Benchmarking and Utilization of NoSQL Databases - A New Vision

Thesis for the Degree of Doctor of Philosophy (PhD)

by Mustafa Alzaidi

Supervisor:
Dr. Aniko Vagner

UNIVERSITY OF DEBRECEN
Doctoral Council of Natural Sciences and Information Technology
Doctoral School of Informatics
Debrecen, 2024

Hereby I declare that I prepared this thesis within the Doctoral Council of Natural Sciences and Information Technology, Doctoral School of Informatics, University of Debrecen in order to obtain a PhD Degree in Informatics at Debrecen University.

The results published in the dissertation are not reported in any other PhD theses.

Debrecen, 202. . . . . .

signature of the candidate

Hereby I confirm that Mustafa Alzaidi candidate conducted his studies with my supervision within the Applied Information Technology and its Theoretical Background Program of the Doctoral School of Informatics between 2019 and 2024. The independent studies and research work of the candidate significantly contributed to the results published in the thesis.
I also declare that the results published in the thesis are not reported in any other theses.
I support the acceptance of the thesis.

Debrecen, 202. . . . . .

signature of the supervisor

Benchmarking and Utilization of NoSQL Databases - A New Vision

Dissertation submitted in partial fulfillment of the requirements
for the doctoral (PhD) degree in informatics

Written by Mustafa Alzidi, certified computer scientist.

Prepared in the framework of the Doctoral School of Inforamtics of University of Debrecen
(Applied Information Technology and its Theoretical Background programme)

Dissertation advisor: Dr. Aniko Vagner

The official opponents of the dissertation:

Dr. ....

Dr. ....

The evaluation committee:

chairperson: Dr. ....

members: Dr. ....

Dr. ....

Dr. ....

Dr. ....

The date of the dissertation defense: ..... 20...

Contents

1

Introduction

This dissertation's first chapter introduces NoSQL databases and underlines the motivation of the research conducted in the dissertation and chapter organization.

In an era dominated by digital information, the ability to manage and process large volumes of data is crucial to many organizations' success. One of the primary tools used to handle this data is the database management system (DBMS). Traditionally, relational database management systems (RDBMS) have been the preferred choice. The expansion of data size in many fields has led to the emergence and adoption of NoSQL (Not Only SQL) databases. These systems are designed to overcome the scalability limitations inherent to relational databases and offer unique advantages in handling certain types of data, particularly in the case of unstructured and semi-structured data.

NoSQL databases arose in the late 20th century and at the beginning of the 21th century, responding to the increased demands for processing speed, volume, and variety of data. As the name suggests, NoSQL databases don't just rely on SQL (Structured Query Language), the standard language for dealing with relational databases. They are built to be flexible, scalable, and capable of handling massive data and real-time applications. NoSQL databases store data in a format that differs from the traditional row-and-column schema used in relational databases. There are four main types of NoSQL databases: key-value stores, column stores, document databases, and graph databases, each with unique use cases and benefits. The emergence of NoSQL databases has drastically changed the landscape of data management. These databases are designed to solve problems where RDBMS could not, such as managing large amounts of data, horizontal scaling, and allowing for flexible schema changes. Furthermore, NoSQL databases provide features such as easy replication support, simple API, eventual consistency, and being able to handle large amounts of data. However, a NoSQL database is not always the best choice for every application. It should depend on the system's specific requirements, including the structure of data, the system's scalability, and the application's consistency requirements.

As NoSQL databases continue to evolve and grow in popularity, there is a pressing need to understand their performance characteristics under various conditions. Benchmarking NoSQL databases refers to evaluating their performance using a standardized set of operations. These performance evaluations are critical for businesses and organizations as they help decide which NoSQL database to select based on their specific

needs. Benchmarking can provide insights into a system's behavior under load, identify its strengths and weaknesses, and indicate areas for improvement.

A common challenge when benchmarking NoSQL databases is that they don't all adhere to a standard query language or data model, unlike traditional SQL databases. This results in a vast heterogeneity among NoSQL databases, making it challenging to develop a one-size-fits-all benchmarking methodology. Therefore, a tailored benchmarking approach, specific to the type of NoSQL database (e.g., key-value, document, column, or graph) and the use case, is often needed to gain meaningful performance insights.

This dissertation will focus on two aspects of NoSQL databases. First, benchmarking of NoSQL databases can lead to better system selection. Second, utilizing NoSQL databases to get a better performance. In this dissertation, 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 dissertation will review benchmarking techniques, their importance, and the commonly 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 will overview GTFS data and introduce a trip planning algorithm (find possible routes between two local transportation stops or stations). Then, the approach of utilizing the Redis NoSQL database will be introduced 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 trip 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, 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 the NoSQL database.

Back to the benchmarking topic, one of the disadvantages of current common benchmarking tools and approaches is that they do 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 the GTFS trip planning application as an example. The chapter will benchmark Redis and MongoDB for GTFS data.

2

Database Benchmarking Comparing Redis and HBase

Dissertation key point and finding: *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.*

Contents

2.1 Introduction

A growing number of NoSQL databases are being developed and used. The promise of quicker and more efficient throughput compared to older Relational Database Management Systems (RDBMS) is one of it is most compelling features [1]. There are several advantages to using NoSQL databases for cloud computing, including the ability to rapidly scale vertically and horizontally as needed and the easiness of application development [2]. However, application developers should be aware that NoSQL databases are not usually equal when it comes to performance [3]. Because NoSQL systems are not yet mature and evolving at various paces, database managers must pick carefully between NoSQL and relational databases based on their demands regarding consistency, security and scalability, performance, prices, and other factors [4]. Choosing a NoSQL system might be challenging for web application developers because of the large variety of open-source and freely accessible NoSQL systems. In other words, a peer-to-peer comparison of NoSQL systems according to the application activity scenarios to identify the most significant match for different situations would be an appropriate next step. A benchmark in this context refers to a performance assessment of NoSQL solutions that have been suggested or have been deployed. Then, compare the performance of different NoSQL databases; it is necessary to utilize experimental interactions that simulate comparable behavior or activities, as could be the case with applications behavior. Selecting a NoSQL system in this manner can be more appropriate for certain types of user interaction and provide better performance and efficiency than a competitor's systems. Key-value, wide column, graph, and document databases are all examples of NoSQL databases [4, 5]. Key-value stores are collections of registers identifiable by a unique key [6]. Usually, this type of NoSQL system is used as a layer that provides cash for the data with time-consuming access [7]. Some researchers [8] use the key-value store when the application needs to retrieve the stored object based on one field value. JavaScript uses Binary Object-Notations (JSON and BSON) as a kind of document-oriented data [9]. Document-based databases provide more flexibility in terms of schema compared to

RDBMS. They store the data in object format in a similar manner to how programming language logically treats objects. The schemaless model enables the developer to store different types of objects in the same storage entity. This flexibility gave the ability to rapid application development [10]. Document store databases can work well on distributed systems that provide cheaper horizontal scaling as the application needs. Databases like MongoDB, CouchDB, and others fall within this category.

Considering the benefits and drawbacks of a schemaless approach, document-oriented NoSQL databases provide an advantageous method to store data, which makes them suitable for applications dealing with complicated data needs or developing data structures. Scalability and data consistency are two areas where document-oriented approaches may struggle in distributed systems. However, key-value NoSQL databases can provide scalable solutions, which makes it a great choice when it comes to rapid development. Nevertheless, applications requiring elaborate data structures and extensive queries may not be well-suited to them. Document databases prioritize simplicity and scalability, whereas key-value stores prioritize data modeling and query flexibility; which one is better for your application will depend on its needs. Modeling and querying, whereas key-value stores prioritize simplicity and scalability.

The success of Google with BigTable seems to have sparked the development of column stores [11]. The column store databases store the table's records fields separately, such that subsequent values of that property are saved sequentially [12]. Wide-column database systems are built on a hybrid method that makes use of both the descriptive qualities of relational databases and the structure of different key-value stores [4]. Accumulo, Cassandra, and HBase fall into this category. Graph databases may be used to store object data, as well as all connections between them [4]. In this way, Graph databases make use of nodes and edges, the two notions from Graph theory. For example, a foreign or primary key link between two nodes is an edge in the data domain. Neo4J and OrientDB are two good examples [13]. This chapter uses the Yahoo Cloud Service Benchmarking (YCSB) tool to benchmark Redis and HBase databases. We did the test with six different workload scenarios for each workload, and we recorded ten results while adding

ten threads each time with ten threads till the last experiment with 120 threads. The finding of this chapter is published in [14]. The reason behind choosing Redis and HBase is that both are competing databases nowadays and are used as real-time storage.

2.2 Database benchmarking

Data is the new oil, powering many of today's essential online services and mobile apps. The importance of databases, which store this information, has so increased. NoSQL (Non-relational or "not only SQL") databases have become increasingly popular among the many available database technologies, particularly for applications that deal with large amounts of mixed-structure data. Traditional relational databases often lack the scalability and adaptability that these databases provide. Benchmarking is used to measure and compare the capabilities of different NoSQL databases to determine the best fit for a given workload. Using a standard set of performance measurements, NoSQL database benchmarking objectively assesses and compares various NoSQL databases. Read/write times, scalability, consistency, fault tolerance, and resource utilization are all examples of metrics to consider. Companies and developers can use the benchmarks to learn more about the relative merits of various NoSQL databases. Given the large variety of NoSQL databases, such as document stores, key-value stores, wide-column stores, and graph databases, the benchmarking process might be difficult. Fair and relevant comparisons need the careful selection of typical workloads and exact measuring procedures. Following this introduction, we will go into the process, challenges, and concerns of benchmarking NoSQL databases, covering a wide range of benchmarking tools and addressing how to understand and use the results successfully.

2.2.0.1 Benchmarkings tools

Benchmarking NoSQL databases often involves simulating different workloads to see how various databases perform under these conditions.

Let's dive into some of the most popular tools and techniques used for this:

- *YCSB*: Short for Yahoo! Cloud Serving Benchmark, YCSB is a top-tier open-source tool tailored for NoSQL databases. Crafted by Yahoo!, it's designed to assess various database operations, be it read-intensive, write-centric, or balanced scenarios. With YCSB, you can craft custom workloads, and it's compatible with several NoSQL options like MongoDB, Cassandra, and HBase. We will go for more details about this tool later in this chapter.

- *TPC Benchmarking*: This is the brainchild of the Transaction Processing Performance Council, a non-profit that sets standards for database benchmarks. Although often linked with SQL databases, variants like TPC-C and TPC-H are also tailored for NoSQL.

- *Jepsen*: Standing out from the crowd, Jepsen focuses less on speed and more on reliability and consistency. It's brilliant for checking how databases react to issues like network interruptions or unexpected shutdowns. This is crucial for distributed NoSQL databases where consistent data amidst failures is paramount.

- *SPECjbb*: Primarily a benchmarking tool for servers that run standard Java business applications, SPECjbb can also gauge the performance effect of the database component within an application.

- *NoSQLBench*: Designed by DataStax, this open-source tool is meant for NoSQL databases, specifically spotlighting Apache Cassandra and DataStax Enterprise. Its strength is Crafting versatile and user-friendly workload scenarios.

- *Sysbench*: Although not made exclusively for NoSQL, Sysbench is versatile. It's often used for system-wide benchmarking and can be used with databases like MongoDB. It's adaptable, letting users generate diverse system loads and even craft custom scripts.

  When picking from these tools, it's vital to consider the NoSQL database in question and the performance facets you're keen to probe. Remember, benchmark results can sway based on factors

like the database setup, underlying hardware, and how the tool's been crafted. So, always analyze results with a discerning eye, ensuring they guide but don't dictate your database choice.

2.2.0.2 Benchmarking challenges and limitations

While benchmarking NoSQL databases offers insightful data and is an integral part of the selection and optimization process, there are several challenges and limitations that need to be recognized.

- *Non-Specific Workloads*: Benchmarking tools such as YCSB often deploy generic workloads for database comparisons. Although this allows a broader analysis, it might not accurately reflect the performance of a specific database for a specific use case. Essentially, these generic workloads might fail to capture the intricacies of real-world applications.
- *Focused Analysis*: Many benchmarking tools might be oriented towards measuring certain aspects like consistency or read/write speeds while overlooking other vital aspects such as sharding, complex query processing, or data replication. Consequently, a tool might showcase impressive performance under certain conditions but might be inadequate under different scenarios.
- *Intricacy*: The process of configuring and executing benchmarks is often complex. Moreover, interpreting results calls for a deep understanding of the databases and the benchmarking tools. A misinterpretation could potentially result in less-than-optimal decisions.
- *Varied Data Models*: Given the variety of NoSQL databases - document, column, key-value, and graph - not all tools are well-equipped to benchmark all types. This makes it challenging to use one tool to benchmark every NoSQL database.
- *Absence of Standardization*: Unlike SQL databases that have standardized benchmarks like the TPC series, NoSQL benchmarks lack standardization due to the diversity among NoSQL databases.

- *Environment Dependent*: The environment - hardware, OS, network - heavily influences benchmark results. Consequently, a benchmark result from one environment might not be applicable to another.
- *Benchmarketing*: Certain organizations might design benchmarks to accentuate the strengths of their systems, downplaying or ignoring the weaknesses. This practice, commonly called "benchmarketing," emphasizes the need for a comprehensive understanding of benchmarks and an objective interpretation of results. Despite these hurdles, benchmarking tools continue to be invaluable, providing crucial insights about various databases. As they evolve, these tools are expected to become more refined, offering a more detailed and accurate evaluation of NoSQL database performance.

