Comparison of semi-structured data on MSSQL and Postgresql

[1]Leandro Alves, [1]Pedro Oliveira, [1]Júlio Rocha, [2]Cristina Wanzeller, [4]Filipe Cardoso, [2]Pedro Martins, [3]Maryam Abbasi

[1] Polytechnic of Viseu, Viseu Portugal
[2] CISeD - Research Centre in Digital Services, Polytechnic of Viseu, Portugal
[3] CISUC - Centre for Informatics and Systems of the University of Coimbra, Portugal
[4] Polytechnic of Coimbra, Coimbra, Portugal

[1]pv23844@alunos.estgv.ipv.pt, [1]estgv9081@alunos.estgv.ipv.pt, [1]estgv13802@alunos.estgv.ipv.pt, [2]cwanzeller@estgv.ipv.pt, [4]filipe@isec.pt, [2]pedromom@estgv.ipv.pt, [3]maryam@dei.uc.pt

Abstract. The present study intends to compare the performance of two Data Base Management Systems, specifically Microsoft SQL Server and PostgreSQL, focusing on data insertion, queries execution, and indexation. To simulate how Microsoft SQL Server performs with key-value oriented datasets we use a converted TPC-H lineitem table. The data set is explored in two different ways, firsts using the key-value-like format and second in JSON format. The same dataset is applied to PostgreSQL DBMS to analyse performance and compare both database engines. After testing the load process on both databases, performance metrics (execution times) are obtained and compared. Experimental results show that, in general, inserts are approximately twice times faster in Microsoft SQL Server because they are injected as plain text without any type of verification, while in PostgreSQL, loaded data includes a validating process, which delays the loading process. Moreover, we did additional indexation tests, from which we concluded that in general, data loading performance degrades. Regarding query performance in PostgreSQL, we conclude that with indexation, queries become three or four percent faster, and six times faster in Microsoft SQL Server.

Keywords: Key-Value, database, mssql, postgresql, tpc-h, performance, gin, computed columns

1 Introduction

The purpose of this experiment is to test unrelated data in two relational databases, and for that we use a dataset converted from the relational model TPC-h[1]. The reason for this is the need to have a dataset structure in key-value format, which will be tested in PostgreSQL and MSSQL. What we propose to test how relational databases handle semi-structured data.
This work summarizes a set of operations over a JSON dataset format on MSSQL

vs. Postgresql, and Key-Value in MSSQL vs. PostgreSQL. Indexes were applied to these three tables in the different databases. The workloads running on all databases, indexed and not-indexed, are explained in more detail in the "Experimental setup" section.
Results show that when using MSSQL, it is important to consider computed columns for the application of indexes, while PostgreSQL can handle semi-structured data without major changes/effort.
This paper is organized into five sections. The first section is this introduction, where the challenge or problem to be solved is addressed and what steps to follow; the second deals with the related work; the third is related to the experimental work, where the technologies and practices on these technologies are presented to solve the challenge/problem; the fourth goes through the presentation and analysis of the different results and, finally, the fifth section will be for the conclusions obtained about the present research.

2 Related Work

In research from authors [2], that analyse the performance in the same DBMSs, PostgreSQL and MSSQL, positioned in the cloud and they concluded, that the most high-performance DBMS was MSSQL, in all the tests performed, but they never approach the theme of a data set oriented to key-value, or JSON format.
Another interesting work [3], also does load tests, records and analyses the times with a data set generated by TPCH, to tables on Oracle and PostgreSQL, and the results obtained differed depending on the method used for each load test. Some of the methods presented are used in our paper, like PostgreSQL "COPY" and "insert into" method.
Another research [4] demonstrates that performance on non-relational DMBS is higher, because MySQL includes a lot of complex queries which involves integrity constraints and joins and NoSQL allows to eke more performance out of the system by eliminating a lot of integrity checks done by relational databases from the database tier.
In another research [5], the behavior of relational and non-relational database engines is also compared, when executing commands 'insert', 'update', 'delete' and 'select'. The comparison is made through the times analyzed when operating on 4 datasets of 100 records, 1000 records, 10000 records and 100000 records. According to the data presented, Redis obtains faster results when the volume of data is greater, specifically in 'delete' and 'insert' operations. For 'update' and 'select' operations, the results show that MariaDB can be faster than Redis. For data volumes around 10000 records, the results are very similar.
In another research [6], tests were carried out using 'insert', 'update', 'delete' and 'select' commands in relational DBMS, such as Oracle, MySQL and MSSQL, and non-relational DBMS, such as Mongo, Redis, Cassandra and GraphQl, and compared the times in obtaining results over a dataset of 10000 and 100000 records. To carry out the tests, a database of a railway with all the stations was used, but, to guarantee the quality of the tests, on a large volume of data, false records

