Application-Based Benchmarking on Redis and MongoDB for Trip Planning using GTFS Data

Mustafa Alzaidi [1], Aniko Vagner [1]

[1] Department of Information Technology Faculty of Informatics University Of Debrecen, Kassai ut 26, Debrecen, Hungary

Abstract **– Benchmarking serves as the foundation for selecting a database in any project. The available benchmarking tools evaluate system performance by subjecting it to random data and a set of arbitrary operations, without considering the specific characteristics of the application. The problem with these tools is that they reflect unrealistic benchmarks as they do not consider the nature, sequence, and type of queries the application will send to the database. In this paper, we introduced the approach of benchmarking the database based on the nature of interaction and queries between the application and database, and we built a benchmarking tool using Java to benchmark Redis and MongoDB as databases for a trip planning application with GTFS data of Budapest local transport data. Our study involved comparing the performance of both databases under ten different stress levels by simulating the number of querying clients. The results show that both database's performance is slightly decreased while increasing the number of clients (stress). However, Redis shows better performance compared to MongoDB.**

Keywords **– Benchmarking, trip-planning, NoSQL, Redis, MongoDB, GTFS.**

DOI: 10.18421/TEM124-70
https://doi.org/10.18421/TEM124-70

Corresponding author: Mustafa Alzaidi,
Department of Information Technology Faculty of Informatics University Of Debrecen, Kassai ut 26, Debrecen, Hungary
Email: mustafa.alzaidi@mailbox.unideb.hu

Received: 02 August 2023.
Revised: 30 October 2023.
Accepted: 14 November 2023.
Published: 27 November 2023.

1. Introduction

Organizations require efficient and reliable databases to support critical operations in today's fast-paced business environment. With the proliferation of various database technologies, it becomes increasingly difficult for organizations to determine the best fit for their specific use cases and requirements. A database benchmarking tool is essential in this context.. Users like analysts and data scientists commonly start the data science procedure by viewing potentially enormous volumes of data via interactions with a graphical user interface, often a data visualization software [1], [2], [3], [4]. However, the underlying data must be processed each time a user interacts with the UI (filtered, aggregated, etc.) and provide fast responses and interactions for the UI user [5], [6], [7]. The database and visualization communities have created several approaches, such as approximate query processing

[8], [9], web-based progressive [10], [11], [12], speculative query execution [13], [14], data cubes [13], [15], [16], spatial indexing [17], and lineage tracking [18], to fit this increasing demands for interactive and real-time performance.

Currently, there are insufficient benchmarks to experimentally determine which of the existing systems give reasonable performance and which systems are superior to others for real-time interactive-querying applications. This problem is made worse by the most demanding and widely used visualization scenarios, like crossfilter [19], [20], – [21], where one interaction with the database may result in hundreds of requests being sent out per second with a requirement for almost instantaneous response. Unfortunately, current database benchmarks like the Star Schema Benchmark (SSB) [22], TPC-DS Benchmark [23], and TPC-H [24] are inadequate for making these comparisons because the workloads depicted in these benchmarks do not accurately reflect how database queries are produced by user activities, such tools like Tableau [25] or Spotfire [20].

Some research benchmark database systems using interactive workload [26], [27], [28], [29], [30] while others use tools like Yahoo Cloud Service Benchmarking tool(YCSB)[31], [32], [33]and HammerDB [34]. In both approaches, a predefined static set of operations is to be performed as the workload; for example, a given workload can perform 1000 operations of 950 read and 50 updates. The problem with these proposals is that the benchmarking does not consider the real-life use cases by the user or required by the application operation scenarios itself. Thus, the benchmark here will not reflect an accurate evaluation of the database for a specific application.

Although research [35] tries to solve these problems and limitations by proposing the approach of benchmarking databases based on user interaction, they depend on visual data exploration, which cannot fit all types of applications. Therefore, as a contribution of this paper, we will introduce the idea of benchmarking the database based on the application nature and expected use case scenarios. First, we build a benchmarking tool using Java designed to generate a series of queries that can be defined based on the application's expected use cases and the database design structure. Then, we measure the performance under different work pressures by simulating the number of requests and simultaneous database access. This approach is different from other available tools as it benchmarks the database by evaluating the number of completed operations per time unit. By operation, here we mean feature or functionality provided by the application. For example, finding a trip-plan (possible path between two local transport stops) using local transport with General Transit Feed Specification data (GTFS) data can be a feature provided by a trip planning or a map application. Many operations or features within a database often require the execution of multiple sub-queries. For instance, when finding a trip plan, this task may involve one or more queries on various tables, such as the route, stoptimes, and stop tables. The response time for retrieving the data can vary significantly based on the database structure or model used for data storage Thus, the traditional database benchmarking tools like YCSB [36] will not reflect the actual database performance for that application and design, as it perform arbitrary read and write operation without benchmarking the performance of the database system and the application database design together. The proposed benchmarking approach and tool calculate the number of complete operations per time unit the database can respond to. This also involves recording the number of sub-quires or database hits, which, in the meantime, provide the same information provided by traditional benchmarking tools.

