A STUDY OF RELATIONAL AND DOCUMENT DATABASES QUERIES PERFORMANCE

Jurgita Lieponienė; Panevėžys University of Applied Sciences
jurgita.lieponiene@panko.lt

Keywords: relational database, document database, queries, denormalization, indexes

1. ABSTRACT

The amount of data being created and used by organizations is growing every day. The databases of modern information systems contain billions of records. The increase of data volumes leads to the increase of the response time of information systems. Modern systems must react to user requests as quickly as possible. The speed of information systems depends on various parameters: program code, parallel processes, database management system, query code efficiency, chosen database model. We can get the same results by writing different queries, executing queries in normalizing or denormalizing data schemes and using or missing indexes, using different database model. The objective of current research is to evaluate how the response time for SQL queries depends on query optimization and database model. The article presents the results of this research and the summarised conclusions.

2. INTRODUCTION

As users of information systems are eager to get the results of their queries as quickly as possible, the programmers who create and upgrade information systems should assess all the parameters that have influence on information system speed, including the effectiveness of used queries. Scientific literature studies different ways of query optimization. Habimana (2015) analysed optimising SQL code. Costel, Luca, and Teodor (2014) studied query optimization techniques offered by Microsoft SQL Server. Mithani, Machchhar, and Jasdanwala (2016) summarised SQL code optimization rules and suggested the model of SQL query converting into optimized query that ensures shorter time of SQL query execution. Bhajipale et al. (2016) analysed query optimization designing more effective structure of data base, optimizing data base indexes and SQL code, analysing query execution plans. Oktavia and Sujarwo (2014) explored the methods of internal SQL queries and data base indexing strategy, which makes positive impact on the duration of internal SQL queries. Besides, scientific literature contains a wide range of studies done on the effectiveness of NoSQL database queries. Scientists who analyse the effectiveness of queries emphasise that although the same results can be received with different queries, a user wants those queries that allow obtaining results the least amount of time possible. The price of unoptimized query execution is very high (Gupta & Chandra, 2011). Ineffective queries have a negative effect on the speed of business systems and reduce business effectiveness (Oktavia & Sujarwo, 2014).

Although scientific literature studies different aspects of query optimization the majority is limited to the theoretical analysis of problems. There is a lack of studies that ground the effectiveness of query optimization methods on the results of experimental testing. In light of this the aim of the current research is to assess the importance of query code optimization, database indexes, structure and chosen database model in the creation of optimized queries by means of experimental testing. The results of this research are important for the programmers of information systems, databases and other IT specialists, who create information systems and analyse the data stored in databases. The results of experimental research can be used in high schools teaching databases.

Research object is query optimization.

Research objective is to perform query optimization by optimizing query code, using or missing database indexes, denormalizing database structure and choosing database model.

Research tasks:

1. To assess the impact of query code on the response time of query.
2. To ascertain how database indexes influence the duration of query execution.
3. To assess the impact of database denormalization on the effectiveness of queries.
4. To ascertain how data model affects query runtime.

Research methods: the impact of query code, database indexes, database denormalization and chosen database model on the effectiveness of queries was assessed my means of experimental testing. The experimental testing was implemented in Oracle XE and MongoDB database management systems conducting tests in databases varying in size, with indices present or not, and with database denormalized or not.

3. COMPARATIVE ANALYSIS OF RELATIONAL AND DOCUMENT DATABASES

Aim of this comparative analysis is to investigate document databases in comparison with relational database approach in terms of data model, data structure, scheme, scalability, query language, transactions. Below provided table represents the generalized results of the comparative analysis of relational and document databases.

Relational Database (RDB) which based on the relational model has been architected more than 30 years ago mainly to serve business data processing since then it has become the best option for storing information that range from financial records, personal data and much more (Mohamed et al, 2014). Relational databases (RDB) appeared when mathematical discipline – relational algebra – was applied in order to ensure data integrity (Makčinskas, 2013). This data storage model ensures data integrity – following standard forms, information is separated into smaller units and relations (connections) are used. M. Shalini and S. Dhamodharan (2014) emphasise that relational database transactions possess qualities that ensure the reliability of transactions. These qualities of transactions are called ACID qualities: atomicity, consistency, isolation, durability.

