Short thesis for the degree of doctor of philosophy (PhD)

Benchmarking and Utilization of NoSQL Databases - A New Vision

by Mustafa Alzaidi

Supervisor: Dr. Aniko Vagner

UNIVERSITY OF DEBRECEN
Doctoral School of Informatics
Debrecen, 2024

Contents

1 Introduction

NoSQL databases continue to evolve and grow in popularity, there is a pressing need to understand their performance characteristics under various conditions.

This thesis will focus on two aspects of NoSQL databases. First benchmarking of NoSQL databases, which can lead to better system selection. Second, utilizing NoSQL databases to get a better performance. In this thesis, we will use trip planning with GTFS data as an example application for utilization and benchmarking after we introduce a new trip planning algorithm.

The second chapter of this thesis will review the benchmarking techniques, their importance, and the common available benchmarking tools. Then, use the Yahoo Cloud Service Benchmarking Tool (YCSB) to benchmark Redis and HBase as competitive key-value NoSQL databases nowadays. And review their performance evaluation.

The third chapter introduces the approach of utilizing the Redis NoSQL database to improve the performance of finding a trip plan using GTFS data.

The trip planning consists of two parts: finding the possible routes and validating the routes based on the trips timetable. While the third chapter did not consider time validation, the fourth chapter introduced an algorithm for validating trips based on the trip timetable. Then as another example of NoSQL database utilization, introduce the Range Mapping Hash which is a Redis structure that aims to speed up the trip time validating process.

Both the third and the fourth chapters measure and compare the performance of trip planning and trip time validation with and without the proposed approach of utilizing NoSQL database.

Back to the benchmarking topic, one of the disadvantages of current common benchmarking tools and approaches is that it not take into consideration the application to be used for benchmarking. The fifth chapter will introduce the approach of benchmarking the database based on application interaction using GTFS trip planning application as an example. The chapter will benchmark Redis and MongoDB for GTFS data.

2 Database Benchmarking

Thesis point: We used the Yahoo Cloud Service Benchmarking (YCSB) tool to benchmark two competing NoSQL databases, Redis and HBase. We used the default workload provided by YCSB and re-conducted the test using a different number of threads every time (1 to 10 threads). The results show that both databases have almost similar performance when fewer threads are used (less than 7). However, when we increase the number of used threads, the HBase shows slightly higher throughput in comparison to Redis.

2.1 Database benchmarking

Putting a database system through its paces with a series of tests and workloads is known as "benchmarking." The fundamental objective of benchmarking is to compare and contrast the effectiveness, dependability, and scalability of the system under real-world conditions of use. Organizations might benefit from benchmarking when it comes to selecting, setting, and optimizing database systems for their unique requirements. For benchmarking and performance testing of distributed database systems, YCSB (Yahoo Cloud Serving Benchmark) is a popular open-source application. Since YCSB was built to mimic the demands of real-world applications, it is a useful tool for testing the efficacy of different DBMSs in distributed and cloud environments. It provides a consistent method for evaluating database performance by performing common operations like reading, writing, updating, and erasing data.

2.2 Banchmarking and results with YCSB

2.2.1 Load A

In this workload, the tool divides the total operation into 50% read and 50% write operations. Thus, this workload can be considered heavy in terms of updates. The result is shown in Figure 2.1. We notified that the HBase started to give better performance when we increased the thread from six to seven threads with this load. However, we got a similar performance gap with more than seven threads. Thus, this load shows better performance for HBase in comparison to Redis.

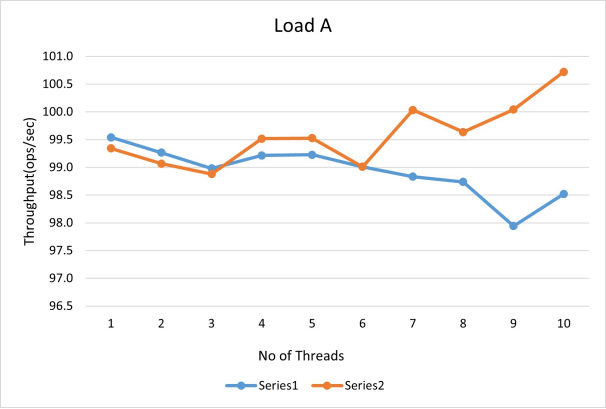

FIGURE 2.1: Load A.

2.2.2 Load B

The read operation takes 95% of the total operations in this workload. Thus, we can denote this workload as a heavy test. The max recorded throughput for Redis and HBase is 99.54 and 100.33 milliseconds, respectively. The result is shown in Figure 2.2. Again, the HBase performs better than Redis with eight

or more threads. Redis has no noticeable change during all the threads experiment.

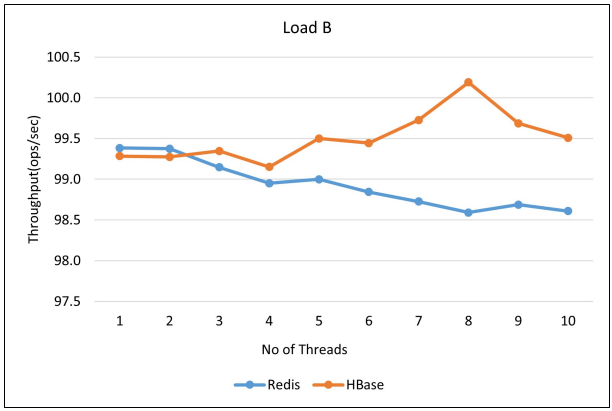

FIGURE 2.2: Load B.

