MySQL as NoSQL: Benchmarking MySQL vs MongoDB

Antonín Vlček
Technical University of Berlin
antonin.vlcek@campus.tu-berlin.de

***Abstract*—This paper benchmarks the performance of MySQL 8.0 and MongoDB for inserting and querying JSON documents, aiming to answer whether MySQL can compete with MongoDB on the NoSQL playing field. The study was conducted on the Google Cloud Platform, and the results showed that MongoDB outperforms MySQL in all scenarios, with a larger difference in insertion times. This confirms the hypothesis that MongoDB will perform better since it is purpose-built for NoSQL operations. For write-heavy applications, MongoDB could provide benefits, while for applications where read operations are more prevalent, it may be worth sticking to MySQL. Limitations of this study include the lack of index optimization and the need for more variables. Future research could include a join operation and consider economic aspects of adding a NoSQL database next to an existing MySQL instance.**

***Index Terms*—MySQL, MongoDB, Document Store, NoSQL, Benchmarking**

I. Introduction

Relational databases have been the go-to choice for many systems for a long time. MySQL has been the most popular open-source relational database for years and is the second most popular, with a very thin gap between it and the proprietary Oracle database. [1]

With the rise in popularity of NoSQL databases, MySQL needed to compete, so it introduced support for storing JSON documents and the NoSQL paradigm in version 5.7. However, it has fully embraced the concept of NoSQL in its latest version 8.0 [2]

As systems evolve and new requirements are put on them, the use-case calling for NoSQL paradigm will often arise. This can leave the development team wondering whether they should provision a new database or stick to what is supported by their existing infrastructure. To provide insights for decision-makers faced with these often challenging decisions, this study aims to compare the performance of storing and retrieving JSON documents from MySQL 8.0 when compared to MongoDB – by far the most popular document-store database out there. [3]

Although there have been some studies comparing MySQL and NoSQL databases [4] [5], including direct comparisons between MySQL and MongoDB, none use MySQL 8. The research question is whether MySQL can compete with MongoDB when inserting and querying documents in a NoSQL fashion. The hypothesis is that MongoDB will perform better since it has been built for this specific purpose.

The study measures the throughput of insert and query requests of various document and dataset sizes. As a real-world parallel, we can consider a scenario of adding audit logging to an existing application that already runs on MySQL 8.0. This paper provides essential information related to benchmarking cloud services and will give more background on the involved technologies, MySQL and MongoDB, and how inserting and querying documents is handled in each.

The paper is structured as follows: a background section will give more information on cloud service benchmarking, MySQL and MongoDB. The section titled Study Design will describe the benchmark architecture, the types of queries, and the data passed into the systems. The implementation section will detail how the benchmarking client was implemented. The results will be presented, then discussed and put into context. The discussion and conclusion sections will provide recommendations for real-world scenarios.

II. Background

This section sets the scene by providing context and outlining the relevant aspects of how MySQL and MongoDB operate. We touch on benchmarking these services in the cloud, specifically in the context of the Google Cloud Platform (GCP). Since this study uses *throughout* as the primary performance measure, we will also formally define this metric.

A. MySQL

MySQL is a popular open-source relational database management system (RDBMS) that has been around for over 25 years. It has a long-standing reputation as a reliable and high-performance database for traditional, structured data. [6] However, with the rise of unstructured data, MySQL introduced a JSON data type in version 5.7, which allows it to handle semi-structured and unstructured data like NoSQL databases. This was a significant shift in paradigm for MySQL, which had been strictly a relational database system. With the latest version 8.0, MySQL has embraced the NoSQL paradigm more fully and added support for more NoSQL features, including faster JSON validation, more efficient indexing, and a fully functional NoSQL API.

B. MongoDB

MongoDB is the most popular document-oriented NoSQL database that stores JSON documents, which allows for flexibility in the data model. It is widely used for real-time analytics and applications requiring scalability and performance. MongoDB stores data in a flexible JSON-like format called

BSON (Binary JSON) that allows for storing complex data structures and also supports a variety of indexing options for faster querying. [7] [8]

C. Benchmarking in the cloud

Both MySQL and MongoDB are available on public cloud platforms such as Google Cloud Platform (GCP), allowing easy deployment and scalability. However, for the purposes of this study, we will deploy both of them into their own VMs to control the allocated resources. This is an important aspect when benchmarking systems in the cloud. In the context of this study, the benchmarking focuses on measuring the throughput of MySQL and MongoDB on GCP.