In this work, we use the trip planning application for GTFS for Budapest city local transport data. GTFS data is a standard format used by transit agencies worldwide to publish their local transport data so that applications like maps or route planning can use these data. As mentioned before, trip planning is the application's feature to find possible routes or trips between two stops using local transport (ex, tram, bus, metro). However, the number of queries (read from database) or the nature of queries (for example, read from hash or lList, or level of joining operation between tables) varies based on the database design for each trip plan depending on the location of the start and end stop points. Thus, using traditional database benchmarking can be unrealistic. In this research, we will benchmark two databases used to store GTFS data, Redis, and MongoDB, and define two models for storing GTFS data in both database systems. We will overview our Java benchmarking tool implementation and use it to evaluate the performance of these two databases in different cases of stress (concurrent user access) and compare both databases' results.

2. Benchmarking

Database benchmarking is a technique used to evaluate the performance of a database system. It involves running a series of tests that simulate the workload of the system and measuring its response time, throughput, and scalability. These tests help to identify bottlenecks, optimize the database configuration, and compare the performance of different database systems.

One of the popular benchmarking tools is Yahoo Cloud Service Benchmarking (YCSB) [33], [37], an open-source tool designed to measure the performance of cloud databases. YCSB supports various NoSQL databases, including Apache Cassandra, MongoDB, and Redis. It provides a set of workloads, such as read-heavy, write-heavy, and mixed workloads, that try to simulate applications' read and write operations. YCSB tool measures the performance of the database system in terms of throughput, latency, and scalability. Throughput refers to the number of operations that the system can handle per second. Latency measures the time it takes for the system to respond to a request. Finally, scalability measures how well the system can handle an increasing workload. The limitation of YCSB is that the workload does not reflect the real application data. The benchmark operation is a set of arbitrary read, write, or update operations that do not reflect the actual application behavior.

3. GTFS Data

This work will benchmark Redis and MongoDB performance for route planning applications using GTFS data. GTFS (General Transit Feed Specification) is a data format created by Google to describe public transportation schedules and related information [38], [39]. Many transit agencies use this format worldwide to provide data to application developers, allowing them to develop tools and services that make it easier for people to use public transit.

GTFS data is typically organized into a set of data tables [40], [41], each containing information on a specific aspect of the transit system. GTFS data include many tables. Below, we describe the tables that we will use during this paper as its related to the route planning process:

Stops: Contains information about individual stops, including their ID, name, location (latitude and longitude), and other details such as wheelchair accessibility.

Routes: Contains information about transit routes, including their ID, name, and type (e.g., bus, subway, train).

Trips: Contains information about individual trips on each route, including their ID, route ID, and other details such as the scheduled start and end times.

StopTimes: Contains information about the scheduled arrival and departure times of vehicles at each stop for each trip.

GTFS contains other tables like Calendar, Calendar Dates, Fare Attributes, Fare Rules, and Agency.

These tables are typically stored in comma-separated values (CSV) format and can be easily imported into databases or used in programming languages to build applications that use transit data.

4. Methodology

Our proposed benchmarking methods involve three steps, identify the application-database use case scenario and main application operations, define the data storage model in each database, and perform data queries based on these models. Next, we will describe this approach using GTFS data trip planning as an application example and Redis with MongoDB as NoSQL database.

4.1. Gtfs Trip Planning Database Interaction

Trip planning for local transport using GTFS data can be defined as the algorithm that takes a starting point, a destination point, and a desired departure or arrival time as input and uses GTFS data to provide a set of possible recommended transit options to reach the destination.

The algorithm would perform the following steps [42]:

- Identify all transit routes that pass through the start and end stops.
- For each candidate route, identify the sequence of stops along the route and the scheduled departure and arrival times at each stop let call this candidate stop set CSS.
- For each stop in CSS if the stop is the destination stop, then add the set of the route leading to it to the solution list else, repeat steps one and two operations for the stop.
- Depending on criteria like the maximum number of transit between routes and the total travel time. If the criteria are not met, stop searching further from that route.

For our benchmarking purpose, the details of retrieving the data from the GTFS tables include routes, stops, and trip data. However, we will ignore the timing information as the following data interaction is a major part of the trip planning, and it is enough for benchmarking. Moreover, the timing data is stored in the stoptimes file, which is already benchmarked here. Therefore, the benchmarking steps will start by picking up a random stop as a start-stop and performing all the trip planning algorithm steps starting from that stop. Note that there is no need to repeat the operations until finding the destination as one iteration of the search will lead to executing all the queries types involved in the full trip planning. The set of benchmark step will be as follow:

- Pick a random stop from the stoptimes table as start-stop,
- Search the stoptimes file to get all trips that pass through the random stop and call it Trips set.
- For each trip, get the trip information from the trips table,
- For each trip, get the route information from the routes table

Next, we will describe our candidate structures for storing GTFS data in both Redis and MongoDB.