were added to the tables. The results show that non-relational database engines are more efficient, with MongoDB standing out as the fastest in all operations. In another research [7], MySQL was also compared with MongoDB. In all cases, with performance or load methods, NoSQL databases have better perform than relational databases and the reason for that is that MongoDB have a very powerful query engines and indexing features.
Not all operations are more efficient in non-relational databases. The research [8] compares the performance of relational and non-relational databases, running complex queries on a large dataset. Results show that 'select' method is significantly faster in MongoDB, however some math queries such as aggregate functions (sum, count, AVG) are better on Oracle RDBMS.

The proposed work differs from the others referenced is the fact that the performance tests are performed on computed columns in MSSQL. One of the key variables that needs to be included in the equation is indexing. MSSQL lacks appropriate NoSQL data indexes and is not JSON friendly. We begin our quest for a method to make that happen. We discovered a mention of MSSQL computed columns that ought to serve as the key. We have discovered a method to use calculated columns and index those columns under this article's citation for sql indx [9]. Computed columns don't actually exist, as far as we know. On demand, the data is extracted from other derived columns. By including a non-clustered index in that column, the DBMS can avoid having to parse data again because the parsed-out information is stored to the index column table.

3 Experimental setup

The main proposal of this experimental is to perform benchmark performance with NoSQL data on a SQL DBMS environment. MSSQL and PostgreSQL were used in this work, both working on Windows 10 Operating System. As for the hardware, the tests in the different DBMS were performed on a laptop computer, with an i7 processor, with 16 GB of RAM, and SSD, whose bandwidth flow, withstand by the interface, is up to 600 MB/s. Regarding the explored dataset, it was achieved through the benchmark TPC-H [10], in which eight tables are generated. However, in our experimental setup we only use one table (lineitem), the reason is related with the fact that SQL Server is a relational data model based, and semi-structured data models does not allow tables and relations, instead, make use of document model hierarchy. In contrast to relational databases, which store data as rows in a table, document databases store entities as documents or JSON documents.

After choosing the hardware and DBMS, the workloads starts as describe above:

- Generate JSON;
- Setup DBMS environment;
- Insert data;

– Query data;
– Update data;

3.1 Generate JSON

To convert data table to data JSON, we used "FOR JSON AUTO" command [11], that, will produce an array of JSON based on a "SELECT" as show on listing 1.1. In this script, first, we made a conversion using "FOR JSON AUTO". This instruction, format the output of the FOR JSON clause automatically, based on the structure of the SELECT statement, converting all columns into JSON properties.

```
INSERT INTO [_lineitem_json]
SELECT [value] FROM OPENJSON(
    SELECT TOP 1000000 *
    FROM [lineitem] FOR JSON AUTO
)
```

Listing 1.1: Convert data table to JSON

3.2 Setup DBMS environment

After generate JSON data, during the last section 3.1, the next step is preparing the DBMS environment. First, we create a clean database on MSSQL and another on PostgreSQL. On MSSQL, add two tables, "lineitem_no_indx" and "lineitem". The reason for two tables is, to test workloads with and without indexes.

On PostgreSQL (hybrid DBMS that allow data-key-value and JSON), was added four tables. All tables have the same mission, test performance workloads with JSON and data-key-value, both, with and without indexes.