No	Criteria	Relation Databases	Document Databases
1.	Data model	Relation data model	Document data model
2.	Data structure	Database structures data into tables and rows (records).	Structures data into collections of documents.
3.	Scheme	Fixed Scheme Each record conforms to fixed scheme.	Flexible Scheme Each document could have different or the same fields.
4.	Scalability	Scaling is vertical. More data means a bigger server.	Scaling is horizontal, meaning across servers.
5.	Consistency	In relational database Consistency means that all users see the same version of data after the transaction. Thus, relational database provides better consistency than no-relational database (Faraj et al, 2014).	In Document database “Eventual Consistency” means that there is no guarantee for reads and writes after the transaction for all entities in the database will be immediately consistent (Faraj et al, 2014).
6.	Query language	SQL	Query language depends on database. MongoDB query language – MQL.
7.	Transactions	The vast majority of relational databases are ACID compliant.	Document Databases sacrifice ACID compliancy for performance and scalability.

Table 1.
Comparative Analysis of Relational and Document databases

Atomicity is an ability of database to ensure that all the actions of the transaction or none of them would be executed. Consistency is the keeping of the permanent state of the database before the start of the transaction and after its finish. Isolation is a possibility to separate an executed transaction from other processes. Durability is the ensuring that if a message concerning successfully executed transaction was received, the results of transaction cannot disappear in themselves. The problem with relational model is that it has some scalability issues that is performance degrades rapidly as data volumes increases (Nayak et al, 2013). The main disadvantage of relational databases is that these databases are not suitable for distributed systems (Shalini & Dhamodharan, 2014). Relational databases are oriented towards data integrity, so the distribution of this model is made owing to data accessibility (Padhy et al, 2011). Strict ensuring of data integrity does not allow systems to be easily distributed, and this means that automatic system scalability and accessibility become a very complicated and expensive task of the infrastructure (Makčinskas, 2013). Relational database can be scaled just vertically, because entire database has to be hosted in a single server. This is necessary in order to ensure reliability and continuous availability of data (Sharma et al, 2015). NoSQL databases were designed to scale horizontally. Instead of increasing power of one single server we just need to add more server instances to get expected power (Sharma et al, 2015).

Document databases are one of the non-relational categories of databases (Faraj et al, 2014). Document databases store data and organize them as document collections, instead of structured tables with uniform sized fields for each record. Document databases are used for the storage, management and processing of structured and semi-structured data. In document databases data are stored in the document collection. Separate documents are considered equivalent to the records of relational database (Faraj et al, 2014). The scheme of document databases is flexible and replaceable (Kaur & Sahib, 2013). In document collection documents of various structures can be stored. Document databases perfectly solve the tasks of data accessibility and distribution (Makčinskas, 2013). Document databases get their type information from the data itself, normally store all related information together, and allow every instance of data to be different from any other. This makes them more flexible in dealing with change and optional values, maps more easily into program objects, and often reduces database size. This makes them attractive for programming modern web applications, which are subject to continual change in place, and speed of deployment is an important issue (Chaitanya et al, 2016).

4. THE RESULTS OF EXPERIMENTAL RESEARCH

The experimental research tested document database MongoDB and relational database Oracle XE. In order to conduct the experiment, four test data sets were generated using DTM Data Generator for JSON tool. The first set of test data contains 10000 records, the second – 50000 records, the third – 250000 records and the fourth – 1250000 records, provided in JSON format. Automatically generated data describe city information: provide city name, city geographical position, number of residents, state, country, the continent where a city is situated. Four test schemes were created in Oracle XE and four document collections were created in MongoDB databases; generated data sets were imported to these databases. Relational database data model is presented in Figure 1.