4.2. Redis GTFS Model

We use Redis hash to represent the data table where each row is stored in a corresponding hash. Redis hash structure is identified by a unique key and contains a set of field-value pairs. We used the field to store the column headers and the value field to store the corresponding value at the row stored in the hash.

First, we form the key by concatenating the table name and primary key value, then all the foreign keys are separated by the "_" character. This format will create a unique identifier for each table row in the GTFS data as follow:

"TableName_PrimeryKey_ForeignKey1_

ForeignKey2_…._ ForeignKeyN".

Note that there is no primary or foreign key for some tables; in that case, it is replaced by a blank. Figure 1 below shows how the stoptimes file is stored in Redis.

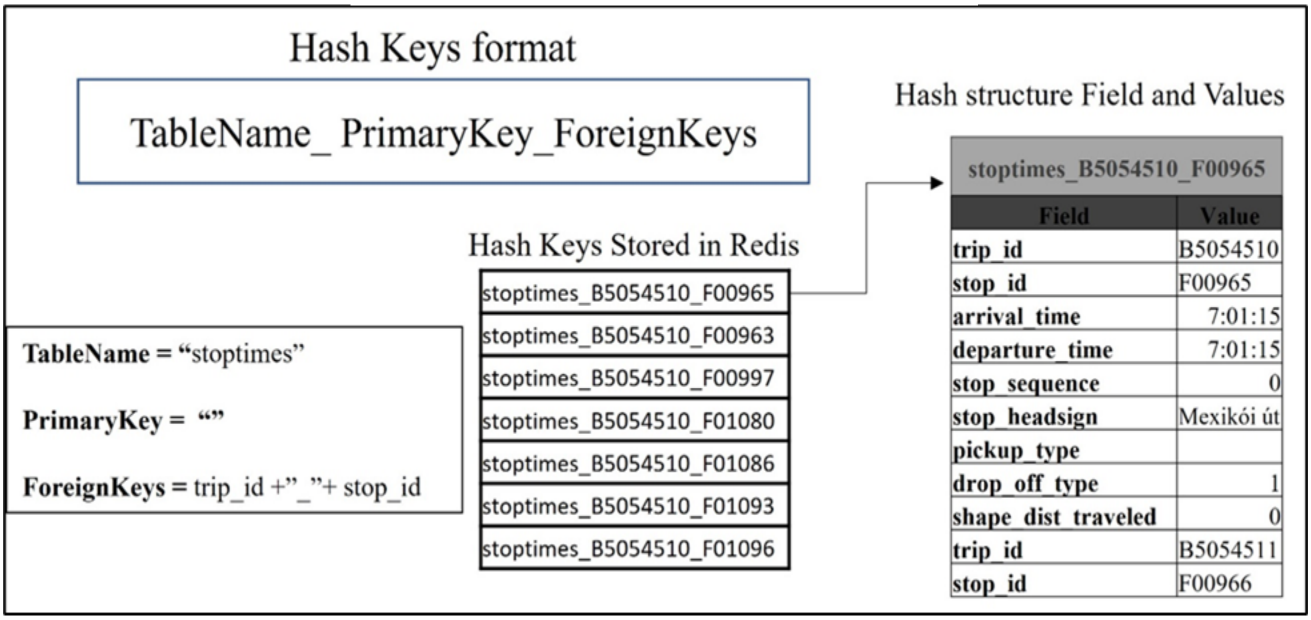

Figure 1. Stoptimes file entity stored in Redis

Thus, the number of keys stored in Redis equals the number of rows in all GTFS data tables. The scan command can be used to retrieve the corresponding hash. For example, to retrieve all the trips that pass through the stop with ID B5054510, we can use the following command: **Scan stoptimes_B5054510_* 0.** By the nature of the scan command, it may be used many times while updating the cursor pointer to retrieve all the keys from the database. Performance variation is expected here, which makes benchmarking more demanding for such an application.

The cycle for fetching the data from this structure includes using the scan command starting with the cursor equal to zero, collecting the set of the key returned by the first scan call using the returned cursor value to start another scan, and repeating the operation till get cursor value equal to zero again (that mean no more key to be found in Redis). For retrieving the trips and route data from the trips and routes table, we can use a simple HGET command as we have the whole key and no need to use the scan for pattern matching. The HGET command will return the data Q(1) complexity and use Redis's fast response as an in-memory database.

4.3. MongoDB Model for GTFS Data

We can define a MongoDB collection to represent each type of GTFS entity, such as stops, routes, trips, and schedules. Each document in the collection will represent a single entity and contain fields corresponding to the entity's properties.

For example, Figure 2 is an example of a MongoDB document that represents a GTFS stop entity unlike the Redis database, where we need to use more than one command type, retrieving data from MongoDB document can be done using the find command.

```
{
    "_id":
ObjectId("617912eb39eaf2a2a8d20260"),
   "stop_id": "1000",
   "stop_name": "Grand Central Terminal",
   "stop_lat": 40.752726,
   "stop_lon": -73.977229
}
```

Figure 2. MongoDB code store stop eintity

5. Our Benchmarking Tool

We developed our benchmarking tool with Java using Gradle version 7.2. The main classes in the project are shown in Figure 3 below.

