Type of the Paper (Article, Review, Communication, etc.)

PERFORMANCE OF GRAPH AND RELATIONAL DATABASES IN COMPLEX QUERIES

Petri Kotiranta 1, Marko Junkkari 2 and Jyrki Nummenmaa 3,\*

|  |
| --- |
| **Citation:** Lastname, F.; Lastname, F.; Lastname, F. Title. *Appl. Sci.* **2021**, *11*, x. https://doi.org/10.3390/xxxxx  Academic Editor: Firstname Lastname  Received: date  Accepted: date  Published: date  **Publisher’s Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.    **Copyright:** © 2021 by the authors. Submitted for possible open access publication under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/). |

1 Affiliation 1; e-mail@e-mail.com

2 Tampere University; marko.junkkari@tuni.fi

3 Tampere University; jyrki.nummenmaa@tuni.fi

**\*** Correspondence: e-mail@e-mail.com; Tel.: (optional; include country code; if there are multiple corresponding authors, add author initials)

**Featured Application: Authors are encouraged to provide a concise description of the specific application or a potential application of the work. This section is not mandatory.**

**Abstract:** In developing NoSQL databases, a main motivation has been to achieve more efficient query performance compared with relational databases. The graph database is a NoSQL paradigm where the navigation is based on links instead of joining tables. Link based navigation has been seen more efficient as query approach than join operations of tables. Existing studies strongly support this assumption. However, query complexity has received less attention. For example, in enterprise information systems queries are usually complex and data should be collected from several data records. In the present study, we compare the query performance of a graph-based database (Neo4j) and relational databases (MySQL and MariaDB). The effect of different efficiency issues (e.g. indexing, optimization) are benchmarked in order to investigate the most efficient solutions in different query types. The outcome is that although Neo4j is more efficient in simple queries, MariaDB is essentially more efficient when the complexity of queries increases. The study also shows how dramatically the efficiency of relational database has been grown during the last decade.

**Keywords:** graph database; relational database; performance; complex queries, Neo4J, MariaDB

1. Introduction

Performance has been one of the motivations to use NoSQL databases instead of traditional SQL databases [Leavitt, 2010; Győrödi, 2020]. With data and queries suitable for the data model, NoSQL databases might offer significant performance benefits. In the present study, we compare traditional relational model and NoSQL graph model. The graph model [Angles, 2008], of the four major NoSQL types, consists of nodes and edges, and it has its own benefits when handling relationship rich data. While in SQL databases multiple tables may need to be joined for a relational query, in graph databases relational information can be queried by navigating through the graph.

Previous studies [4,5,6,7,8] where the performance of graph databases, especially Neo4j, have been compared with the traditional SQL databases, indicate that graph databases possess better performance than relational databases. However, those studies are mainly focused on quite simple queries. In the contrast of the earlier studies, we investigate the performance of databases in the situation where query complexity grows. In a complex query, the needed data must be collected from several data record (tables in SQL databases, nodes in graph databases) or a long path must be gone through recursively. Based on a complex query an aggregated value (e.g., count or average) from a large data set can be calculated. Complex queries are typical in various application domains like Enterprise information systems [Niemi, Junkkari, Järvelin, 2014], Geographical information systems [Marie-Aude Aufaure-Portier, Claude Trépied, 1996], Bioinformatics [Leser, 1996] and CAD systems [Hardwick, Samaras and Spooner, 1987]. The sample data of the present paper relate to Enterprise information systems.

In the present study, MariaBD is selected as a relational database because it is a modern database and, to our best knowledge, it is not earlier compared to graph databases. MySQL is also included in the present benchmarking in order to investigate how relational databases have been developed. MariaBD is a modern version of MySQL. For the reason, that they belong to the same database family, their comparison shows well, how relational databases have been developed during the last decade. Second, earlier studies are based on basic setting of queries. Instead, we aim to find the most effect way to query formulation. Indexing is a traditional method to improve performance and it can be applied to relational and graph databases. Further, for the used graph database (Neo4J), a more efficient query type (call-function) has been developed. Recursive queries can be optimized in modern versions of Neo4J. In benchmarking of databases, we take into account all these settings.

In order to benchmark the databases in complex queries we designed and implemented a new test bed that also support complex queries unlike existing benchmarks such as [1] or [2]. Our test bed is designed for testing queries for MariaDB, MySQL and Neo4J. The test database relates to enterprise information systems, but it is worth to note that query types are general and processing of data is similar independently of application domains. The test bench is called Invoicing Database Test Bench and its source code is available from GitHub [3]. The program generates a selected amount of data for the test invoicing database schema and performs various query tests. Our dataset is public. The source code for generating the data is available in GitHub, and, thus, it is possible for anyone to repeat this test by installing the same test settings.

The rest of the paper is organized as follows. Section 2 reviews previous work related to Neo4j and MariaDB performance analysis. Section 3 introduces the schema that is used for the test data. Section 4 presents the implemented benchmarking program. Section 5 presents the test queries. Section 6 presents the test results. Section 7 discusses the results and Section 8 has the conclusions.

2. Related work

An old MySQL version is involved in the comparison to make our research compatible with earlier studies. From this perspective MariaDB is a natural choice for a modern database because Maria DB is a descendant of MySQL. When making the present study, DB-Engines site ranks MariaDB as 8th out of 138 of relational databases [14]. Neo4j ranks the first out of 32 databases on the same sites. As both of the databases are quite popular, they are often candidates to be used in many enterprises. One of the goals of this study is to find out differences in what use case the databases should be used.