Indexing is one of the main variables that should be added to the equation. MSSQL is not JSON friendly and, has no proper indexes for NoSQL data. With that in mind, we start a research to find a way to accomplish that. We found a reference to MSSQL computed columns that should be the key. Under this article [9], we have found a way to use computed columns and, index those columns. As we know, computed columns do not exist physically. The data is parsed out on runtime from other derived columns. Adding a non-clustered index to that column, the parsed-out data is written to the index column table, and the DBMS does not have to parse data out again. We've created a computed column to each JSON property as showed on fig.1. The improvement is vast, as we show in the results and analyse section 4. So, with this in mind, we start to add all JSON properties as computed columns and index the ones used on query where condition, as non-clustered indexes.

On PostgreSQL, we found the GIN (Generalized Inverted Indexes) index, specialized for semi-structured data, are quite useful when an index must map multiple values to a row, and good for array indexing values, as well as full text search programs.

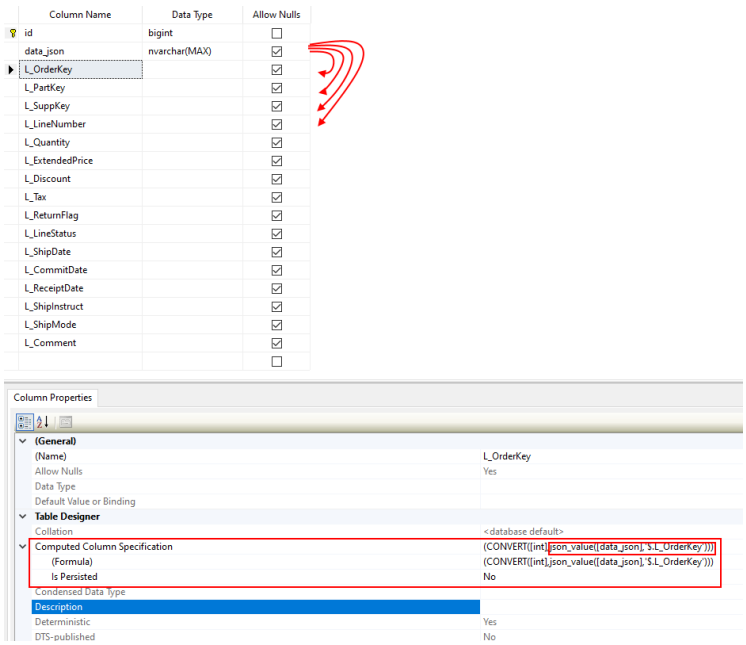

Fig. 1: Example of computed columns in MSSQL

3.3 Insert data

DBMS environment concluded, we perform the scripts to insert the same amount of JSON in each table, one million registries to be precise. Each insert should be executed three times. The lowest time registered, should be considered. All scripts have been executed against the indexed and non indexed tables. To collect times, turn on time collecting before the script "SET STATISTICS TIME ON" and turn it off after the insert "SET STATISTICS TIME OFF" as showed on listing 1.2.

```
SET STATISTICS TIME ON
    insert into [lineitem]
    select top 1000000 data_json from [_lineitem_json]
SET STATISTICS TIME OFF
```

Listing 1.2: Insert using clause INSERT INTO

Bulk insert is a faster way to insert data on MSSQL. The listing 1.3, is a script example how it works. First the table is defined on "BULK" instruction, and then the script point to a CSV file (JSON data generated on section 3.1) where the data is located. Bulk only allow insertions over a file.

```
SET STATISTICS TIME ON
        BULK INSERT [lineitem]
```

```
        FROM 'C:\TPC-H\json_postgress.csv'
        WITH (FORMAT = 'CSV'
                      ,FIRSTROW = 2
                      ,KEEPIDENTITY
                      ,KEEPNULLS
                      )
SET STATISTICS TIME OFF
```

Listing 1.3: Insert using clause BULK

For PostgreSQL, the process is identical. First, was made an insertion using method "INSERT INTO", as described on 1.4, and a second method, make use of massive insertion's instruction "COPY" on PostgreSQL, "BULK" equivalent on MSSQL as showed on 1.5

```
INSERT INTO lineitem_json (info)
SELECT info from lineitem_json_to_copy_source
```

Listing 1.4: Insert using clause INSERT INTO on PostgreSQL

```
COPY lineitem_json
FROM 'E:\ESTGV\json_postgress.csv'
    DELIMITER ',' CSV HEADER;
```

Listing 1.5: Insert using clause COPY on PostgreSQL

3.4 Query data

This section, describes the process adopted to test performance on both DBMS.