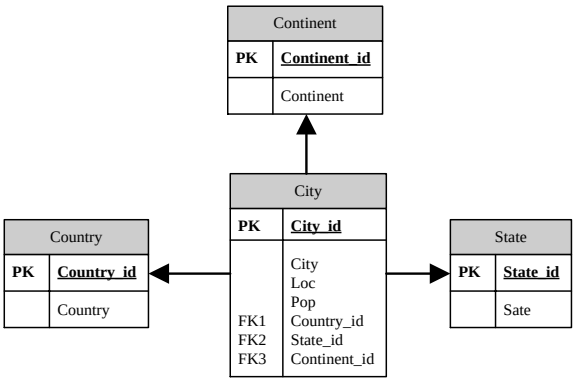

Figure 1.
Relational database structure

In order to conduct the experimental testing RozorSQL tool was used. By means of this tool queries in both Oracle XE, and MongoDB test databases were executed. The duration of query execution was recorded in query execution log file. Each test query is executed 10 times; the average of response time was calculated. The experiment started from optimizing SQL queries by writing SQL statement in different ways. Five test cases were created. Two queries were created for each test case: original query and improved query. Test was performed by running original and improved query in four test schemes and average time of each query was recorded. Also improved query was written in

MQL language and this query was executed in each MongoDB document collections for ten times and average time across queries was recorded. The experiment started from running SELECT statement including asterisk symbol instead of column names. Though it is easier to write, it takes more time for the database to complete the query. By selecting only the column you need, you are reducing the size of result table, reducing the network traffic. Test results are provided in Table 2. The speed of improved query increased 27%. Running improved query in MongoDB reduced 62% of time.

The next test case tested how unnecessary use of HAVING clause influence query speed. Improved query reduced 32% of time in ORACLE XE database. The HAVING clause is used to filter the rows after all the rows are selected and it is used like a filter. It is useless in a SELECT statement. It works by going through the final result table of the query parsing out the rows that don't meet the HAVING condition. Improved query written in MQL language also was executed in MongoDB. The results of the conducted experiment show that calculating query execution time is shorter in relational databases. The query execution time in MongoDB increased 17%. Test results are provided in Table 3.

Queries	Average Responds Time (in seconds)	Time Reduction (in percent)
Original SQL Query: *SELECT * FROM Cities*	0,2811	
Improved SQL Query: *SELECT City FROM Cities*	0,2063	26,61
MQL Query: *db.collection.find({City:1})*	0,1056	62,44

Table 2.
The SELECT Statements with * and Column Name

Queries	Average Responds Time (in seconds)	Time Reduction (in percent)
Original SQL Query: *SELECT country_id, count(country_id)* *FROM Cities* *GROUP BY country_id* *HAVING country_id!='155' and country_id!='50'*	0,9145	
Improved SQL Query: *SELECT country_id, count(country_id)* *FROM Cities* *WHERE country_id!='155' and country_id!='50'* *GROUP BY country_id*	0,6232	31,85
MQL Query: *db.collection.aggregate({$match: {country: {$nin:[Italy, Greece]}}}, $group: {_id: "$country", count: {$sum:1}}})*	1,068	-16,79

Table 3.
The SELECT Statements with and without Unnecessary HAVING Clause

The next test case tested how the unnecessary use of DISTINCT conditions influences query speed. Improved query resulted in 85% time reduction in Oracle XE. On MongoDB, 93% reduction in time was observed. Test results are provided in Table 4.

The next test case tested how the unnecessary use of SUBQUERIES influences query speed. The experiment showed that rewriting nested queries as joins leaded to a more efficient execution and more effective optimization. In our case improved query reduced 38% of time in ORACLE XE and 71% in MongoDB. Testing results are provided in Table 5.

Queries	Average Responds Time (in seconds)	Time Reduction (in percent)
Original SQL Query: *SELECT DISTINCT c.City* *FROM Countries cc* *JOIN Cities c ON cc.id=c.country_id* *WHERE cc.country='Lithuania'*	1,4395	
Improved SQL Query: *SELECT c.City* *FROM Countries cc* *JOIN Cities c ON cc.id=c.country_id* *WHERE cc.country='Lithuania'*	0,2157	85,02
MQL Query: *db.collection.find({country: "Lithuania"},{city: 1})*	0,1016	92,94