SQL database and Neo4j have been compared in several studies including [4,5,6,7,8]. Khan et al. compared tuned Oracle 11g and Neo4j 3.03 Community Edition [4]. Healthcare data was used including data of patients, medicines and medical staff. Performance of the databases was evaluated with ten different count(\*) queries. Many of the queries also performed some table joins. Physical database tuning technique called tablespaces was used for Oracle. The same databases were compared without physical database tuning by Khan et al. [6]. The physical database tuning technique decreased the overall average query time of Oracle from 4.34 to 2.78 seconds. However, the overall average query time for Neo4j in query tests was only 0.67 seconds. Thus, Neo4j performed better compared to Oracle.

Holzschuher et al. tested Neo4j version 1.8 performance with different backend solutions [5]. Neo4j was benchmarked as embedded with native object access, as a dedicated server through RESTful Web Services, with embedded Cypher queries, with Cypher through REST optimized for remote execution and with Gremlin queries through REST. MySQL version 5.5.27 was also included with Java Persistence API based backend. Queries were done using Cypher, Gremlin and SQL query languages. The test data consisted of data of persons and their relationships. Relational test queries were executed such as friends of friends. As the database got larger, the advantages of Neo4j over MySQL become more prevalent. Neo4j performance stayed nearly constant when MySQL performance dropped by factors 5 and 7-9. Both Neo4j query languages Gremlin and Cypher had performance benefits over MySQL with JPA.

Vicknair et al. compared MySQL Community Server version 5.1.42 and Neo4j version 1.0-b11 in 2010 [7]. The graph database was stored into a relational database as nodes and edges. Three types of structural and three types of data queries were made. First structural query found all the orphan nodes and the two other ones traversed the graph in the depths of 4 and 128. The data queries were count(\*) queries counting nodes with certain payloads. Neo4j performed better in structural queries. However, with the integer-based queries MySQL was more efficient due to the fact that the tested Neo4j used Lucene indexing. As it treated the data as text by default, conversions had to be made and thus they impacted the performance. The work [7] by Vicknair et al. has been referenced in [4,5,6].

Batra et al. compared MySQL version 5.1.41 and Neo4j Community version 1.6 in 2012 [8]. They used a schema with tables user, friends, fav\_movies and actors for testing, and they tested the databases with three queries: “Find all friends of Esha”, “Find all favourite movies of Esha’s friends” and “Find the lead actors of Esha’s friends favourite movies”. Queries were executed on 100 and 500 objects. Neo4j had 2-5 times faster query times with 100 objects data set and 15-30 times faster in 500 objects data set. The work by Batra et al [8] has similarity to the present study as the data is stored into SQL database with a relational schema unlike in the work by Vicknair et al [7]. The work [8] by Batra et al is referenced in [5].

There also exist previous performance studies where MariaDB is involved. Tongkaw et al. compared the performance of MariaDB 10.0.21 and MySQL 5.6 [9]. They used Sysbench and OLTP [2] software. OLTP-Simple and OLTP-Seats workloads were used. Both databases consumed the same number of resources. However, when an increasing the number of threads in OLTP-Simple and the number of workers in OLTP-Seats was used, MySQL became clearly more effective outperforming MariaDB. Shalygina et al. studied the Common Table Expression capabilities of MariaDB along with Postgres [10]. The study showed that Postgres had better results, when only a few steps of recursion was needed. However, MariaDB was a better choice for a long recursive process on a huge amount of data.

3. Test database

The test database is a general example from enterprise information systems, where different information is associated with customers. In the example, a customer may have several targets for which different types of works and items are associated. When sending an invoice for a customer, the stored works and items must be collected and calculated which may require a complex navigation among stored data. As it is an invoicing database, one of the most important use cases is the calculation of the price for a customer invoice. This is done by calculating the used time for work of different work types and the price of the items used when working. Invoices might also have relations to other invoices if several invoices are sent to the customer.

The relational database has 10 tables. The basic tables represent entities of the application domain, and they are customer, invoice, target, work, worktype and item. These tables contain the customer information, customer’s invoices, the target (or project) where the work is done, a listing of each work, a listing of different worktypes with different prices and information about the items used for each work. Relational data between the entity tables are stored into relationship tables worktarget, workinvoice, useditem and workhours. These represents many-to-many relationships between entities. Figure 1 shows the database structure or relation schema. Arrows illustrate how the tables are associated with each other. For example, the arrow from the invoice to the customer means that customer\_id in the invoice refers to the identity of the customer.

Diagram

Description automatically generated

Figure 1: Database structure in relational format.

In the graph format, entities are represented by nodes and edges to represent the relationships. Bidirectional edges are used for many-to-many relationships. Customer, invoice, target, work, and worktype are entities and they are represented as nodes. PAYS from the customer to invoice, CUSTOMER\_TARGET from the customer to the target are directed edges, PREVIOUS\_INVOICE from an invoice to another are directional edges. The latter is recursive. WORK\_TARGET, WORK\_INVOICE, WORKHOURS and USED\_ITEM are bidirectional edges. Figure 2 represent the database structure in a graph format. The attributes of nodes and edges are not illustrated.



Figure 2: Database structure in graph format.

4. Test program

The test data are generated using a Java program. The customer and target generation uses sample data that are based on openly available name and address data sets [13], [12]. The generation process is divided into three parts. Items and work types need to be generated first, then work and customer data. The ~~Java~~ program has threaded classes for each part. Multiple threads can be used to insert the generated data. For random data generation, controlled random seeds are used, making the generation repeatable.

The generation is controlled with parameters for: numbers of work types; numbers of items; numbers of related invoices, targets and work for a customer; number of works; number of customers; numbers of relations between worktypes and works; numbers of invoices and targers for each customer; and numbers of workinvoice and worktarget relationships.

The program has a class called QueryTester that is used to perform the query tests. Query tests are repeated the selected number of times. The test program collects the performance figures from the executions into a list structure. The program removes the biggest and the smallest number from the list and calculates an average and a standard deviation from the rest of the results.