D. Throughtput

Throughput is a performance metric that describes the number of requests a cloud service can handle in parallel. As this includes both a number and some notion of time, the typical representation of throughput uses the unit requests per second. This means that within a given time interval, the number of requests that were successfully completed is counted. [9]

III. Study Design

Starting from the architecture, this chapter describes the different components of the benchmark and how they relate to each other. It then focuses on the client and how it is used to benchmark the Systems Under Test (SUTs), before concluding with a description of data that client stores and queries from the two databases.

A. Architecture

On a high level, this study deals with two SUTs and a client. The client generates queries to the respective SUT and records how well the system performs. This infrastructure is deployed in one availability zone in GCP. The SUTs are two Virtual Machines (VMs): one VM runs MySQL, and the other runs MongoDB. The client is designed in a way that a change of configuration determines whether it generates queries for MySQL or MongoDB. Two instances of the client – each with a different configuration – are deployed and connected to their respective SUT.

The client records metrics and saves them locally to introduce as little overhead as possible. Only once the benchmarking sequence concludes are these metrics transferred to Cloud Storage Bucket before the client's VM terminates. A data analysis tool such as Looker Studio or Jupyter Notebook can later query this persistent storage to scrutinize the data.

Finally, the Cloud Build service turns the client's code into a Docker image. Cloud Build persists this image in Google's Container Registry. This simplifies the deployment of the infrastructure – more on that in section IV.

Figure 1 summarises the different components and their connections.

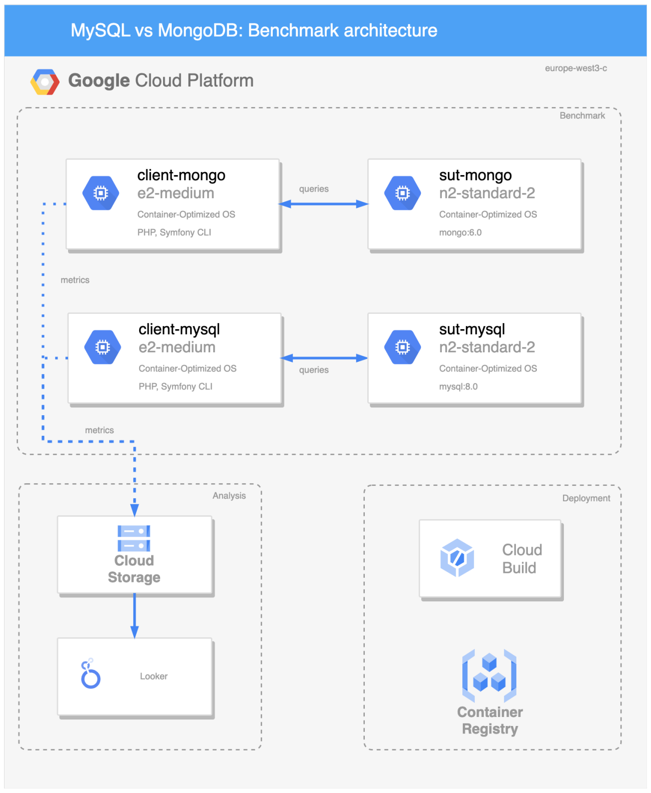

Fig. 1. Architecture of the MySQL vs MongoDB benchmark

B. Client

Let us zoom in on the client, look at its inner workings, and show the different sub-benchmarks it runs against the SUT. The client executes a benchmarking sequence which consists of multiple runs – sub-benchamrks. To assess the performance of the database both from the read and write perspective, these sub-benchmarks fall into these two categories: *Insert Reqests* (write) and *Query Reqests* (read). For insert requests, the size and depth of the inserted document vary between runs. For query requests, the size of the dataset and the query type vary. Datasets of 1, 10, and 100 thousand documents are used, and there are three types of queries:

- *Query UUID*: Queries a document where a uuid attribute in the root of a document exactly matches a specific value.
- *Query deep UUID*: Queries a document where uuid attribute palced seven levels deep in the document matches a specific value.
- *Query LIKE*: Queries all documents where a string placed seven levels deep in the document matches contains a specific substring. [1]

Section V breaks down the results and the impact of these variables.

[1]In MySQL this is done using the `LIKE "%substring%"` construct. MongoDB uses regular expressions match `collection.find("attribute": "/.*substring.*/")`.

