# Introduction

One of the benefits of using NoSQL databases has usually been performance over traditional SQL databases. With an appropriate use case, NoSQL databases might offer significant performance benefits. Key-value based queries is one typical use case when NoSQL can give better performance. A graph database is one of the NoSQL database types. As the graph model consists of nodes and edges, it should be theoretically more optimal for relational queries. While in SQL database multiple tables have to be joined for a relational query, in graph databases relational information can be queried by navigating through the graph.

This study compares the relational query performance of SQL database with MariaDB version and graph database Neo4J. The databases are tested with various queries including a relation, an aggregation, a key-based and a recursive query. There are already several studies where graph databases, especially Neo4J, have been compared to traditional SQL databases. In many cases the results often show better performance in favour of graph database.

MySQL version 5.1.41 that was used in previous study [graph6] is also included into tests in this study. We were partially able to repeat similar results when comparing Neo4J with MySQL, showing performance benefits of Neo4J. However the inclusion of MariaDB to the tests gives interesting results indicating how SQL databases can also still offer good performance.

In addition of including MariaDB, this study focuses on a use case with a practical approach. A general example of invoicing database is used for the tests. The test queries aim to be practical examples of the usage of such database. A diverse set of queries is used to query invoice price including multiple joins, calculations and aggregations. A recursive query is also included to query sequential invoices. In this study it will be seen how the tested databases perform with a practical use case scenario.

# Related study

Various studies about graph database performance exist including [graph1], [graph2], [graph3] and [graph4]. [graph1] conducts the qualitative and performance comparison of 12 databases which are capable of storing and querying graph. Algorithms single source shortest path (SSSP), connected components (SV) and PageRank (PR) are computed on these databases. Besides these, update to graph per second is measured on databases. All the databases successfully completed the tests with small graph (32K vertices and 256K unidirected egdes) and medium graph (1M vertices and 8M unidirected edges). Only 5 of the original 12 databases completed the tests with large graph (16M vertices and 128M unidirected edges). The top five performers in these tests are STINGER, MTGL, Boost, Graph and NetworkX. Neo4J is one of the compared databases, but does not fit into top performers in these tests.

In [graph2] Neo4J enterprise version 1.9.1 query performance is tested with two datasets and two kinds of queries. The queries were executed 11 times and the first time was thrown away as it warms up Neo4J caches. The dataset consists author nodes and paper nodes. The relationship between these nodes is called ref. The first query is “find coAuthor of a paper written by an author” and the second query is “find a paper written by an author”. The queries are done three ways. With Java APi, with TraverserFrameWork and with Cypher. This study shows the clear differences between the three ways. Both Core Java API and TraverserFrameWork offer faster response time compared to Cypher. Based on this study, Core Java API should be chosen for shorter response times. Cypher’s strength is it’s easiness and maintainability. TraverseFrameWork is a good compromise between the two.

[graph3] compares tuned Oracle 11g and Neo4J 3.03 Community Edition.A healtcare related dataset is used including data about patients, medicines and medical staff. Performance of the databases is evaluated with ten different count(\*) queries. Many of the queries also do some table joins. Physical database tuning technique called tablespaces is used for Oracle. The same databases were also compared without physical database tuning in [graph7]. 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.

In [graph4] Neo4J version 1.8 performance is tested with different backend solutions. Neo4J is benchmarked as embedded with native object access, as a dedicated server trough 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 is also included with Java Persistence API based backend. Queries are done using Cypher, Gremlin and SQL query languages. The test data consist of people related data. Relational test queries are executed such as friends of friends.

When database gets larger advantages of Neo4J over MySQL become more prevalent. Neo4J performance stays nearly constant when MySQL performance drops by factors 5 and 7-9. Both Neo4J query languages Gremlin and Cypher have performance benefits over MySQL with JPA. Gremlin and Cypher were also compared Gremling having performance benefits in certain queries. When comparing Cypher with Neo4J native object access Cypher was between 10% and 200% worse.

There also exist previous studies that compare Neo4J and MySQL [graph5] [graph6]. Both of these articles demonstrate Neo4J performance over MySQL. Article [graph5] is from the year 2010 and compares MySQL Community Server version 5.1.42 and Neo4J version 1.0-b11. In the article the graph database is stored into a relational database as nodes and edges. Three types of structural and three types of data queries are made. First structural query finds all the orphan nodes and the two other ones traverse the graph in depths of 4 and 128. The data queries are count queries counting nodes with certain payload. Neo4J performed better in structural queries. However with 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 article [graph5] is referenced in articles [graph1], [graph3], [graph4] and [graph7].