5. Test queries

The query tests contain different queries for calculating the price of work and invoice and a recursive query. These queries test different capabilities of the databases. As both relational and graph databases are tested, the queries are in SQL and Cypher form. The query tests are ordered from simple to complex starting from the work price and the work price with items ending in the work prices and work prices for a given customer. Finally, there is a recursive query test.

The queries were chosen as they represent typical queries that would be executed in the chosen test database. Finding and calculating invoice related information is what the database would be most used for, and this is what all the test queries do. As querying all the information required for invoice ends up in complex query, simpler queries were included in order to see how databases perform with different complexity of queries.

Calculating the invoice prices is one of the most important queries. The schema does not store invoice prices explicitly. The price has to be calculated based on the amount of the workhours and the items used. The “price of work” and the “price of work with items” are the subqueries for calculating this price. The query calculating invoice prices for a given customer add customer information into this query. The recursive query queries all the interrelated invoices. One practical example is that customer has not paid the invoice and there will be additional invoices based on the same invoice.

5.1. Query optimization

In addition to just testing databases with the queries, we investigated improving the query performance, and in particular indexing. Both MySQL and MariaDB index primary key and foreign key by default while Neo4j does not create indexes for properties by default. The effect of indexing is also different when comparing an SQL database with a graph database. When querying relations, in SQL databases the relations are formed by joining the tables based on primary key and foreign key information. Thus, SQL databases usually benefit from the indexing of primary keys and foreign keys. In a graph database, we are traversing the graph when querying data. As such, it does not benefit from indexing the properties the way SQL databases do.

In order to study the effects of indexing, certain columns and properties that were used in queries were indexed in all the databases. Table 1 shows the extra indexes created. As ids in customer and invoice tables are indexed by default in MySQL and MariaDB, an extra index was not needed.

|  |  |  |
| --- | --- | --- |
| Table/Node | SQL | Neo4j |
| Customer | - | customerId |
| Invoice | previousinvoice | invoiceId, previousinvoice |
| Item | purchaseprice | purchaseprice |
| Workhours | hours, discount | hours, discount |
| Worktype | price | price |
| Useditem | amount, discount | amount, discount |

Table 1: Indexed columns/properties in SQL and Neo4j.

Besides indexing, in Neo4j 4.1.3 the queries can be optimized using CALL subqueries [13]. The CALL clause makes it possible to execute subqueries in other queries. It is like a function that gets input parameters from the main query and returns some values. The subquery is executed for each incoming input row from calling the query from the main query. CALL has been supported from Neo4j 4.1 onwards. In the present study, Cypher queries with and without CALL are used in order for backward compatibility. This way it is also possible to see how much CALL subquery improves the query performance.

5.2. Short query, price of work

The first test query calculates the price of works. One work can have different work types with different prices. The price of one work is defined by the number of hours done of the work type. There can also be a discount on the prices and the discount is included in calculation. This query shows how databases perform with a fairly simple query. Table 2 shows the queries to query the price of work in SQL and Cypher.

Table 2: Price of work query in SQL and Cypher.

|  |
| --- |
| SQL |
| SELECT work.id AS workId, SUM((worktype.price \* workhours.hours \*  workhours.discount))  AS price  FROM work INNER JOIN workhours ON work.id = workhours.workId  INNER JOIN worktype  ON worktype.id = workhours.worktypeId  GROUP BY work.id |
| Cypher |
| MATCH (wt:worktype)-[h:WORKHOURS]->(w:work)   WITH SUM(h.hours\*h.discount\*wt.price)AS price, w  RETURN w.workId as workId, price |
| Cypher with CALL |
| MATCH (w:work) CALL {  WITH w  MATCH (wt:worktype)-[h:WORKHOURS]->(w)  RETURN SUM((h.hours\*h.discount\*wt.price))AS price} RETURN w.workId as workId, price |
|  |
|  |
|  |
|  |
|  |

5.3. Long query, price of work with items

The query for the price of work with items is an extended version of the query for the price of work. This query adds item prices into work prices. As items are also included, a longer relational query is needed. With this query it is possible to see how databases perform when more relations and calculations are included in the query. Item purchase price is a floating-point number so this will add more challenges to the calculations. Table 3 shows the queries for the price of work with items in SQL and Cypher.

Table 3: The price of work with items query in SQL and Cypher.

|  |
| --- |
| SQL |
| SELECT work.id AS workId, SUM((worktype.price \* workhours.hours \*  workhours.discount) + (item.purchaseprice \* useditem.amount \*  useditem.discount)) AS price  FROM work INNER JOIN workhours  ON work.id = workhours.workId  INNER JOIN worktype  ON worktype.id = workhours.worktypeId  INNER JOIN useditem  ON work.id = useditem.workId  INNER JOIN item  ON useditem.itemId = item.id  GROUP BY work.id |
| Cypher |
| MATCH (wt:worktype)-[h:WORKHOURS]->(w:work) -[u:USED\_ITEM]->(i:item)  WITH SUM((h.hours\*h.discount\*wt.price)+  (u.amount\*u.discount\*i.purchaseprice)) AS price, w  RETURN w.workId as workId, price |
| Cypher with CALL |
| MATCH (w:work)  CALL {  WITH w  MATCH (wt:worktype)-[h:WORKHOURS]->(w)- [u:USED\_ITEM]->(i:item)  RETURN  SUM((h.hours\*h.discount\*wt.price) + (u.amount\*u.discount\*i.purchaseprice))  AS price }  RETURN w.workId as workId, price |

5.4. Complex query, invoice price

This query calculates the sum of a work price for each invoice. The query contains two subqueries. The first one finds the relation of the invoices and work. The second query is the previously presented “price of work with items”. The results of these queries are joined and the sums of prices are aggregated based on the id of the invoice. This is one of the heaviest queries and as such it is useful to see the performance differences when executing a complex query. Table 4 present the queries for calculating the invoice price in SQL and Cypher.

