# 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 studies [graph5] and [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.

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 [graph5] is referenced in articles [graph1], [graph3], [graph4] and [graph7]. The article [graph6] is referenced in articles [graph2] and [graph4]. As these articles are much referenced in the relevant studies and MySQL version 5.1.41 is used in both of these articles, the same version was included into tests of this article to see how the results compare with the previous studies.

# 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)

# Data generation

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 amount 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 aswell 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 work for certain invoice id and 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.

## Short query, price of work

This query calculates the price of work. Different work types have the different prices so the prices vary based on the hours for each work type. This query tests how the databases perform with a simple relational query example.

SELECT SUM(worktype.price \* workhours.hours \* workhours.discount) as price, work.id as workId FROM worktype,workhours,work WHERE worktype.id=workhours.worktypeId AND workhours.workId=work.id GROUP BY workId;

MATCH (wt:worktype)-[h:WORKHOURS]->(w:work) RETURN SUM(h.hours\*h.discount\*wt.price) as price, w.workId as workId

## Long query, price of work with items

This query is an extended version of the query that queries workhours prices. This query also includes item prices for each work which is summed to the workhours 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.

SELECT SUM((worktype.price \* workhours.hours \* workhours.discount) + (item.purchaseprice \* useditem.amount \* useditem.discount)) as price, work.id as workId FROM worktype,workhours,work,useditem,item WHERE worktype.id=workhours.worktypeId AND workhours.workId=work.id AND work.id=useditem.workId AND useditem.itemId=item.id GROUP BY workId

MATCH (wt:worktype)-[h:WORKHOURS]->(w:work)-[u:USED\_ITEM]->(i:item) RETURN SUM((h.hours\*h.discount\*wt.price)+(u.amount\*u.discount\*i.purchaseprice)) as price, w.workId as workId

## Query with defined key

This query tests the performance of a relational query for a specific invoice. When an invoice id is known, querying relationships should be straightforward in a graph database. In a relational database, relation information should be queried from a relation table ja other tables should be joined with it in order to get the needed information. This query is tested especially for technical reasons.

SELECT \* FROM invoice,workinvoice,work WHERE invoice.id=workinvoice.workId AND workinvoice.workId=work.id AND invoice.id=0

MATCH (i:invoice { invoiceId:0 })-[wi:WORK\_INVOICE]->(w:work) RETURN \*

## Aggregate 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 invoices and work. The second query is “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.

SELECT q1.invoiceId AS invoiceId, sum(q2.price) AS invoicePrice FROM (SELECT invoice.id AS invoiceId, work.id AS workId FROM invoice, workinvoice, work WHERE invoice.id=workinvoice.invoiceId and workinvoice.workId=work.id ) AS q1, (SELECT work.id AS workId, SUM((worktype.price \* workhours.hours \* workhours.discount) + (item.purchaseprice \* useditem.amount \* useditem.discount)) AS price FROM worktype,workhours,work,item,useditem WHERE worktype.id=workhours.worktypeid AND workhours.workid=work.id AND work.id=useditem.workid AND useditem.itemid=item.id GROUP BY work.id) AS q2 WHERE q1.workId = q2.workId GROUP BY q1.invoiceId

MATCH (inv:invoice)-[:WORK\_INVOICE]->(w:work)<-[h:WORKHOURS]-(wt:worktype) WITH inv, w, SUM(wt.price\*h.hours\*h.discount) as workTimePrice OPTIONAL MATCH (w)-[u:USED\_ITEM]->(i:item) WITH inv, workTimePrice + SUM(u.amount\*u.discount\*i.purchaseprice) as workItemPrice RETURN inv, sum(workItemPrice) as invoicePrice

## Recursive query, invoices related to invoice id 100000

This is a recursive query that gets all the invoices related to a given invoice. A common use case for this kind of query is when several related bills are sent to a customer. This query is useful to test the recursive query capabilities of the databases. MariaDB has support for Common Table Expressions which is more efficient way to do this query. However as the MySQL version that we tested does not support them, the traditional way to make recursive query is used.

SELECT id,customerid,state,duedate,previousinvoice FROM (SELECT \* FROM invoice ORDER BY previousinvoice, id) invoices\_sorted, (SELECT @pv := '100000') initialisation WHERE find\_in\_set(previousinvoice, @pv) AND length(@pv := concat(@pv, ',', id))

MATCH (i:invoice { invoiceId:100000 })-[p:PREVIOUS\_INVOICE \*0..]->(j:invoice) RETURN \*

# Query tests

## Test settings

The query tests were performed with MacBook Pro Laptop that has following specifications:

* macOS Catalina version 10.15.3
* 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 that contains 10000 customers, 100000 invoices, 100000 targets, 10000 works, 100000 items, 100000 worktypes, 100000 useditems, 100000 workhours, 1000000 workinvoices and 1000000 worktargets was generated. 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. A list of results was formed after each query. The biggest and the smallest number was removed from the list and average and standard deviation was calculated from the result set.

* + 1. Short query, price of work

Results for a short query can be found in table 1. Neo4J is clearly faster that MySQL as in previous studies. However MariaDB performs the best.



Table 1: Results for the query "price of work"

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

Results for a long query can be found in table 2. When query gets more complex Neo4J does no longer perform so efficiently. Both MySQL and MariaDB outperform Neo4J. MariaDB performs the best.



Table 2: Results for the query "Price of work with items"

* + 1. Query with the defined key, work of invoice

Results for a query with defined key can be found in table 3. Neo4J would be expected to have the best performance with this query as when the key is known, relations can be found by navigating the path. However both MySQL and MariaDB seem to perform better with no significant difference between the two.



Table 3: Results for the query with defined key

* + 1. Aggregate query, invoice price

The invoice price calculation is the heaviest query in these tests. When comparing MySQL with Neo4J the result has similarity to previous studies Neo4J outperforming MySQL. However MariaDB has far better performance clearly outperforming both.



Table 4: Results for the query "Price of Invoice"

* + 1. 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 5 shows the results when querying 100 sequential invoices and table 6 shows results when querying 1000 invoices. With 100 invoices MySQL 5.1.41 and MariaDB 10.5.6 are both faster than Neo4J. MariaDB performs a bit better than MySQL.



Table 5: Results for the recursive query **with 100 invoices**

As the number of invoices was increased to 1000, Neo4J performance seemed to be very low. Both MySQL 5.1.41 and MariaDB 10.5.6 seemed to significantly perform better. The interesting finding with 1000 invoices is that MySQL 5.1.41 is a bit faster than MariaDB 10.5.6.



Table 6: Results for the recursive query with 1000 invoices

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.

# Conclusions

The performance tests represented in this article show interesting differences in performance for the three databases. The short query and the aggregate query test has similarities to results in previous research done in [graph5] and [graph6] the way that Neo4J outperforms MySQL version 5.1.41. However in the other tests Neo4J 4.1.3 seem to have clearly the worst performance even compared to old MySQL version 5.1.41. In overall MariaDB 10.5.6 has the best performance. Except when the number of invoices was increased the old MySQL 5.1.41 seemed to be a bit faster. Perhaps because the old way of recursive query was used. In MariaDB usage of Common Table Exrpessions is recommended when doing recursive queries.

The performance of MariaDB indicates that the situation in the market has changed in favour of a relational database gaining back the position when it comes to performance. While the graph model should be in theory more effective in relational queries, the relational database seem to be still a viable option when it comes to the performance.

[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