Article [graph6] is from the year 2012 and compares MySQL version 5.1.41 and Neo4J Community version 1.6. A schema with tables user, friends, fav\_movies and actors is used for testing. The databases are tested 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 are done with 100 and 500 objects. Neo4J has 2-5 times lower query times with 100 objects data set and 15-30 times lower in 500 objects data set. The article [graph6] is referenced in articles [graph2] and [graph4].

There exist previous performance studies where MariaDB is involved. In [MariaDB1] the performance of MariaDB 10.0.21 and MySQL 5.6 is compared using Sysbench and OLTP [OLTP] software. OLTP-Simple and OLTP-Seats workloads are used. Both databases consumed the same number of resources. However, when increasing number of threads in OLTP-Simple and number of workers in OLTP-Seats MySQL is clearly more effective outperforming MariaDB. Common Table Expression capabilities of MariaDB is studied in [MariaDBCTE] along with PostgresSQL. This study showed that Postgres has better results when only few steps of recusion is needed. However, MariaDB is better choice for a long recursive process on a huge amount of data.

In [MariaDB2] MariaDB is compared with Cassandra, Mongo, PostgreSQL, CockroachDB and Elasticsearch. The databases are tested with Last.fm Million Song Dataset. The test data is in JSON format. Both insertion and querying the dataset is tested. MariaDB has mediocre performance in insertions. In queries MariaDB is clearly the slowest.

MariaDB has also been previously used in context of studying Non-Volatile Memory (NVM) performance in [MariaDBNVM1] and [MariaDBNVM2]. There is also a study about A high availability MariaDB Galera Cluster [MariaDBHA].

# Invoicing database

The test database is a general example of an invoicing database. One of the most important use cases is 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, listing of each work, 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.

Kuva, joka sisältää kohteen näyttökuva, tie

Kuvaus luotu automaattisesti

Figure 1: Invoicing database in relational format (obsolete)



Figure 1: Invoicing database in relational format (Dia diagram) (obsolete)

In graph format edges are used to represent the relationships. For N:M relationships, bidirectional edges are used. Customer, invoice, target, work, worktype are represented as nodes. The edges between the nodes are PAYS between customer and invoice, CUSTOMER\_TARGET between customer and target, WORK\_TARGET between work and target, WORK\_INVOICE between work and invoice, WORKHOURS between work and worktype and USED\_ITEM between work and item.Kuva, joka sisältää kohteen pieni, varuste, kaulakoru

Kuvaus luotu automaattisesti

Figure 2: Invoicing database in graph format (from Neo4J)



Figure 2: Invoicing database in graph format (Dia diagram)

# Test program

The test data is generated using a Java program. The entire source code for the program is available in GitHub [graafitietokantaprojekti]. The program uses sample data that is based on openly available name and address data sets [1], [2]. The sample data is used when generating customer and target names. The generation process is divided into three parts. Items and work types should be generated first, then work and customer data. The Java program has threaded classes for each part. Multiple threads can be used to insert data as well as coefficients for related data.

When generating work data, amount of the related worktypes and items can be defined by setting worktype coefficient and item coefficient. When generating customer data, the amount of the related invoices, targets and work can be defined by setting invoice coefficient, target coefficient and work coefficient. When generating work and customers, the given amount of work and customers are generated. For each work, the given coefficient of relations to worktypes and invoices are generated. For each customer the given number of invoices and targets are generated. The generator will also generate workinvoice and worktarget relationships based on the given coefficient.

# 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 test queries aim to query something related to a practical use case as well as finding the differences of the databases in performance.

Calculating the invoice price is one of the most important use cases. 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 long query are the subqueries for calculating this price. The aggregation query will calculate the whole price. A query with defined key will get the invoice price for certain customer. The recursive query will query all the interrelated invoices. One practical example is that customer has not paid the whole invoice and there will be additional invoices based on the same invoice.