Table 4: The query for invoice prices in SQL and Cypher.

|  |
| --- |
| SQL |
| SELECT q1.invoiceId, SUM(q2.price) AS invoicePrice  FROM (   SELECT workinvoice.invoiceId, workinvoice.workId   FROM workinvoice  INNER JOIN invoice ON workinvoice.invoiceId = invoice.id)  AS q1 INNER JOIN (   SELECT workhours.workid AS workId,  SUM((worktype.price \* workhours.hours \* workhours.discount) +  (item.purchaseprice \* useditem.amount \* useditem.discount))AS price   FROM workhours INNER JOIN worktype ON workhours.worktypeid = worktype.id   INNER JOIN useditem ON workhours.workid = useditem.workid INNER JOIN item  ON useditem.itemid = item.id   GROUP BY workhours.workid)  AS q2 USING (workId)  GROUP BY q1.invoiceId |
| Cypher |
| MATCH (inv:invoice)-[:WORK\_INVOICE]->(w:work)   WITH inv, w   OPTIONAL MATCH (wt:worktype)-[h:WORKHOURS] ->(w:work)-[u:USED\_ITEM]->(i:item)   WITH inv, w, SUM((h.hours\*h.discount\*wt.price) +  (u.amount\*u.discount\*i.purchaseprice)) AS workPrice  RETURN inv, SUM(workPrice) as invoicePrice |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| Cypher with CALL |  |
|  |
|  |
|  |
|  |
| CALL {   WITH inv   MATCH (inv)-[:WORK\_INVOICE]->(w:work)   RETURN w }  CALL {   WITH w   MATCH (wt:worktype)-[h:WORKHOURS]->(w)-[u:USED\_ITEM]->(i:item)  RETURN SUM((h.hours\*h.discount\*wt.price)+(u.amount\*u.discount\*i.purchaseprice))  AS workPrice }  RETURN inv, SUM(workPrice) AS invoicePrice |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

5.5. Query with defined key, invoice prices for customer with id 0

It is often necessary to find out all the invoice prices for a given customer. The query that calculates invoice prices for a given customer is an extended query from the query that calculates invoice prices. A subquery to get customer’s relation to invoices is included. This query is the most complex of the tested queries. From the technical point of view this query shows how databases perform when there is a certain key defined for which the data should be related to. Table 5 presents the queries for calculating invoice prices for a given customer.

Table 5: The query for the Invoice prices of a defined customer in SQL and Cypher.

|  |
| --- |
| SQL |
| SELECT q1.customerId, q2.invoiceId, SUM(q3.price) AS invoicePrice   FROM (SELECT customer.id AS customerId, invoice.id AS invoiceId   FROM invoice INNER JOIN customer ON invoice.customerId=customer.id) AS q1   INNER JOIN (  SELECT workinvoice.invoiceId, workinvoice.workId   FROM workinvoice INNER JOIN invoice  ON workinvoice.invoiceId = invoice.id) AS q2 USING (invoiceId)  INNER JOIN (   SELECT workhours.workid AS workId, SUM((worktype.price \* workhours.hours  \* workhours.discount) + (item.purchaseprice \* useditem.amount \*  useditem.discount)) AS price   FROM workhours   INNER JOIN worktype ON workhours.worktypeid = worktype.id   INNER JOIN useditem ON workhours.workid = useditem.workid   INNER JOIN item ON useditem.itemid = item.id  GROUP BY workhours.workid) AS q3 USING (workId)  WHERE q1.customerId=0  GROUP BY q2.invoiceId |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| Cypher |  |
| MATCH (c:customer)-[:PAYS]->(inv:invoice) WHERE c.customerId=0   WITH c, inv  OPTIONAL MATCH (inv)-[:WORK\_INVOICE]->(w:work)   WITH c, inv, w  OPTIONAL MATCH (wt:worktype)-[h:WORKHOURS]->(w:work)-[u:USED\_ITEM]->(i:item)   WITH c, inv, w,  SUM((h.hours\*h.discount\*wt.price)+(u.amount\*u.discount\*i.purchaseprice))  AS workPrice  RETURN c, inv, SUM(workPrice) as invoicePrice |  |
|  |
|  |
|  |
|  |
|  |
| Cypher with CALL |  |
| MATCH (inv:invoice) WHERE inv.customerId=0  CALL {WITH inv   MATCH (c:customer)-[:PAYS]->(inv)   RETURN c} CALL {WITH c, inv   MATCH (inv)-[:WORK\_INVOICE]->(w:work)   RETURN w}  CALL {WITH w   MATCH (wt:worktype)-[h:WORKHOURS]->(w)-[u:USED\_ITEM]->(i:item)   RETURN SUM((h.hours\*h.discount\*wt.price)+(u.amount\*u.discount\*i.purchaseprice))  AS workPrice}  RETURN c, inv, SUM(workPrice) AS invoicePrice |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

5.6. Recursive query, invoices related to invoice id 100000

This recursive query finds all the sequential invoices related to given invoice id. The query is useful to test the recursive query capabilities of the databases. In SQL, Common Table Expressions are used to make the query. In Cypher there is a way to optimize a recursive query by negating irrelevant relationships. The optimized query does not return exactly the same result as the basic query. While the basic query returns a set of individual nodes, the optimized query returns a list structure containing nodes. However, it still returns similar results and as such it is a relevant query. Table 6 presents the queries for finding sequential invoices for a given invoice.