2.3 Redis NoSQL database

Redis is an open-source in-memory key-value store database that is very customizable and claims to be extremely quick in terms of performance. VMware initially maintained it; later, Pivotal Software has taken over as the company that is sponsoring its development. Typically, the databases in Redis is specified by a numerical value. The number of databases is set at 16 by default, although this may be changed as a custom configuration. It is more customizable than a generic key-value structure in terms of data organization. For example, a value in Redis may be saved as a string, a list of strings with insertions at the beginning and end of the list. Furthermore, searching for objects towards the two ends of a huge list is incredibly quick, but querying for an item in the center of a large list is much more time-consuming. The collection of keys stored in Redis does not allow duplication, which implies that adding the same key (string) more than once will result in just one copy of the collection. The operations of adding and removing only need a constant amount of time (O(1)). Redis provides other structures like Hash, Set, and Sorted Set. Hash is referred to by a unique key and can store a set of unique fields, where each field can have one value. Hash provides high-speed data access in comparison to other structures. For

instance, in comparison to List, a Hash can retrieve any key-field value with O(1). Every value (or structure) stored in Redis can be accessed by its key. The key is determined by the user when he inserts the structure. Here are some basic Redis commands to get you started:

SET key value: Set a key to hold a string value.

GET key: Get the value of a key.

HSET key field value: Set the field in a hash stored at the key to a value.

HGET key field: Get the value associated with a field in a hash.

LPUSH key value [value ...]: Prepend one or multiple values to a list.

LRANGE key start stop: Get a range of elements from a list.

SADD key member [member ...]: Add one or more members to a set.

SMEMBERS key: Get all the members of a set.

In this dissertation, we will mainly use the hash structure. Redis hashes provide the same functionality as dictionaries or maps in other programming languages. They provide the ability to store fields and their corresponding values. Simply, hash consists of two parts: the key and a set of field and value pairs. Hash can be a good choice to store objects. Redis also provides special commands that support synchronized data access. For example, BRPOP takes keys of List structures (one or more) as parameters and an integer number to specify the timeout in seconds. The command checks the specified lists in the same order given to the command and removes and returns the last element on that list. If all the lists are empty, the command blocks the current connection and waits for the time specified by the timeout parameter for any other user connection that may be inserted into one of the lists before it releases the connection and returns a value to the client.

2.4 HBase NoSQL database

A distributed, fault-tolerant, and with high scalability column-store NoSQL database implemented on top of the Apache Hadoop Distributed File System (HDFS). HBase is an Apache open-source database that provides real-time store and retrieving ability for massive data. The data in HBase is arranged logically into named indexed tables. HBase tables are stored as multidimensional sparse maps with rows and columns, where rows include a sorting key and an arbitrary number of columns. Versioning is used in table cells. When cells are added to HBase, HBase assigns a timestamp to them that is used to identify the version of that particular cell. For the same row key, many versions of a specific column might exist for that column. Column family and column name are assigned to each cell so that software can always tell what types of data items a particular set of cells contains. The content of a cell is an unbroken array of bytes that is uniquely recognized by the following combinations: Table + Row-Key + Column-Family: Column + Timestamp [15, 16]. A byte array, which also acts as the database's primary key, is used to sort the rows of the table

2.5 Experiments tool setup

2.5.1 Yahoo cloud service benchmarking tool

We will use the Yahoo Cloud Service Benchmarking (YCSB) as the database performance evaluation tool. YCSB was created in 2010 by the research department at Yahoo. The task was to develop a tool that provides the ability to test and compare performance over the service provided by the cloud. Later, this tool became widely used by application developers to test database systems. In addition, this test can help during the decision-making to select the system to be used in the project. Figure 2.1 shows the tool architecture [3]. YCSB

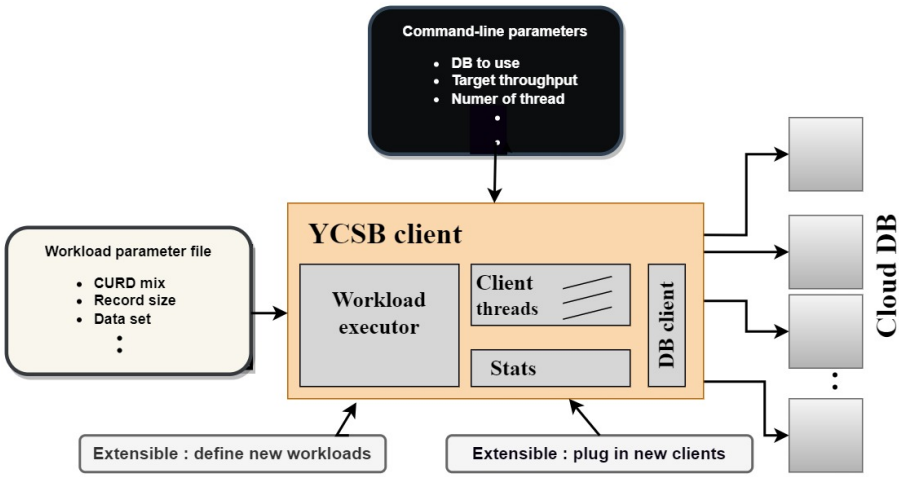

Figure 2.1: Architecture of YCSB.

is developed using the Java programming language as an open-source project [17]. The code can be compiled with Maven and worked as a command line base. The tool supports a variety of NoSQL databases. The test is done by specifying the workload to be used. A workload can determine the number of operations and the types of these operations (Read, Write, and Update). There is a set of predefined workloads provided with tools default source code; we will use these workloads in this work, denoting them as (Load A, Load B, Load C, Load D, Load E, Load F). The test is done in two steps: the Load command and the Run command. The database connection information can be provided as a parameter to the tool with the Run and the Load command. In summary, the YCSB tool performs the following operation:

Load Phase: Data is initially loaded into the target system to create a dataset for testing. This phase simulates the process of populating the database with initial data.

Read Operation: Read operations involve retrieving data from the database. These operations measure the system's ability to fetch existing records efficiently.

Write Operation: Write operations involve inserting or updating data in the database. These operations assess the system's ability to handle

data modifications.

Update Operation: Similar to write operations, update operations specifically involve modifying existing data in the database.

Delete Operation: Delete operations involve removing records from the database. This evaluates the system's ability to handle data deletion efficiently.

Scan Operation: Scan operations retrieve a range of records based on specified criteria, such as a range of keys. These operations test the system's ability to perform range queries.

Complex Queries: Some YCSB workloads include complex queries that combine multiple operations, like read-modify-write sequences. These tests assess the system's ability to handle more intricate data access patterns.

Mixed Workloads: In some cases, YCSB tests involve a mix of read, write, and other operations, simulating real-world usage patterns where multiple types of operations occur concurrently.

2.5.2 Hardware and software specifications

Table 2.1 shows the system specification we used for this work.

System	
Operating System	Window 10 64 bit
Memory (RAM)	8GB
CPU	Intel Core i5-1135G7 4 x 2.4 - 4.2 GHz
Software	
Yahoo Cloud Service Benchmarking Redis	Version ycsb-0.17.0 Version 6.2.6
HBase	Version 2.4.9
Maven	Version apache-maven-3.8.4z

Table 2.1: Hardware and software specifications.

2.5.3 Experiments work load

We conducted the test using six workloads. We recorded the result by changing the number of threads used in the test. For each test, we build a chart that shows the recorded performance (throughput measured by operation per second) for both databases while changing the number of used threads. The number of threads can be determined in practice according to the application. Next, we will review the result for each workload.

2.5.3.1 Load A

In this workload, the tool divides the total operation into 50% read and 50% write operations. This workload can be considered heavy in terms of updates. The result is shown in Figure 2.2. 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.

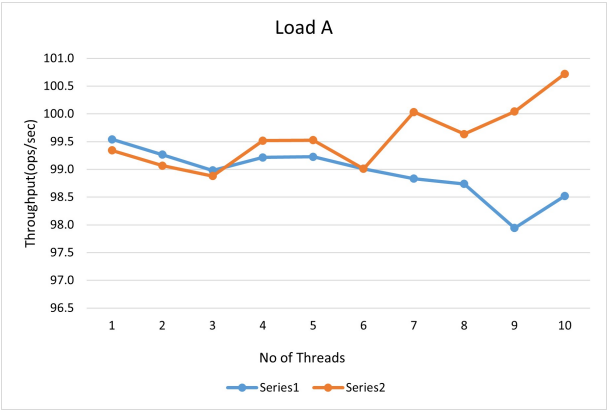

Figure 2.2: Load A.

2.5.3.2 Load B

The read operation takes 95% of the total operations in this workload. Thus, we can denote this workload as a reading-heavy test. The maximum recorded throughput for Redis and HBase is 99.54 and 100.33 seconds, respectively. The result is shown in Figure 2.3. Again, the HBase performs better than Redis with eight or more threads. Redis has no noticeable change during all the threads experiment.

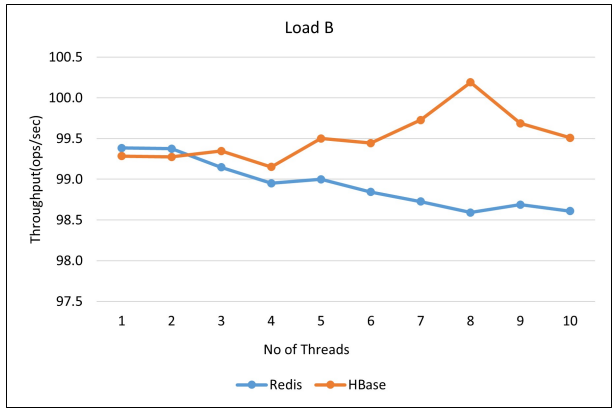

Figure 2.3: Load B.

2.5.3.3 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 maximum recorded throughput for Redis and HBase is 99.36 and 100.39 seconds, respectively. The result is shown in Figure 2.4.

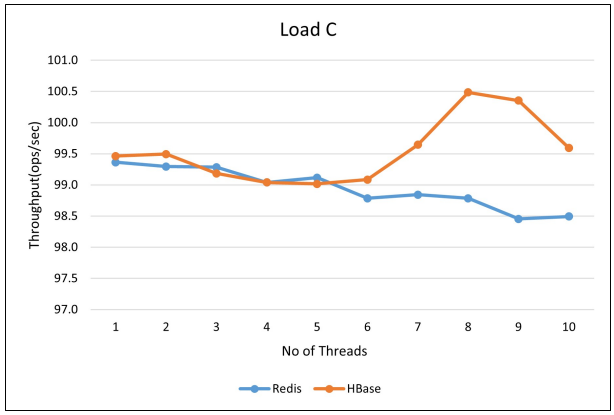

Figure 2.4: Load C.

2.5.3.4 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 maximum recorded throughput for Redis and HBase is 99.48 100.61 seconds, respectively. Figure 2.5 shows the Load D result. HBase shows better performance with increasing the number of threads.

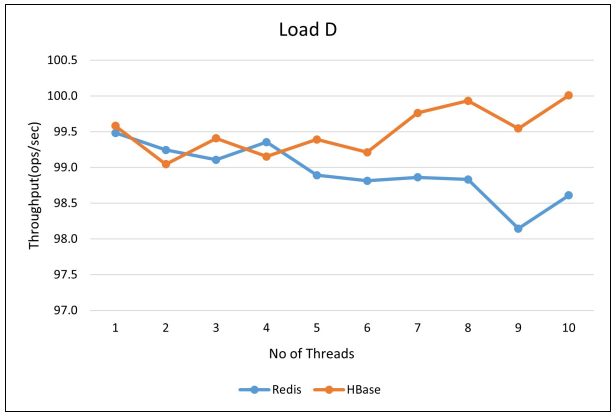

Figure 2.5: Load D.

2.5.3.5 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 2.6 shows the result comparison for both databases. The maximum recorded throughput for Redis and HBase are 99.48, and 100.87 seconds. 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.

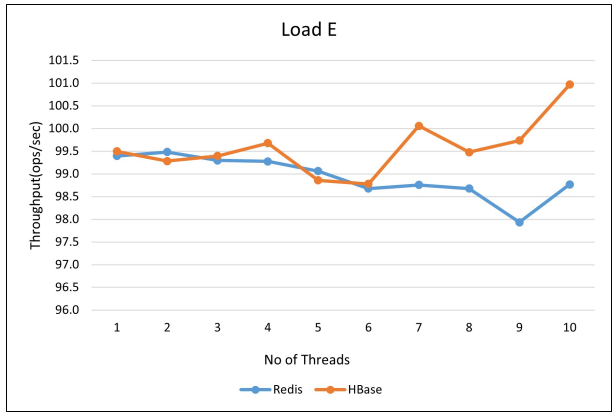

Figure 2.6: Load E.

2.5.3.6 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 2.7 shows the result for load F.

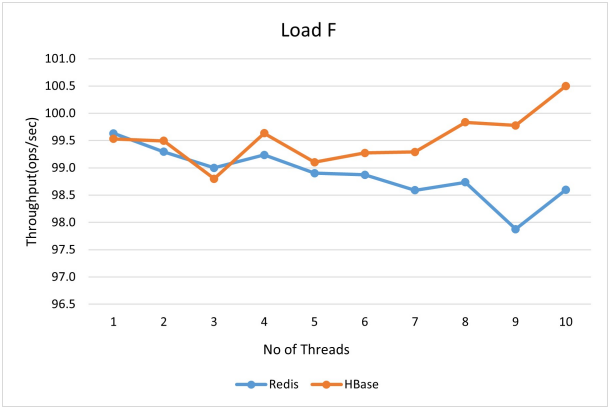

Figure 2.7: Load F.

2.5.4 Result from others research

To have a wider view, let's take a look at an experiment from another research [18]. The experiment benchmarks five databases, including Redis and HBase using workload A. Unlike our benchmark, the experiment benchmarked this database while increasing the number of records in the workload, where, in our case, we use the default workload records count while changing the number of threads. Figure 2.8 shows bench the results of the experiment. Redis and HBase show similar performance when the recorded count is low. However, HBase throughput is going higher when the record count is increased.