C. Data

The first step was to test a hypothesis about the impact of the document structure on throughput. The hypothesis was that inserting many identical documents would allow for higher throughput when compared to a document where values are randomly generated. To test this, documents with static values and a document where values were randomly generated using the Faker [2] library were inserted into both databases. While these tests were run only locally – more on that in IV – the difference between static and random documents over a number of test runs on both SUTs was *negligible*. The difference could be attributed to the varying size of values in the randomly generated document.

With the hypothesis rejected, all the Insert sub-benchmarks work with static documents. Some variation needs to be added when seeding the database prior to running the Query sub-benchmarks in order to be able to filter the dataset in a non-trivial way. For both types of experiments, the documents and the traffic are synthetic.

When inserting, there are three types of documents:

- *Small*: A tiny document with one attribute-value pair.
- *Deep*: Still a relatively small document. However, there are seven levels of nesting. [3].
- *Long*: Anonymised audit log document taken from a production system. It has roughly 50 attributes, value data types vary, and the document also contains multiple nested objects.

D. Benchmark run

After the system is deployed and a configuration setup is completed in the SUTs, the client creates its local logs file. Then the client starts sending requests to the target SUT.

First, a series of insert benchmarks starting with inserting small documents, then deep documents, and finally, long documents. Each experiment type runs five times for the duration of two seconds and ten times for the duration of ten seconds.

Once Insert experiments are complete, the Query experiments start. Before the Query experiment starts sending query requests, it populates the database with a varying number of documents. After the experiment runs, the database gets re-set, and all records get dropped. The time it takes to populate and re-set the database is not considered when calculating the throughput. The *Query UUID* experiment runs atop a dataset of 1, 10, 100 thousand documents. For each dataset, the experiment runs five times for a duration of five seconds. After that, the *Query deep UUID* and *Query LIKE* experiments run atop the 10-thousand-document dataset. Each of these two experiments runs five times for five seconds per run.

IV. Implementation

This section describes in detail the implementation of the benchmarking client. It explains the design patterns that were used in order to make it easy to extend the client in the future to add more experiments or even to benchmark other databases. Then we focus on the docker-compose setup that allows for local prototyping. The final aspect of implementation is deploying this setup into GCP and the script that makes this possible.

The benchmarking client is implemented in PHP 8.1; it uses a suite of packages from the contributte [4] ecosystem, a library to work with google cloud storage, and the aformetiond Faker library to produce random data.

[2]https://fakerphp.github.io/

[3]`it.is.turtles.all.the.way.down`

[4]https://contributte.org/

A. Database layer abstraction

Using a database abstraction in the benchmarking client provides a number of benefits. First, it centralizes the implementation details of interacting with each database, making it easier to maintain and update the codebase. Second, it provides a consistent interface to interact with each database, making it easier to compare their performance. Finally, adding new databases to compare is straightforward as long as the database client implements the required interface.

To achieve this, there is a singleton service called `Client` through which the rest of the codebase can interact with the database. When the benchmarking client starts, the service decides which one of the supported adapters it should use. This is done through an environment variable or starting the client with the `--target <target>` CLI parmater.

Each adapter must implement four methods, which are *seed*, *read*, *write*, and *teardown*. The seed method is used to initialize the database before running the benchmark. For MongoDB, no seed is needed, while for MySQL, a new database schema needs to be created. The read method is used to execute a select query. The write method is used to execute an insert query. Finally, the teardown method resets the database to its original state.

For the purposes of this study, adapters for MySQL and MongoDB were implemented.

B. Sub-benchmarks

As discussed earlier, the entire benchmark run consists of batches of sub-benchmarks. Each sub-benchamrk tests the impact of a vraible change on the performance of the two SUTs. Figure 2 shows the hierarchy of classes implemented in the benchamrking client to capture this structure. This design pattern has the following benefits:

- Lifecycle of a sub-benchmark can be defined.
- Each sub-benchmark can only implement the parts of the lifecycle that differ from others. This promotes consistency: e. g. the collection of metrics is done in one place.
- Adding a new sub-benchmark is straightforward without duplicating a lot of code.