2.2.3 Load C

This workload consists only of read operations and can be used to test the database when the application is critical to data retrieval, and there is no rapid insertion or update operation that can affect the software. The max recorded throughput for Redis and HBase is 99.36 and 100.39 milliseconds, respectively. The result is shown in Figure 2.3.

2.2.4 Load D

This load contains only 5% insert operations with 95% read operations. The read operations are done on the data that was inserted recently. The max recorded throughput for Redis and HBase is 99.48 and 100.61 milliseconds, respectively. Figure 2.4 shows the Load D result. HBase shows better performance with increasing the number of threads.

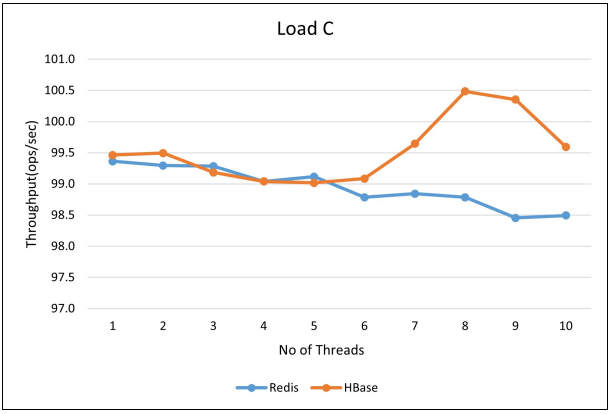

FIGURE 2.3: Load C.

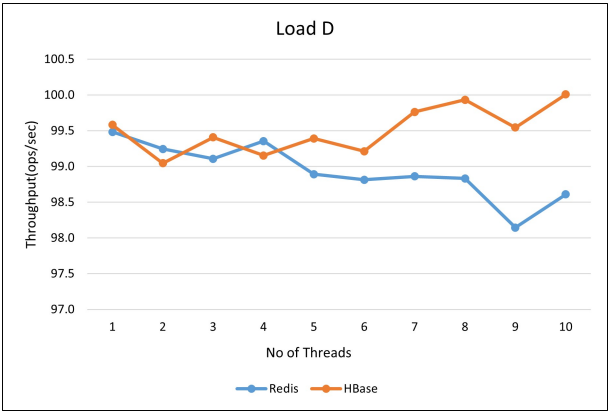

FIGURE 2.4: Load D.

2.2.5 Load E

95% of the time is spent scanning, and just 5% is spent inserting. It is scanned for a short number of records rather than a single one. Figure 2.5 shows the result comparison for both databases. The max recorded throughput for Redis and HBase

is 99.48 and 100.87 milliseconds, respectively. Both databases show similar performance till we use seven threads. However, the performance gap after seven or more threads was smaller compared to the gap we got with the other tests. Again, the HBase was slightly better than Redis for this test.

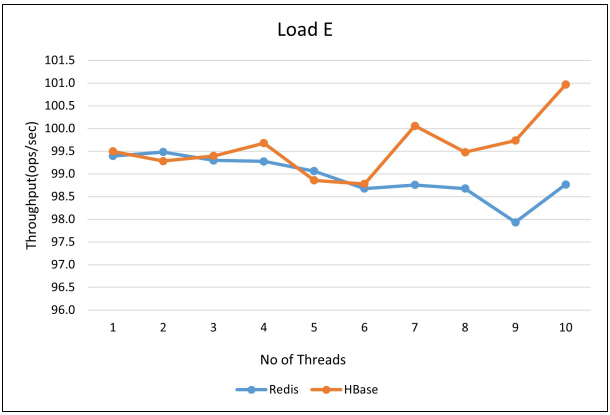

FIGURE 2.5: Load E.

2.2.6 Load F

This load simulates the situation when the application retrieves the data from the database, updates it, and then stores it back in the database. Figure 2.6 shows the result for load F.

2.3 Conclusion

We use the Yahoo Cloud Service Benchmarking tool to compare two popular NoSQL databases. We used the default workload provided by the tool, and we re-conducted the test using a different number of threads every time (1 to 10 threads). The results show that both databases have almost similar performance when fewer threads are used (less than 7). However,

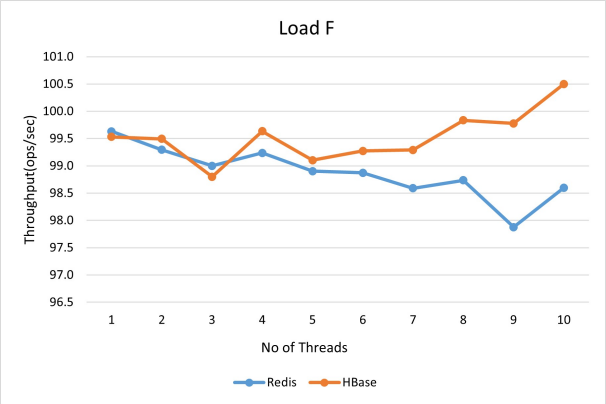

FIGURE 2.6: Load F.

when we increase the number of used threads, the HBase shows higher throughput in comparison to Redis.