After run section 3.3, all tables have been loaded with the exactly same data. The performed scripts, in the end, should return the exact same results, with same where conditions "L_ShipMode='TRUCK' and L_LineNumber=1 and L_LineStatus='O' and L_Quantity between 8 and 60". As showed on listing 1.6 and 1.7. This is a 'sine qua non' condition, for a benchmark test.

Since we are using computed columns, as described in section 3.2 and fig.1, no column's definition is needed, only "*" to show all. The where condition complies with criteria defined on beginning of section 3.4, and times will be collected making use of "SET STATISTICS TIME ON" and "SET STATISTICS TIME OFF" instructions.

```
SET STATISTICS TIME ON
        select * from [lineitem]
        where [l_shipmode] ='TRUCK'
                AND [L_LineNumber] = 1
                and L_lineStatus='O'
```

```
                and L_Quantity between 8 and 60
        order by L_ShipDate asc
SET STATISTICS TIME OFF
```

Listing 1.6: Select on MSSQL

On this script, listing 1.7, we start to define all properties with correspondent data types, with where condition meeting the agreed criteria. To list data tabularly, we found function "jsonb_to_record", we pass all JSON record data, define data type for each JSON property, and a tabular view should be listed. This, has been used for legible proposes.

```
select x.*
from lineitem_json,
        jsonb_to_record(info) as x(
                "L_OrderKey" int,
                "L_PartKey" int,
                "L_SuppKey" int,
                "L_LineNumber" int,
                "L_Quantity" int,
                "L_ExtendedPrice" decimal,
                "L_Discount" decimal,
                "L_Tax" decimal,
                "L_ReturnFlag" text,
                "L_LineStatus" text,
                "L_ShipDate" date,
                "L_CommitDate" date,
                "L_ReceiptDate" date,
                "L_ShipInstruct" text,
                "L_ShipMode" text,
                "L_Comment" text)
where (info->>'L_ShipMode' = 'TRUCK')
        and (info->>'L_LineNumber' = '1')
        and (info->>'L_LineStatus' = 'O')
        and (info->>'L_Quantity')::numeric between '8' and '60'
order by (info->>'L_Quantity')
```

Listing 1.7: Select on PostgreSQL

3.5 Update data

The last workload section updates, have been performed with the same where conditions and same changes. Where condition, "L_ShipMode = 'MAIL'". Changes, update all "L_ShipMode" equal to "MAIL" to "TRUCKKKKKK". In the end, all results have been verified.

On MSSQL listing 1.8, we make use of "JSON_MODIFY" function, to navigate through JSON data on field "json_data" and perform changes as listed on listing 1.8.

```
SET STATISTICS TIME ON
UPDATE U
SET data_json =
    JSON_MODIFY(data_json, '$.L_ShipMode', 'TRUCKKKKKK')
FROM AEABD.dbo.[lineitem] as U
WHERE L_ShipMode = 'MAIL'
SET STATISTICS TIME OFF
```

Listing 1.8: Update data on MSSQL

On PostgreSQL listing 1.9, same changes have been performed. First, we filtered all registries where property "L_ShipMode" value equal to "MAIL", and change it to "TRUCKKKKKK".

```
UPDATE lineitem_json
SET info = info || '{"L_ShipMode":"TRUCKKKKKK"}'
WHERE (info->>'L_ShipMode' = 'MAIL')
```

Listing 1.9: Update data on PostgreSQL

4 Results and analysis

4.1 Tables sizes

On Table.1 and Fig.2, it is clear that MSSQL, because is not optimized for unstructured data, spend more space to accommodate same data. The performed test only have been done in one table and the difference is huge.