## Query optimization

In addition to just testing databases with the queries, there was also interested to find out how to make the query performance better. Indexing is one of the ways making queries faster. It has to be taken into consideration when comparing MySQL, MariaDB and Neo4J that both MySQL and MariaDB index primary key and foreign key by default. Neo4J does not create indexes for properties by default. The effect of indexing is also different when comparing SQL database with 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 database benefits of indexing of primary key and foreign key. In graph database we are traversing the graph when querying data. Thus, it does not benefit indexing 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. Id in customer and invoice tables are indexed by default in MySQL and MariaDB and thus 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 [Neo4JCALL]. The CALL clause makes it possible to execute subqueries in other queries. It is like a function that gets input parameters from main query and returns some values back. Subquery is executed for each incoming input row from calling query from main query. CALL has been supported from Neo4J onwards. In this study Cypher queries with and without CALL are used in order for backwards compatibility. This way it is also possible to see how much CALL subquery improves the query performance.

## Short query, price of work

Short query query calculates the price of work. 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 discount for the prices and the discount is included into calculation. This query shows how databases perform with fairly a simple query. Table A shows the queries for 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 |  |
|  |
|  |
|  |
|  |
|  |

## Long query, price of work with items

This query 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, the longer relational query is needed. With this query it is possible to see how databases perform when more relations and calculations are included into the query. Item purchase price is a floating-point number so this will add more challenges to the calculations. Table B shows the queries for 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 |  |
|  |
|  |
|  |
|  |
|  |

## 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 invoice id. This is an important query as one of the most important use cases is to calculate invoice price for the customer. This is one of the heaviest queries and as such it is useful to see the performance differences when executing a complex query. Table C shows the queries for calculating 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.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 |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

## Query with defined key, invoice prices for customer with id 0

There is often interest 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. Customer information is also included. This query is the most complex of the tested queries. From 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 D shows 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.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 |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

## Recursive query, invoices related to invoice id 100000

The recursive query gets all the sequential invoices related to given invoice id. A common use case for this kind of query is when several related bills are sent to a customer. The query is useful to test the recursive query capabilities of the databases. In SQL Common Table Expressions is used to make the query. In Cypher there is a way to optimize the 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 nodes the optimized query returns a list of nodes. However it still returns similar results and as such it is a relevant query. Table E shows 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) |  |

# Query tests

## Test settings

The query tests were performed with MacBook Pro Laptop that has 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 driver version 2.7 and Neo4J driver version 4.1.1 were used. A dataset was generated using the test program. Table F shows the number of rows/objects generated for the dataset. A double the amount of useditem, workhours, workinvoice and worktarget relationships were generated into Neo4J as in graph form N:M relationship is expressed as a bidirectional relationship.



## Test results

Each query test was executed with 12 iterations. The test program collects the results into a list structure. The program removes biggest and the smallest number from the list and calculates an average and a standard deviation from the rest of the results. Each query result contains average time for query in milliseconds and standard deviation of the result list. As Neo4J has outperformed SQL in many previous studies, there a percentage related to Neo4J performance is shown. Non-indexed tests show percentage slower related to non-indexed Neo4J and indexed tests show percentage slower related to indexed Neo4J.

* + 1. Short query, price of work

Results for a short query can be found in table G.. Neo4J is clearly faster that MySQL as in previous studies. However, MariaDB performs the best. Inclusion of CALL into query does not seem to bring benefits in Neo4J with this query. Indexing does not also seem to bring benefits with this query.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Database | Avg | Sd | Slower than Neo4J 4.1.3 CALL | Avg indexed | Sd indexed | Slower than Neo4J 4.1.3 CALL indexed |
| MySQL 5.1.41 | 405 | 10,39 | 64 % | 413 | 2,93 | 65 % |
|  |
| MariaDB 10.5.6 | 209 | 18,59 | 31 % | 204 | 14,55 | 28 % |  |
|  |
| Neo4J 4.1.3 | 144 | 1,66 | -1 % | 148 | 7,35 | 1 % |  |
|  |
| Neo4J 4.1.3 CALL | 145 | 1,55 | 0 % | 146 | 2,37 | 0 % |  |
|  |

* + 1. Long query, price of work with items

Results for a long query can be found in table H. When query gets more complex Neo4J does no longer perform so efficiently. Even MySQL 5.1.41 is faster. However when including CALL to the Cypher query Neo4J has the best performance. Indexing does not seem to improve performance with this query either.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Database | Avg | Sd | Slower than Neo4J 4.1.3 CALL | Avg indexed | Sd indexed | Slower than Neo4J 4.1.3 CALL indexed |
| MySQL 5.1.41 | 4952 | 31,28 | 62 % | 4901 | 16,48 | 40 % |
|  |
| MariaDB 10.5.6 | 2197 | 9,43 | 15 % | 2193 | 5,14 | -34 % |  |
|  |
| Neo4J 4.1.3 | 6880 | 1294,72 | 73 % | 8250 | 1797,56 | 64 % |  |
|  |
| Neo4J 4.1.3 CALL | 1859 | 161,23 | 0 % | 2931 | 396,45 | 0 % |  |
|  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