3 Utilizing Redis for GTFS trip planning

Thesis point: We implemented a trip planning algorithm for GTFS data and proposed a NoSQL utilization method to improve trip planning performance. We introduced a Redis NoSQL structure that can pre-store trip plans and take advantage of the Redis Hash structure to provide better performance. With this structure, we eliminate the need to rerun the trip planning algorithm with every user request. We compared the performance of finding Trip plans using the algorithm and Redis structure. The result shows that using Redis can provide better performance and less server overhead.

3.1 Introduction

Google developed General Transit Feed Specification (GTFS) as a format to define public transport systems data. Its main objective was to allow public transit agencies to upload their data to Google Transit schedules so that Google Maps users could easily decide which bus, train, or other vehicles to take for transport between two specific locations. GTFS can describe the public transportation schedules and associated geographic information and make this information easy to access and use standardly by users and transport application developers.
Some research proposes algorithms to find trips in local transport [1, 2]. In trip-planning applications, every time a user runs a trip planner, the server runs the route searching or trip-planning algorithm, which is a time-consuming process. In this chapter, we implemented a trip planning algorithm [3] for GTFS data

and propose a new technique to enhance the server response time and reduce the server overhead by introducing a Redis NoSQL data structure to store all possible plans between any two stops, eliminating the need to run the algorithm with every user request. As any search request can be served using the data in the Redis structure. We experiment with this strategy, the GTFS data of Debrecen and Budapest cities, and measure and compare the performance with and without using Redis (using a route searching algorithm). The generated data can increase the ability to analyze transportation services and help the city provide better public transportation using a Smart City paradigm. We implement the algorithm and the strategy as an open-source project using C#.

3.2 Improve the performance using Redis

Any path-planning algorithm is time-consuming, especially with extensive data. For trip-planning applications, the server handles the overhead of running the planning algorithm. Every time a user requests a plan, the server must run the algorithm and search the GTFS data. Here, we proposed a technique to increase the server performance and reduce the overhead by searching the trip plans between all the stops of the GTFS data and storing them in a Redis database. The result will be retrieved from the Redis server for all user search requests without rerunning the algorithm.

3.2.1 Redis structure for trip plans

The trip plan structure resulting from the algorithm is a PATH data structure defined earlier; the most informative part of the PATH data structure is the WAY attribute, a list of MOVE objects. The task here is to define a Redis data structure model that stores all possible plans between two stops (list of PATHs). Both MOVES and PATH attributes are converted by concatenating to a single string using a special-character string "||" as a value separator. We experiment with two models. The first model

separates the PATH and the MOVE list it contains, and the second model keeps PATH and its MOVE list in a single string.

3.2.2 Model-1 hash and list

This model uses the indexing concept. We use a Hash with a Key equal to both stops IDs separated by "____". The Hash stores the number of values fields equal to the number of solutions for these two stops; the value for each field stores a List name (a reference to a Redis List) where each element in the list represents a MOVE made by this Path.

3.2.3 Model-2 LIST

We use a Redis List structure to store all possible solutions (PATH list) for any two stops. The list Key will be the two stop IDs separated by "____". Each value in the list represents a PATH object. All the data stored in the Path object is converted to a string and concreted together as a single string. Both models are illustrated in Figure 3.1 After the experiment, we found that the second model provided a faster performance. As retrieving the hash value takes O(1) time, then for n PATH stored in the hash will take O(n) besides O(1) for the indexing list. The second model needs only O(1) to retrieve the solutions, and it is easier for implementation, although it needs more back-end processing to extract the data from the string, so we depend on it in the C# project.

3.3 Measure and compare the performance

As performance enhancement is the key point behind using Redis, we execute the algorithm by choosing two random stops and recording the execution time. Then, we compared that time with the time taken to request and retrieve the solution for the same stops from the Redis server, as we already stored all possible solutions between any two stops in Redis. Because GTFS data size affects the algorithm performance, we conducted

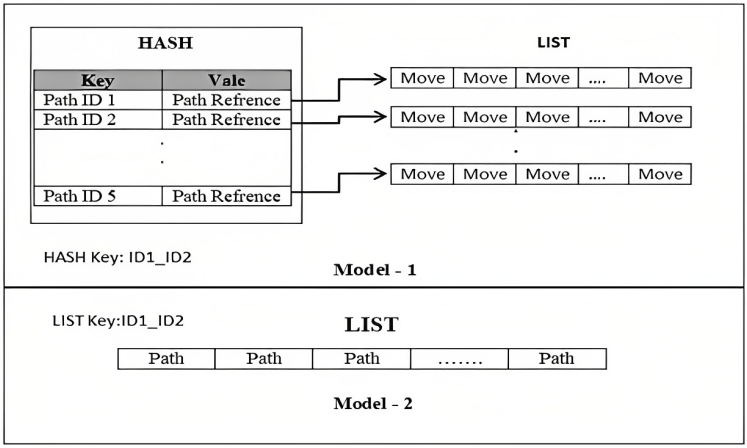

FIGURE 3.1: Redis models for PATH data structure.

28 experiments with the GTFS data for two cities varying in size, Debrecen(1483 KB) and Budapest(42128KB). The experiment results are shown in Table 3.1.