Table 4.
The SELECT Statements with and without Unnecessary DISTINCT Conditions

Queries	Average Responds Time (in seconds)	Time Reduction (in percent)
Original SQL Query: *SELECT c.City* *FROM Cities c* *WHERE c.country_id IN (SELECT cc.id* *FROM Countries cc* *WHERE cc.country='Lithuania')*	0,3474	
Improved SQL Query: *SELECT Country* *FROM Countries cc* *JOIN Cities c ON cc.id=c.country_id* *WHERE cc.country='Lithuania'*	0,2157	37,91
MQL Query: *db.collection.find({country: "Lithuania"},{city: 1})*	0,1017	70,73

Table 5.
The SELECT Statements with and without Unnecessary SUBQUERIES

The next test case tested how to reduce query speed by changing DISTINCT Clause with Exists statement. In our case improved query reduced 39% of time in Oracle XE and 69% in MongoDB. Testing results are provided in Table 6.

The second part of the experiment evaluated how queries can be optimized by using or missing indexes. For this purpose three test cases were created. For the first two cases, created queries were executed in indexed and not indexed relational database schemes and indexed document collections and average time of each query was recorded. For the last test case, the original and improved query was executed in indexed Oracle XE database schemes and MongoDB document collections. For the first test case, the databases and document collections were indexed by field Country. This field was used in a WHERE Clause of SQL Query. The query was faster only in the indexed database schemes where the query returns less than 10 percent of records. Test results are provided in Table 7.

Queries	Average Responds Time (in seconds)	Time Reduction (in percent)
Original SQL Query: *SELECT Distinct c.City* *FROM Countries cc* *JOIN Cities c ON cc.id=c.country_id* *WHERE cc.country='Lithuania'*	1,4395	
Improved SQL Query: *SELECT c.City* *FROM Cities c* *WHERE EXISTS (SELECT 'X' FROM Countries cc* *WHERE (c.country_id=cc.id) and* *(cc.country='Lithuania'))*	0,8775	39,04
MQL Query: *db.collection.find({country: {$exists: true, $in: [“Lithuania“]}},{city: 1})*	0,4460	69,02

Table 6.
The SELECT Statements with EXISTS Instead of DISTINCT

Queries	Average Responds Time (in seconds)	Time Reduction (in percent)
SQL Query: *SELECT c.City* *FROM Countries cc* *JOIN Cities c ON cc.id=c.country_id* *WHERE cc.country='Lithuania'* executed in not indexed database schemes	0,2157	
SQL Query: *SELECT c.City* *FROM Countries cc* *JOIN Cities c ON cc.id=c.country_id* *WHERE cc.country='Lithuania'* executed in indexed database schemes	0,1956	Depends on fetch records in ORACLE XE
MQL Query: *db.collection.find({country: "Lithuania"},{city: 1})* executed in indexed document collections	0,0863	59,99

Table 7.
Execution of Selection Queries in Indexed and not Indexed Database Schemes and Indexed Document Collections

The next test case measures the efficiency of calculated queries in indexed and not indexed database schemes and document collections. The field pop was indexed in all test database schemes and MongoDB document collections. This field was used in calculation in queries. In this case, the use of indexes was inefficient; the query execution time increased in both databases. Test results are provided in Table 8.

The last test evaluates the use of indexes with use of IN Predicate and Logical operator OR in a WHERE Clause. The test showed that using predicate IN with indexes in SQL queries resulted in 75% reduction in time. Improved query in MongoDB resulted in 91% reduction in time. Test results are provided in Table 9.

Queries	Average Responds Time (in seconds)	Time Reduction (in percent)
SQL Query: *SELECT cc.Country, sum(pop)* *FROM Countries cc* *JOIN Cities c ON cc.id=c.country_id* *GROUP BY cc.Country* executed in not indexed database schemes	0,8935	
SQL Query: *SELECT cc.Country, sum(pop)* *FROM Countries cc* *JOIN Cities c ON cc.id=c.country_id* *GROUP BY cc.Country* executed in indexed database schemes	1,3323	-49,11
MQL Query: *db.collection.aggregate($group: {_id: "$country", count:{$sum:pop}}})* executed in indexed document collections	1,3960	-56,23

Table 8.
Execution of Calculated Queries in Indexed and not Indexed Database Schemes and Indexed Document Collections