* + 1. Complex query, invoice price

Table J shows results for the query that calculates invoice prices. Without CALL included into query Neo4J peforms even worse than old MySQL 5.1.41. Inclusion of CALL gives significant performance benefit. However MariaDB clearly outperforms Neo4J. Indexing gives also performance benefits in MairaDB. However with other databases it does not seem to improve performance.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Database | Avg | Sd | Slower than Neo4J 4.1.3 CALL | Avg indexed | Sd indexed | Slower than Neo4J 4.1.3 CALL indexed |
| MySQL 5.1.41 | 235817 | 2419,67 | 3 % | 236875 | 1801,43 | -28 % |
|  |
| MariaDB 10.5.6 | 4282 | 1047,05 | -5254 % | 3840 | 885,57 | -7800 % |  |
|  |
| Neo4J 4.1.3 | 753587 | 5517,32 | 70 % | 957325 | 12929,71 | 68 % |  |
|  |
| Neo4J 4.1.3 CALL | 229256 | 3805,37 | 0 % | 303365 | 59489,12 | 0 % |  |
|  |

* + 1. Query with a defined key, invoice prices for customer 0

Results for the query that gets invoice prices for a customer 0 can be found in table I. This query is the most complex. However, it gives just ten results as it is limited to specific customer. MySQL 5.1.41 has the worst performance with this query. In practical use it would be unusable. With Neo4J inclusion of CALL does not give performance benefits. However, indexing seems to bring improvements with Neo4J unlike in other queries. With indexing Neo4J finds the customer 0 from graph faster. Indexing seems to have benefits with SQL databases aswell. Altough Neo4J performs well, MariaDB outperforms it by a small margin.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Database | Avg | Sd | Slower than Neo4J 4.1.3 CALL | Avg indexed | Sd indexed | Slower than Neo4J 4.1.3 CALL indexed |
| MySQL 5.1.41 | 4013818 | 542004,37 | 100 % | 3609818 | 65871,35 | 100 % |
|  |
| MariaDB 10.5.6 | 42 | 0,49 | -38 % | 30 | 5,41 | -97 % |  |
|  |
| Neo4J 4.1.3 | 56 | 14,09 | -4 % | 50 | 8,96 | -18 % |  |
|  |
| Neo4J 4.1.3 CALL | 58 | 0,94 | 0 % | 59 | 7,13 | 1. % |  |

6.2.5 Recursive query, invoices related to invoice id 100000

The recursive query lists all the sequential invoices related to invoice with given id. The tests were performed with different amounts of invoices. Table K shows the results when querying 100 sequential invoices and table L shows results when querying 1000 invoices.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Database | Avg | SD | Slower than Neo4J 4.1.3 Optimized | Avg indexed | SD indexed | Slower than Neo4J 4.1.3 Optimized indexed |
| MariaDB 10.5.6 | 2493 | 12,28 | 82 % | 1 | 0 | -103100 % |
|  |
| Neo4J 4.1.3 | 97 | 17,2 | -351 % | 127 | 24,95 | -713 % |  |
|  |
| Neo4J 4.1.3 Optimized | 437 | 39,14 | 0 % | 1032 | 110,34 | 0 % |  |
|  |

With 100 invoices Neo4J seems to have the best performance when no indexes are used. Optimized query does not seem to improve performance with this query. However, when indexes are used MariaDB clearly benefits dramatically from indexing. The query takes just 1ms average clearly making MariDB the best performer. Indexing does not improve performance for Neo4J.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Database | Avg | SD | Slower than Neo4J 4.1.3 Optimized | Avg indexed | SD indexed | Slower than Neo4J 4.1.3 Optimized indexed |
| MariaDB 10.5.6 | 37242 | 5108,36 | 90 % | 10 | 0,5 | -44950 % |
|  |
| Neo4J 4.1.3 | 1045119 | 339434,83 | 100 % | 990249 | 184368,55 | 100 % |  |
|  |
| Neo4J 4.1.3 Optimized | 3789 | 1009,68 | 0 % | 4505 | 174,16 | 0 % |  |
|  |

As the number of invoices was increased to 1000, Neo4J performance drops dramatically. With optimized query though, Neo4J becomes much faster outperforming MariaDB. However as MariaDB benefits from indexing, with indexing the query takes only 10 milliseconds making it yet again the best performer. Indexing does not improve performance for Neo4J.