TABLE 3.1: Experiments results

NO	Budapest		Debrecen	
	No Redis	Using Redis	No Redis	Using Redis
1	1.0381	0.08930	0.8575	0.08908
2	1.0452	0.08915	0.8587	0.08919
3	0.9909	0.08907	0.8510	0.08901
4	1.0320	0.08909	0.8946	0.08925
5	1.0551	0.08902	0.8953	0.08920
6	1.0398	0.08929	0.8720	0.08919
Average	1.0368	0.08915	0.8750	0.08913

The above results are computed using the same hardware and computation power (CPU: Intel Core i7-3720QM 2.6 GHz, 6MB L3 cache, RAM: 8GB, OS: WIN10 64bit); it is clear that using Redis can produce high performance and reduce the server overhead as the computation time will be reduced. We can note that,

with both cities, retrieving Redis data is faster than performing a real-time search. Also, we note that the average time required to search the Budapest data (1.0280 milliseconds) is greater than the average time required for Debrecen (0.08915). But using Redis, we do not have this variation as the recorded time is 0.08915 and 0.08914 for the two cities.

4 GTFS Trip Timing Performance Enhancement

Thesis point: We implement an algorithm for validating the trip plans based on the trip timetable and propose a Redis model called Range Mapping Hash (RMH) as a method of utilizing Redis NoSQL database to retrieve the timing information. We experimented and tested the performance with and without using RMH. The test shows that RMH provides notable improvement.

4.1 Find trip time information

Finding the trip plans is the first phase in the trip planning process. The second step is trip time validation, which verifies that the plans match the schedule for the trip. Certain studies provide improvement in time validation performance [4]. In terms of NoSQL utilization, we introduce Redis Range Mapping Hash (RMH), a structure that can obtain trip time information for any trip with two operations, each with a time complexity of Q(1). We tested and compared the performance both with and without RMH (using the traditional algorithm to search the GTFS data for timing data).

4.2 Range mapping hash (RMH)

We propose the RMH as a Redis model to avoid the time-consuming liner data scanning by mapping the input parameters to the output directly without any liner search or scan using the power of hash structure in Redis. For each route between any two stops, we need to map a route and two stops ID and time T to the ID of the trip with the nearest time to T on that route, the trip departure time at the start-stop, and trip arrival time at the end stop. The RMH consists of two structures. Both structures' querying results are combined to form the timing answer. We use a Redis Hash structure for the implementation. The Hash structure syntax has three-part, the KEY, which refers to the Hash name; the FIELD which uniquely identifies a row in the hash; and the VALUE. The HGET and HSET commands are used to retrieve and insert data into Redis Hash.

4.2.1 RMH trip departure time structure

The first structure is used to map the start-stop ID, route ID, and Time (T) into the next trip's ID and time at this stop using that route. For this structure, we create a Redis hash for each stop route pair to store all trip visiting time at the stop. The Hash KEY part will mention the stop ID and the route ID separated by the "__" string. The FIELD will contain the time to be examined and denote it as T. The hash VALUE part holds the time of the next coming trip according to T, and the trip ID, separated by the "__" string. Figure 4.1 shows the trip time structure.
All this information is available in the stoptimes file except the time (T). Any possible T value belongs to the set of sharp minutes in the day. Thus a maximum of 1440 entries is needed to cover all the possibilities. For each stop ID, route ID pair from the stoptimes data and a time T entry, the value field contains the time of the next coming trip and the trip ID separated by "___". For example, in Figure 4.1, we take route ten and stop 1100905. Three trips, 208,209, and 210 are listed in the stoptimes file with departure times 11:44:00, 11:54:00, and 12:04:00. At the first hash

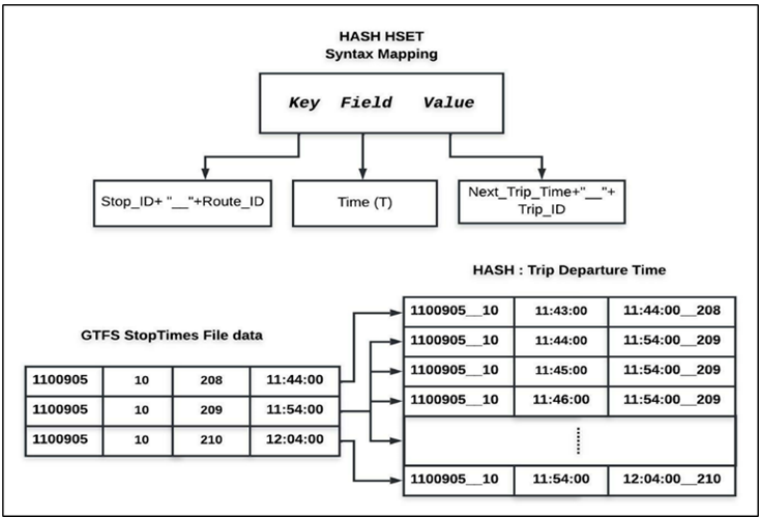

FIGURE 4.1: RMH trip departure time structure.

entry, where the T value is 11:43:00, the answer (the Hash value field) shows the time 11:44:00 and trip ID 208. For any value of T equal to or greater than 11:44:00, the answer will be trip 209 as its time is 11:54:00. When T is greater or equal to 11:54:00, the answer will be trip 210 and its time 12:04:00.