2.6 Conclusion

Applications programmers may choose between SQL and NoSQL databases. Despite their antiquity, SQL databases are still popular among programmers and web designers alike. The NoSQL database systems have become a good alternative to relational databases in some applications during the last decade. As they provide better scalability and schema-less structure, what can make software project development faster and easier? This advantage and popularity led to the introduction

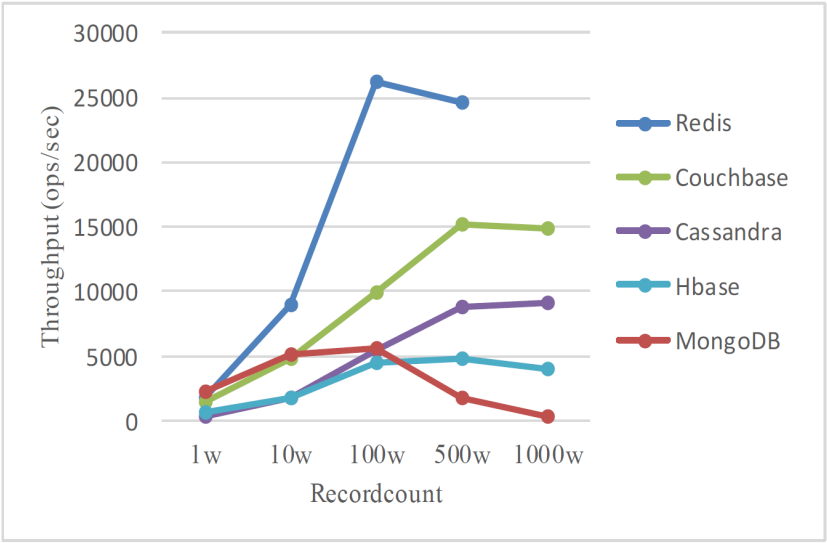

Figure 2.8: Throughput of executing workload A with increasing record count.

of many NoSQL database systems. However, each may provide some features and miss some others that are provided by another system. Thus, the selection between the available NoSQL databases becomes more complex and needs a comparison between the candidate systems. 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, when we increase the number of used threads, the HBase shows higher throughput in comparison to Redis.

3

Utilizing Redis for GTFS trip planning

Dissertation key point and finding: *We introduced 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.*

Contents

3.1 Introduction

In 2005, Google started using the Google Maps web application to create a transit trip planner. TriMet and Google in Portland formulated the General Transit Feed Specification (GTFS) [11]. In 2007, Google published transit feed specifications and encouraged transit agencies to use and depend on the GTFS format to create and post transit data on the web for public use as open sources. Later, the feed became the most commonly used standard for static transit data exchange in the United States [19]. Because of its rising popularity, the transit industry adopted the GTFS format as a standard for sharing their schedule data. At that time, over 170 transit agencies in the United States and Canada generated and published their schedules as GTFS [19]. Later, applications that use GTFS like CarFreeAtoZ, Hopstop, and MapQuest were created, providing on-map stop locations, bus timetables, and trip-planners features. The essential part of such applications is the trip planner, where the server searches GTFS data to find a possible route between two locations. Considering trips, routes, and transit between trips in stops (bus stops or transport stations), planning a trip is more complicated than finding a path in a graph.

Finding the shortest path in a graph is a common problem, and there are several algorithms like Dijkstra [20], Bellman-Ford [21, 22], Floyd-Warshall [23], Johnson [24]. The algorithm may be a bi-criterion or multi-criterion, where the criterion here is how we consider the weight for the graph edges. For example, if the graph represents a road network, the road weight may be a bi-criterion by considering the distance and the cost or maybe a multi-criterion if we consider more factors. Many algorithms try to solve the shortest path problem by reducing the weight value to a single value; these algorithms fall into categories like k-th shortest path algorithms [22], two phases algorithms [25], label setting algorithms [26, 27, 28], label correcting algorithms[29, 30, 31, 32], and others[33, 34, 35, 36]. Some research [37, 38] uses a variation of these algorithms to introduce an algorithm to find paths in local transport networks. The considered weight criteria and the data structure used for the implementation affect the algorithm's performance. This chapter will

introduce a new algorithm as a new variation, where the performance is enhanced by using a data structure that reduces the time required to search the data. The algorithm ignores the weight criteria using the number of transitions made to evaluate results instead of calculating and examining weight to choose the best possible next transition every time.

In trip-planning applications, every time a user runs a trip planner, the server runs the route searching or trip-planning algorithm, which is time-consuming. This chapter also proposes 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. 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, from [39, 40] and measure and compare the performance with and without using Redis. One of the disadvantages of this approach is that we generate extra data. However, the generated data can increase the ability to analyze the transportation service and help the city provide better public transportation using a Smart City paradigm[41]. We implement the algorithm and the strategy as an open-source project using C#. The project code is available at https://github.com/mustafamajid/GTFS.git. The finding of this chapter is published in [8]

3.2 Overview of GTFS data

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.

3.2.1 GTFS files and structure

GTFS is a set CSV of files that contains data about local transport trips, routes, stops, and timetable data in a CSV format. Each file represents a database table. GTFS includes three types of tables: Required, Optionally Required, and Optional tables [42]. This categorization gives the GTFS data the flexibility to include more or less data according to the agency or the application's needs. The required file must be present in all of the GTFS. The optionally required files must be present depending on some other GTFS data. Next, we will describe the required files:

- *agency.csv*: Contains information about the transport agencies providing the transport service described by the GTFS data set. The file lists information like agency ID, agency name, and URL of the agency website. It can also optionally show information like the URL of the ticket pushes site for the users and contact information like email and phone number.

- *stops.csv*: Contains information about transport stops and stations like stops name, stops ID, and the geographical location of the stops as latitude and longitude values. This location information is useful for transport applications, especially for planning trips. It can be used to calculate the distance between the stops, which helps decide the next transition or determine the walkable distance between stops.

- *routes.csv*: describes the trajectories for each trip with general and geographical information. Each route is pass-through a set of stops or stations. However, in the GTFS structure, routes are linked to trips, and trips are linked to stops. Information like route ID, route name, and agency ID links each route with the agency that provides the service.

- *trips.csv*: describes the trip information like trip ID, head-sign, which is the text that appears on signs showing the destination of the trip for riders. The file contains a service ID field to refer to

the service that this trip belongs to. Also, the file shows the route ID of the route the trip uses.

- *stop_times.csv*: In this file, the arrival and departure times of the trip at each stop are listed.

3.2.2 Understand GTFS data

There are three main objects in the GTFS: data routes, stops, and trips. The route is a path that passes through a set of stops or stations. The trip refers to a journey made by a vehicle from an initial stop to an end stop. We can view or read GTFS data as one or more routes listed in the routes.txt file. Each route has one or more trips in the trips.txt file. Each trip visits a series of stops (from stops.txt) at specified times in a specific sequence (in stoptimes.txt). Trips.txt and stoptimes.txt contain only time of day. The calendar.txt and calendar_dates.txt files determine which days a given trip runs [43]. Each file has a reference to the other related files. For example, trips.txt has a reference to the routes.txt file using the route ID field as a foreign key or reference. GTFS can contain other files like fare_rules, shapes, and feeds_info [44]. Figure 3.1 shows the structure and the relations between the fields of the GTFS data file. As the GTFS aims to exchange transit information, specific preprocessing is needed so it can be used for other purposes. The GTFS data set is generally loaded into a relational database (e.g., MySQL, PostgreSQL, Oracle), where the developer will process the scheduling and transit data. In some cases, a database with a particular geographical object querying ability is used.

3.2.3 GTFS-Realtime

GTFS-Realtime is a standard created by Google to enable transit agencies to provide updates about their services in real time. The set of data the GTFS-realtime feed provides contains vehicle positions, trip updates, and service alerts. Vehicle Positions contain data about the

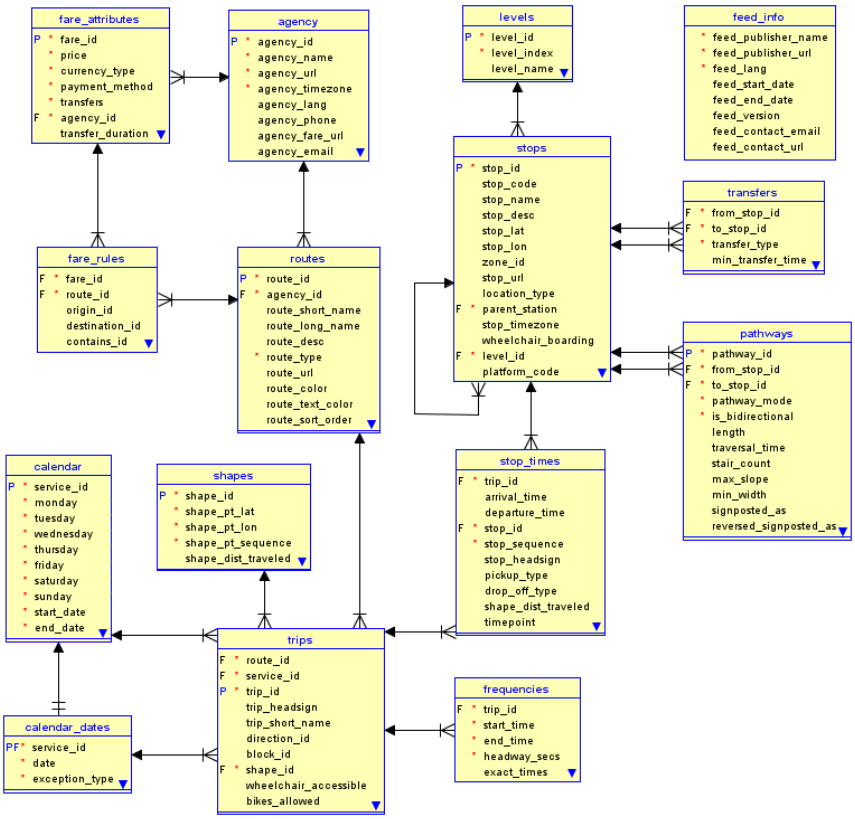

Figure 3.1: GTFS Entity Relationship.

past, but Trip Updates contain data about the future. Typically, one GTFS-realtime feed can contain only one type of data out of these three types. Many governmental and business agencies have multiple GTFS - real-time feeds (one for vehicle positions, one for trip updates and one for service alerts). GTFS-Realtime uses Protocol Buffers to ensure efficient data transmission and has three core message types: VehiclePosition for tracking vehicle location TripUpdate for trip-related information such as expected timings and delays, and Alerts for service disruptions. These messages are contained within a FeedMessage, which acts as a container and contains feed-specific metadata. GTFS-Realtime aligns with the General Transit Feed Specification (GTFS), ensuring compatibility with static schedule data, and can be extended for custom fields. However, this dissertation works only on GTFS data.

3.3 The algorithm

Usually, a variation of label-setting and label-correcting algorithms, especially Dijkstra algorithms, are used to implement trip planning algorithms. The data structure used by the trip planning algorithm to hold the GTFS data plays a primary role in the algorithm's performance and behavior, as it can affect the speed and logic of data access. A trip-planning algorithm can be implemented at the stop level where every time the algorithm searches for a possible next stop or at the route level where the algorithm checks the available route each time. This algorithm will use both the stop level and route level search. We will consider the data in Figure 3.2 as an example to describe the algorithm steps. Each circle represents a stop. The number inside the circle is the stop_id. The routes are represented by a set of arrows denoted by letters.

3.3.1 Preprocessing

In the preprocessing steps, we create and fill the data structures with the GTFS data. The task here is to make the data more understandable by reducing the joining effort and faster and easier access by the algorithm code. Preprocessing contains the following steps.

3.3.1.1 Find the routes

The route file in the GTFS data provides an ID for each route, but some routes may have different stops set to visit during different trips [45]. We distinguish each route by the list of stops it visits in a specific order. We use the stoptimes file to extract this list for each route and create a route data structure containing all information provided about that route in the route file and a list of all stops this route visits, ordered by the visiting sequence. For example, in Figure 3.2, the list of stops for route

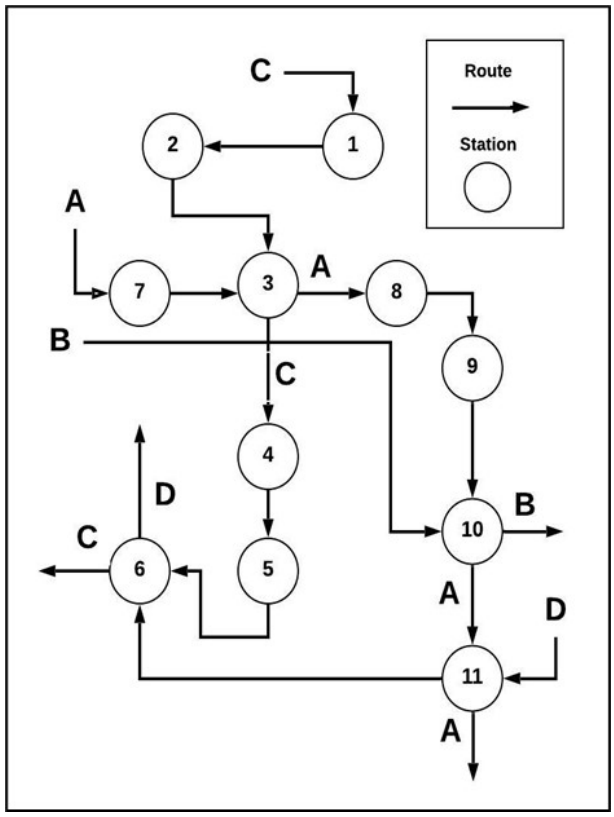

Figure 3.2: GTFS Routing Network Example.

A is 7,3,8,9,10,11. Finally, a list of route objects will be created. The route structure has a VISITED index variable used to indicate whether the route is visited or not. Initially, the VISITED variable will initialize to -1 to indicate that no stop in this route has been visited yet and prevent rechecking the already checked routes. For example, if we use route B to reach stop10 and then from that stop, the algorithm starts to check all possible routes and put all the next stops in the waiting list to be checked, the VISITED variable for route A will be set to stop10. Thus, only the stops before stop10 can be checked next time using route A, and if all the stops in the route are checked, the VISITED will be set to zero, the index of the first stop.