Table 6: The Recursive query for getting sequential invoices related to defined invoice in SQL and Cypher

|  |
| --- |
| SQL |
| WITH RECURSIVE sequential\_invoices AS  (SELECT id, customerId, state, duedate, previousinvoice   FROM invoice   WHERE id=10000  UNION ALL   SELECT i.id, i.customerId, i.state, i.duedate, i.previousinvoice   FROM invoice AS i INNER JOIN sequential\_invoices AS j ON i.previousinvoice = j.id   WHERE i.previousinvoice <> i.id)  SELECT \* FROM sequential\_invoices |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| Cypher |  |
| MATCH (i:invoice { invoiceId:10000 })-[p:PREVIOUS\_INVOICE \*0..]->(j:invoice)  RETURN \* |  |
| Cypher optimized |  |
| MATCH inv=(i:invoice { invoiceId:10000})-[p:PREVIOUS\_INVOICE \*0..]->(j:invoice)  WHERE NOT (j)-[:PREVIOUS\_INVOICE]->()  RETURN nodes(inv) |  |

6. TEST EXECUTIONS

6.1. Test settings

The tests were performed with MacBook Pro Laptop with following specifications:

• MacOS Catalina version 10.15.5

• 1,4 GHz quad core Intel Core i5

• 8 GB 2133 MHz LPDDR3

• Intel Iris Plus Graphics 645, 1536 MB

MySQL version 5.1.41, MariaDB version 10.5.6 and Neo4j community edition version 4.1.3 were installed on this computer. MariaDB and Neo4j were the latest when making the present study. MySQL version 5.1.41 is already considered as "end-of-life" when making the present study. However, it was used in a previous study [8] and version 5.1.42 in [7] as shown in related studies. MariaDB driver version 2.7 and Neo4j driver version 4.1.1 were used.

A dataset was generated using the test program. Table 7 presents the number of rows/objects generated for the dataset. For each row in the relationship tables useditem, workhours, workinvoice and worktarget, two respective edges were generated for the Neo4j graph database, as a many-to-many relationship is expressed as a bidirectional relationship, i.e. double edges.

Table 7: The numbers of the generated rows/objects in SQL and Neo4j.

|  |  |  |
| --- | --- | --- |
| Table/Object | Rows in SQL | Object in Neo4J |
| Customer | 10000 | 10000 nodes |
| Invoice | 100000 | 100000 nodes |
| Item | 100000 | 100000 nodes |
| Target | 100000 | 100000 nodes |
| Work | 10000 | 10000 nodes |
| Workhours | 100000 | 200000 edges |
| Workinvoice | 1000000 | 2000000 edges |
| Worktarget | 1000000 | 2000000 edges |
| Worktype | 100000 | 100000 nodes |
| UsedItem | 100000 | 200000 edges |
| Pays | - | ? |
| Customertarger | - | ? |
| Previousinvoice | ? | ? |

6.2. Test results

Each query test was executed with 12 iterations. Each query result contains an average time for the query in milliseconds. As Neo4j has outperformed SQL databases in many previous studies, Neo4j was chosen as a reference database where others are compared with. Overall, the inclusion of CALL into a Cypher query made queries faster so Cypher queries with CALL were chosen as the reference queries. If Neo4j version above 4.1 would be used, CALL would be preferred for more performance. Non-indexed tests show percentages slower related to non-indexed Neo4j and indexed tests show percentages slower related to indexed Neo4j. Indexed is the same query for indexed database.

The results of queries are given in Tables 8 and 9. Table 8 contains the results of the short query, long query aggregate query and the query with a defined key. Table 9 contains the result of recursive queries. The table does not contain MySQL because it does not support them. The results are illustrated and analyzed in the following subsections.

Table 8. Query performance of the MySQL, MariaDB and Neo4J

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | MySQL | MariaDB | Neo4J | Neo4J CALL |
| Short Query | | | | |
| Avg | 405 | 209 | 144 | 145 |
| Avg, ind | 413 | 204 | 148 | 146 |
| Long Query | | | | |
| Avg | 4952 | 2197 | 6880 | 1859 |
| Avg, ind | 4901 | 2193 | 8250 | 2931 |
| Aggregate Query | | | | |
| Avg | 235817 | 4282 | 753587 | 229256 |
| Avg, ind | 236875 | 3840 | 957325 | 303365 |
| Aggregate Query with defined key | | | | |
| Avg | 4013818 | 42 | 56 | 58 |
| Avg, ind | 3609818 | 30 | 50 | 59 |

Table 9. Query performance in recursive queries.

|  |  |  |  |
| --- | --- | --- | --- |
|  | MariaDB | Neo4J | Neo4J Optimized |
| Recursive Query, 100 entities | | | |
| Avg | 4473 | 150 | 852 |
| Avg, ind | 1 | 153 | 869 |
| Recursive Query, 1000 entities | | | |
| Avg | 45004 | 1239446 | 3662 |
| Avg, ind | 10 | 1637223 | 5117 |

6.2.1. Short query, price of work

The performance results for the short query are illustrated in Figure 3. From the generated dataset, the query returned 10000 rows/objects. With this query, Neo4j outperforms SQL databases as in several previous studies. MariaDB is the second and it was about 30% slower than Neo4j. MySQL is the slowest. The difference of the relational databases shows how the performance of relational databases has developed during xx. MariaBD is about two times faster that old MySQL. In general indexing plays a minor role in efficiency. Inclusion of CALL does not seem to bring benefits to Neo4j with this query. The strict performance values are given in Table 8.