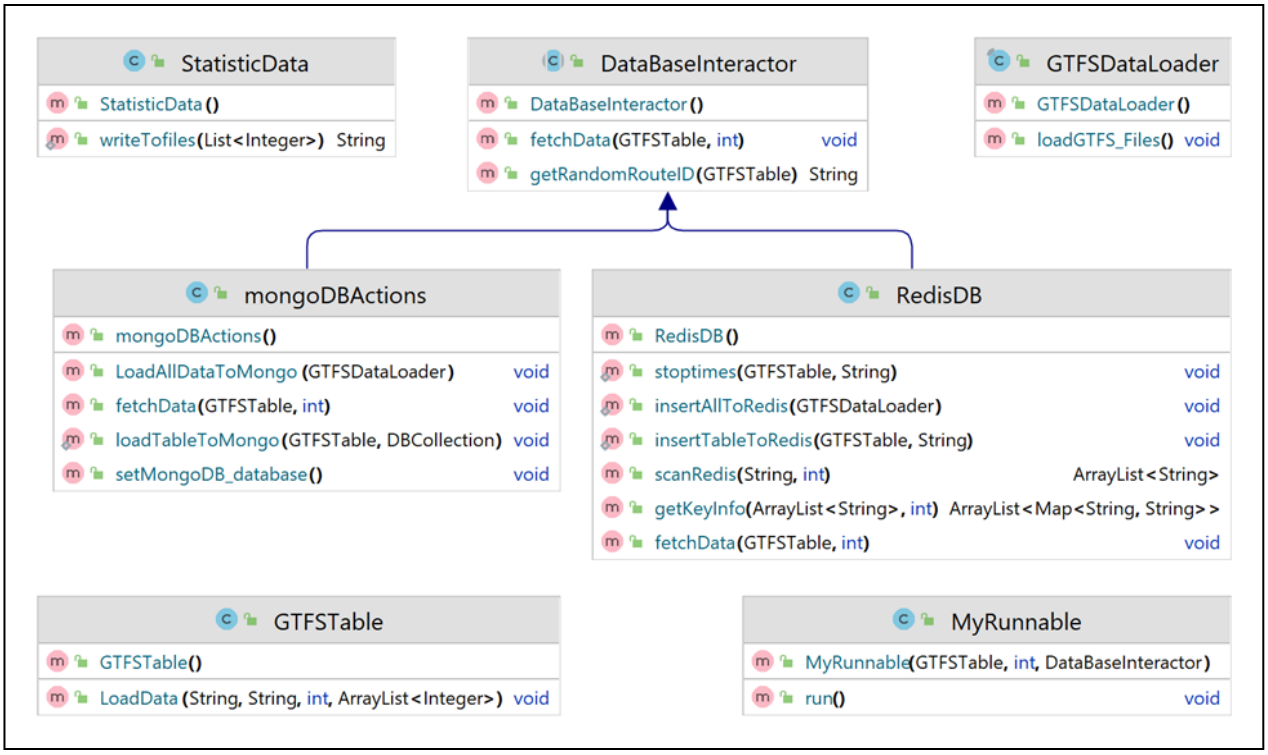

Figure 3. UML design for main classes in the benchmark tool

We used the strategy design pattern so that in the future, support for new databases can be easily added. To support a database, the DataBaseInteractor abstract class must be implemented. To support Redis and MongoDB, RedisAction and MongoDBAction classes implement the DataBaseInteractor abstract class. The whole use case scenario must be implemented using the fechData function. The tool simulates many clients connecting to the database simultaneously using multi-threads. Each thread records the performance information and stores it in the StatisticData object. This data is then exported to CSV files. The tool can be configured to run a specific number of threads for a particular time. Each thread will go in a loop, picking up a random stop as a start-stop and fetching the route planning data for that stop. While looping, the client will record information like how many databases hit are served and how many complete queries are served. By database hit, we mean any read or write to the database, while a complete query is a set of hits that performs a route planning operation.

6. Benchmarking Results

In this section, we will describe the setting we used for our benchmarking tool during the experiments and then overview the results of the experiment.

6.1. Settings

The maximum number of threads and the test duration time must be defined to use our proposed benchmarking tool. Our proposed benchmarking tool can run with a different number of threads to simulate different levels of stress on the database. For example, if the user sets the max number of threads to 100 with a shift window of 10 threads per run, then the tool will start by benchmarking the database with 10 threads and then do a second run with 20 threads, and so on.

Until the last run with 100 threads, which is the max number, this approach can give more details about the database performance with different stress levels and more flexibility for test under different computation power and hardware specifications. This benchmarking tool outputs three types of files. The first type contains the thread ID, and the number of database hits done by each thread.