4.2.2 RMH arrival time structure

After using the trip departure time structure, we have the departure time from the start-stop and the trip ID. Finally, we can find the end stop's arrival time using the end-stop ID and the trip ID using the arrival time structure. Figure 4.2 shows the arrival time structure, a single Redis Hash for each stop, to list all the trip arrival time combinations. The KEY part of the Redis Hash is used to refer to stop using the stop ID; the FIELD is used to refer to the trip ID, where the VALUE stores the arrival time.

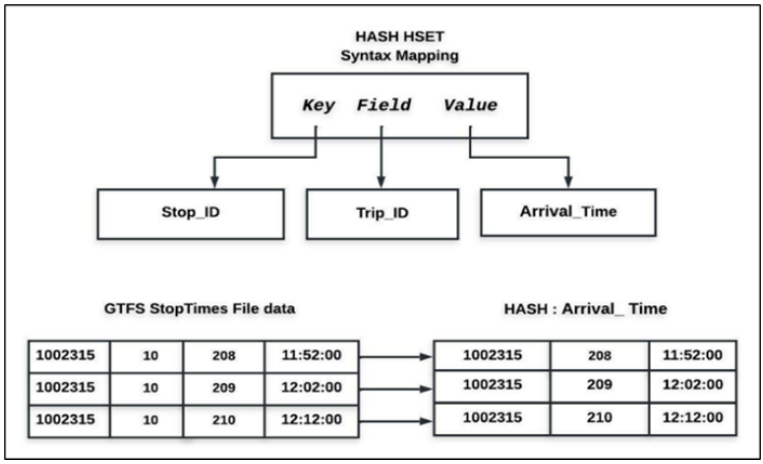

FIGURE 4.2: RMH arrival time structure.

For example, in Figure 4.2, the trip planning algorithm results in a MOVE with start-stop 1100905, end-stop 1002315, and route 10. If the T value is 11:45:00, the timing validation will work as follows: HGET statement using the trip departure time structure is used to retrieve the trip ID and the departure time from the start-stop. The HGET KEY will be "1100905__10" (start-stop ID and the route ID); the FIELD part is "11:45:00" (T). The returned value from this HGET statement will be "11:54:00__209" (the departure time and the trip ID), as shown in Figure 4.1. Now, the trip is known, and the end stop's arrival time is the only missing part. Another HGET statement with KEY is "1002315", and FIELD "209" is used with the arrival time structure; the return value from this statement will be "12:02:00 (the arrival time at the end stop). If any of the two structures do not return a match for the HGET command, the MOVE is rejected, and the whole PATH (path). Thus, the total time complexity of RMH is formed by two Hash table read operations only. As the time complexity for reading from a Hash structure is O(1), then RMH total complexity is O(2).

4.3 Experiment and results

We experimented using RMH (Redis model) with Budapest and Debrecen cities GTFS data for 30 random start and end stop combinations. Each experiment finds the timing data with and without using Redis (the RMH) and records the time (in milliseconds) the computer takes to retrieve the result for each combination. Table 4.1 shows the recorded results for six experiments.

TABLE 4.1: Experiments results

NO	Budapest Without Redis	Budapest Using Redis	Debrecen Without Redis	Debrecen Using Redis
1	1202.4	8.368	1204.6	8.559
2	1305.4	8.238	1106.7	8.278
3	1105.2	8.38	908.2	8.48
4	1409.3	8.749	907.3	7.237
5	1408.4	8.198	904.1	8.107
6	1206.3	8.558	1008.1	7.848
Average	1259.17	8.389	1038.48	8.135

Figure 4.3 visualizes the execution time difference between the time taken to find the timing information for a trip using the run-time algorithm without Redis and the execution time for retrieving the exact data for the same pair of start and end stops using Redis. We can notice that the run-time performance varies during the experiments, around an average of 1219.89 milliseconds. Conversely, Redis's execution time is more stable. It has ignorable variation during the experiments, with an average of 7.985 milliseconds forming a straight line in the chart close to zero compared to the run-time performance.
With Debrecen data, the experiment shows a similar performance compared to Budapest experiments. The average execution time is 7.808 milliseconds, which is very close to the execution time for Redis with Budapest data. However, again, the experiments show notifiable variation in the performance using the run-time

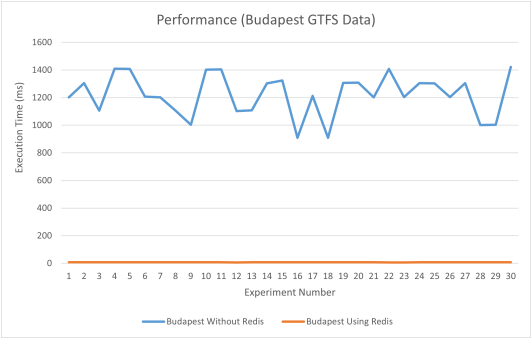

FIGURE 4.3: Budapest data experiments result.

algorithm. Figure 4.4 shows Debrecen data experiments' performance using Redis and the run-time algorithm (without Redis).

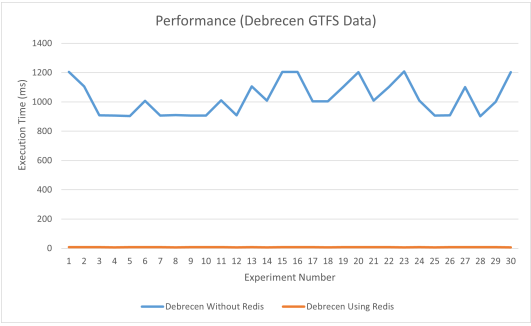

FIGURE 4.4: Debrecen data experiments result.