3.3.1.2 Create stops list

The stop data structure contains all the static information about a stop provided by the stops file of the GTFS data with a list of all routes that pass through this stop. The route list can be extracted using the route data from the previous step. For example, in Figure 3.2, the route list of stop3 is A, C.

3.3.1.3 Find distance between stops

The stops file contains the latitude and longitude of each stop location. We use the Haversine formula [46] to calculate each stop's distance from all the other stops. The formula is shown in Equation 3.3. If the distance between two stops is walkable, we create a walkable route between these stops. A distance is walkable if the distance between two stops is short enough for the passenger to walk. The walking route is usually preferred or taken when there is no trip between these two stops or when the waiting time is much more than the time required by the passenger to walk the distance. Some research defines the distance as a walkable distance if it is less than or equal to 400m [47]. However, we set the walkable distance in our project to 300m to reduce the need to consider non-straight paths. The walkable route uses the same route data structure with all the fields set to the "walk" string, and the stops list contains only the two stops, which are linked by this walkable distance. Walkable routes are important when the user must change the route by walking to another stop.

$$a = \sin^2\left(\frac{\Delta\text{lat}}{2}\right) + \cos(\text{lat}_1) \cdot \cos(\text{lat}_2) \cdot \sin^2\left(\frac{\Delta\text{long}}{2}\right) \tag{3.1}$$

$$c = 2 \cdot \text{atan2}\left(\sqrt{a}, \sqrt{1-a}\right) \tag{3.2}$$

$$d = R \cdot c \tag{3.3}$$

where

- $\Delta\text{lat} = \text{lat}_2 - \text{lat}_1$
- $\Delta\text{long} = \text{long}_2 - \text{long}_1$
- R is the Earth's radius (mean radius = 6,371 km)
- $\text{lat}_1, \text{long}_1$ are the coordinates of the first point
- $\text{lat}_2, \text{long}_2$ are the coordinates of the second point
- d is the distance between the two points

3.3.1.4 Trips timing data extraction

After using the algorithm to find possible solutions, a time check is needed to validate the routes according to the trip's timetable, considering the time parameter provided by the user's query. Time data is collected from the stoptimes file and calendar_date file. The stopstimes file shows the arrival and departure times at the stops the trip visits. Each set of trips on a specific route is denoted as a service with a unique service_id. The calander_dates file has two fields, start_date, and end_date, which denote the period when the service is available. The trips file links each trip with the route and the service it belongs. In this step, we extract this data into a time data structure and create a dictionary to link each route and trip with its service id.

3.3.1.5 Creating dictionaries

The above steps are about loading the GTFS data into memory. All the stops, routes, and time data are loaded into lists of stop data structure, route data structure, and time data structure. We increase the algorithm execution speed by using dictionaries for the indexes of these lists. A dictionary will map the stop ID to an integer value, representing the index of the stops data structure in the list. Thus, we can access a stop object in a list of stops with the stop ID using the dictionary. That will

eliminate the need to search the list for a specific stop and increase the execution speed as the dictionary's mapping time is O(1) [48].

3.3.2 The output data structure

The algorithm input is the initial stop ID and the destination stop ID. The output is a list of all possible trip plans where each plan may contain one or a combination of routes. We used the word PATH to denote a plan (a solution). Each PATH contains a list of one or more transitions made from one stop to another using a specific route. We denote these transitions as MOVE and the list of MOVE in each PATH as WAY. The PATH data structure contains attributes to denote the initial stop and the destination stop, with their names. The MOVE data structure contains the start-stop ID where this transition starts from, the end-stop ID, the route ID used to make this transition, and other information like service ID, the stops, and the route names. Figure 3.3 shows the structure of PATH and MOVE. For example, in Figure 3.2, the algorithm is used to find the possible solution from stop2 to stop6. Solutions will be a list that contains two PATH objects. The first solution is a PATH containing only one MOVE in the WAY list. This MOVE describes a transition from stop2 to stop6 using route C. The second solution is a PATH, which contains three MOVEs in the WAY list. The first MOVE mentions a transition from stop2 to stop3 using route C. The second MOVE mentions a transition from stop3 to stop11 using route A; the third MOVE is from stop11 to stop6 using route D.

3.3.3 Definition

- TRANSIT LIMIT (TL): A constant number determines the limit of change between transport trips (for example, between two buses). This value is used to stop the searching and prevent the algorithm from getting stuck in an endless loop. Studies show that the typical transit limit is four transits per plan [49], as people did not prefer

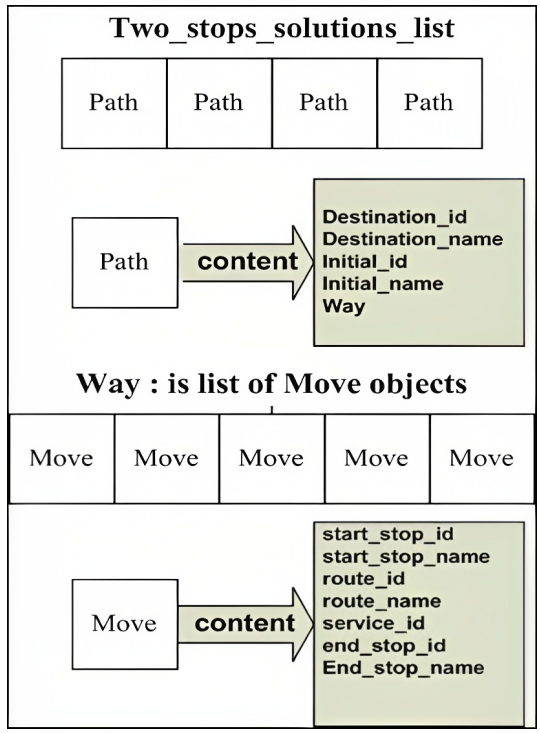

Figure 3.3: Path and Move Data Structure.

to use plans with many transits, as it causes a waste of time and extra cost.

- *INITIAL*: The initial stop where the planning starts from.
- *FINAL*: The final step of (the destination).
- *MOVE* (S1, S2, R): Denote a possible transfer from the stop S1 to stop S2 using the route R.
- *PATH*: A list of MOVEs where for each consecutive two MOVEs PATH[i] and PATH [i+1], PATH[i].S2 = PATH[i+1].S1 each PATH represents a possible route solution.
- *SOLUTION LIST (SOL_LIST)*: A list of PATH used to store all the possible solutions.
- *TRANS_COUNT*: Counter represents the number of transit during the travel till the Current stop

- *QUEUE ITEM(C, TRANS_COUNT, WAY_PASSED)*: a data structure used to store algorithm parameters, where WAY_PASSED is a PATH variable, C is a stop currently reached starting from INITIAL stop and going through WAY_PASSED, TRANS_COUNT is an integer represents a count of transit mad to travel the WAY_PASSED.
- *QUEUE*: A Queue of Queue_Item, stored according to the execution order

3.3.4 Algorithm

Figure 3.4 shows the pseudocode of the algorithm.

The algorithm starts from the initial stop by creating a queue item that contains the WAY_PASSED that has nothing but the initial stop and sets the transit count to zero. The algorithm then will pop items from the waiting queue till the queue is empty. For each route pass through the current stop of the popped item step4 will check if these routes contain the final stop then the WAY_PASSED will be updated by adding a transition from current stop to the final stop, and then add this WAY_PASSED as a solution in the SOLUTION LIST. In step5, the algorithm simply creates a new waiting item and adds it to the waiting queue by add a transition from the current stop to every new possible stop using all unvisited routes that pass through the current. VISITED is not a status flag to mention if the whole route is visited or not. For any route Ri, VISITED is a variable (part of the route implementation data structure) that indicates the index of the earliest visited stop in the route; checking the VISITED variable will decide if the route is visited from a current stop or not. For example, for a route R1 passing through stops 1, 2, 3, 4, 5, 6 in order. If the algorithm visits stop 4 coming from another route let's say R2 then step 5 will create transitions (MOVE) to use R1 from stop four going to every next stop (5 and 6) and store these transitions to wait for execution in the QUEUE. When these moves for stops 5 and 6 are popped from the queue for execution, the algorithm will consider route R1 as visited and no action will be taken for R1. The

INPUT : INITIAL, FINAL

OUTPUT: SOL_LIST

Step1: Initialize QUEUE by add QUEUE_ITEM(INITIAL,0,EMPTY WAY_PASSED) and set SOL_LIST =empty

Step 2: IF (Queue is empty) then Exit

Else

pop the first QUEUE_ITEM Q and Set Current = Q.C , WAY_PASSED = Q.WAY_PASSED ,

TRANS_COUNT =Q.TRANS.

Step3: IF ((Current == FINAL) OR(TRANS_COUNT== TRANS_LIMIT)) then Goto Step2.

Step4: For each unvisited route Ri that pass through Current:

IF Ri.VISITED not refer to Current or any earliar stop in Ri then:

starting from Current, IF next Stops in Ri contain the Final then :

Add MOVE(Current, FINAL, Ri) to a copy of WAY_PASSED.

Create new PATH using the newally created WAY_PASSED

Add PATH to SOL_LIST.

Step5: For each unvisited route Ri that pass through Current:

For each Stop S in Ri after Current :

Add MOVE(Current, S, Ri) to a copy of WAY_PASSED name it NEW_WAY_PASSED.

Add QUEUE_ITEM(S,TREANS_COUNT + 1, NEW_WAY_PASSED) to QUEUE

Mark Ri as visited from Current by updating Ri.VISITED

Step6: Goto Step2

Figure 3.4: Route planning algorithm.

algorithm will check only the possibility of other routes passing on these stops.

3.3.5 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 it in a Redis database. The result will be retrieved from the Redis server for all user search requests without rerunning the algorithm.

3.3.5.1 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.

3.3.5.2 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.

3.3.5.3 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 3.5

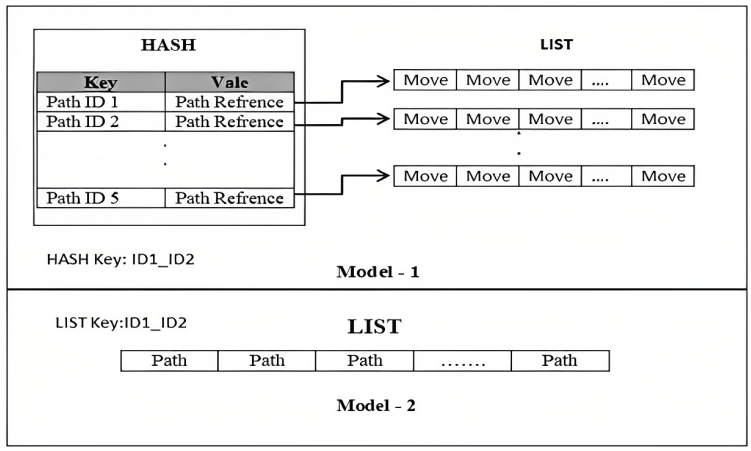

Figure 3.5: Redis models for PATH data structure.

After the experiment, we chose the second model due to the fact that the first model accessed the database twice in comparison to the second model, which needs one database access, although this shows a very small difference when experimenting with both models, which was done on a local machine DB server, it can logically show more difference in a production server in the case where database and back-end reside in different servers. Moreover, it is easier to implement, although it needs more back-end processing to extract the data from the string. Thus, we depend on it in the C project. Worth mentioning that the GTFS data size on the file system for Budapest is 41MB and 1.4MB for Debrecen, whereas the generated data size in the Redis dump file is 306MB for Budapest and 98MB for Debrecen.

3.3.6 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 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

The 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, as shown in Figure 3.6. Also, we note that the average time required to search the Budapest data (1.0280 seconds) is greater than the average time required for Debrecen (0.08915 seconds). However, using Redis, we do not have this variation as the recorded time is 0.08915 and 0.08914 seconds for both cities.

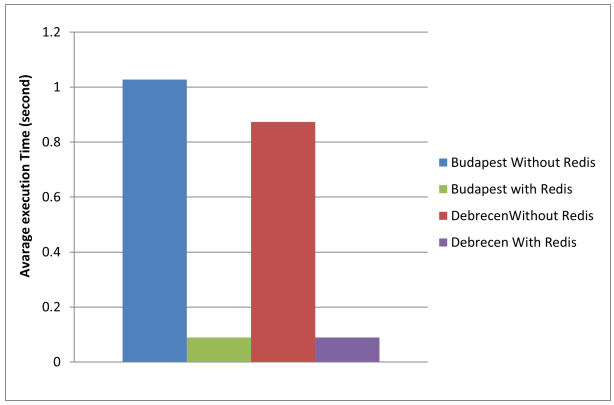

Figure 3.6: Execution Time Comparison.

3.4 Chapter Summery

Sharing transport information is a critical factor in a successful transportation system in modern and Smart Cities. Thus, the need to use a standard format for sharing transport data is increased. Transport agencies widely use GTFS (General Transit Feed Specification) worldwide as a standard format to share and publish their data. Thus, the need to efficiently process and utilize GTFS is increased. Trip planning is one of the most demanded applications that use GTFS data. Such an application can tell the user which local transport trip or trips (e.g., bus, tram, metro) can travel between two locations. Algorithms that find a path in a graph are a bi-criterion or multi-criterion, where the criterion is the number of weights for the graph edges. We introduce a new variation of the trip planning algorithm, which ignores the criterion factor using transition limit to evaluate the search and considering both stop level and route level search. Thus, we enhance the performance by eliminating the time required to assign and process the arc's weight and reduce the algorithm implementation complexity. We introduce data structure and an implementation method that can produce better performance. Querying the server to run a trip planning algorithm with each user request is time-consuming and can cause server overhead, especially with large city data. We introduce a method to enhance the server performance and reduce the server overhead using the Redis NoSQL database structures. Redis data structures are used to store all the trip plans between any two GTFS stops. Thus, we can serve user requests by querying the Redis server, which is much faster than running a search algorithm using GTFS data in real time. We propose two Redis models. The first model uses Redis HASH and LIST structure, whereas the second model uses LIST structure only. Both models provide similar performance but can be utilized differently by the software developer according to the development and the application needs. We experimented with the server performance and showed the performance enhancement using Budapest and Debrecen cities' GTFS data. The experiments show that this method can produce a faster performance with nearly constant time regardless of the data size (city size). The data generated can also

provide a source of information for the analysis and planning of the local transport system in smart cities.