In contrast, the second type shows the number of complete operations each thread does. The third file contains a summary of all runs together in one table. We run the experiment using a PC with the following hardware specification 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz, 8GB RAM, Windows 11 OS. Under the same specifications, we benchmark MongoDB and Redis databases. We set the Max Thread Number to 110, and the thread shifts up size to 10 and test time to 40 seconds. Thus, the tool will start benchmarking the database using 10 threads for 40 seconds, then another test run with 20 threads for 40 seconds, and so on till the last run, with 110 threads for 40 seconds.

For every run, the tool will output the first two types of files mentioned above. After the last run, the tool will produce the third type file, summarizing all runs and providing easier comparison and visualization.

6.2. Results

To compare the database accessibility, we select three test runs 20 threads, 60 threads, and 110 threads. Table 1 shows the experiment results for the first 10 threads in each test run for MongoDB and Redis.

Table 1. Test results for 20,60 and 110 threads run

Thread ID	MongoDB			Redis		
	Hits 20 threads	**Hits 60 threads**	**Hits 110 Threads**	**Hits 20 Threads**	**Hits 60 Threads**	**Hits 110 Threads**
0	8997	375	1078	28797833	7225407	3210808
1	13086	939	499	27675503	7278314	3237905
2	11116	2790	2865	27929125	7285905	3198596
3	8262	327	970	18741209	7189834	3165443
4	14519	1289	1258	29333027	7426133	3154295
5	22247	169	0	27305631	7151923	3169713
6	10360	645	1374	27938556	7289945	3355994
7	9716	349	489	17456620	7251756	3221892
8	6566	3067	0	29369999	7051873	3115420
9	11463	8213	9146	27462338	7150764	3223875
10	13604	845	3764	27738747	7056592	3099929

The number of hits in the table represents the number of times the thread sent a request to the databases and got the response back.

This data shows that both MongoDB and Redis performance decreased with increasing the number of threads. However, Redis offers faster response times of several million queries per 40 seconds than MongoDB, which serves thousands of queries per 40 seconds. Of course, such results may be shown by any other benchmarking tool.

Still, as we consider benchmarking the databases based on specific application use cases, we will go further and analyze the throughput of finding trip planning results.

Tables 2 and 3 below show the summary of all ten runs with different numbers of threads, including the total number of database hits, the total number of completed operations for 40 seconds, and the number of complete trip planning operations done per second for Redis and MongoDB, respectively.

Table 2. Experiments summary for Redis

No Of Threads	No of DB Hits	No of Complete Operation	DB Hits per/Sec	Complete Operation per/Sec
10	577554675	261525	14438866	6538
20	542528715	243550	13563217	6088
30	461164242	208894	11529106	5222
40	434841597	196189	10871039	4904
50	361386447	163077	9034661	4076
60	431072073	195215	10776801	4880
70	399627851	180677	9990696	4516
80	375112260	169421	9377806	4235
90	365071720	165548	9126793	4138
100	347442017	156039	8686050	3900
110	341046978	154682	8526174	3867

Table 3. Experiments summary for MongoDB

No Of Threads	No of DB Hits	No of Complete Operation	DB Hits per/Sec	Complete Operation per/Sec
10	296746	132	7418	3
20	263563	129	6589	3
30	243914	112	6097	2
40	256990	96	6424	2
50	164201	87	4105	2
60	100021	69	2500	1
70	210965	89	5274	2
80	133628	80	3340	2
90	115265	59	2881	1
100	213504	86	5337	2
110	193595	89	4839	2

Figures 4 and 5 below highlight that the throughput decreases when threads increase for the Redis database. While for MongoDB, the number of threads does not affect the database performance at the same level as Redis. However, Redis's throughput is much more than what MongoDB can provide.

