Performance of Graph and Relational Databases in Complex Queries

Petri Kotiranta, Marko Junkkari, Jyrki Nummenmaa

Tampere University, Finland

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 a relational database (MariaDB). The outcome is that although Neo4j is more efficient in simple queries, MariaDB is essentially more efficient when the complexity of queries increases.

Keywords: Query performance, Performance comparison, Graph database, Relational database, Neo4j, MariaDB. MySQL, Enterprise information systems, SQL, Cypher

1. Introduction

Performance has been one of the motivations to use NoSQL databases instead of traditional SQL databases. 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, 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.

There are already several studies where the performance of graph databases, especially Neo4j, have been compared with the traditional SQL databases, see e.g. [4,5,6,7,8]. Often the results show better performance of graph databases. However, those studies are mostly based on quite simple queries. In the present study, we investigate the performance of databases in the situation where query complexity grows. Complex queries are substantial for example in enterprise information systems where data are strongly structured, and queries collect data from various record (tables in SQL databases.) As an example of enterprise related data, we use a simulated structured database containing data needed to compose an invoice piece by piece. We also investigate what are the effects of query optimization and indexing in differed databases. Thus, we aim to find the most efficient setting in the query formulation of complex queries.

In enterprise use, performance is often crucial. Thus, it is very important to take into consideration when choosing the database model. Our main question is to find out how an SQL database and a graph database performance compare when query complexity grows. Recursive query performance and the effect of indexing on the performance are also studied. The databases compared in the present study are MySQL 5.1.41, MariaDB 10.5.6 and Neo4j 4.1.3. Instead of using existing benchmarks such as [1] or [2], a dedicated test bench was implemented for the present study. 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.

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

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. Invoicing database

The test database is a general example of an invoicing database. 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 database has 10 tables. The basic tables are customer, invoice, target, work, worktype and item. These tables contain the customer information, customer's invoices, the target 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 tables is stored into M:N tables worktarget, workinvoice, useditem and workhours. Figure 1 shows database in relational format.