Another evidence was on indexed tables. Because the indexes require space to organize and split table data, all indexed tables in all DBMS's, are heavier than the ones without indexes. This means, performance improvement, will pay a high storage price.

Table 1: Tables Sizes

Table Name	Rows	SQL	PG-KEY-VALUE	PG-JSONb
LineItem	1 000 000	988.21 MB	678 MB	726 MB
LineItem_no_indx	1 000 000	835.38 MB	493 MB	691 MB

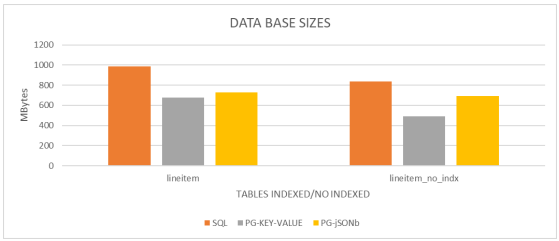

Fig. 2: Databases size

4.2 Inserts

The data, loaded via "BULK" or "COPY", was slower than "INSERT INTO" or "SELECT INTO".

In the "INSERT INTO", the data was loaded from a select that already resides in memory due to DBMS optimizations. "SELECT INTO" is the fast one. When the script runs, the target table should not exist and will be created without any indexes. So, in the "INSERT INTO", the table already exists with the primary key; consequently, the primary key has a clustered index by default. The DBMS needs to manage it, creating a delay during the process, as revealed in Fig.3.

All indexed tables are slowest, as shown in Fig.4, again, because of the need to manage all indexes. Therefore, adding indexes always be a trade-off, if many updates or inserts need to be executed on the Database. We should not add unnecessary indexes to tables. The reason is related to the fact that, more indexes means more work, rebuilding and reorganize them during the insertions or updates and consequently more time-consuming.

On Fig. 5 we can figure out that PostgreSQL is much time-consuming during the inserts than MSSQL. MSSQL doesn't do any validation during the process, while PostgreSQL verify each insertion to validate data. For that reason, has a poor performance.

4.3 Selects

All selects exposed a better performance on PostgreSQL than in MSSQL. PostgreSQL is optimized for unstructured data. That is a fact and is visible in Fig.6. On MSSQL, the selects are prolonged on tables without indexes. On the other hand, MSSQL revealed, despite a worse performance, been not too far from PostgreSQL, when indexed.

4.4 Updates

This section reveals the statistics against indexed and non-indexed tables during the update process. The problem mentioned above, regarding the management of the indexes, gets highlighted in Fig.7. As we can see, all updates in both DBMS