Recursive query test with 10000 invoices was also considered. However query tests with Neo4J were so slow that the test would have taken too long time to complete.

# Analysis

With the query tests performed Neo4J is often outperformed by MariaDB in many cases. In some tests Neo4J performs even worse than old MySQL 5.1.41. When comparing Neo4J to MariaDB we are comparing a Java program with C/C++ program. Obviously, the latter can be optimized better. MariaDB clearly seems to benefit from indexing while Neo4J only seems to benefit from it when the starting point in the graph is indexed. Otherwise indexing can even make the performance worser. It has to be also taken into consideration that MariaDB indexes primary keys and foreign keys by default. This gives benefits in every query.

The benefit of indexing in MariaDB can also seen as the 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 this study, calculating price is done with complex queries. If this database was used in some real case, usage table views would be probably preferred to simplify the queries. When it comes to using views it is also a benefit of SQL databases over Neo4J as the time writing this article Neo4J does not have an exact equivalent for such feature as views in SQL databases.

# Conclusions

This study compared MySQL 5.1.41, MariaDB 10.5.6 and Neo4J 4.1.3 with various query tests related to invoicing database. The data was generated, and the query tests were performed using a Java program developed for this study. The query tests included relational queries with various complexity and recursive queries. With a simple query Neo4J had good performance as shown in [graph5] and [graph6]. However with more complex queries MariaDB outperformed Neo4J in many cases. The data and the queries in this study targeted to be more practical than in previous studies simulating a real-life use case.

This study 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, the tested database Neo4J seemed often not to benefit from indexing. In fact, indexes often made the performance much worse. However, Neo4J benefited from indexing when a starting point in the graph was indexed.

Overall MariaDB is the clear winner in this study when it comes to performance. Especially the query test for querying invoice price when complex query is excited that selects a big datasets MariaDB shows its performance. Indexing also gives clear benefits in MariaDB. Especially with recursive queries the improvement was dramatical. The results in this study show how a relational database is still a strong alternative when it comes to performance when compared to NoSQL graph database.

[1] https://data.world

[2] <http://results.openaddresses.io/>

[graph1], McColl, Robert Campbell, et al. "A performance evaluation of open source graph databases." *Proceedings of the first workshop on Parallel programming for analytics applications*. 2014.

[graph2], Huang, Hongcheng, and Ziyu Dong. "Research on architecture and query performance based on distributed graph database neo4j." *2013 3rd International Conference on Consumer Electronics, Communications and Networks*. IEEE, 2013.

[graph3], Khan, Wisal, et al. "SQL Database with physical database tuning technique and NoSQL graph database comparisons." *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. IEEE, 2019.

[graph4] 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.

[graph5] Vicknair, Chad, et al. "A comparison of a graph database and a relational database: a data provenance perspective." *Proceedings of the 48th annual Southeast regional conference*. 2010.

[graph6] 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.

[graph7 Khan, Wisal, and Waseem Shahzad. "Predictive Performance Comparison Analysis of Relational & NoSQL Graph Databases." *Int. J. Adv. Comput. Sci. Appl* 8 (2017): 523-530.

[graafitietokantaprojekti]. GitHub, graafitietokantaprojekti <https://github.com/homebeach/graafitietokantaprojekti>

[MariaDB1] 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.

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

[MariaDBNVM1] Das, Dhananjoy, et al. "{NVM} Compression—Hybrid Flash-Aware Application Level Compression." 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads ({INFLOW} 14). 2014.

[MariaDBNVM2] Lindstrom, Jan, et al. "NVM aware MariaDB database system." 2015 IEEE Non-Volatile Memory System and Applications Symposium (NVMSA). IEEE, 2015.

[MariaDBHA] Aditya, Bagus, and Tutun Juhana. "A high availability (HA) MariaDB Galera Cluster across data center with optimized WRR scheduling algorithm of LVS-TUN." 2015 9th International Conference on Telecommunication Systems Services and Applications (TSSA). IEEE, 2015.

[OLTP] Difallah, Djellel Eddine, et al. "Oltp-bench: An extensible testbed for benchmarking relational databases." *Proceedings of the VLDB Endowment* 7.4 (2013): 277-288.

[Neo4JCALL] https://neo4j.com/docs/cypher-manual/current/clauses/call-subquery/