (2021) compared SQL Server 2009 and Neo4j 4.0, using datasets with up to 2.1 million entries and complex query structures [5]. Neo4j outperformed SQL Server as both query and dataset complexity increased. This highlights Neo4j’s efficiency in handling complex relationships and queries, especially in recommendation systems.

Čerešňák and Kvet (2019) compared query performance in relational (MySQL, Oracle, MS SQL) and non-relational databases (MongoDB, Redis, GraphQL, Cassandra) [2]. Initially, the difference in query time was not significant. But after implementing range scan on a key attribute, non-relational databases proved more effective. Cheng et al. (2019) compared RocksDB 5.8, Hbase 2.2, Cassandra 3.11, Neo4j 3.4.6, and MySQL 5.7 using both relational and graph datasets [6]. They found that relational databases performed better in workloads involving group by, sort, and aggregation. Graph databases performed well in tasks like multiple table joins and pattern matching.

Shalygina et al. (2017) investigated the Common Table Expression capabilities of MariaDB by comparing them to Postgres [7]. The study revealed that for short tasks with few repetitions Postgres is faster and for complex tasks with many repetitions MariaDB is faster. Vicknair et al. (2010) compared MySQL v5.1.42 and Neo4j v1.0-b11, transferring a graph database to a relational one [8]. They tested three structural and three data queries. Neo4j performed better in structural queries. But MySQL was more efficient in data queries due to Neo4j’s Lucene indexing, impacting integer data conversion.

III. Methodology

A. Test Database

An e-commerce system database is selected as test database. Initially database is created without any additional indexes.

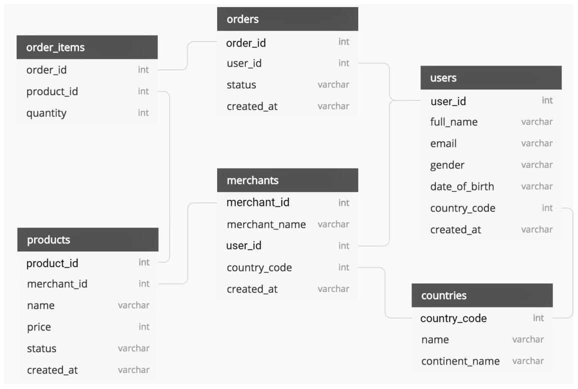


Fig. 1: SQL Database ER diagram

My SQL relational database has 6 tables. They are Orders,Products, Merchants, Users, Countries and order items. Orders table stores information about orders placed by users. It includes attributes like order ID (primary key), user ID (foreign key referencing the Users table), status and creation date. Products table stores information about products sold on the platform. It includes attributes like product ID (primary key), merchant ID (foreign key referencing the Merchants table), product name, price, status, and creation date. Merchants table stores information about merchants selling products on the platform. It includes attributes like merchant ID (primary key), merchant name, user ID (foreign key referencing the Users table), country code (foreign key referencing the Countries table), and creation date. Users table stores information about users of the platform. It includes attributes like user ID (primary key), full name, email, gender, date of birth, country code (foreign key referencing the Countries table), and creation date. Countries table stores information about countries where users reside. It includes attributes like country code (primary key), country name, and continent name. order items table stores information about ordered items in the orders.It is a relationship table. It includes attributes like order id, product id (composite primary key), and quantity.

In this database, One user can place many orders. One order can contain many products, and one product can be included in many orders. One merchant can sell many products, and one product belongs to one merchant. One user can be a merchant, and one merchant is associated with one user. One user belongs to one country, and one country can have many users.