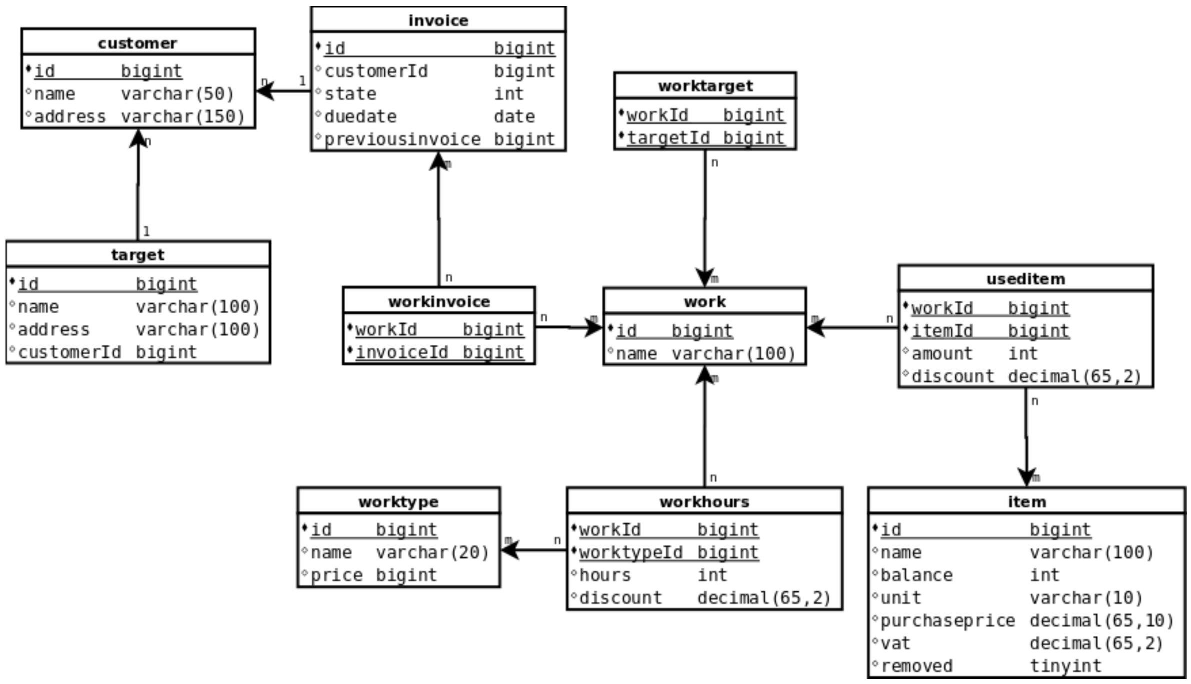

Figure 1: Invoicing database in relational format.

In graph format, edges are used to represent the relationships. For N:M relationships, bidirectional edges are used. The tables customer, invoice, target, work, and worktype are represented as nodes. The edges between the nodes are PAYS between the customer and invoice, CUSTOMER_TARGET between the customer and target, WORK_TARGET between work and target, WORK_INVOICE between work and invoice, WORKHOURS between the work and worktype and USED_ITEM between work and item. Figure 2 shows database in graph format.

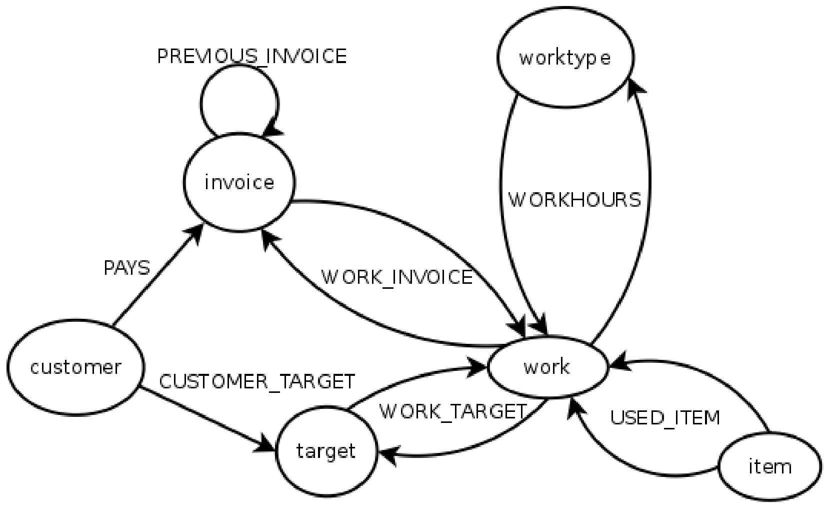

Figure 2: Invoicing database in graph format.

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 relational 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. There is also a recursive query test.

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 looked into 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.

Type	Query

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`

Table 2: Price of work query in SQL and Cypher

5.2. 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.

Type	Query

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`

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

5.3. 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.

Type	Query

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.purchasep`
	`rice)) 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.purchasep`
	`rice)) as workPrice`
	`}`
	`RETURN inv, SUM(workPrice) as invoicePrice`

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

5.4. 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.

Type	Query

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.purchasepr`
	`ice)) 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.purchasepr`
	`ice)) as workPrice`
	`}`
	`RETURN c, inv, SUM(workPrice) as invoicePrice`

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

5.4. 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.

Type	Query

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)

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

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.

We wanted to see how the old MySQL compares with the new versions of MariaDB and Neo4j and whether it was possible to repeat the previous results such as in [8]. MariaDB was chosen because MySQL is nowadays often replaced with MariaDB. There are various reasons for this, including more open development compared with modern MySQL. There are also not yet many studies about MariaDB yet, especially comparing it to a graph database.

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. 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.