Table 3.1: Experiments results (time in seconds)

NO	Budapest Without Redis	Budapest Using Redis	Debrecen Without Redis	Debrecen 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
7	1.0492	0.08901	0.8600	0.08906
8	0.9930	0.08903	0.8795	0.08902
9	1.0418	0.08911	0.8901	0.08902
10	1.0335	0.08932	0.8684	0.08901
11	1.0071	0.08930	0.8562	0.08921
12	1.0004	0.08914	0.8786	0.08928
13	1.0255	0.08911	0.8583	0.08914
14	1.0050	0.08908	0.8625	0.08923
15	1.0268	0.08918	0.8557	0.08910
16	1.0254	0.08935	0.8540	0.08906
17	1.0515	0.08917	0.8544	0.08919
18	1.0300	0.08909	0.8763	0.08923
19	1.0476	0.08905	0.8998	0.08907
20	1.0519	0.08926	0.8849	0.08921
21	1.0005	0.08918	0.8615	0.08906
22	1.0585	0.08900	0.8785	0.08906
23	0.9961	0.08919	0.8954	0.08918
24	1.0340	0.08911	0.8865	0.08921
25	1.0253	0.08915	0.8932	0.08901
26	1.0215	0.08924	0.8815	0.08922
27	1.0317	0.08916	0.8848	0.08921
28	1.0381	0.08930	0.8575	0.08908
Average	1.0280	0.08915	0.8737	0.08914

4

GTFS Trip Timing Performance Enhancement

Dissertation key point and finding: *We introduced an algorithm for validating the trip plans based on the trip timetable and proposed 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.*

Contents

4.1 Introduction

For two key reasons, public transportation accessibility has become a hot topic for scholars and transit organizations. First, improved transit accessibility promotes active transportation (such as walking and bicycling) while decreasing private vehicle use. As a result, it will enhance public health and reduce Greenhouse gas emissions (GHG) [50, 51, 52]. Second, transit-dependent people primarily rely on public transportation to reach key services (e.g., workplaces, universities, and shopping centers). Thus, transit accessibility is crucial to attaining socioeconomic fairness [53, 54]. Also, transit accessibility research may help guide decisions about transportation investment and land use development [55].

Handling a user trip planning request requires two actions or steps. First, identify all feasible pathways or routes between two places as candidate solutions; second, filter and validate these solutions according to the user's schedule and start time. Route planning is more complex than identifying a path or route in a graph since it considers journeys, directions, and intermediate transfers between bus stops or stations. We have already introduced a trip planning algorithm variation [8] in the previous chapter. This algorithm ignores the weight criteria while checking for all possible next transitions from the current and uses a limit for the number of made transitions to find the best route (trip plan). This trip planning algorithm can find the possible trip plans without considering the timing factor. However, some plans may be rejected due to time conflict between the plan trips and the GTFS trips timetable or trip unavailability at the time the user determines to start the journey. Therefore, we must calculate and find the trips time and transit accessibility as the next step.

To calculate transit accessibility in spatiotemporal dimensions, trip time for station pairs must be calculated at any particular time of day, which is practically impossible with a standard computer as it is time-consuming and needs high computation power [56]. Although previous studies [57] introduce algorithms that try to calculate the trip

time and transit accessibility while reducing the time complexity and computational power, there is still a need to find an approach to simplify the complexity of such problems solution and reduce the required time and resources, which is the aim of this chapter. Algorithm enhancement is a common research topic [58, 59, 60]. This chapter focuses on two parts. First, we introduce a new time validation algorithm that can find the timing information for a trip plan or reject the plan if there is a time conflict according to the trip timetable in the GTFS. Second, we go beyond the algorithm enhancement and propose RMH (Range Mapping Hash), which is a new method that can find and extract the timing information for any trip using GTFS data with O(1) time complexity. Our new approach (RMH) eliminates the need for an algorithm to search the GTFS timing records. We use Redis NoSQL Hash to create RMH. Thus, we provide a solution by turning the problem of simplifying the existing algorithms into a simple database interaction that can run even on a standard computer. The idea is that for a route going through a station, at any minute between the last going bus and the next bus, the answer to the question "When is the next bus time?" will be the time of the next bus. The RMH is applicable not only for the GTFS timing data but also for improving the performance of similar problems, as we describe later. We experiment with the performance of RMH and compare it to run-time search algorithm performance using arbitrary search input for 30 pairs of origin and destination stops using the GTFS data of Debrecen and Budapest. We implement the algorithm and RMH as an open-source project using C# and Redis, available at https://github.com/mustafamajid/GTFS-csharp. Next, we review our route planning algorithm and its output data structure [8]. Then, we introduce the time validation algorithm, which will use this output to provide the trip's timing information. Later, we introduce the RMH approach and the Redis implementation. Finally, we list our experiment's results and performance evaluation. The findings of this research are published in [61].

4.2 GTFS and trip planning (considering time)

The GTFS data is a set of tables, usually in CSV file format. There are three main objects in the GTFS: route, trips, and stop [43]. The routes represent the pathway used by the vehicle, a bus, tram, train, etc., and are usually denoted by the vehicle name. The route visits a set of stops in a specific sequence, where the stop can be a bus or tram stop or a train or subway station.

Planning a trip between two locations (the start and destination) requires finding all possible single or combinations of trips that can take the passenger from start to destination points. Finding trip plans can be divided into two steps. First, find all possible routes that can connect the start to the destination point, and these will be the candidate solutions list. Then, find the trip's timing information and check for any time conflict in the candidate's plans. Time conflict here is the case when one of the trips in the plan is arranged not earlier than the last trip on the next route. This case can happen only when the plan contains more than one trip. To understand the problem, we use Figure 4.1 as an example of the GTFS data. The figure shows three routes, A, B, and C, going through a set of stops denoted by circles with numbers. We consider that the user wants to start the trip from stop seven at 11:10:00 going to stop 6. Therefore, the candidate solutions will be as follows: first, the user takes route C to stop nine and then takes route A from stop 9 to stop 6. The second solution is that the user walks from stop 7 to stop three and then takes route B to stop six if the distance is walkable [62]. The next step is to validate the solution according to the timetable. For the first solution, as in the figure, if the user starts at 11:07:00, 11:17:00, or 11:27:00, he will arrive at stop nine at 11:35:00, 11:35:00, or 11:45:00, respectively. Thus, the user will take the trip at 11:17:00 because it is the earlier trip and then arrive at stop nine at 11:35:00, where the next trip using route A will be at 11:45:00 and reach the destination at 12:05:00. In the same way, we can find the time information for the second solution. As we mentioned earlier, a solution can be rejected if there is a time conflict; for example, the first solution may be rejected if there is no outgoing trip using route A from stop nine

any time after 11:35:00.

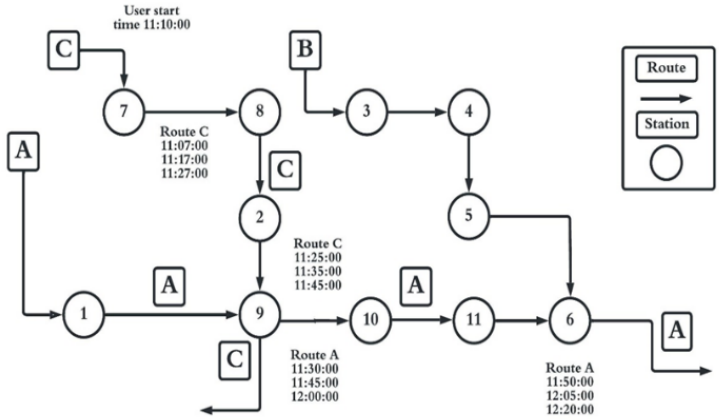

Figure 4.1: GTFS routes and stops example.

4.3 Find trip time information

4.3.1 Data structure

Finding trip plans according to the time is a complex problem and needs computation power that is not provided by standard computers [57]. For this work, we use the output structure (the candidate solutions) provided by our previously proposed trip routes planning algorithm [8] as an input to introduce our new trip timing algorithm. Figure 4.2 shows the UML design of the trip routes planning algorithm output with additional fields to store the time data.

The structure contains three main objects SOLUTION_LIST, PATH, and MOVE. Each MOVE represents a single transition from a start-stop to an end-stop using a route (e.g., a bus), and PATH denotes a trip plan or solution containing at least one or more transitions (MOVE) stored in a list called Way. The algorithm's final output is a list of PATH called the Solution_List. The MOVE_WITH_TIME class was inherited from

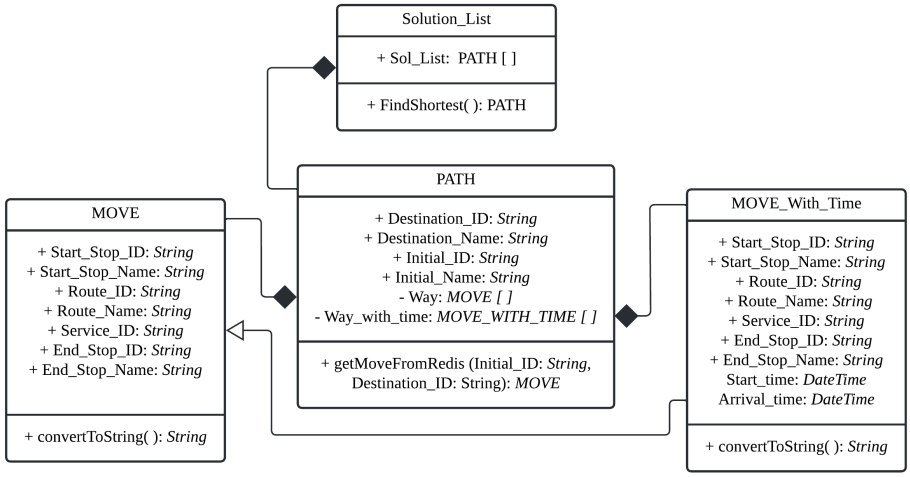

Figure 4.2: Algorithm data structure.

the MOVE class and contained the arrival and departure time fields. Finally, a list of MOVE_WITH_TIME is added to the PATH class called Way_With_Time. The task of the next trip timing algorithm is to validate the PATH by checking every MOVE object in its Way list. If a time conflict is found in any MOVE, the whole PATH will be rejected; Otherwise, a new MOVE_WITH_TIME object will be created from the current MOVE by adding the timing fields. The newly created list of MOVE_WITH_TIME objects will form the Way_With_Time list.

4.3.2 Algorithm

The stoptimes.txt file lists a set of records for each trip; each record contains stop ID, trip ID, trip arrival, and departure time at that stops. The set of trip records is present in the file ordered by trip ID and the arrival time. Thus, if the file starts to list a trip that visits ten stops at row number N, then row N contains timing data about the first stop, row N + 9 shows the data about the last stop that the trip visits and row N + 10 will list data for the first stop of another new trip if any. Every PATH must be checked by examining the MOVEs in its WAY list using T's time. The stoptimes.txt file record is checked

sequentially to find the trip with the closest time to T. Initially, T is set to the time determined by the user (USER_TIME) to start the trip, and during the next MOVEs check, T is set to the arrival time at the last checked MOVE end stop. The check starts from the first record in the stoptimes.txt until finding the first record (i) with stop_id equal to the MOVE start_stop_id and with the same route used by the MOVE and the departure time is greater than T and one of the next record (i + j) in the same trip with stop_id equal to the MOVE end_stop_id. Where j is the number of intermediate stops, if such records are found, a MOVE_WITH_TIME object is created using the examined MOVE and record (i) departure time as Start_time and record (i + j) arrival_time as arrive time for the new MOVE_WITH_TIME object and as the new T value for the next MOVE check. If no such record is found in the stoptimes.txt file, then the whole PATH is rejected and mentioned as an unacceptable solution. The new resulting MOVE_WITH_TIME objects are used to form a WAY_WITH_TIME list. Figure 4.3 shows the Trim timing algorithm that validates the MOVE according to the trip's timing information. The algorithm input is the start-stop from which the MOVE starts, the end-stop where the MOVE ends, the route used to make that MOVE, and the user's time. The algorithm output must be the trip on that route with the nearest time to T

4.3.3 Time complexity

Searching the stoptimes.txt file record is a time-consuming process as with any linear search, the algorithm performs a linear search for timing information. Let N be the number of records present in the stoptimes file. Then, the best case is if the stop with time greater than T is at the stoptimes file's first record. The worst case is O(N-1), and the average case is O((N-1)/2). That mean both worst and average cases are equal to O(N). Thus, the algorithm time is increased by increasing the number of records. We ignore the number of records between the start and the end stops, as this number is minimal compared to N.

```
Input: WAY a List of MOVE, USER_TIME, Stoptimes object list (data loaded from GTFS
       stoptimes file).
Output: WAY_WITH_TIME as List of MOVE_WITH_TIME objects
Step1: T=USER_TIME , WAY_WITH_TIME = empty
Step2: ForEach MOVE M in WAY
Do Step3 To Step5
Step3: Set M_WITH_TIME =NULL,
Step4: For (i=0 ; i< Stoptimes.Length-2 ; i++)
        IF(Stoptimes[i].stop_id == M.start_stop_id && Stoptimes[i].trip_id == M.trip_id &&
        Stoptimes[i].Departure_time >T ) Then:
        For (j=i+1 ; i< Stoptimes.Length-1 && Stoptimes[i].trip_id == M.trip_i; j++)
                IF (Stoptimes[j].stop_id == M. end_stop_id)
                        M_WITH_TIME = MOVE_WITH_TIME (
                        M,
                        Stoptimes[i].Derparture_time,
                        Stoptimes[j].Arraival_time).
                        ADD M_WITH_TIME to WAY_WITH_TIME,
                        T= Stoptimes[j].Arrival_time
        Goto Step2 check the next MOVE
Step5:  IF M_WITH_TIME ==NULL :
                Return FALS and Exit
Step6: Return WAY_WITH_TIME
```

Figure 4.3: Trip timing algorithm.