The experiments also show that the difference between GTFS data size for the cities (Debrecen 1483 KB and Budapest 42128 KB) affects the performance when run-time algorithm used. This effect can be clear if we compare the average execution time with both cities' data. We can notice that the average run-time execution time increases with larger cities (in this case, Budapest), while the data size has no effect in the case of using Redis.

5 Application-Based Database Benchmarking

Thesis point: We proposed a database benchmarking method based on the application's nature and how the application will interact with the database. We build a benchmarking tool that can be customized by developers to benchmark the database according to specific applications. We used this tool to benchmark Redis and MongoDB as databases for trip-planning applications. The tool can simulate different levels of workload on the database server by simulating multiple simultaneous requests. The test shows that performance with Redis was way higher than MongoDB.

5.1 Introduction

The conventional method, such as YCSB, has drawbacks. The benchmark was conducted using arbitrary database operations according to a predetermined workload, which is one of the disadvantages. Because such workload ignores the use case and the interaction between the application and database, such a tool will not accurately reflect the performance of the database for a certain application. Some research benchmark database systems using interactive workload [5, 6]. We suggested a benchmarking strategy and created a benchmarking tool (using Java) that developers and researchers can modify to benchmark the database in accordance with particular applications. Using Redis and MongoDB, we used this tool to assess the database performance for trip planning applications. The tool mimics numerous simultaneous user requests, which

allows the simulation of various levels of load stress on the database server. According to the test, Redis performed significantly better than MongoDB.

5.2 Methodology

Our proposed benchmarking methods involve three steps: identify the application-database use-case scenarios and main application operations, define the data storage model for 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 a NoSQL database.

5.2.1 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:

- 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 query types involved in the full trip planning. The set of benchmark steps will be as follows:

- 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 trip 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.

5.2.2 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 follows:

*TableName_PrimeryKey_ForeignKey*1*_ForeignKey*2*_...._ForeignKeyN*

Note that there is no primary or foreign key for some tables; in that case, it is replaced by a blank. Figure 5.1 shows how the stoptimes file is stored in Redis.

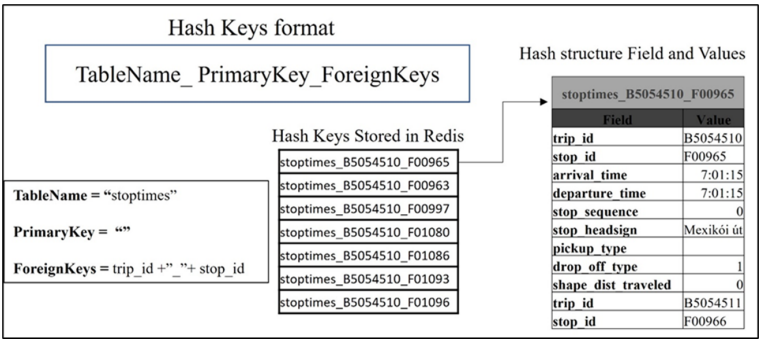

FIGURE 5.1: stoptimes stored in Redis

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 it is an in-memory database.

5.2.3 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, here 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
}

5.3 Benchmarking result

5.3.1 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 ten 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.

5.3.2 Results

To compare the database accessibility, we select three test runs 20 threads, 60 threads, and 110 threads. Table 5.1 shows the experiment results for the first 10 threads in each test run for MongoDB and Redis.

TABLE 5.1: Results for 20, 60, and 110 threads.

	MongoDB			**Redis**		
ID	**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 5.2 and 5.3 shows 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 5.2: Experiments (Redis).

No Of Threads	No of DB Hits	No of Complete Operation	DB Hits per/Sec	Complete Operation per/Sec
10	577,554,675	261,525	14,438,866	6,538
20	542,528,715	243,550	13,563,217	6,088
30	461,164,242	208,894	11,529,106	5,222
40	434,841,597	196,189	10,871,039	4,904
50	361,386,447	163,077	9,034,661	4,076
60	431,072,073	195,215	10,776,801	4,880
70	399,627,851	180,677	9,990,696	4,516
80	375,112,260	169,421	9,377,806	4,235
90	365,071,720	165,548	9,126,793	4,138
100	347,442,017	156,039	8,686,050	3,900
110	341,046,978	154,682	8,526,174	3,867

Figure 5.1 and Figure 5.1 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.
It is important to highlight here that the number of the required queries (database hits) to perform one complete trip plan for the

TABLE 5.3: Experiments (MongoDB).

No Of Threads	No of DB Hits	No of Complete Operation	DB Hits per/Sec	Complete Operation per/Sec
10	296,746	132	7,418	3
20	263,563	129	6,589	3
30	243,914	112	6,097	2
40	256,990	96	6,424	2
50	164,201	87	4,105	2
60	100,021	69	2,500	1
70	210,965	89	5,274	2
80	133,628	80	3,340	2
90	115,265	59	2,881	1
100	213,504	86	5,337	2
110	193,595	89	4,839	2

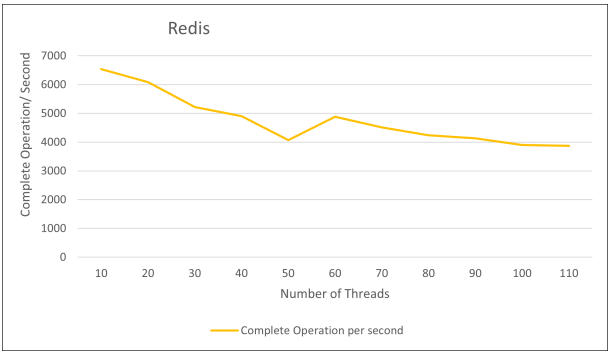

FIGURE 5.2: Relation between throughput and number of used threads (Redis)

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.