When executing a sub-benchmark batch, we specify the number of repeats, the run time, and the number of documents to be inserted before the sub-benchmark commends. The lifecycle then comprises of `before` and `after` methods that

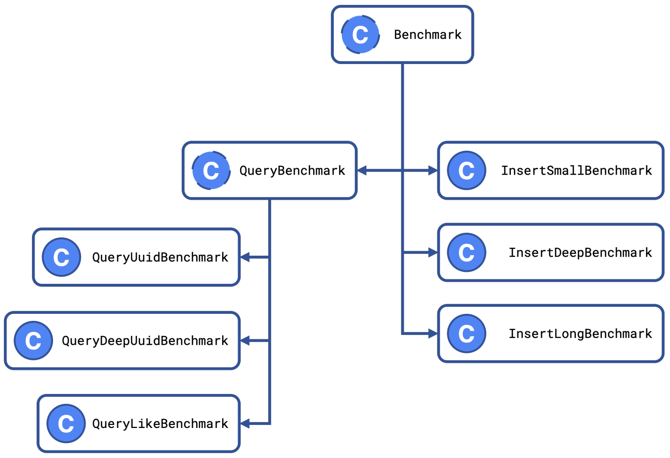

Fig. 2. Hierarych of benchmark classes in benchmarking client

execute operations prior to and after the batch runs. Their duration is not measured as part of the runtime and thus does not affect the measured throughput. For example, the `before` method can insert documents into the database before a batch of sub-benchamrks querying these documents gets executed. For each sub-benchmark that runs within a batch, we can then define what happens at the start and in the end using the `setup` and `tear_down` methods. Finally, each benchmark runs for a specified amount of time, and the operations (e. g. insert a document) that are repeatedly being executed during this period of time are defined in the `do` method. The number of successful operations is recorded and used to calculate throughput.

C. *Local development and testing*

For local development, this study makes use of Docker Compose. Thanks to this tool, we can define and run multi-container Docker applications. It enables quicker development and testing of the benchmarking client. The Docker Compose setup in this study includes three services: `client`, `sut-mysql`, and `sut-mongodb`.

This setup saves both time and money since there is no need to deploy the infrastructure every time a new sub-benchmark is tested. The infrastructure can be set up once using Docker Compose and then be utilized to test multiple sub-benchmarks. Reusing the infrastructure eliminates the need for manual setup, reducing the time and effort required to perform the testing.

D. *Deployment to the clouds*

Deploying the benchmarking client to GCP involves several steps. First, the Cloud Build service is used to build the client benchmarking image, which is then stored in Google Container Registry. The image is then referenced and pulled from the registry to set up the two client VMs (see 1. All the VMs in this infrastructure are configured to run an operating system optimized for containers. This setup enables the benchmarking client to run efficiently and effectively within the GCP environment.

In addition, the GCP internal DNS service enables VMs to be referred to by their names rather than requiring the manual management of IP addresses. The GCP internal DNS automatically registers the names of the VMs and assigns them an IP address, enabling seamless communication between the VMs. This maps onto the Docker Compose described in the previous sub-section, so no changes are needed between the local and cloud environments.

To automate the process of local testing, deployment, all the way to the download of recorded data from the storage bucket, a custom script called `toolbox` is part of the repository. Thanks to this script, all these tasks can be performed by executing a single command.

V. Results

This section presents the results and answers the research question.

A. *Inserting documents*

In the first set of experiments, the benchmarking client inserts different types of documents (as described in III-C) into MySQL and MongoDB. For each document type, we run two experiments - the first batch runs for two seconds and gets repeated five times, and the second batch runs for 10 seconds and gets repeated ten times. We end up with 15 throughput values for each document type and database. Figure 3 shows a violin plot representing the distribution density for each document type and each dataabse. We observe significantly lower throughput for MySQL. MongoDB seems to be more sensitive to the type of document as we can see propotionally more significant drop in throughput when comparing the insertion of a small and long document. This observation seems to support our initial hypothesis.

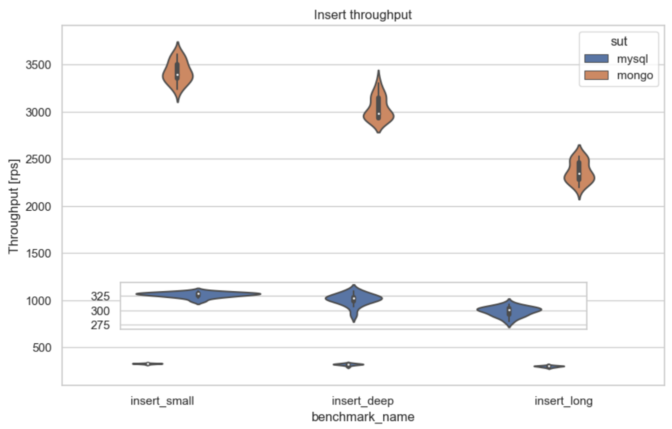

Fig. 3. Throughput distributions when inserting different types of documents.