4.4 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 parts: 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 [63].

4.4.1 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 4.4 shows the trip time structure.

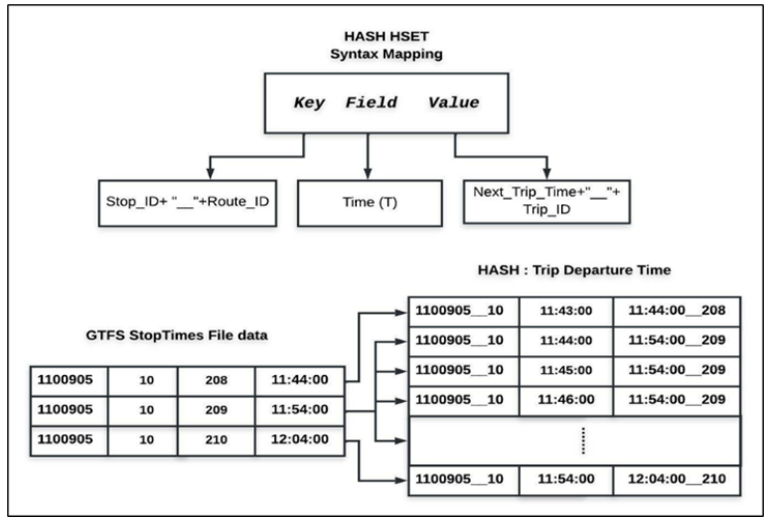

Figure 4.4: RMH trip departure time structure.

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 4.4, 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 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.

4.4.2 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 4.5 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.

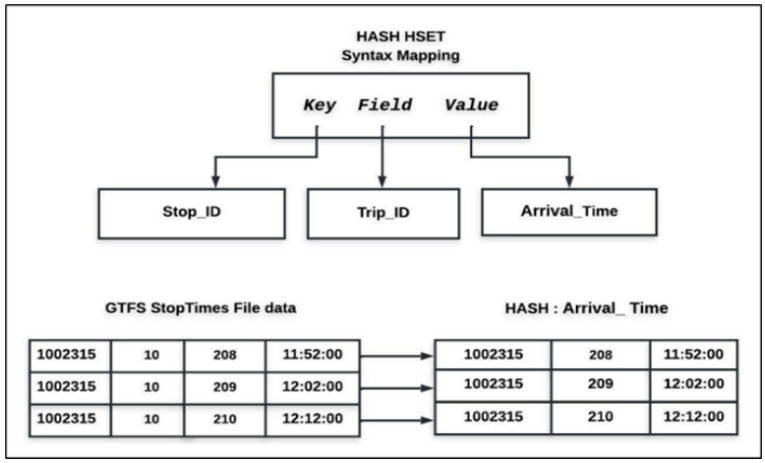

Figure 4.5: RMH arrival time structure.

For example, in Figure 4.5, 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: The 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 4.4. 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 does 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)[64], then RMH total complexity is two operations of O(1) complexity.

4.5 Experiment and results

4.5.1 Experiment tool

We implement the trip timing algorithm as a C# project. The project is a WinForm application (GUI) containing a set of classes: GTFSData for loading and preprocessing the GTFS data, Algorithm class contains the implementation of our previously published trip planning algorithm, TimeCalculator class contains the implementation of the trip timing algorithm (trip timing), Redis action class include the code for connecting to Redis database load the data to Redis and retrieve the solution and other classes. Figure 4.6 shows the UML design of the main classes in the project.

We used the GetMovesWithTime( ) function from the TimeCalculator

class for this experiment, which takes a MOVE list and start-time as a parameter and returns a MOVE_WITH_TIME list. We calculate the execution time for this function and compare it with the execution time for reviving the timing information using the RMH using the GetSolFromRedis( ) function from the RedisAction class. The GetMovesWithTime() illustrates the implementation of the trip timing algorithm given earlier, whereas the GetSolFromRedis() represents the interaction with the Redis RMH model, which is implemented as mentioned before. This function will retrieve the same result returned by the GetMovesWithTime() function (for the same parameters). The Redis RMH serves similarly to a data warehouse model where redundant data is stored and utilized to serve the application purpose. float

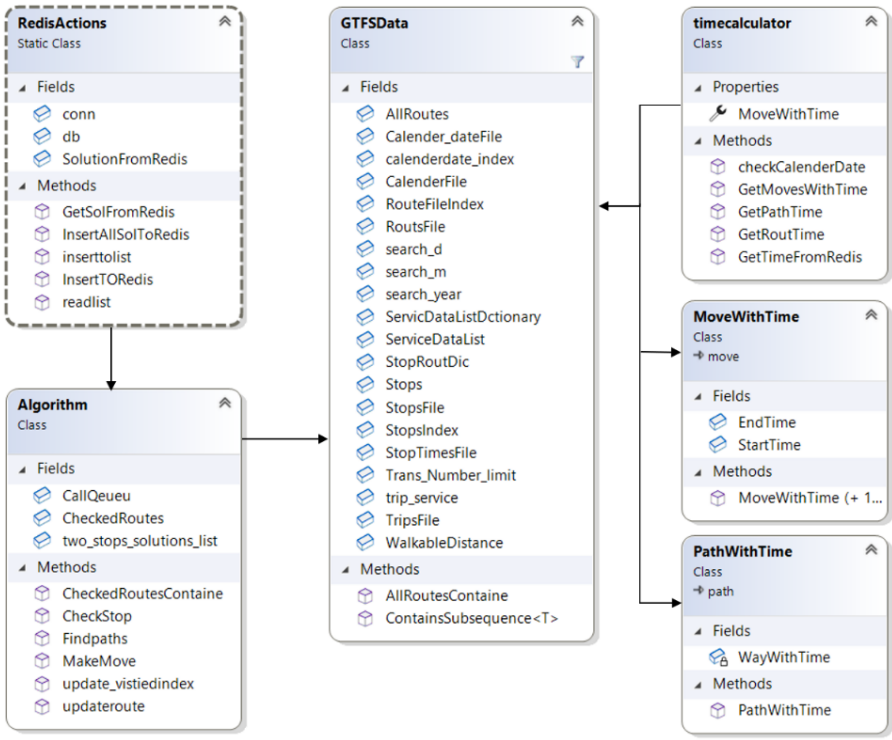

Figure 4.6: Test tool UML design.

4.5.2 Result

We experiment 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) that the computer takes to retrieve the result for each combination. Table 4.1 shows the recorded results.

Figure 4.7 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.

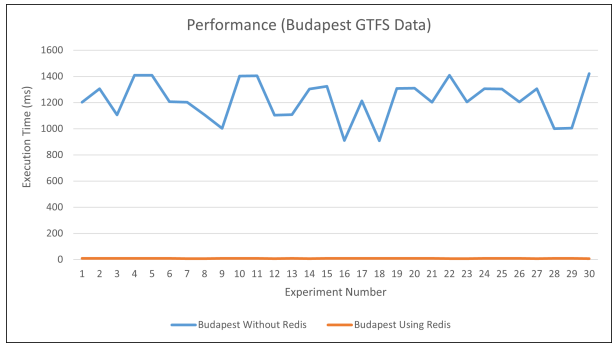

Figure 4.7: Budapest data experiments result.

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 algorithm. Figure 4.8 shows Debrecen data experiments' performance using Redis and the run-time

algorithm (without Redis).

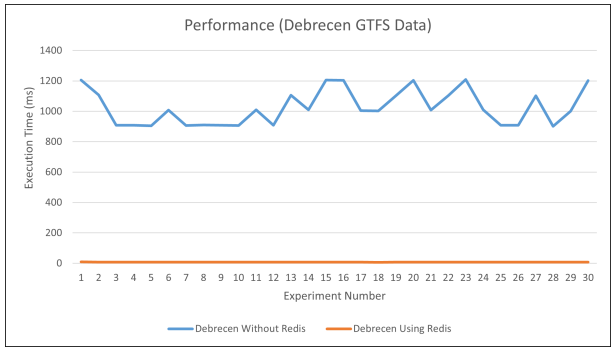

Figure 4.8: Debrecen data experiments result.

The experiments also show that the difference between GTFS data size for the cities (Debrecen 1483 KB and Budapest 42128 KB) affects the performance in the case of run-time algorithm use. This effect can be clear if we compare the average execution time with both cities' data, as shown in Figure 4.9. 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.

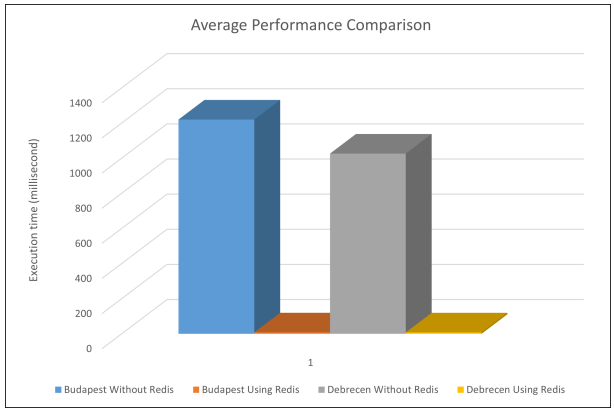

Figure 4.9: Budapest and Debrecen experiments result comparison.

4.5.3 Conclusions

In contemporary and Smart Cities, sharing transportation data is crucial for a successful transportation system. As a result, the necessity for a uniform format for communicating transportation data has grown. Transportation authorities extensively use GTFS (General Transit Feed Specification) across the globe as a standard format for sharing and publishing data. In addition, trip planning and computing transit accessibility are common topics between researchers and transit organizations as they can affect society's life and productivity. However, computing transit accessibility and finding a trip plan with timing information is complex and requires more computation than a standard computer can provide. Find a trip plan and transit accessibility consists of two steps. First, find all possible routes that can lead from the start to the destination, mark them as candidate solutions, and then validate them according to the user's time to start the trip(start time) and the trip timetable in the GTFS. The first part is done using the trip route planning algorithm, and the second part is accomplished using the trip timing algorithm. Also, they can be combined into one algorithm. This work uses our already published trip planning algorithm output as input to introduce a new trip timing algorithm. The trip planning algorithm output is a set of trip plans; each has one or more transitions. The trip timing algorithm in this chapter validates these transitions by searching the trips to find the trip with the closest departure time to the time T (where T is initially specified by the user to start the trip) on the specified route.

Some researchers try to improve the time complexity of trip planning and trip timing algorithms. We introduced the Range Mapping Hash RMH as a Redis model that provides fast access to the timing data and eliminates the need to run the trip timing algorithm as it does the same task with better performance.

The model contains two structures. The first structure can map any route ID, start-stop ID, and the time T; to the next going trip's ID, and the departure time at the start-stop. The idea behind this method is that T

can be any time during the day with sharp minutes part. Thus, we have 1440 possible values for T during the day. We use a Redis hash for the implementation. For each stop route combination, we create a hash with 1440 entries such that the key part will mention the stop ID and the route ID separated by the string "___", the hash fields will hold the T possible values, and the value field will hold the next trip_ID and its departure time. If T falls between two trips' departure times, then the answer (the value field) should be the trip with a later departure time. If T is earlier than the departure time of the first trip, then the answer will be the first trip and its departure time. If T is later than the departure time of the last trip in the GTFS data, then no entry will be stored in the hash, and a null value will be returned for such search, leading to rejecting the transition and the plan and in this case, we will have less than 1440 entries in the hash list. The second structure is a Redis hash with the key part holding the destination stop ID, the field part containing the trip ID that goes through that stop, and the value part containing the arrival time. Thus, both structures can form answers for any trip timing request. Both structures are Redis hash, and each can provide the response within O(1) complexity. Thus, the RMH can solve the timing problem with two read operations of O(1) complexity. Using GTFS data from Budapest and Debrecen, we tested the performance of RMH and the normal trip planning algorithm using the same computation hardware and software specifications. Experiments show that RHM can provide better complexity than the time validation algorithm. The experiments also show that RHM provides consistent time independent of data size (city size) in comparison to the run-time algorithm, where the performance is decreased when the data size is increased. In future work, the RMH can be applied to any similar problem where the input can be divided into sets or ranges with identical output. The RMH sacrifices the space to provide better performance.