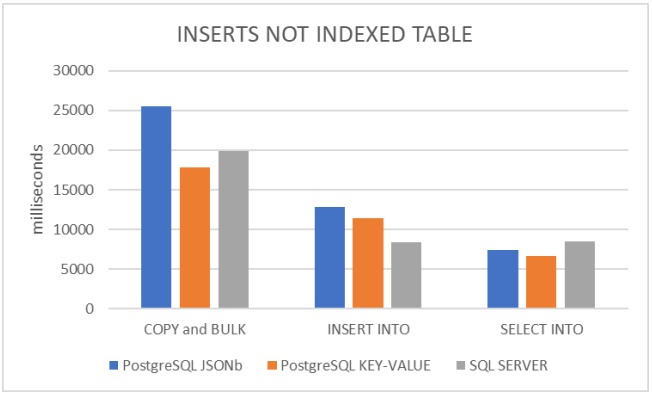

Fig. 3: Inserts not indexed tables

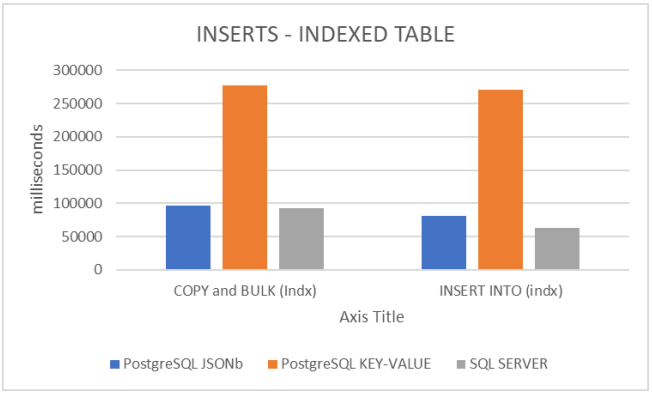

Fig. 4: Inserts in indexed tables

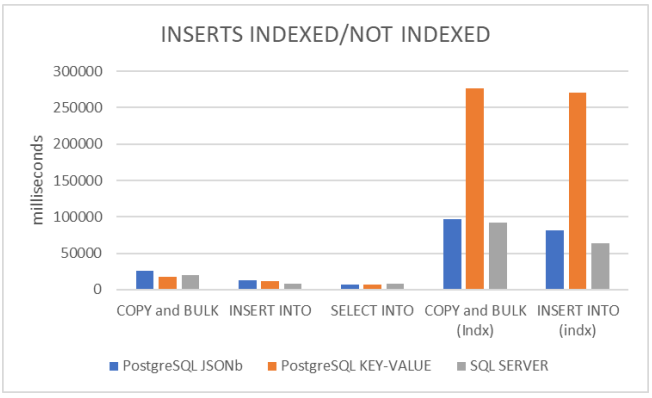

Fig. 5: Inserts in indexed and not indexed tables

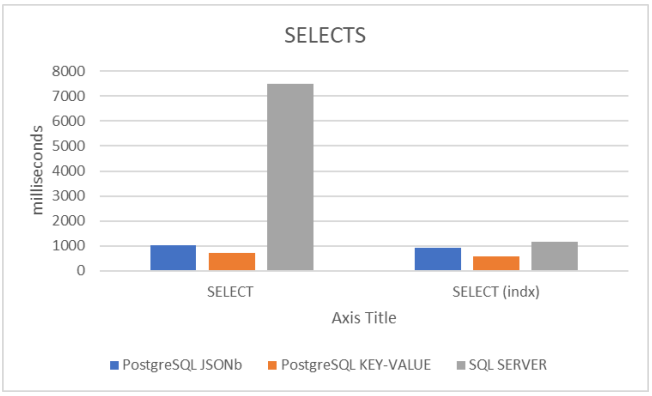

Fig. 6: Queries

on indexed tables get worst and worst performance every time a new index has been added. In conclusion, indexes are powerful, but we should moderate the way we add them to databases, since we can solve query performance issues, and get new ones on updates/inserts.

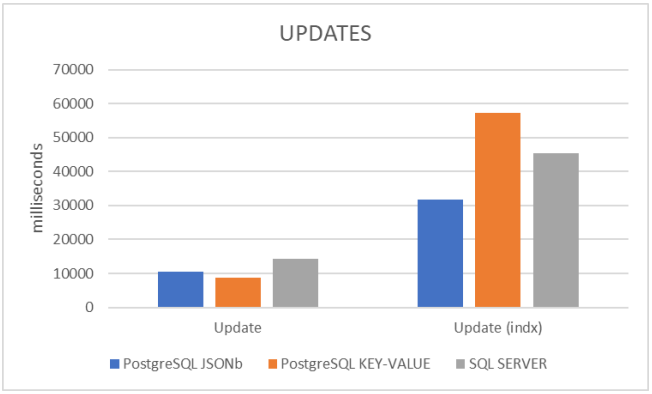

Fig. 7: Updates

5 Conclusions

MSSQL revealed a worse performance during the selects, but not so bad, when using indexes. This means, MSSQL made good improvements, in last years, getting better results, making use of new features that we'll explain above.