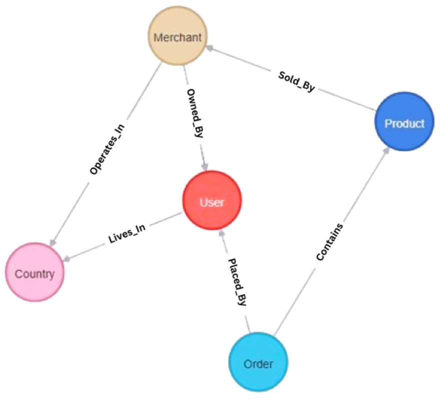
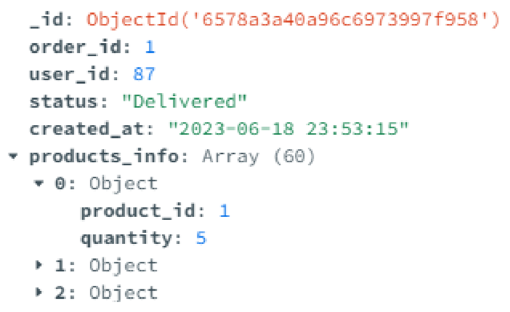


Fig. 2: Neo4j Database schema

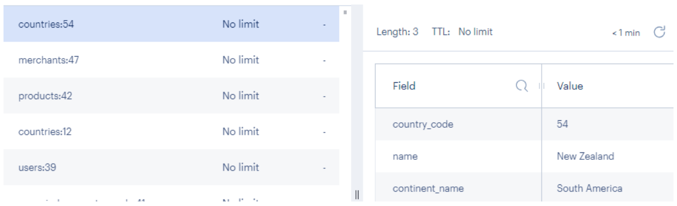
In the Neo4j database, entities such as User, Order, Product, Merchant, and Country are represented as nodes, with their properties encoded as node properties. The relationship between the Order and Product entities, analogous to the order items table in a relational database, is depicted as a 'contains' relationship in Neo4j. The properties of the order items table are modelled as properties of that relationship. Also relevant relationships are created among other entities.



```
_id: ObjectId('6578a3a40a96c6973997f958')
order_id: 1
user_id: 87
status: "Delivered"
created_at: "2023-06-18 23:53:15"
▾ products_info: Array (60)
  ▾ 0: Object
      product_id: 1
      quantity: 5
  ▸ 1: Object
  ▸ 2: Object
```

Fig. 3: A Document of Order Entity in MongoDB

In MongoDB, User, Order, Product, Merchant, and Country entities are structured as collections, and records of these entities are stored as documents within their respective collections. Each document contains attributes corresponding to the entity's properties. In MongoDB, the order items table from the relational database is integrated into the Orders collection as an attribute called "products info," where the relevant details of each order item are embedded within the document representing the order.



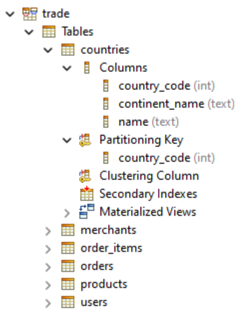
countries:54	No limit	-
merchants:47	No limit	-
products:42	No limit	-
countries:12	No limit	-
users:39	No limit	-

Length: 3 TTL: No limit <1 min

Field	Value
country_code	54
name	New Zealand
continent_name	South America

Fig. 4: Redis Database

In Redis, keys for each record are formulated in the following format: 'entity:unique identifier of record'. Subsequently, key-value pairs representing the attributes of the record are stored within the respective key.



```
trade
  Tables
    countries
      Columns
        country_code (int)
        continent_name (text)
        name (text)
      Partitioning Key
        country_code (int)
      Clustering Column
      Secondary Indexes
      Materialized Views
    merchants
    order_items
    orders
    products
    users
```

Fig. 5: Cassandra DB schema of countries table

In Cassandra, tables similar to those in relational databases are created with their primary and foreign keys.