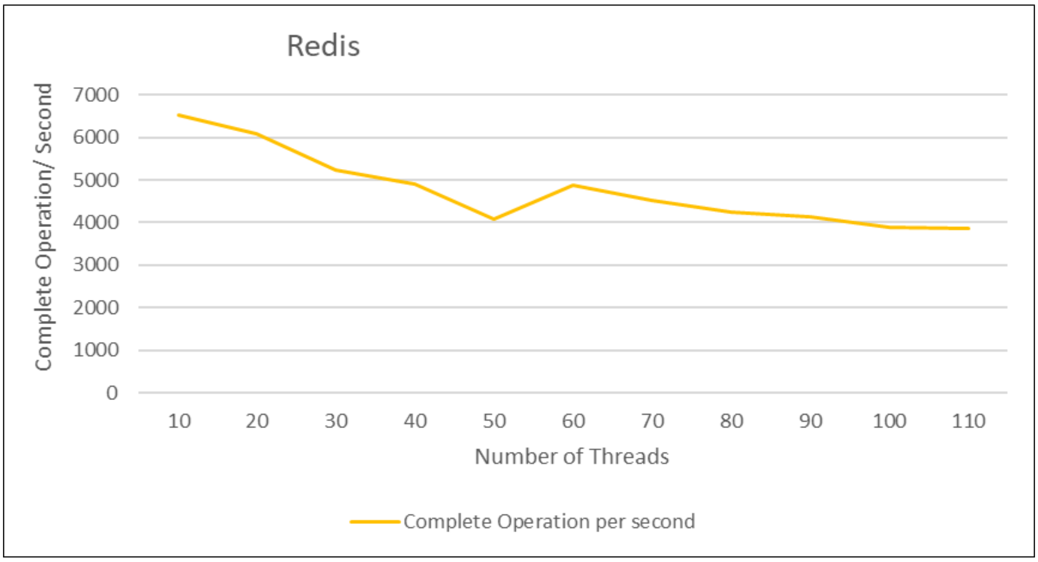

Figure 4. Relation between throughput and number of used threads (Redis)

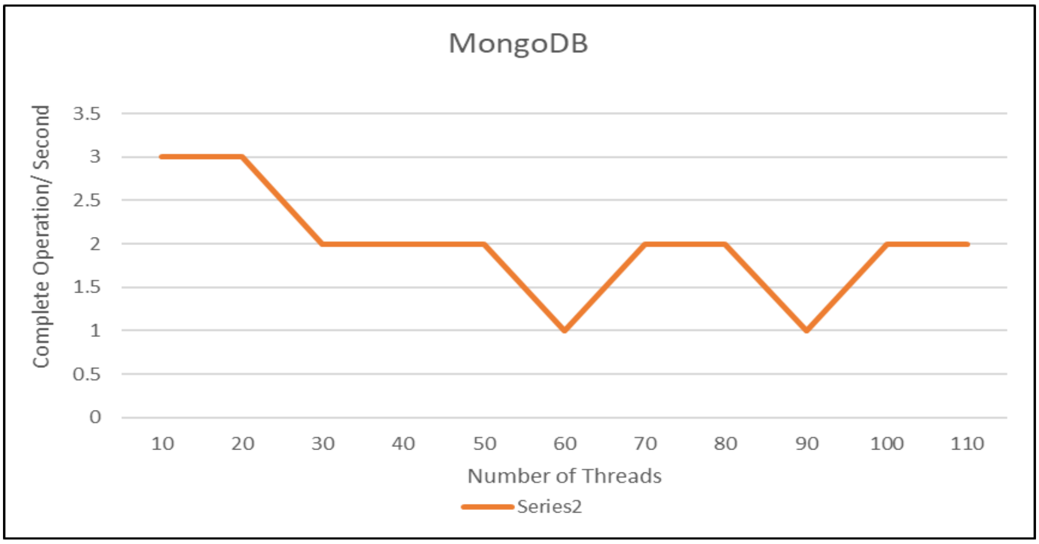

Figure 5. Relation between throughput and number of used threads (MongoDB)

It is important to highlight here that the number of the required query (database hits) to perform one complete trip plan for the same input differs between Redis and MongoDB as each database uses a different model to store the GTFS data, as we described before. Therefore, although the throughput in the above charts depends on the GTFS use scenario, we can still have a benchmark on general query response time if we consider the database hits information in the tables.

7. Conclusion

Selecting the proper database system is essential for any project and application, which increases the need for database benchmarking. Available benchmarking tools like Yahoo Cloud Service Benchmarking Tool (YCSB) evaluate the performance of the database using a predefined workload containing a set of queries that may not reflect the need of the application. Recent research introduced the idea of benchmarking the database depending on user interactions and exploring data. This study proposed a benchmarking tool to evaluate the performance of databases under different stress levels depending on application interaction and use case scenarios. We use a trip planning application for GTFS data of Budapest city to benchmark Redis and MongoDB databases. The tool allows flexible testing by varying the number of threads used to simulate different stress levels on the database. The results showed that the performance of both databases decreased as the number of threads increased, but Redis had a faster response time than MongoDB. However, the study also analyzed the throughput of finding trip planning results and found that Redis had a higher throughput. Still, MongoDB throughput was less affected by the number of threads used in each experiment. It should be noted that the number of required queries to perform a complete trip plan differed between the two databases due to their different GTFS data storage models. Still, the benchmarking tool allowed for a comparison of the general query response time using the database hit output information.

References:

[1]. Alspaugh Nava; Liu Andrea; Jin Cindy; Hearst Marti A., S. Z. (2018). Futzing and Moseying: Interviews with Professional Data Analysts on Exploration Practices. *IEEE Transactions on Visualization and Computer Graphics, 25*(1), 22–31. Doi: 10.1109/tvcg.2018.2865040.

[2]. Battle Jeffrey, L. H. (2019). Characterizing Exploratory Visual Analysis: A Literature Review and Evaluation of Analytic Provenance in Tableau. *Computer Graphics Forum, 38*(3), 145–159. Doi: 10.1111/cgf.13678.

[3]. Card, S. K., Mackinlay, J., & Shneiderman, B. (Eds.). (1999). *Readings in information visualization: using vision to think*. Morgan Kaufmann.