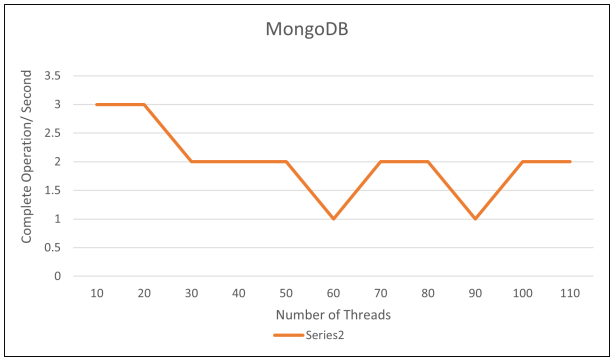

FIGURE 5.3: Relation between throughput and number of used threads (MongoDB)

6 Summary

This thesis focuses on two aspects of NoSQL databases, the benchmarking and the utilization of NoSQL databases to obtain better application performance. We started by using Yahoo Cloud Service Benchmarking Tool (YCSB), a well-known benchmarking tool to benchmark two of the competing NoSQL databases nowadays Redis and HBase. We used the default workload provided by the tool, and we re-conducted the test using a different number of threads every time (1 to 10 threads). The results show that both databases have almost similar performance when fewer threads are used (less than 7). However, when we increase the number of used threads, the HBase shows slightly higher throughput in comparison to Redis. Next, we focused on application performance improvement using NoSQL databases. As a case study, we used the Trip Planning application for General Transit Feed Specification data (GTFS). Such an application provides the user with the ability to find local transportation routes between two locations (two local transport stops). We started by implementing a trip-planning algorithm, and then we introduced a Redis NoSQL structure that can pre-store trip plans and take advantage of the Redis Hash structure, which can retrieve data in Q(1) time complexity. With this structure, we eliminate the need to rerun the trip planning algorithm with every user request. We implemented the algorithm using C# and the Redis structure, and then we compared the performance of finding Trip plans using both approaches. The result shows that using Redis can provide better performance and less server load. The trip planning involves two steps: first, finding the trip plans (possible routes), and second, trip time validation, which validates the plans according to the trip timetable. Some research spots

the light on time validation performance enhancement. We implemented a trip timing validation algorithm, and in terms of NoSQL utilization approach, we introduced the Redis Range Mapping Hash (RMH). This structure can retrieve the trip time information for any trip with a time complexity of Q(2). We implemented the approach using C# on the top of trip planning implementation, and we experimented with the performance with and without using RMH. The test shows that RMH provides notable improvement. Back to benchmarking, the traditional approach (like YCSB) has its disadvantages. One of the biggest concerns is that the benchmark was done with random database operations based on a predefined workload. Thus such a tool will not reflect the actual performance of the database for a specific application, as suck workload will not take into consideration the application-database interaction or use case scenarios. We propose a benchmarking approach where we build a benchmarking tool (using Java) that can be customized by developers to benchmark the database according to specific applications. We used this tool to benchmark the performance of the database for trip-planning applications for Redis and MongoDB. The tool can simulate the different levels of load stress on the database server by simulating multiple simultaneous requests for users. The test shows that performance with Redis was way higher than that of MongoDB.

As final conclusion, this thesis encourages the utilization and use of NoSQL databases to go outside the common or traditional way of use. Moreover, as benchmarking is an important part of database system selection, the benchmarking tool should provide an actual and real view of the system performance according to the nature of the application and the candidate database systems to be used.

Bibliography

[1] Jacek Widuch. “A Label Correcting Algorithm for the Bus Routing Problem”. In: *Fundamenta Informaticae* 118 (Aug. 2012), pp. 305–326. DOI: 10.3233/FI-2012-716.

[2] Chao-Lin Liu et al. “Path-planning algorithms for public transportation systems”. In: 100. 2001, pp. 1061–1066. ISBN: VO -. DOI: 10.1109/ITSC.2001.948809.

[3] Mustafa Alzaidi Aniko Vagner. “Trip Planning Algorithm For Gtfs Data With Nosql Structure To Improve The Performance”. In: *Journal of Theoretical and Applied Information Technology* Vol.99. No (10 31st May 2021 May 2021), pp. 2290–2300.

[4] S Kiavash Fayyaz S., Xiaoyue Cathy Liu, and Guohui Zhang. “An efficient General Transit Feed Specification (GTFS) enabled algorithm for dynamic transit accessibility analysis”. In: *PLOS ONE* 12 (10 Oct. 2017), e0185333–. URL: https://doi.org/10.1371/journal.pone.0185333.

[5] Leilani; Chang Battle Remco; Heer Jeffrey; Stonebraker Michael. *DSIA - Position statement: The case for a visualization performance benchmark*. Vol. NA. 2017, pp. 1–5. DOI: 10.1109/dsia.2017.8339089.