Table/Object	Rows SQL	Objects Neo4J
Customer	10000	10000
Invoice	100000	100000
Item	100000	100000
Target	100000	100000
Work	10000	10000
Workhours	100000	200000
Workinvoice	1000000	2000000
Worktarget	1000000	2000000
Worktype	100000	100000
Useditem	100000	200000

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

6.2. Test results

Each query test was executed with 12 iterations. Each query result contains an average time for the query in milliseconds and coefficient of variation (CV) of the result list. 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 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.

6.2.1. Short query, price of work

Results for the query that queries the price of work are given in Table 8. From the generated dataset, the query returned 10000 rows/objects. With this query, Neo4j outperforms SQL databases as in several previous studies. MySQL is the slowest and MariaDB the second. Inclusion of CALL does not seem to bring benefits to Neo4j with this query. Indexing does not seem to bring much benefits either. CV values do not have a remarkable difference in these results.

Database	Avg	CV	Slower than Neo4j 4.1.3 CALL	Avg indexed	CV indexed	Slower than Neo4j 4.1.3 CALL indexed
MySQL 5.1.41	405	2,56 %	64 %	413	0,71 %	65 %
MariaDB 10.5.6	209	8,90 %	31 %	204	7,13 %	28 %
Neo4j 4.1.3	144	1,15 %	-1 %	148	4,97 %	1 %
Neo4j 4.1.3 CALL	145	1,07 %	-	146	1,63 %	-

Table 8: Results for the query for the price of work.

6.2.2. Long query, price of work with items

Results for the query that queries the price of work can be found in Table 9. From the generated dataset, the query returned 10000 rows/objects. With this query, Neo4j outperforms SQL databases as in several previous studies. MySQL is the slowest and MariaDB the second. Inclusion of CALL does not seem to bring benefits to Neo4j with this query. Indexing does not seem to bring benefits either. The CV values of MySQL and MariaDB results are very low, Neo4j CV values being significantly higher.

Database	Avg	CV	Slower than Neo4j 4.1.3 CALL	Avg indexed	CV indexed	Slower than Neo4j 4.1.3 CALL indexed
MySQL 5.1.41	4952	0,63 %	62 %	4901	0,34 %	40 %
MariaDB 10.5.6	2197	0,43 %	15 %	2193	0,23 %	-34 %
Neo4j 4.1.3	6880	18,82 %	73 %	8250	21,79 %	64 %
Neo4j 4.1.3 CALL	1859	8,67 %	-	2931	13,53 %	-

Table 9: Results for price of work with items query.

6.2.3. Complex query, invoice price

Table 10 shows results for the query that calculates invoice prices. From the generated dataset, the query returns 100000 rows/objects. Without CALL included in query Neo4j performs even worse than old MySQL 5.1.41. Inclusion of CALL gives significant performance benefit. However, MariaDB is still radically faster. Indexing gives also some performance benefits in MariaDB. However, with other databases it does not seem to improve performance. The CV values of MariaDB and indexed Neo4j with CALL are significantly higher than others, which are quite low.

Database	Avg	CV	Slower than Neo4j 4.1.3 CALL	Avg indexed	CV indexed	Slower than Neo4j 4.1.3 CALL indexed
MySQL 5.1.41	235817	1,03 %	3 %	236875	0,76 %	-28 %
MariaDB 10.5.6	4282	24,45 %	-5254 %	3840	23,06 %	-7800 %
Neo4j 4.1.3	753587	0,73 %	70 %	957325	1,35 %	68 %
Neo4j 4.1.3 CALL	229256	1,66 %	-	303365	19,61 %	-

Table 10: Results for the query for the invoice prices.

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 given in Table 11. 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. With Neo4j, the inclusion of CALL does not give performance benefits. However, indexing seems to bring improvements with basic Cypher query. With indexing Neo4j finds the customer 0 from the graph faster. Although Neo4j performs well, MariaDB outperforms it by a small margin. The CV values of these results are overall high compared to previous results ones non-indexed MariaDB and Neo4j being quite low.