B. Test Program and Query Creation

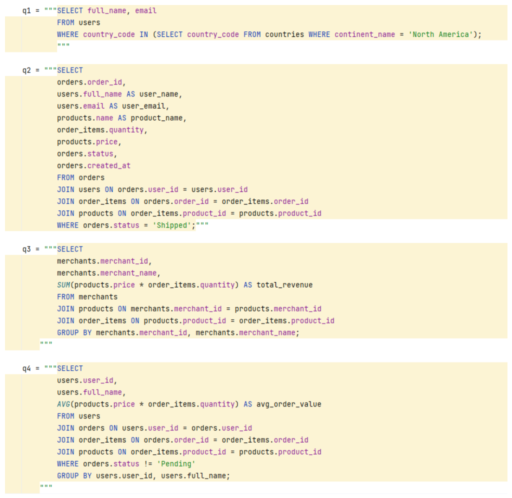
A Python program is utilized to create above databases and relevant tables or documents within those databases. Faker library is used to generate and insert random data samples into the database. Initially, data is inserted into the MySQL database, and then that data is saved as a CSV file. Subsequently, that CSV file is imported into other databases to ensure that the same data is inserted across all databases.

The following queries have been created in their respective query languages to analyze query performance across varying complexities.

- short complex : selects full name and email of users in north American countries
- long complex : Selects information about all shipped orders
- Aggregation : Calculates total revenue for each merchant
- Aggregation with defined key : Calculates average order value per user

C. Indexing and Indexed Query Creation

In MySQL, the primary key also serves as an index. Therefore, additional indexes are created as needed for efficient querying. For query 1, indexes are created on the country



```
q1 = """SELECT full_name, email
        FROM users
        WHERE country_code IN (SELECT country_code FROM countries WHERE continent_name = 'North America');
        """

q2 = """SELECT
        orders.order_id,
        users.full_name AS user_name,
        users.email AS user_email,
        products.name AS product_name,
        order_items.quantity,
        products.price,
        orders.status,
        orders.created_at
        FROM orders
        JOIN users ON orders.user_id = users.user_id
        JOIN order_items ON orders.order_id = order_items.order_id
        JOIN products ON order_items.product_id = products.product_id
        WHERE orders.status = 'Shipped';"""

q3 = """SELECT
        merchants.merchant_id,
        merchants.merchant_name,
        SUM(products.price * order_items.quantity) AS total_revenue
        FROM merchants
        JOIN products ON merchants.merchant_id = products.merchant_id
        JOIN order_items ON products.product_id = order_items.product_id
        GROUP BY merchants.merchant_id, merchants.merchant_name;
    """

q4 = """SELECT
        users.user_id,
        users.full_name,
        AVG(products.price * order_items.quantity) AS avg_order_value
        FROM users
        JOIN orders ON users.user_id = orders.user_id
        JOIN order_items ON orders.order_id = order_items.order_id
        JOIN products ON order_items.product_id = products.product_id
        WHERE orders.status != 'Pending'
        GROUP BY users.user_id, users.full_name;
    """
```

Fig. 6: SQL Queries

code column of users table and the continent name column of countries table. For query 2,indexes are created on user id column of the orders table, order id column of the order items table, the product id column of the order items table and status column of the orders table. For queries 3 and 4, the indexes created previously can be utilized to optimize query performance. Query structure remains unchanged in MySQL. Because MySQL's query optimizer automatically evaluates the available indexes and uses them.

In Neo4j, when initially setting up the database, there are no indexes established by default. So, specific indexes are created. Indexes are established on the 'name' and 'continent name' of Country nodes, the 'email', 'gender', and 'date of birth' of User nodes, the 'merchant name' of Merchant nodes. the 'status' and 'created at' of Order nodes, as well as the 'price', 'status', and 'created at' of Product nodes. Furthermore, an index is created for the 'quantity' property of the 'contains' relationship. Here also queries do not need to be modified after indexing.

In MongoDB, a compound index is created on the fields 'continent name' and 'country code' within the 'countries' collection. Another compound index is constructed on the fields 'product id' and 'merchant id' within the 'products' collection. Additionally, unique indexes are generated on the 'order id' field within the 'orders' collection, the 'country code' field in the 'countries' collection, the 'user id' field in the 'users' collection, and the 'merchant id' field in the 'merchants' collection. Here also queries do not need to be modified after indexing.