[6] Philipp Eichmann et al. “Towards a Benchmark for Interactive Data Exploration”. In: *IEEE Data Eng. Bull.* 39 (2016), pp. 50–61.

List of Publications

List of Papers [J]:

J1 Mustafa Alzaidi, Vagner Aniko. (2023). *Trip Timing Algorithm for GTFS data With Redis Model to Improve the Performance.* Journal of Information Systems and Telecommunication (JIST), **11(2322–1437), 260–268**.

J2 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, **Vol.99. No(10 31st May 2021), 2290–2300**.

J3 Mustafa Alzaidi, Aniko Vagner. *Application-Based Benchmarking on Redis and MongoDB for Trip Planning using GTFS Data.* TEM JOURNAL - Technology, Education, Management, Informatics, **12(4):2583, 2023.**

List of Conference Proceedings [C]:

C1 Alzaidi, Mustafa and Vagner, Aniko (2022) *Benchmarking Redis and HBase NoSQL Databases using Yahoo Cloud Service Benchmarking tool.* Annales Mathematicae et Informaticae, **56. pp. 1-9. ISSN 17876117**.

C2 Vágner, Anikó Al-Zaidi, Mustafa. (2023). *Sharding and Master-Slave Replication of NoSQL Databases: Comparison of MongoDB and Redis.* **576-582. 10.5220/0012142700003541.**

C3 Al-Zaidi, M., Vagner, A. (2020). *Effective Smart Sensors Based Cognitive Infocommunications Weight Loss System.* **https://doi.org/10.1109/CogInfoCom50765.2020.9237872.**

C4 H2020 INTEGRITY – Research INTEGRITY Conference European Student Convention (8-9 Sept).

C5 Al-Zaidi, Mustafa, Aniko Vagner , *Station Based Real Time Trip Planning Algorithm.* in conference On Information Technology And Data Science November 6–8, 2020 Debrecen, Hungary, 2020.

UNIVERSITY of DEBRECEN

UNIVERSITY AND NATIONAL LIBRARY
UNIVERSITY OF DEBRECEN
H-4002 Egyetem tér 1, Debrecen
Phone: +3652/410-443, email: publikaciok@lib.unideb.hu

Registry number: DEENK/533/2023.PL
Subject: PhD Publication List

Candidate: Mustafa Majid Hayder Al-Zaidi
Doctoral School: Doctoral School of Informatics
MTMT ID: 10078747

List of publications related to the dissertation

Foreign language scientific articles in international journals (3)

1. **Al-Zaidi, M. M. H.**, Vágner, A.: Application-Based Benchmarking on Redis and MongoDB for Trip Planning using GTFS Data.
   TEM Journal. 12 (4), 2583-2592, 2023. ISSN: 2217-8309.
   DOI: http://dx.doi.org/10.18421/TEM124-70
   IF: 0.7 (2022)

2. **Al-Zaidi, M. M. H.**, Vágner, A.: Trip Timing Algorithm for GTFS data With Redis Model to Improve the Performance.
   Journal of Information Systems and Telecommunication. 11 (3), 260-268, 2023. ISSN: 2322-1437.

3. **Al-Zaidi, M. M. H.**, Vágner, A.: Trip planning algorithm for gtfs data with nosql structure to improve the performance.
   J. Theor. Appl. Inf. Technol. 99 (10), 2290-2300, 2021. ISSN: 1992-8645.

Foreign language conference proceedings (1)

4. **Al-Zaidi, M. M. H.**, Vágner, A.: Benchmarking Redis and HBase NoSQL Databases Using Yahoo Cloud Service Benchmarking Tool.
   Ann. Math. Inform. 56, 1-9, 2022. ISSN: 1787-5021.
   DOI: https://doi.org/10.33039/ami.2022.12.006
   IF: 0.3

UNIVERSITY AND NATIONAL LIBRARY
UNIVERSITY OF DEBRECEN
H-4002 Egyetem tér 1, Debrecen
Phone: +3652/410-443, email: publikaciok@lib.unideb.hu

List of other publications

Foreign language conference proceedings (2)

5. Vágner, A., **Al-Zaidi, M. M. H.**: Sharding and master-slave replication of NoSQL databases - comparison of MongoDB and Redis.
   In: Proceedings of the International Conference on Data Science, Technology and Applications DATA (DATA 2023). Eds.: Oleg Gusikhin, Slimane Hammoudi, Alfredo Cuzzocrea, [Institute of Electrical and Electronics Engineers Inc.], [Piscataway], 576-582, 2023, (ISSN 2184-285X) ISBN: 9789897586644

6. **Al-Zaidi, M. M. H.**, Vágner, A.: Effective Smart Sensors Based Cognitive Infocommunications Weight Loss System.
   In: 11th IEEE International Conference on Cognitive Infocommunications : CogInfoCom 2020 : Proceedings. Ed.: Baranyi Péter, IEEE, Piscataway, 73-78, 2020. ISBN: 9781728182131

Total IF of journals (all publications): 1
Total IF of journals (publications related to the dissertation): 1

The Candidate's publication data submitted to the iDEa Tudóstér have been validated by DEENK on the basis of the Journal Citation Report (Impact Factor) database.

05 December, 2023

DEBRECENI EGYETEM * Egyetemi és Nemzeti Könyvtár *