Database	Avg	CV	Slower than Neo4j 4.1.3 CALL	Avg indexed	CV indexed	Slower than Neo4j 4.1.3 CALL indexed
MariaDB 10.5.6	42	1,17 %	-38 %	30	18,02 %	-97 %
Neo4j 4.1.3	56	25,17 %	-4 %	50	17,91 %	-18 %
Neo4j 4.1.3 CALL	58	1,63 %	-	59	12,09 %	-

Table 11: Results for the query for the invoice prices for a defined 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. Table 12 presents the results when querying 100 sequential invoices and Table 13 presents results when querying 1000 invoices. The CV value of MariaDB is significantly higher because of its radically different result. Average of 1 divided by 0.4 standard deviation yields this result. Without the index, MariaDB has quite low CV, others having a notably higher value.

Database	Avg	CV	Slower than Neo4j 4.1.3 Optimized	Avg indexed	CV indexed	Slower than Neo4j 4.1.3 Optimized indexed
MariaDB 10.5.6	4473	0,33 %	81 %	1	40,00 %	-86800 %
Neo4j 4.1.3	150	12,00 %	-468 %	153	4,33 %	-468 %
Neo4j 4.1.3 Optimized	852	9,63 %	-	869	12,44 %	-

Table 12: Results for the recursive query with 100 rows/objects.

With 100 invoices, Neo4j seems to have the best performance without query optimization. The optimized query does not seem to improve performance. However, when indexes are used MariaDB benefits dramatically from indexing. The query takes just 1ms average clearly making MariaDB the best performer. Indexing does not improve performance for Neo4j. There is a variation in CV values, non-indexed MariaDB having the lowest value. Most variation was in the results of optimized Neo4j query results.

Database	Avg	CV	Slower than Neo4j 4.1.3 Optimized	Avg indexed	CV indexed	Slower than Neo4j 4.1.3 Optimized indexed
MariaDB 10.5.6	45004	0,46 %	92 %	10	6,40 %	-51070 %
Neo4j 4.1.3	1239446	12,09 %	100 %	1637223	16,71 %	100 %
Neo4j 4.1.3 Optimized	3662	27,24 %	-	5117	5,71 %	-

Table 13: Results for the recursive query with 1000 rows/objects.

As the number of invoices 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 is 51070 % faster. Indexing does not improve performance for Neo4j.

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 SQL databases over Neo4j as at the time of writing this article Neo4j does not have an exact equivalent of such feature as views in SQL databases.

8. Conclusions

The present study compared MySQL 5.1.41, MariaDB 10.5.6 and Neo4j 4.1.3 with various query tests related to an invoicing database. A simulated enterprise dataset was used as test data. The query tests were performed using a Java program developed for the present study. The query tests included relational queries with increasing complexity and recursive queries. With the simpler relational queries, Neo4j had the best performance which correlates with previous results shown in [7] and [10]. 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. 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.

Overall, MariaDB is the clear winner in the present study when it comes to performance with more complex queries. Especially in the query test for querying the invoice prices, MariaDB shows its performance. Indexing also gives clear benefits in MariaDB. Especially with recursive queries the improvement was dramatical. 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* (pp. 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." *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. (pp. 110-116). IEEE, 2019.

[5] Holzschuher, Florian, and René Peinl. "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, Wisal, and Waseem Shahzad. "Predictive Performance Comparison Analysis of Relational & NoSQL Graph Databases." International Journal of Advanced Computer Science and Applications 8 (2017): 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*. (pp. 1-6). 2010.

[8] Batra, Shalini, and Charu Tyagi. "Comparative analysis of relational and graph databases." *International Journal of Soft Computing and Engineering (IJSCE)* 2.2 (2012): 509-512.

[9] Tongkaw, Sasalak, and Aumnat Tongkaw. "A comparison of database performance of MariaDB and MySQL with OLTP workload." 2016 IEEE conference on open systems (ICOS). IEEE, 2016.

[10] Shalygina, Galina, and Boris Novikov. "Implementing common table expressions for MariaDB." Second Conference on Software Engineering and Information Management (SEIM-2017) (full papers). 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