B. *Querying uuid - exact match*

In this experiment, we seed both databases with 1, 10, and 100 thousand documents. For each of these datasets, we then run five experiments querying one of the existing uuids for five seconds. Figure 4 shows the relative throughput differences between MySQL and MongoDB. As opposed to inserting the results here are much closer together. However, Mongo DB can still achieve higher throughputs regardless of the dataset size.

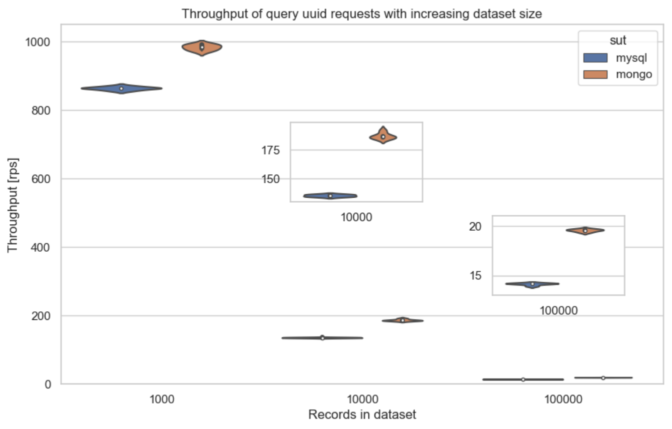

Fig. 4. Throughput distributions when querying a document by uuid in datasets of increasing sizes.

C. Varying queries

In this final experiment, the dataset size we will query is fixed at 10 thousand documents. Instead, the type of query varies: `query_uui` experiment is the same as the previous section; `query_deep_uuid` also selects a document by an exact match, in this case, the uuid attribute is nested seven levels deep inside of the document; finally, the `query_like` experiment searches for a document where a string value contains a sub-string. Each experiment runs for 5 seconds and is repeated five times. In figure 5, we again observe higher throughput achieved by MongoDB for all these experiments. As opposed to the previous query experiment, we can see that MySQL performs relatively worse compared to MongoDB as the query types are getting more difficult. Where Mongo is able to perform the `query_deep_uuid` and `query_like` experiments at nearly the same throughput, we can see a large dropoff in the case of MySQL.

D. Evaluation

Throughout these experiments, we consistently observe better performance from MongoDB. This confirms our hypothesis that MongoDB, purpose-built for storing documents, will perform better when compared to MySQL. In the next section, we discuss the implications of these results, and we elaborate on the possible limitations of this study.

VI. Discussion

In this section, we discuss the results and their implications for real-world situations. We also elaborate on the limitations and possible future work.

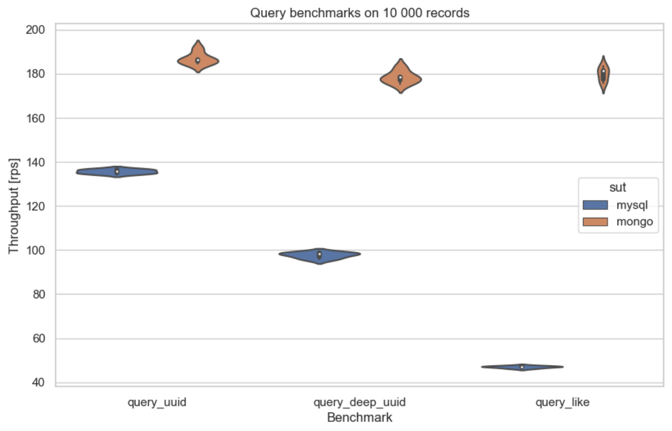

Fig. 5. Throughput distribution of different types of query requests in a dataset with 10 000 documents.

A. Implications

The findings of our analysis show that MongoDB outperforms MySQL when inserting and querying documents, with greater differences in throughput observed during the insertion phase. However, it is important to consider the specific needs of the application in question, as this performance advantage may not necessarily translate to all use cases.

For applications that prioritize write-heavy workloads, MongoDB may be a more suitable choice. On the other hand, for applications where read operations are more common, the differences in performance between the two databases may be less pronounced. Building on the example from the introduction chapter: If a development team already uses MySQL in a project when they encounter a NoSQL use case, with MySQL 8.0, they might benefit from simply using MySQL's JSON data type for document storage rather than provisioning MongoDB.