[4]. David John W., F. N. ; T. (1977). Exploratory data analysis. *Biometrics, 33*(4), 768-NA. Doi: 10.2307/2529486.

[5]. Miller, R. B. (1968). Response Time in Man-Computer Conversational Transactions. *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, 267–277. Doi: 10.1145/1476589.1476628.

[6]. Nielsen, J. (1993). *Response Times: The 3 Important Limits.* Nngroup. Retrieved from: https://www.nngroup.com/articles/response-times-3-important-limits/ [accessed: 02 June 2023].

[7]. Shneiderman, B. (1984). Response time and display rate in human performance with computers. *ACM Computing Surveys, 16*(3), 265–285. Doi: 10.1145/2514.2517.

[8]. Agarwal, S., Mozafari, B., Panda, A., Milner, H., Madden, S., & Stoica, I. (2013). BlinkDB: queries with bounded errors and bounded response times on very large data. *Proceedings of the 8th ACM European Conference on Computer Systems*, 29–42.

[9]. Chaudhuri, S., Ding, B., & Kandula, S. (2017). Approximate Query Processing. *Proceedings of the 2017 ACM International Conference on Management of Data*, 511–519. Doi: 10.1145/3035918.3056097.

[10]. Albers, S. (2003). Online algorithms: a survey. *Mathematical Programming, 97*(1), 3–26. Doi: 10.1007/s10107-003-0436-0.

[11]. Fekete Danyel, Nandi Arnab, Sedlmair Michael, J.-D. F. (2019). Progressive Data Analysis and Visualization (Dagstuhl Seminar 18411). In *Dagstuhl Reports, 8*(10). Doi: 10.4230/dagrep.8.10.1.

[12]. Hellerstein Peter J.; Wang Helen J., J. M. ; H. (1997). Online aggregation. *ACM SIGMOD Record, 26*(2), 171–182. Doi: 10.1145/253262.253291.

[13]. Battle, L., Chang, R., & Stonebraker, M. (2016, June). Dynamic prefetching of data tiles for interactive visualization. In *Proceedings of the 2016 International Conference on Management of Data*, 1363-1375. Doi: 10.1145/2882903.2882919.

[14]. Kamat, N., Jayachandran, P., Tunga, K., & Nandi, A. (2014). Distributed and interactive cube exploration. *2014 IEEE 30th International Conference on Data Engineering*, 472–483.

[15]. Lins James T., Scheidegger Carlos, L. K. (2013). Nanocubes for Real-Time Exploration of Spatiotemporal Datasets. *IEEE Transactions on Visualization and Computer Graphics, 19*(12), 2456–2465. Doi: 10.1109/tvcg.2013.179.

[16]. Liu Biye, Heer Jeffrey, Z. J. (2013). imMens : real-time visual querying of big data. *Computer Graphics Forum, 32*, 421–430. Doi: 10.1111/cgf.12129.

[17]. Tao Xiaoyu, Wang Yedi, Battle Leilani, Demiralp Çağatay, Chang Remco, Stonebraker Michael, W. L. (2019). Kyrix: Interactive Pan/Zoom Visualizations at Scale. *Computer Graphics Forum, 38*(3), 529–540. Doi: 10.1111/cgf.13708.

[18]. Psallidas, F., & Wu, E. (2018). Provenance for Interactive Visualizations. *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, 1–8. Doi: 10.1145/3209900.3209904.

[19]. Moritz, D., Howe, B., & Heer, J. (2019). Falcon: Balancing interactive latency and resolution sensitivity for scalable linked visualizations. In *Proceedings of the 2019 CHI conference on human factors in computing systems*, 1-11. Doi: 10.1145/3290605.3300924.

[20]. Spotfire. (1995). *TIBCO Spotfire*. Tibco. Retrieved from: https://www.tibco.com/products/tibco-spotfire [accessed: 05 June 2023].

[21]. Tweedie, L., Spence, B., Williams, D., & Bhogal, R. (1994, April). The attribute explorer. In *Conference companion on Human factors in computing systems*, 435–436 Doi: 10.1145/259963.260433.

[22]. O'Neil, P., O'Neil, E., Chen, X., Revilak, S. (2009). The Star Schema Benchmark and Augmented Fact Table Indexing. In Nambiar, R., Poess, M. (eds) *Performance Evaluation and Benchmarking. TPCTC 2009. Lecture Notes in Computer Science*, *5895*. Springer, Berlin, Heidelberg. Doi: 10.1007/978-3-642-10424-4_17.

[23]. TPC. (n.d.). *TPC-DS*. TPC. Retrieved from: http://www.tpc.org/tpcds/, [accessed: 17 June 2023].

[24]. TPC. (n.d.). *TPC-H. TPC*. Retrieved from: http://www.tpc.org/tpch/, [accessed: 18 June 2023].