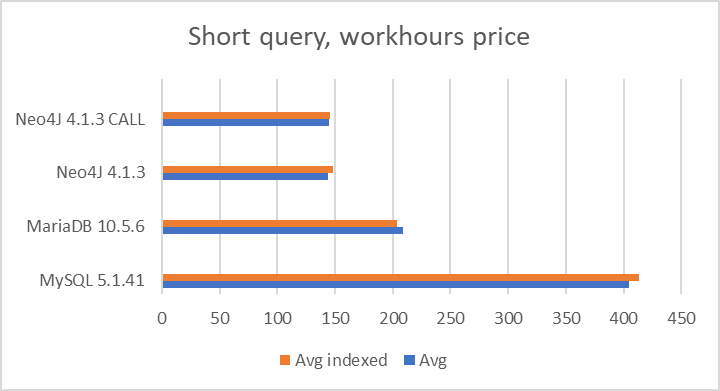


Figure 3. Results of the short query (workhours price)

6.2.2. Long query, price of work with items

Results for the query that queries the price of work is represented in Figure 4. From the generated dataset, the query returned 10000 rows/objects. The mutual difference of relational database is similar than in the short query. Instead, both the relational databases are much faster than Neo4j. MariaDB is about three times faster that Neo4j. However, if the Neo4j query is implement in terms of CALL-function the Neo4j is the fastest. However, difference is not essential. In this query, indexing of relational databases has minor effect. In Neo4j, indexing has negative effect.

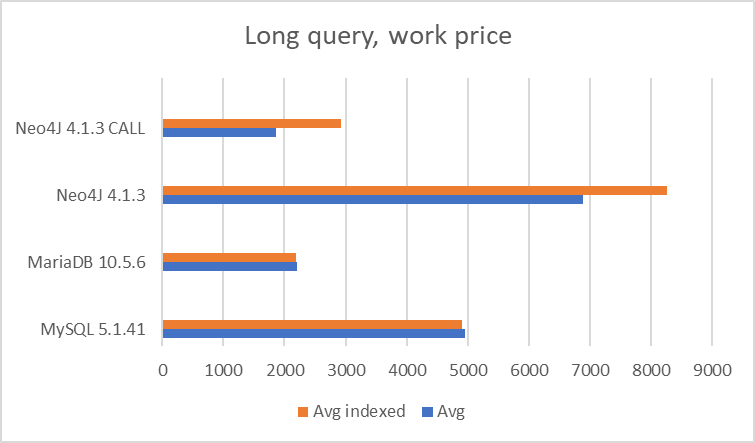


Figure 4: Results of the long query (price of work with items)

6.2.3. Complex query, invoice price

In the aggregation query, radical differences between the database appear. Now, MariaDB is 55 times faster than old MySQL and 176 times faster than basic Neo4j. Inclusion of CALL gives significant performance benefit compared to basic Neo4j, but it is still slower than MySQL. Among Neo4j queries, inclusion of CALL gives essential performance benefit because it is about three times slower without CALL function. Anyway, with CALL function has about the same performance than MySQL. MariaDB is over 50 times faster than any Neo4J run. Figure 5 illustrates the difference on the databases and their settings.

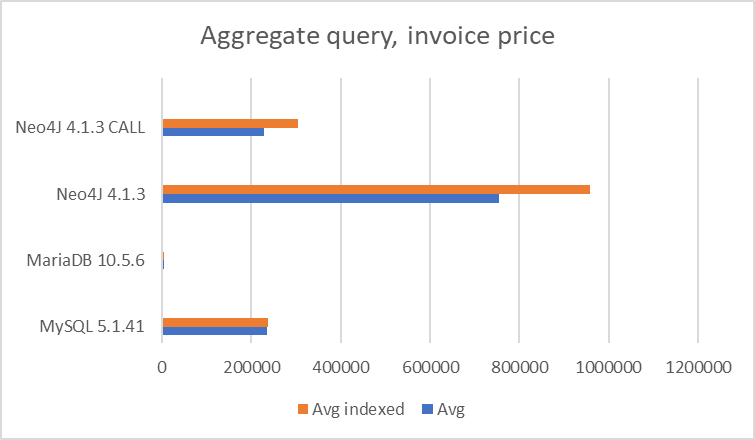


Figure 5: Results of an aggregate query (invoice price)

6.2.4 Query with a defined key, invoice prices for a given customer

Results for the query that gets invoice prices for a given customer are illustrated in Figure 6. From the generated dataset, the query returns 10 rows/objects. MySQL 5.1.41 was left out as the performance was too poor: the query took over one hour on average. In practice, it would be unusable. MariaDB was xx times faster than MySQL. Neo4J performs well. However, the inclusion of CALL does not give performance benefits. Instead, indexing seems to bring improvements with basic Cypher query. With indexing Neo4j finds the customer from the graph faster. Although Neo4j performs well, the best case of MariaDB outperforms the best case of Neo4J by 40%.

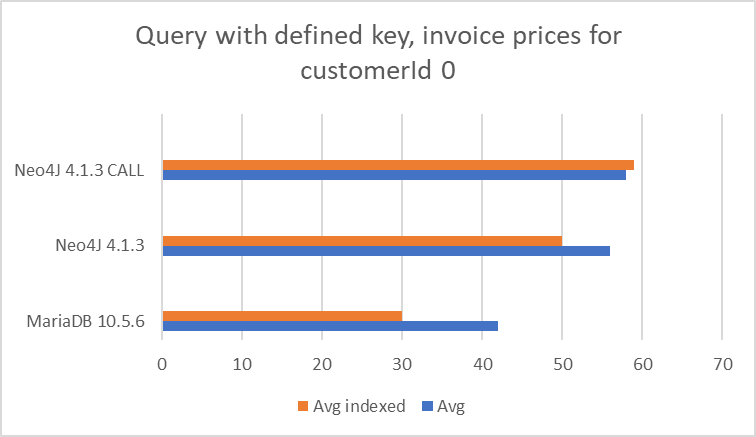


Figure 6: Results of the aggregate query for a single customer.