It should also be noted that our analysis focused solely on throughput and did not consider factors such as the scalability of databases. In cases where scaling or the amount of data is a major concern, more research would be needed to provide insight into the impact on the performance of these two databases.

B. Limitations

This study's limitations should be considered when interpreting the results. Firstly, it is important to note that the databases were running on their default configurations without any specific optimizations. It is possible that adding indexes, for example, could have a large impact on both the insert and query operations. Secondly, this study only varied a limited set of parameters, including the query type, type of document, and dataset size. In order to gain a more comprehensive understanding of the performance of these databases, it would be beneficial to explore additional variables and factors, such as the use of more realistic trace-based or synthetic traffic that combines both write and read operations in the same experiment.

C. Future work

Several avenues for future research can build upon the findings of this study. Firstly, the limitations of this study could be addressed by running experiments that combine read and write operations or studying the impact of additional variables. This could enable a more comprehensive analysis of the performance of the databases under conditions that more closely mimic their real-world usage.

Another area of research that could be explored is the use of JOIN[5] operations in both MySQL and MongoDB. This would provide insight into how well MySQL can connect the SQL and NoSQL paradigms and how MongoDB can approach such scenarios. Furthermore, the analysis could focus on how the performance of the databases changes when JOIN operations are introduced into the experiments.

Finally, the economic perspective could be considered by analysing the vertical scaling of the MySQL database using the resources saved by not provisioning a new MongoDB workload. This would provide an insight into how the two systems perform in terms of economics when compared against each other. Additionally, this could help identify scenarios where one database system may be more suitable than the other based on economic considerations.

VII. Conclusion

In this paper, we aimed to answer the research question of whether MySQL could compete with MongoDB when inserting and querying JSON documents in a NoSQL fashion. Our results show that MongoDB performs better than MySQL in all tested scenarios, confirming our hypothesis. The difference between the two systems is more pronounced when inserting documents, indicating that for write-heavy applications, MongoDB might bring significant benefits. However, in scenarios where read operations are more dominant, it might be worth considering using the NoSQL features that MySQL 8.0 provides.

Although our study provides valuable insights into the performance of MySQL and MongoDB, it is important to consider the limitations of our approach. Our experiments were performed using the default configuration of the databases without any optimization or indexing. Furthermore, we only varied a limited number of parameters, leaving room for additional variables to be explored in future research. For example, experiments including a *join* operation could provide further insights.

Finally, it might be worth considering the economic aspect of adding another database to an existing infrastructure. While our experiments focused on the throughput of the databases, other factors such as licensing, maintenance costs, and skills within the organization also play a role in the decision-making process. Future research could explore these economic factors to provide a more comprehensive picture of the pros and cons.

[5]MongoDB introduced a JOIN-like operation called `$lookup` in version 3.2

References

[1] db-engines "DB-Engines Ranking - Trend of Relational DBMS Popularity". https://db-engines.com/en/ranking_trend/relational+dbms (accessed Mar. 12, 2023).

[2] C. Bell, "Introducing the MySQL 8 Document Store". Berkeley, CA: Apress, 2018. doi: 10.1007/978-1-4842-2725-1.

[3] db-engines "DB-Engines Ranking - Trend of Document Stores Popularity" https://db-engines.com/en/ranking_trend/document+store (accessed Mar. 12, 2023).

[4] N. Dilkov, "Analysis and comparison of document-based databases with SQL relational databases: MongoDb vs MySQL". Proceedings of the International Conference on Information Technologies (InfoTech-2018), 2018.

[5] E. Andersson and Z. Berggren, 'A Comparison Between MongoDB and MySQL Document Store Considering Performance', 2017.

[6] Foster, E.C., Godbole, S. "Overview of MySQL. In: Database Systems". Apress, Berkeley, CA, 2016 https://doi.org/10.1007/978-1-4842-1191-5_24

[7] E. Mehmood and T. Anees, "Performance Analysis of Not Only SQL Semi-Stream Join Using MongoDB for Real-Time Data Warehousing," in IEEE Access, vol. 7, pp. 134215-134225, 2019, doi: 10.1109/ACCESS.2019.2941925.

[8] A. Chauhan, "A review on various aspects of MongoDB databases". International Journal of Engineering Research & Technology (IJERT), 2019, 8(05), pp.90-92.

[9] D. Bermbach, E. Wittern, and S. Tai, "Cloud Service Benchmarking". Cham: Springer International Publishing, 2017. doi: 10.1007/978-3-319-55483-9.