Queries	Average Responds Time (in seconds)	Time Reduction (in percent)
Original SQL Query: *SELECT city* *FROM Cities* *WHERE (pop=1000) OR (pop=10000)*	0,5935	
Improved SQL Query: *SELECT city* *FROM Cities* *WHERE pop IN (1000, 10000)*	0,1490	74,90
MQL Query: *db.collection.find({pop: {$in: [1000, 10000]}},{city: 1})*	0,0534	91,00

Table 9.
Execution of Selection Queries in Indexed Database Schemes and Document Collections

The last part of experiment evaluated how SQL queries can be optimized by optimizing the Database Design via denormalization. We implement normalization in order to maintain data integrity. During the normalization process, we decompose tables into more tables. The more tables we have, the more joins we have to perform in our queries. The aim of the last part of the experiment was to evaluate the impact of data scheme denormalization on SQL queries. For this part of the experiment, three test cases were performed. In each test case, we had original query with three joins and improved query with decreased number of joins. In the first test, improved query had two joins, in the second test, there was one join and in the last, there were no joins. Also we executed query in MongoDB document collections without joins. In the first test case the decreased number of joins led to 49% reduction in response time of SQL queries. Running query in MongoDB document collections without relations (references and subdocuments), a 95% of reduction in time was achieved. Test results are provided in Table 10.

In the second test case, the decreased number of joins reduced 65% of SQL queries time in Oracle database and running a query in MongoDB document collections without relations (references and subdocuments) 95% of time was saved. Test results are provided in Table 11.

Queries	Average Responds Time (in seconds)	Time Reduction (in percent)
Original SQL Query: *SELECT c.City, cc.Country, s.State, cn.Continent* *FROM ((Countries cc* *JOIN Cities c ON cc.id=c.country_id)* *JOIN States s ON c.state_id=s.id)* *JOIN Continents cn ON c.continent_id=cn.id*	1,957	
Improved SQL Query: *SELECT c.City, c.Country, s.State, cn.Continent* *FROM (Cities c JOIN States s ON c.state_id=s.id)* *JOIN Continents cn ON c.continent_id=cn.id*	0,9994	48,93
MQL Query: *db.collection.find({City:1, Country:1, State:1, Continent:1})*	0,1076	94,50

Table 10.
Query in Normalized and Denormalized Database Structure (Improved Query with two Joins)

Queries	Average Responds Time (in seconds)	Time Reduction (in percent)
Original SQL Query: *SELECT c.City, cc.Country, s.State, cn.Continent* *FROM ((Countries cc* *JOIN Cities c ON cc.id=c.country_id)* *JOIN States s ON c.state_id=s.id)* *JOIN Continents cn ON c.continent_id=cn.id*	1,957	
Improved SQL Query: *SELECT c.City, c.Country, c.State, cn.Continent* *FROM Cities c JOIN Continents cn* *ON c.continent_id=cn.id*	0,6848	65,01
MQL Query: *db.collection.find({City:1, Country:1, State:1, Continent:1})*	0,1076	94,50

Table 11.
Query in Normalized and Denormalized Database Structure (Improved Query with one Join)

Queries	Average Responds Time (in seconds)	Time Reduction (in percent)
Original SQL Query: *SELECT c.City, cc.Country, s.State, cn.Continent* *FROM ((Countries cc* *JOIN Cities c ON cc.id=c.country_id)* *JOIN States s ON c.state_id=s.id)* *JOIN Continents cn ON c.continent_id=cn.id*	1,957	
Improved SQL Query: *SELECT c.City, c.Country, c.State, c.Continent* *FROM Cities*	0,2936	85,00
MQL Query: *db.collection.find({City:1, Country:1, State:1, Continent:1})*	0,1076	94,50

Table 12.
SQL Query in Normalized and Denormalized Database Structure (Improved Query without Joins)

In the last test case the decreased number of joins reduced 85% of SQL queries time. Running a query in MongoDB document collections without relations (references and subdocuments) 95% of time was saved. Test results are provided in Table 12.

5. CONCLUSIONS

(A) During the assessment of the impact of code on the effectiveness of query it was established that properly written query code reduces the duration SQL query execution by 85%.