In redis sorted set Indexes are created for 'countries' based on the 'continent name' field, 'users' based on the 'country code' field, 'products' based on the 'merchant id' field, and 'orders' based on the 'status' field. Here all queries are rewritten to use the created indexes using sorted sets.

In Cassandra, the primary key also serves as a partitioning key and clustering columns. so additional secondary indexes are created on 'continent name' column of the 'countries' table, 'country code' column of the 'users' table, 'status' column of the 'orders' table, 'user id' column of the 'orders' table, 'product id' column of the 'order items' table, 'merchant id' column of the 'products' table. Here all queries are rewritten to take advantage of the indexed columns.

D. *Test Execution*

The tests were conducted on a Dell 5593 laptop with the following specifications:

- Operating System: Windows 11
- Processor: Intel Core i7-1065G7 Quad-Core
- Memory: 16GB DDR4 RAM
- Graphics: Intel Iris Plus Graphics

MySQL 8.0.34, MongoDB 7.02, Neo4j 5.12.0, Redis 7.0.14 and Cassandra 3.11.16 version were installed on this computer.

After creating the database and tables, All queries are executed ten times. Then their average is calculated to determine the average execution time. Subsequently, all databases are indexed, and queries are updated to utilize indexes. Then, the queries are evaluated ten times again, and the average is calculated to determine the correct value.

IV. Results and Discussion

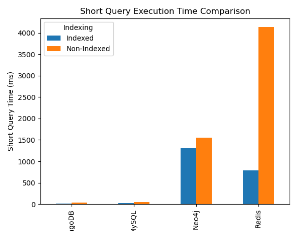
The test results for the above databases are listed in the table below.

TABLE I: Test Query Execution Time(ms) of Databases

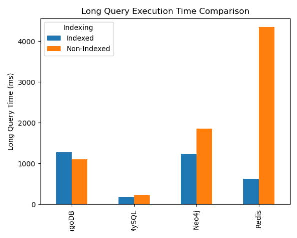
Database	Index	Short	Long	Aggregation	Agg.with Defined key
MySQL	False	49	221	450	253
MySQL	True	30	180	360	219
MongoDB	False	39	1099	4030	1128
MongoDB	True	9	1270	1230	1344
Neo4j	False	1557	1854	2168	2260
Neo4j	True	1311	1242	1465	222
Redis	False	4133	4347	1994	4790
Redis	True	791	625	2362	4335
Cassandra	False	5914	193318	34524	404481
Cassandra	True	5885	193218	34572	404300

The above results show that Cassandra takes more time than all other databases with or without indexing, and that time has a greater difference compared to the other databases. For better visualization, Cassandra is not plotted in the comparison graphs.

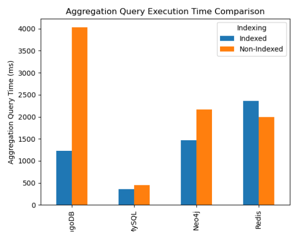
In short queries without an index, MySQL and MongoDB performed the best, while Neo4j and Redis took longer times. specifically, Redis takes more time than Neo4j. The reason for this difference in short query execution performance lies in the nature of each database. Due to MySQL's relational model and MongoDB's document-oriented model, they take less time. Neo4j takes longer due to navigation through relationships. Reason for Redis's under performance is its in-memory processing may not be optimized. Indexing improved the performance of all databases. With the help of indexing, Redis overtakes Neo4j. This is because Redis's indexes are well-suited for small queries.



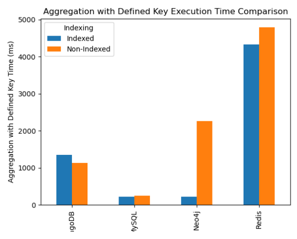
(a) Short Query



(b) Long Query



(c) Aggregation Query



(d) Aggregation with defined key

Fig. 7: Query Performance Comparison

MySQL maintained its lead for long queries, but MongoDB's performance decreased, even though MongoDB outperforms others except MySQL. Like short queries, Redis and Neo4j take a long time. specifically, Redis taking more time than Neo4j. MySQL engine's efficient optimization and MongoDB's efficient join access resulted above performance. However with indexing, MongoDB's performance decreased, and Neo4j over-performed it, securing second place after MySQL. The reason for the performance degradation of MongoDB is inefficient indexing.