6.2.5 Recursive query, invoices related to invoice id 100000

The recursive query lists all the sequential invoices related to the invoice with given id. The tests were performed with 100 and 1000 invoices. With 10000 invoices Neo4j performance without optimization was so poor that it would have taken too long to complete. Figure xx presents the results when querying 100 sequential invoices and Figures 7 and 8 present results when querying 1000 invoices.

When not using indexing, Neo4j had the best performance without query optimization with 100 entities. It was about 30 times faster than MariaBD. The optimized query did not improve performance, because the optimized query was about six times slower than without optimization. When indexes are used MariaDB benefits dramatically from indexing. The query takes just 1ms average clearly making MariaDB the best performer. This is 153 times faster than the best result of Neo4j. Therefore, there is no balk in the context of the indexed MariaDB.

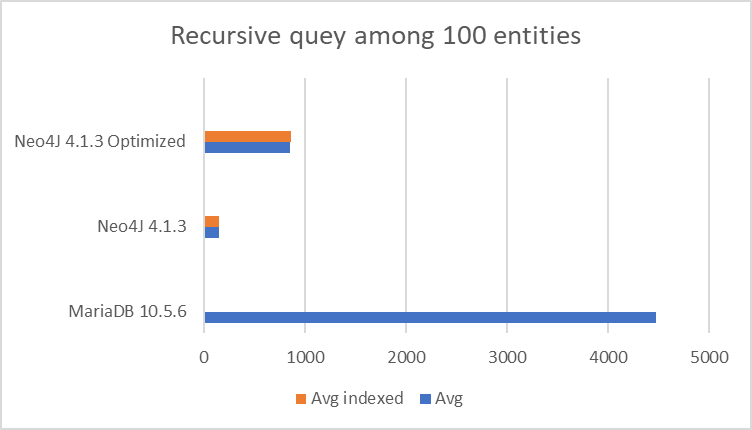


Figure 7. Results of the recursive query among 100 entities.

As the number of relationships is increased to 1000, Neo4j performance drops dramatically. With the optimized query, though, Neo4j becomes much faster outperforming MariaDB. However, as MariaDB benefits from indexing, with indexing the query takes only ten milliseconds on average making it yet again the best performer. The performance is radically faster in comparison with Neo4j. MariaDB was 366 times faster than the best setting of Neo4j and 163722 times faster than the worst setting of Neo4j. Indexing does not improve performance for Neo4j.

Figure 8 illustrates the difference of basic Neo4J case to MariaDB and optimized Neo4j. Figure illustrated the differences between optimized Neo4j and MariaDB. Indexed MariaDB was outstanding, and therefore, there is no balk for it.

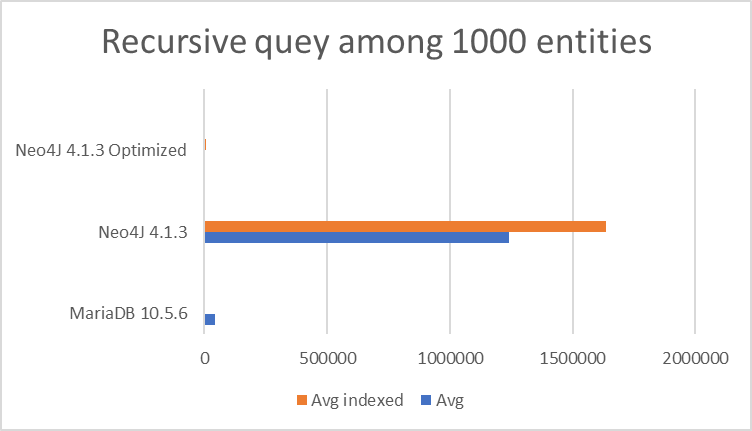


Figure 8: Results of the recursive query among 1000 entities.

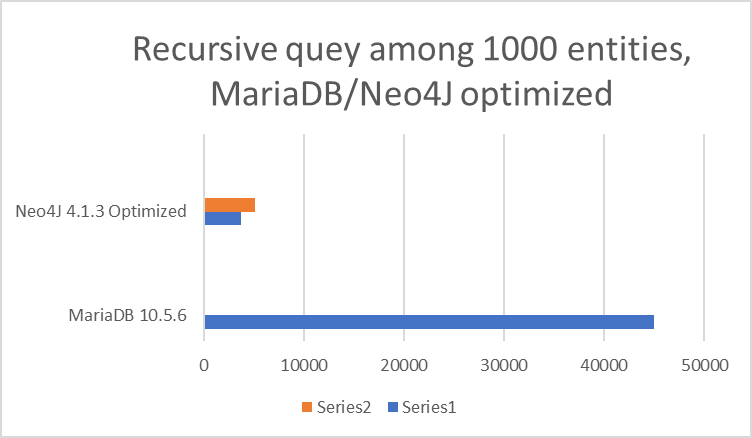


Figure 9: Comparison of the best settings in the recursive query among 1000 entities.

7. Discussion

With the query tests performed, Neo4j was often outperformed by MariaDB. In some tests, Neo4j performed even worse than old MySQL 5.1.41. When comparing Neo4j with MySQL and MariaDB we are comparing a Java program with a C/C++ program. Obviously, the latter can be optimized better. It has to be also taken into consideration that MariaDB indexes primary keys and foreign keys by default. This gives benefits in every query where the table joins are done. Neo4j does not seem to benefit from indexing in many cases. One such case where indexing had benefits was when Neo4j needed to find the starting point from the graph.

The benefit of indexing in MariaDB is a benefit of the traditional relational database model. As the relations with the tables are created when executing the SQL query, indexing the keys becomes beneficial. The graph model does not benefit from such indexing as there are no tables that are joined by keys. Querying a graph database is done by traversing the graph. One of the benefits of the graph model can be seen in recursive query tests. By optimizing the query, performance becomes clearly better and, in this case, even better than SQL database with CTE query. However, with recursive queries, indexing still brings dramatical benefits for SQL database.