The insertions in MSSQL are generally faster, since the DBMS doesn't do any validation, but this, could be a problem under production environments,

since not well-formatted data could be inserted, triggering exceptions, causing a huge pain on development teams.

Storage could be an issue too. MSSQL is not optimized for this kind of data, taking up much more space than PostgreSQL.

Generally speaking, PostgreSQL should be the right choice for accommodating semi-unstructured data, faster, and better storage space managed.

Our major research, during this test case, was at MSSQL performance level, when we start to indexing computed columns. MSSQL, doesn't have a way to index NoSQL data as PostgreSQL. The way to accomplished that, was indexing computed columns. Since all JSON data inserted on MSSQL, are treated as text, we try to find a method to convert the data into tabular structure. Recurring to computed columns, as explained on section 3.2, we have a chance to index does columns. At this level, the benefit was huge. The same query, made an improvement 82.4% faster than without indexes as showed on fig.6.

Acknowledgements

"National Funds fund this work through the FCT—Foundation for Science and Technology, IP, within the scope of the project Ref UIDB/05583/2020. Furthermore, we would like to thank the Research Centre in Digital Services (CISeD), the Polytechnic of Viseu, for their support."

References

[1] *TPC-H*. visited on 2022-06. URL: https://www.tpc.org/tpch/.

[2] I. S. Vershinin and A. R. Mustafina. "Performance Analysis of PostgreSQL, MySQL, Microsoft SQL Server Systems Based on TPC-H Tests". In: *2021 International Russian Automation Conference (RusAutoCon)*. 2021, pp. 683–687. DOI: 10.1109/RusAutoCon52004.2021.9537400.

[3] Pedro Martins et al. "A performance study on different data load methods in relational databases". In: *2019 14th Iberian Conference on Information Systems and Technologies (CISTI)*. 2019, pp. 1–7. DOI: 10.23919/CISTI.2019.8760615.

[4] *Relational V S Key Value Stores Information Technology Essay*. visited on 2022-06. 2018. URL: https://www.ukessays.com/essays/information-technology/relational-v-s-key-value-stores-information-technology-essay.php.

[5] Wittawat Puangsaijai and Sutheera Puntheeranurak. "A comparative study of relational database and key-value database for big data applications". In: *2017 International Electrical Engineering Congress (iEECON)*. 2017, pp. 1–4. DOI: 10.1109/IEECON.2017.8075813.

[6] Roman Čerešňák and Michal Kvet. "Comparison of query performance in relational a non-relation databases". In: *Transportation Research Procedia* 40 (2019). TRANSCOM 2019 13th International Scientific Conference on Sustainable, Modern and Safe Transport, pp. 170–177. ISSN: 2352-1465. DOI: https://doi.org/10.1016/j.trpro.2019.07.027. URL: https://www.sciencedirect.com/science/article/pii/S2352146519301887.

[7] Benymol Jose and Sajimon Abraham. "Performance analysis of NoSQL and relational databases with MongoDB and MySQL". In: *Materials Today: Proceedings* 24 (2020). International Multi-conference on Computing, Communication, Electrical & Nanotechnology, I2CN-2K19, 25th & 26th April 2019, pp. 2036–2043. ISSN: 2214-7853. DOI: https://doi.org/10.1016/j.matpr.2020.03.634. URL: https://www.sciencedirect.com/science/article/pii/S2214785320324159.

[8] Azhi Faraj, Bilal Rashid, and Twana Shareef. "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 (2014), pp. 11–22.

[9] *One SQL Cheat Code For Amazingly Fast JSON Queries*. URL: https://bertwagner.com/posts/one-sql-cheat-code-for-amazingly-fast-json-queries/.

[10] Transaction Processing Performance Council. "TPC-H benchmark specification". In: *Published at http://www. tcp. org/hspec. html* 21 (2008), pp. 592–603.

[11] Microsoft. *Format JSON Output Automatically with AUTO Mode (SQL Server)*. URL: https://docs.microsoft.com/en-us/sql/relational-databases/json/format-json-output-automatically-with-auto-mode-sql-server?view=sql-server-ver16.