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
7	1201.9	7.387	906.7	8.298
8	1106.3	7.938	909.4	6.897
9	1002.6	8.638	907.5	8.318
10	1402.3	8.727	906.3	8.278
11	1404.2	8.648	1009.8	8.67
12	1103.2	7.018	908.7	6.917
13	1107.1	7.449	1105.7	7.53
14	1303.4	7.418	1009.2	7.227
15	1324.1	8.027	1205.4	7.887
16	909.7	8.199	1204.2	8.509
17	1213	8.73	1004.5	7.488
18	908.5	8.319	1003.3	6.809
20	1308.7	7.679	1202.9	8.037
22	1408.4	6.829	1103.2	7.748
23	1203.8	6.839	1208.9	7.05
24	1304.8	8.307	1009.1	7.807
25	1303.5	8.678	907.7	6.909
26	1204.3	8.098	908.5	7.677
27	1305.1	7.099	1101.8	8.3
28	1001.1	7.697	901.8	8.437
29	1003.9	7.158	1001.2	8.047
30	1421.4	7.758	1202.4	6.849
Average	1219.89	7.985	1025.94	7.808

5

Application-Based Database Benchmarking

Dissertation key point and finding: *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.*

Contents

5.1 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. This is where a database benchmarking tool comes into play. 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 [65, 66, 67, 68]. However, the underlying data must be processed each time a user interacts with the user interface (filtered, aggregated, etc.) and provide fast responses and interactions for the UI user [69, 70, 71]. The database and visualization communities have created several approaches, such as approximate query processing [72, 73], web-based progressive [74, 75, 76], speculative query execution [77, 78], data cubes [77, 79, 80], spatial indexing [81], and lineage tracking [82], 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 cross filter [83, 84, 85], 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) [86], TPC-DS Benchmark [87], and TPC-H [88] 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 [89] or Spotfire [84]. Some research benchmark database systems using interactive workload [90, 91, 92, 93, 94] while others use tools like Yahoo Cloud Service Benchmarking tool (YCSB) [1, 2, 95] and HammerDB [96]. 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 [97] 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 dissertation, 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. Such an operation or feature may involve many sub-queries, querying more than one table or data structure, like the stops, routes, and stopstimes tables to retrieve all the data required to find the trip plans. The response time to retrieve the data needed to complete one operation varies depending on the database structure (or model) used to store the data. Thus, the traditional database benchmarking tools like YCSB [98] will not reflect the actual database performance for that application and design, as it performs arbitrary read and write operations 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 List, 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 that tool to evaluate the performance of these two databases in different cases of stress (concurrent user access) and compare both databases' results. The findings of this research are published in [99].

5.2 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.

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 [8]:

- 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 step 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:

$$TableName\_PrimeryKey\_ForeignKey1\_ForeignKey2\_....\_ForeignKeyN$$

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

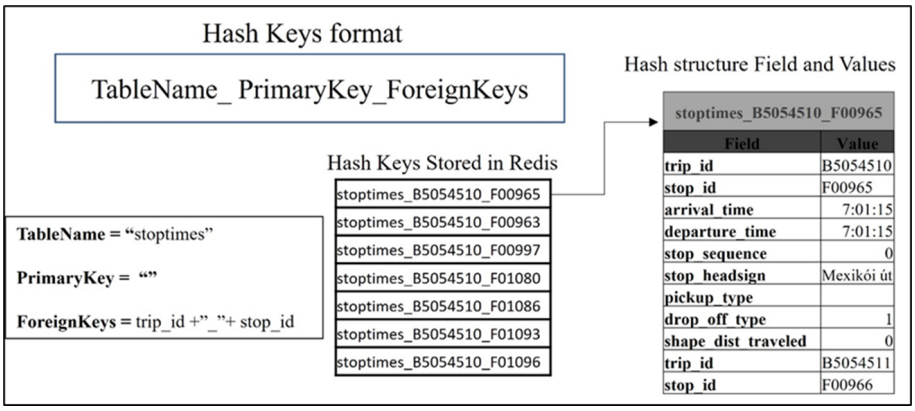

Figure 5.1: Stoptimes file entity stored in Redis

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 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, Below 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. The primary key for each GTFS table was used as an index in our MongoDB model.

{

"_id": ObjectId("617912eb39eaf2a2a8d20260"),

"stop_id": "1000",

"stop_name": "Grand Central Terminal",

"stop_lat": 40.752726,

"stop_lon": -73.977229

}

5.3 Our benchmarking tool

We develop our benchmarking tool with Java using Gradle version 7.2. The main classes in the project are shown in Figure 5.2 This kind of benchmarking requires more investment, and it is suitable only for a given application and is not as flexible as universal tools. 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.

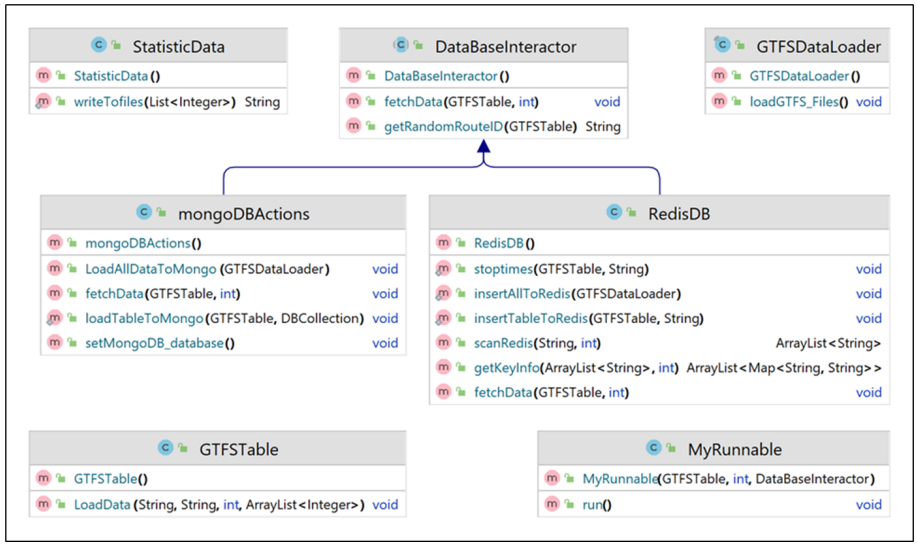

Figure 5.2: UML design for main classes in the benchmark tool

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 database hits 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.

5.4 Benchmarking result

5.4.1 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 ten 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.

5.4.2 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: Test results for 20, 60, and 110 threads run

Hits 20 threads	Hits 60 threads	Hits 110 threads	Hits 20 threads	Hits 60 threads	Hits 110 threads
8,997	375	1,078	28,797,833	7,225,407	3,210,808
13,086	939	499	27,675,503	7,278,314	3,237,905
11,116	2,790	2,865	27,929,125	7,285,905	3,198,596
8,262	327	970	18,741,209	7,189,834	3,165,443
14,519	1,289	1,258	29,333,027	7,426,133	3,154,295
22,247	169	0	27,305,631	7,151,923	3,169,713
10,360	645	1,374	27,938,556	7,289,945	3,355,994
9,716	349	489	17,456,620	7,251,756	3,221,892
6,566	3,067	0	29,369,999	7,051,873	3,115,420
11,463	8,213	9,146	27,462,338	7,150,764	3,223,875
13,604	845	3,764	27,738,747	7,056,592	3,099,929

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's and Redis's performance decreased with the increasing 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 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 5.2: Experiments summary for Redis

No Of Threads	No of DB Hits	No of Complete Operation	DB Hits per/Sec	Complete Operation per/Sec
10	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

Table 5.3: Experiments summary for MongoDB

No Of Threads	No of DB Hits	No of Complete Operation	DB Hits per/Sec	Complete Operation per/Sec
10	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 5.3 and Figure 5.4 showing 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.

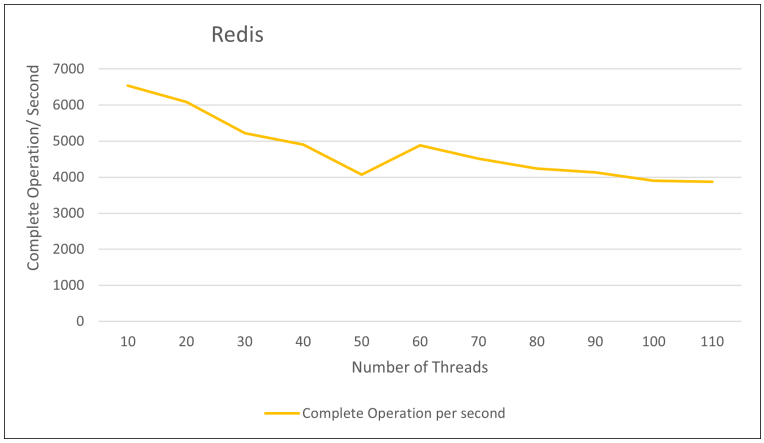

Figure 5.3: Relation between throughput and number of used threads (Redis)

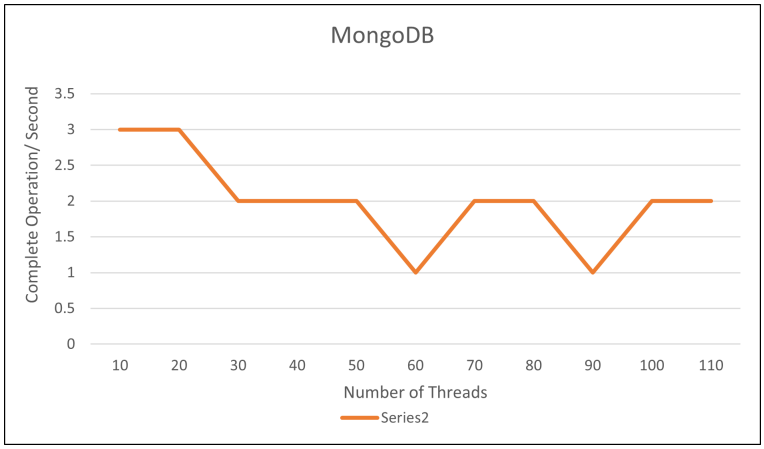

Figure 5.4: Relation between throughput and number of used threads (MongoDB)

It is important to highlight here that the number of the required queries (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.

5.5 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.

6

Summary

The work of this dissertation is concluded and summarized in this chapter.

Data is certainly one of the most significant assets in today's digital age, as it drives decisions, ideas, and business strategies across many different industries. There has been a change in how data is stored, retrieved, and interacted with in the midst of this data-driven revolution. Structured query language (SQL)-based relational databases have been the backbone of the data storage industry for decades. However, the demand for more adaptable, scalable, and reliable data management systems arose in response to the exponential growth of data (in volume and diversity) and the rising complexities of modern applications. Here comes NoSQL storage. NoSQL (short for "not only SQL") databases, which represent a departure from the strictures of relational models, have proven crucial in handling the large and varied nature of modern data, such as social network posts, user-generated content, real-time logs, and sensor-generated data. These databases were developed with horizontal scaling in mind, making it possible to continue normal business activities even as data volumes increase. They are well-suited for dealing with vast amounts of constantly changing, unstructured, or semi-structured data due to their schema-less structure, which allows for more dynamic and diversified data modeling. As distributed data storage, effective caching, and real-time analytics become increasingly important for cloud and web applications, the value of NoSQL databases has exploded. They signal a new era of data management and optimization and are essential components of the modern developer's toolkit. The dissertation focuses on two aspects of NoSQL databases: benchmarking and utilizing them 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 introducing a new 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. As some research spots the light on time validation performance enhancement. We propose a trip timing validation algorithm, and in terms of the NoSQL utilization approach, we introduce 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 MongoDB.

As final conclusion, this dissertation 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.

7

Acknowledgements

Completing this dissertation has been a journey marked by the guidance, encouragement, and support of many. First and foremost, my deepest gratitude goes to my supervisor, Dr. Aniko Vagner. Her vast knowledge, meticulousness, and unwavering support have been invaluable from the inception of my research topic, "Benchmarking and Utilization of NoSQL Databases," to its culmination. Dr. Vagner's accessibility, insightful feedback, and commitment to excellence inspired me to meet challenges with determination and rigor.

A special place in my heart is reserved for my Mom and Dad. Their unwavering love, constant support, and endless encouragement have been my strength and motivation throughout this journey. Their lessons and values have shaped me, and their faith in me has been a guiding light, pushing me to strive for excellence.

My heartfelt appreciation goes to the University of Technology, Iraq, for laying the educational foundation during my bachelor's degree that made this advanced research possible.

The financial assistance from Stipendium Hungaricum deserves special mention. Their generosity greatly eased the financial burdens of my academic journey, allowing me to focus on my research.

Mustafa Alzaidi

Debrecen, Hungary, 2024

Reference

[1] Tiroshan Madushanka, Laksheen Mendis, Dananji Liyanage, and Chamath Kumarasinghe. Performance comparison of nosql databases in pseudo distributed mode: Cassandra, mongodb redis. 2 2015.

[2] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. pages 143–154, 2 2010.

[3] Chandranil Chakraborttii. Performance evaluation of nosql systems using yahoo cloud serving benchmarking tool. 2 2015.

[4] Ahmed Oussous, Fatima-Zahra Benjelloun, Ayoub Ait Lahcen, and Samir Belfkih. Comparison and classification of nosql databases for big data. 2 2015.

[5] Hamzeh Khazaei, Marios Fokaefs, Saeed Zareian, Nasim Beigi, Brian Ramprasad, Mark Shtern, Purwa Gaikwad, and Marin Litoiu. How do i choose the right nosql solution? a comprehensive theoretical and experimental survey. *Journal of Big Data and Information Analytics (BDIA)*, 2, 10 2015.

[6] Eric Anderson, Xiaozhou Li, Mehul Shah, Joseph Tucek, and Jay Wylie. What consistency does your key-value store actually provide? *HP Laboratories Technical Report*, 2 2010.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. *Sigmetrics Performance Evaluation Review - SIGMETRICS*, 40, 2 2012.

[8] Mustafa Alzaidi Aniko Vagner. Trip planning algorithm for gtfs data with nosql structure to improve the performance. *Journal of Theoretical and Applied Information Technology*, Vol.99. No:2290–2300, 5 2021.

[9] Kun Ma and Ajith Abraham. Toward lightweight transparent data middleware in support of document stores. pages 253–257, 2013.

[10] Craig Chasseur, Yinan Li, and Jignesh M Patel. Enabling json document stores in relational systems. In *WebDB*, volume 13, pages 14–15, 2013.

[11] Rick Cattell. Scalable sql and nosql data stores. *SIGMOD Record*, 39:12–27, 2 2010.

[12] Daniel Abadi. Column stores for wide and sparse data. pages 292–297, 2 2007.

[13] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. volume 2, pages 505–516, 2 2005.

[14] Mustafa Alzaidi and Aniko Vagner. Benchmarking redis and hbase nosql databases using yahoo cloud service benchmarking tool. In *Annales Mathematicae et Informaticae*, volume 56, pages 1–9, 2022.

[15] L George. *HBase: The Definitive Guide: Random Access to Your Planet-Size Data*. O'Reilly Media, Inc.: Sebastopol, CA, USA, 1st edition, 2011.