(B) The conducted experimental research showed that rewriting nested queries as joins, eliminating unnecessary DISTINCT conditions and avoiding include a HAVING clause in SELECT statements lead to more efficient query execution and more effective optimization. The designation of query field in SELECT phrase can reduce the duration of query execution by 27%.

(C) The conducted experimental research showed that the indexing of query fields used in calculations has negative impact on the duration of query execution.

(D) The indexing of query fields that form selection criteria is effective only when selection query returns less than 10% percent of data base records.

(E) The conducted experimental research showed that the denormalization of database has positive impact on the effectiveness of SQL queries. By denormalizing database to avoid connections in SQL query use, it is possible to reduce the time of query execution by 85%.

(F) Document databases are more superior in the course of selection queries; however, aggregate queries are faster in relational databases.

6. REFERENCES

[1] Bhajipale R.; Bisen P.; Meshram A.; Thakur S. (2016). SQL Tuner. In International Journal of Computer Trends and Technology (IJCTT), 33(1), 29-32.

[2] Chaitanya P.; Ranjan H. P.; Kiran T. S.; Anitha K. (2016). Implementation of an Efficient MongoDB NoSQL Explorer for Big Data Visualization. In International Journal of Advanced Networking & Applications (IJANA), 444-447.

[3] Chunxia Qi. (2016). On index-based query in SQL Server database. In Control Conference (CCC), 56-65.

[4] Costel G.; Luca V.; Teodor O. (2014). Query Optimization Techniques in Microsoft SQL Server. In Database Systems Journal, 2, 33-48.

[5] Faraj A.; Rashid B.; Shareef T. (2014). Comparative Study of Relational and Non-relations database performances using ORACLE and MONGODB systems. In International Journal of Computer Engineering and Technology (IJCET), 5(11), 11-22.
[6] Kaur J.; Sahib, G. A. (2013). Review on Document Oriented and Column Oriented Databases. In International Journal of Computer Trends and Technology, 4(3), 338-344.
[7] Gupta M.; Chandra P. (2011). An Empirical Evaluation of LIKE Operator in Oracle. In International Journal of Information Technology, 55-65.
[8] Habimana J. (2015). Query Optimization Techniques – Tips For Writing Efficient And Faster SQL Queries. In International Journal of Scientific & Technology Research, 4(10), 22-26.
[9] Makčinskas T. (2013). Jau skęstame duomenyse, o norime vis daugiau. In BDC NEWS, 10, 1-3.
[10] Mithani F.; Machchhar S.; Jasdanwala F. (2016). A novel approach for SQL query optimization. In Computational Intelligence and Computing Research (ICCIC), 15-25.
[11] Mohamed M.; Altrafi O. G.; Ismail M. O. (2014). Relational vs. NoSQL Databases: A Survey. In International Journal of Computer and Information Technology, 3(3), 598-601.
[12] Nayak A.; Poriya A.; Poojary D. (2013). Type of NOSQL Databases and its Comparison with Relation Databases. In International Journal of Applied Information Systems (IJAIS), 5(4), 16-19.
[13] Oktavia T.; Sujarwo S. (2014). Evaluation of Sub Query Performance in SQL Server. In EPJ Web of Conferences, 16-22.
[14] Ordonez C. (2010). Optimization of Linear Recursive Queries in SQL. In IEEE Transactions on Knowledge & Data Engineering, 22(2), 264-277.
[15] Padhy R. P.; Patra M. R.; Satapathy S. Ch. (2011). RDBMS to NoSQL: Reviewing Some Next-Generation Non-Relational Database's. In International Journal of Advanced Engineering Sciences and Technologies, 11(1), 15-30.
[16] Ratkevičius D. (2011). Neprograminiai verslo valdymo sistemų atrankos kriterijai. In Socialinių mokslų studijos, 13(1), 1359-1374.
[17] Shalini M.; Dhamodharan S. (2014). Performance and Scaling Comparison Study of RDBMS and NoSQL (MongoDB). In An international journal of advanced computer technology, 3 (11), 1270-1275.
[18] Sharma N., Charu J., Chauhan M. (2015). Comparative Study of Distributed, Scalable, & High Performance NoSQL Databases. In International journal in IT and Engineering, 3 (1), 28-35.