In the aggregation queries, MongoDB exhibits the longest execution time. Neo4j and Redis perform better than MongoDB, with MySQL performing the best. The reason for MongoDB's performance degradation is the lack of optimized aggregation and overhead joins. Indexing reduced the execution time for all of these databases except Redis. With indexing, MongoDB's execution time reduced significantly, but MySQL still performs the best.

In the Aggregation with defined Key queries, MySQL performs well. MongoDB performs slightly less than MySQL, while Neo4j and Redis perform the worst. Indexing reduced the execution time for all databases except MongoDB. With indexing, Neo4j's execution time was reduced significantly. It gives equivalent performance to MySQL after indexing.

V. Conclusion

The present study compared the query execution performance of SQL and NoSQL databases (document database, graph database, key-value database, column database) in executing complex queries with and without indexing. It has been discovered that some databases benefit from indexing.

- MySQL leads in overall performance in every type of query.
- Neo4j excels with indexing for Aggregation with defined Key queries. Because its graph-based data model and indexing capabilities are well-suited for traversing relationships and aggregating data based on defined keys.
- Redis shines in small queries with indexing because of its in-memory data storage.
- MongoDB is generally good for handling small queries. But indexing may be problematic at sometimes. It happens due to the overhead introduced by maintaining indexes or inefficient index usage.
- Cassandra struggles in a single-node setup. Because, its architecture is optimized for distributed and scalable deployments. In a single-node setup, Cassandra may not fully utilize its distributed nature and may encounter performance limitations

References

[1] P. Kotiranta, M. Junkkari, and J. Nummenmaa, "Performance of Graph and Relational Databases in Complex Queries," Applied Sciences, vol. 12, no. 13, p. 6490, Jan. 2022, doi: https://doi.org/10.3390/app12136490.

[2] R. Čerešňák and M. Kvet, "Comparison of Query Performance in Relational a non-relation Databases," Transportation Research Procedia, vol. 40, pp. 170–177, 2019, doi: https://doi.org/10.1016/j.trpro.2019.07.027.

[3] Khan, M.Z., Zaman, F.U., Adnan, M., Imroz, A., Rauf, M.A., and Phul, Z. "Comparative Case Study: An Evaluation of Performance Computation Between SQL And NoSQL Database." Journal of Software Engineering, vol. 1, no. 2, pp. 14-23, 2023.

[4] R. J. Sholichah, M. Imrona, and A. Alamsyah, "Performance Analysis of Neo4j and MySQL Databases Using Public Policies Decision Making Data," IEEE Xplore, Sep. 01, 2020. https://ieeexplore.ieee.org/document/9239206 (accessed May 09, 2022).

[5] L. Stanescu, "A Comparison between a Relational and a Graph Database in the Context of a Recommendation System," InProceedings of the 16th Conference on Computer Science and Intelligence Systems, Sep. 2021, doi: https://doi.org/10.15439/2021f33.

[6] Y. Cheng, P. Ding, T. Wang, W. Lu, and X. Du, "Which Category Is Better: Benchmarking Relational and Graph Database Management Systems," Data Science and Engineering, vol. 4, no. 4, pp. 309–322, Nov. 2019, doi: https://doi.org/10.1007/s41019-019-00110-3.

[7] D. Bartholomew, MariaDB and MySQL Common Table Expressions and Window Functions Revealed. 2017. doi: https://doi.org/10.1007/978-1-4842-3120-3.

[8] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A Comparison of a Graph Database and a Relational Database," Proceedings of the 48th Annual Southeast Regional Conference on - ACM SE '10, 2010, doi: https://doi.org/10.1145/1900008.1900067.