With the invoicing database schema used in the present study, calculating the price is done with complex queries. If this database was used in some real case, the usage of table views would probably be preferred to simplify the queries. When it comes to using views, it is also a benefit of the fact that a SQL database outperformed a graph database is a new finding that has not been presented in previous studies. As presented in previous studies, Neo4j often outperformed SQL databases. In study [4] for example, Neo4j outperformed Oracle in various tests using count(\*) queries. In the present study aggregation queries were also used but the result was different. The present study also indicated the benefit of indexing in SQL database in many of the tests. SQL databases seemed to benefit from indexing and in some cases very dramatically. However, Neo4j did not seem to benefit from indexing, apart from when a starting point in the graph was indexed.

In further studies, it is essential to compare other NoSQL databases to a modern relational database. Nowadays a general understanding is that NoSQL databases are more efficient than relational databases in general. We see that in the comparison indexing, optimization and query complexity should be taken into account like in the present study.

8. CONCLUSIONS

The present study compared relational databases (MariaDB, old version of MySQL,) and graph database (Neo4j) by queries having different complexity. The results verify earlier studies where graph databases outperform relational databases in structurally simple dataset and simple queries. However, with more complex queries MariaDB outperformed Neo4j.

The fact that a SQL database outperformed a graph database is a new finding that has not been presented in previous studies. In general, the more complex query, the more efficient a modern relational database (MariaDB) is. In the most complex query, MariaDB was 176 times faster than the used graph database. When optimized the graph database query MariaDB was still 50 times faster than the graph database. In general, optimization and indexing may play very essential role in performance especially in modern relational databases. This appears in the perform of a long recursive query where MariaDB was 366 times faster than the best setting of Neo4j and more than 160000 times faster than the basic setting of Neo4j. The study also showed how much relational databases have been developed during the one and half decadence.

Our general conclusion is that on the basis of tests with simple data set and queries, it cannot be concluded whether one database is more efficient than another. In other words, the efficiency depends on the complexity of data and queries. Furthermore, query optimization and indexing may play important roles. This means that when choosing a database for an application domain, the query need must be analyzed carefully beforehand. The results in the present study show how a relational database is still a strong alternative when it comes to performance compared with a NoSQL graph database.

References

1. Cooper, B. F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; Sears, R. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM Symposium on Cloud Computing* **2010,** 143-154.
2. Difallah, D. E.; Pavlo, A.; Curino, C.; Cudre-Mauroux, P. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment* 7.4 **2013,** 277-288.
3. GitHub. InvoicingDBTestBench repository. https://github.com/homebeach/InvoicingDBTestBench, Accessed 13.12.2020
4. Khan, W.; Ahmad, W.; Luo, B.; Ahmed, E. SQL Database with physical database tuning technique and NoSQL graph database comparisons. *Proceedings of 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*.  **2019**, 110-116.
5. Holzschuher, F.; Peinl. R. Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j. *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. **2013**.
6. Khan, W.; Shahzad, W. Predictive performance comparison analysis of relational & NoSQL graph databases. *International Journal of Advanced Computer Science and Applications* **2017,** 8, 523-530.
7. Vicknair, C.; Macias, M.; Zhao, Z.; Nan, X.; Chen, Y.; Wilkins, D. A comparison of a graph database and a relational database: a data provenance perspective. *Proceedings of the 48th Annual Southeast Regional Conference*. **2010,** 1-6.
8. Batra, S.; Tyagi, C. Comparative analysis of relational and graph databases. *International Journal of Soft Computing and Engineering (IJSCE)* **2012**, 2(2), 509-512.
9. Tongkaw, S.; Tongkaw, A. A comparison of database performance of MariaDB and MySQL with OLTP workload. 2016 *Proceedings of* *IEEE Conference on Open Systems* (ICOS) **2016**.
10. Shalygina, G.; Novikov, B. Implementing common table expressions for MariaDB. *Proceedings of Second Conference on Software Engineering and Information Management* (SEIM-2017) **2017**.
11. Data.world. Names datasets. https://data.world/datasets/names, **2020.**
12. OpenAddresses. *A summary view of OpenAddresses data*. http://results.openaddresses.io, **2020.**
13. Neo4j, Inc. *The Neo4j Cypher Manual* v4.2. CALL {subquery}. https://neo4j.com/docs/cypher-manual/current/clauses/call-subquery, **2020.**
14. [DB-Engines. https://db-engines.com/, Accessed 13.12.2020.

Leavitt, N. Will NoSQL databases live up to their promise?. *Computer* **2010,** 43(2), 12-14.

Győrödi, C.; Dumşe-Burescu, D.; Zmaranda, D.; Győrödi, R.; Gabor, G.; Pecherle, G. Performance analysis of NoSQL and relational databases with CouchDB and MySQL for application’s data storage. *Applied Sciences* **2020,** 10(23), 8524.

Angles, R.; Gutiérrez C. Survey of graph database models. *ACM Comput. Surv.* **2008,** 40(1), 1-39.

Niemi, T.; Junkkari, M.; Järvelin K. Concept-based query language approach to enterprise information systems. *Enterprise Information Systems* **2014,** 8(1), 26-66.

Aufaure-Portier, M.-A.; Trépied C. A survey of query languages for geographic information systems. *IDS* **1996,** 3

Leser, U. A query language for biological networks. *Bioinformatics* **2005,** 21(2), ii33–ii39.

Hardwick, M.; Samaras, G.; Spooner, D. Evaluating recursive queries in CAD using an extended projection function, *IEEE Third International Conference on Data Engineering* **1987,** 138-148.