[25]. Stolte Diane L.; Hanrahan Pat, C. T. (2002). Polaris: a system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics, 8*(1), 52–65. Doi: 10.1109/2945.981851.

[26]. Battle, L., Chang, R., Heer, J., & Stonebraker, M. (2017, October). Position statement: The case for a visualization performance benchmark. In *2017 IEEE Workshop on Data Systems for Interactive Analysis (DSIA)* 1-5. IEEE. Doi: 10.1109/dsia.2017.8339089.

[27]. Eichmann, P., Zgraggen, E., Zhao, Z., Binnig, C., & Kraska, T. (2016). Towards a Benchmark for Interactive Data Exploration. *IEEE Data Eng. Bull., 39*, 50–61.

[28]. Idreos, S., Papaemmanouil, O., & Chaudhuri, S. (2015). Overview of data exploration techniques. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 277–281.

[29]. Jiang, L., Rahman, P., & Nandi, A. (2018). Evaluating interactive data systems: Workloads, metrics, and guidelines. *Proceedings of the 2018 International Conference on Management of Data*, 1637–1644.

[30]. Tang, N., Wu, E., & Li, G. (2019). Towards democratizing relational data visualization. *Proceedings of the 2019 International Conference on Management of Data*, 2025–2030.

[31]. Cooper, B., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010). Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10*, 143–154. Doi: 10.1145/1807128.1807152.

[32]. Claesen, C., Rafique, A., Van Landuyt, D., & Joosen, W. (2022). A YCSB Workload for Benchmarking Hotspot Object Behaviour in NoSQL Databases. In *Performance Evaluation and Benchmarking: 13th TPC Technology Conference, TPCTC 2021, Copenhagen, Denmark, August 20, 2021, Revised Selected Papers 13*, 1-16. Springer International Publishing.

[33]. Madushanka, T., Mendis, L., Liyanage, D., & Kumarasinghe, C. (2015). Performance Comparison of NoSQL Databases in Pseudo Distributed Mode: Cassandra, MongoDB & Redis. *Researchgate*.

[34]. Chen, Y., Xie, X., Wu, J. (2015). A Benchmark Evaluation of Enterprise Cloud Infrastructure. In Cheng, R., Cui, B., Zhang, Z., Cai, R., Xu, J. (eds) *Web Technologies and Applications. APWeb 2015. Lecture Notes in Computer Science(), 9313*. Springer, Cham. Doi: 10.1007/978-3-319-25255-1_68.

[35]. Battle, L., Eichmann, P., Angelini, M., Catarci, T., Santucci, G., Zheng, Y., Binnig, C., Fekete, J.-D., & Moritz, D. (2020). Database Benchmarking for Supporting Real-Time Interactive Querying of Large Data. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 1571–1587. Doi: 10.1145/3318464.3389732.

[36]. Ose, O., Okokpujie, K., Nkordeh, N., Ndujiuba, C., John, S., & Uzairue, I. (2018). Performance Benchmarking of Key-Value Store NoSQL Databases. *International Journal of Electrical and Computer Engineering (IJECE), 8*(6), 5333. Doi: 10.11591/ijece.v8i6.pp5333-5341.

[37]. da Silva, L. F., & Lima, J. V. F. (2021). An evaluation of Cassandra NoSQL database on a low-power cluster. *2021 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, 9–14. Doi: 10.1109/SBAC-PADW53941.2021.00012.

[38]. Velasquez, R., Rodriguez, F., Vargas Martin, M., & Ponce, J. (2020). Mapping of the Transportation System of the City of Aguascalientes Using GTFS Data for the Generation of Intelligent Transportation Based on the Smart Cities Paradigm, 177–185. In Botto-Tobar, M., León-Acurio, J., Díaz Cadena, A., Montiel Díaz, P. (eds) *Advances in Emerging Trends and Technologies. ICAETT 2019. Advances in Intelligent Systems and Computing, 1066*. Springer, Cham. Doi: 10.1007/978-3-030-32022-5_17.

[39]. Wessel, N., & Widener, M. J. (2017). Discovering the space–time dimensions of schedule padding and delay from GTFS and real-time transit data. *Journal of Geographical Systems*, *19*(1), 93–107. Doi: 10.1007/s10109-016-0244-8.

[40]. Q. Zervaas. (2014). *The Definitive Guide to GTFS: Consuming open public transportation data with the General Transit Feed Specifcation (1st ed).* Gtfsbook. Retrieved from: http://gtfsbook.com/gtfs-book-sample.pdf [accessed: 19 June 2023].

[41]. Fortin, P., Morency, C., & Trépanier, M. (2016). Innovative GTFS data application for transit network analysis using a graph-oriented method. *Journal of Public Transportation*, *19*(4), 18-37. Doi: 10.5038/2375-0901.19.4.2.

[42]. Mustafa Alzaidi, Aniko Vagner. (2021). Trip Planning Algorithm For Gtfs Data With Nosql Structure To Improve The Performance. *Journal of Theoretical and Applied Information Technology, 99*(10), 2290–2300.