[16] Mehul Nalin Vora. Hadoop-hbase for large-scale data. *Proceedings of 2011 International Conference on Computer Science and Network Technology*, 1:601–605, 2011.

[17] Yahoo could service benchmarking tool (ycsb), https://github.com/brianfrankcooper/ycsb.

[18] Enqing Tang and Yushun Fan. Performance comparison between five nosql databases. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pages 105–109. IEEE, 2016.

[19] James Wong. Leveraging the general transit feed specification for efficient transit analysis. *Transportation Research Record: Journal of the Transportation Research Board*, 2338:11–19, 12 2013.

[20] E W Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 9.

[21] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16:87–90, 1958. 1.

[22] João Carlos Namorado Climaco and Ernesto Queirós Vieira Martins. A bicriterion shortest path algorithm. *European Journal of Operational Research*, 11:399–404, 1982. 3.

[23] Robert W Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5:345, 1962. 12.

[24] Donald B Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24:1–13, 1 1977. 17.

[25] John Mote, Ishwar Murthy, and David L Olson. A parametric approach to solving bicriterion shortest path problems. *European Journal of Operational Research*, 53:81–92, 1991. 26.

[26] Pierre Hansen. Bicriterion path problems bt - multiple criteria decision making theory and application. pages 109–127. Springer Berlin Heidelberg, 1980. 15.

[27] Ernesto Queirós Vieira Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16:236–245, 1984. 24.

[28] Chi Tung Tung and Kim Lin Chew. A multicriteria pareto-optimal path algorithm. *European Journal of Operational Research*, 62:203–209, 1992. 33.

[29] J Brumbaugh-Smith and D Shier. An empirical investigation of some bicriterion shortest path algorithms. *European Journal of Operational Research*, 43:216–224, 1989. 2.

[30] H W Corley and I D Moon. Shortest paths in networks with vector weights. *J. Optim. Theory Appl.*, 46:79–86, 5 1985. 6.

[31] H G Daellenbach and C A De Kluyver. Note on multiple objective dynamic programming. *Journal of the Operational Research Society*, 31:591–594, 7 1980. 7.

[32] A J V Skriver and Kim Andersen. A label correcting approach for solving bicriterion shortest-path problems. *Computers Operations Research*, 27:507–524, 5 2000. 31.

[33] Paolo Dell'Olmo, Monica Gentili, and Andrea Scozzari. On finding dissimilar pareto-optimal paths. *European Journal of Operational Research*, 162:70–82, 4 2005. 8.

[34] Enrique Machuca, L Mandow, and J Cruz. An evaluation of heuristic functions for bicriterion shortest path problems. *New Trends in Artificial Intelligence. Proceedings of EPIA'09*, 1 2009. 20.

[35] L Mandow and J L de la Cruz. Frontier search for bicriterion shortest path problems. page 480–484. IOS Press, 2008. 21.

[36] Andrea Raith and Matthias Ehrgott. A comparison of solution strategies for biobjective shortest path problems. *Computers Operations Research*, 36:1299–1331, 4 2009. 28.

[37] Jacek Widuch. A label correcting algorithm for the bus routing problem. *Fundamenta Informaticae*, 118:305–326, 8 2012.

[38] Chao-Lin Liu, Tun-Wen Pai, Chun-Tien Chang, and Chang-Ming Hsieh. Path-planning algorithms for public transportation systems. pages 1061–1066, 2001. 100.

[39] Debrecen regional transport association, http://www.derke.hu.

[40] Bkk gtfs - openmobilitydata, https://transitfeeds.com/p/bkk/42?fbclid=iwar1vi6 3txibkxtlggfptjw4li5sdxecuvflk7xsgq45rauwr0qsvkwlgrdo.

[41] Raul Velasquez, Francisco Rodriguez, Miguel Vargas Martin, and Julio Ponce. *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*, pages 177–185. 1 2020.

[42] Google developers/google transit, https://developers.google.com/transit/gtfs/reference.

[43] Quentin Zervaas. *The Denitive Guide toGTFS-realtime (How to consume and produce real-time public transportation data with the GTFS-rt specication)*. 2014.

[44] General transit feed specification, https://gtfs.org/reference/static.

[45] Andreza Queiroz, Veruska Santos, Dimas Nascimento, and Carlos Santos Pires. *Conformity Analysis of GTFS Routes and Bus Trajectories*. 11 2019.

[46] C Carl Robusto. The cosine-haversine formula. *The American Mathematical Monthly*, 64:38–40, 1957.

[47] Lucy Dubrelle Gunn, Tania L King, Suzanne Mavoa, Karen E Lamb, Billie Giles-Corti, and Anne Kavanagh. Identifying destination distances that support walking trips in local neighborhoods. *Journal of transport & health*, 5:133–141, 2017.

[48] Dictionary class, microsoft docs, https://docs.microsoft.com/en us/dotnet/api/system.collections.generic.dictionary-2?view=net-5.0.

[49] S S., Xiaoyue Cathy Liu, and Guohui Zhang. An efficient general transit feed specification (gtfs) enabled algorithm for dynamic transit accessibility analysis. *PLOS ONE*, 12:e0185333, 10 2017.

[50] Todd Litman. Exploring the paradigm shifts needed to reconcile transportation and sustainability objectives. *Transportation Research Record*, 1670:8–12, 1 1999. doi: 10.3141/1670-02.

[51] James F Sallis, Lawrence D Frank, Brian E Saelens, and M.Katherine Kraft. Active transportation and physical activity: opportunities for collaboration on transportation and public health research. *Transportation Research Part A: Policy and Practice*, 38:249–268, 2004.

[52] Tya Shannon, Billie Giles-Corti, Terri Pikora, Max Bulsara, Trevor Shilton, and Fiona Bull. Active commuting in a university setting: Assessing commuting habits and potential for modal change. *Transport Policy*, 13:240–253, 2006.

[53] Aaron Golub and Karel Martens. Using principles of justice to assess the modal equity of regional transportation plans. *Journal of Transport Geography*, 41:10–20, 2014.

[54] Karel Martens, Aaron Golub, and Glenn Robinson. A justice-theoretic approach to the distribution of transportation benefits: Implications for transportation planning practice in the united states. *Transportation Research Part A: Policy and Practice*, 46:684–695, 2012.

[55] Kathryn Coffel, Jamie Parks, Conor Semler, Paul Ryus, David J Sampson, Carol Kachadoorian, Herbert S Levinson, and Joseph L Schofer. Guidelines for providing access to public transportation stations. 2012.

[56] Steven Farber, Benjamin Ritter, and Liwei Fu. Space–time mismatch between transit service and observed travel patterns in the wasatch front, utah: A social equity perspective. *Travel Behaviour and Society*, 4:40–48, 2016.

[57] S Kiavash Fayyaz S., Xiaoyue Cathy Liu, and Guohui Zhang. An efficient general transit feed specification (gtfs) enabled algorithm for dynamic transit accessibility analysis. *PLOS ONE*, 12:e0185333–, 10 2017.

[58] Peyman Goli and MOLLAEI MOHAMAD REZA KARAMI. Speech intelligibility improvement in noisy environments for near-end listening enhancement. 2016.

[59] Sara Motamed, A Broumandnia, and Azamossadat Nourbakhsh. Multimodal biometric recognition using particle swarm optimization-based selected features. *Journal of Information Systems and Telecommunication*, 1:79–87, 3 2013.

[60] Ghazi Maghrebi Saeed, Haji Bagher Naeeni Babak, and Lotfizad Mojtaba. Achieving better performance of s-mma algorithm in the ofdm modulation. 2013.

[61] Trip timing algorithm for gtfs data with redis model to improve the performance.

[62] Nur Amirah, Diana Mohamad, and Ahmad Hilmy. *Acceptable walking distance accessible to the nearest bus stop considering the service coverage.* 7 2021.

[63] Redis, https://redis.io/clientsc-sharp.

[64] Santiago Tapia-Fernández, Daniel García-García, and Pablo García-Hernandez. Key concepts, weakness and benchmark on hash table data structures. *Algorithms*, 15, 2022.

[65] Sara; Zokaei Alspaugh Nava; Liu Andrea; Jin Cindy; Hearst Marti A. Futzing and moseying: Interviews with professional data analysts on exploration practices. *IEEE transactions on visualization and computer graphics*, 25:22–31, 2018.

[66] Leilani; Heer Battle Jeffrey. Characterizing exploratory visual analysis: A literature review and evaluation of analytic provenance in tableau. *Computer Graphics Forum*, 38:145–159, 2019.

[67] Stuart K.; Mackinlay Card Jock D.; Shneiderman Ben. *Readings in Information Visualization: Using Vision to Think*, volume NA. 1999.

[68] F N.; Tukey David John W. Exploratory data analysis. *Biometrics*, 33:768–NA, 1977.

[69] Jakob Nielsen. Response times: The 3 important limits. *https://www.nngroup.com/articles/response-times-3-important-limits/*.

[70] Robert B Miller. Response time in man-computer conversational transactions. pages 267–277. Association for Computing Machinery, 1968.

[71] Ben Shneiderman. Response time and display rate in human performance with computers. *ACM Computing Surveys*, 16:265–285, 1984.

[72] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. pages 29–42, 2013.

[73] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. Approximate query processing. pages 511–519. ACM, 5 2017.

[74] Susanne Albers. Online algorithms: a survey. *Mathematical Programming*, 97:3–26, 2003.

[75] *Jean-Daniel; Fisher Fekete Danyel; Nandi Arnab; Sedlmair Michael. Progressive Data Analysis and Visualization (Dagstuhl Seminar 18411), volume 8.*

[76] Joseph M.; Haas Hellerstein Peter J.; Wang Helen J. Online aggregation. *ACM SIGMOD Record*, 26:171–182, 1997.

[77] Leilani Battle, Remco Chang, and Michael Stonebraker. Dynamic prefetching of data tiles for interactive visualization. pages 1363–1375. ACM, 6 2016.

[78] Niranjan Kamat, Prasanth Jayachandran, Karthik Tunga, and Arnab Nandi. Distributed and interactive cube exploration. *2014 IEEE 30th International Conference on Data Engineering*, pages 472–483, 2014.

[79] Lauro; Klosowski Lins James T.; Scheidegger Carlos. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE transactions on visualization and computer graphics*, 19:2456–2465, 2013.

[80] Zhicheng; Jiang Liu Biye; Heer Jeffrey. immens : real-time visual querying of big data. *Computer Graphics Forum*, 32:421–430, 2013.

[81] Wenbo; Liu Tao Xiaoyu; Wang Yedi; Battle Leilani; Demiralp Çağatay; Chang Remco; Stonebraker Michael. Kyrix: Interactive pan/zoom visualizations at scale. *Computer Graphics Forum*, 38:529–540, 2019.

[82] Fotis Psallidas and Eugene Wu. Provenance for interactive visualizations. pages 1–8. ACM, 6 2018.

[83] Dominik Moritz, Bill Howe, and Jeffrey Heer. Falcon. pages 1–11. ACM, 5 2019.

[84] Spotfire 1995. Tico spotfire., 7 2019.

[85] Lisa Tweedie, Bob Spence, David Williams, and Ravinder Bhogal. The attribute explorer. pages 435–436. ACM Press, 1994.

[86] Patrick; O'Neil O'Neil Elizabeth; Chen Xuedong; Revilak Stephen. Tpctc - the star schema benchmark and augmented fact table indexing. *Lecture Notes in Computer Science*, 5895:237–252, 2009.

[87] TPC-DS 2016. Tpc-ds, http://www.tpc.org/tpcds/, 11 2017.

[88] TPC-H 2016. Tpc-h, http://www.tpc.org/tpch/, 11 2017.

[89] Chris; Tang Stolte Diane L.; Hanrahan Pat. Polaris: a system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8:52–65, 2002.

[90] Leilani; Chang Battle Remco; Heer Jeffrey; Stonebraker Michael. *DSIA - Position statement: The case for a visualization performance benchmark*, volume NA. 2017.

[91] Philipp Eichmann, Emanuel Zgraggen, Zheguang Zhao, Carsten Binnig, and Tim Kraska. Towards a benchmark for interactive data exploration. *IEEE Data Eng. Bull.*, 39:50–61, 2016.

[92] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of data exploration techniques. pages 277–281, 2015.

[93] Lilong Jiang, Protiva Rahman, and Arnab Nandi. Evaluating interactive data systems: Workloads, metrics, and guidelines. pages 1637–1644, 2018.

[94] Nan Tang, Eugene Wu, and Guoliang Li. Towards democratizing relational data visualization. pages 2025–2030, 2019.

[95] Casper Claesen, Ansar Rafique, Dimitri Van Landuyt, and Wouter Joosen. A ycsb workload for benchmarking hotspot object behaviour in nosql databases. pages 1–16, 2021.

[96] Yong Chen, Xuanzhong Xie, and Jiayin Wu. *A Benchmark Evaluation of Enterprise Cloud Infrastructure*, pages 832–840. 2015.

[97] Leilani Battle, Philipp Eichmann, Marco Angelini, Tiziana Catarci, Giuseppe Santucci, Yukun Zheng, Carsten Binnig, Jean Daniel Fekete, and Dominik Moritz. Database benchmarking for supporting real-time interactive querying of large data. pages 1571–1587. Association for Computing Machinery, 6 2020.

[98] Omoruyi Ose, Kennedy Okokpujie, Nsikan Nkordeh, Charles Ndujiuba, Samuel John, and Idiake Uzairue. Performance benchmarking of key-value store nosql databases. *International Journal of Electrical and Computer Engineering (IJECE)*, 8:5333, 12 2018.

[99] Mustafa Alzaidi and Aniko Vagner. Application-based benchmarking on redis and mongodb for trip planning using gtfs data. *TEM Journal*, 12(4):2583, 2023.

My 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.

Software developed during the PhD:

- *Trip planning for GTFS data using C*
- *Benchmarking tool using Java and Mavin*