

Database Benchmarking Suite for Survival Analysis Data

A Project

Presented to

The Faculty of the Department of Computer Science San  
José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Aarsh Patel

Dec 2023

©2023

Aarsh Patel

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Database Benchmarking Suite for Survival Analysis Data

by

Aarsh Patel

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

December 2023

Dr. Chris Pollett                      Department of Computer Science

Dr. Robert Chun                      Department of Computer Science

Dr. William Andreopoulos    Department of Computer Science

## **ABSTRACT**

### **Database Benchmarking Suite for Survival Analysis Data**

by Aarsh Patel

Survival analysis data is crucial for predicting future events and making informed decisions. Storing this data in databases enables researchers and analysts to easily access and analyze it, facilitating more accurate predictions and better decision-making. There is a growing demand to store such data utilizing databases. While benchmarking tools are available to aid in selecting the appropriate database, there is currently no benchmarking suite designed explicitly for survival analysis data. In this report, I present the development and analysis of a benchmarking suite for survival analysis data. The suite encompasses performance metrics for both read and write operations and has been applied to several popular databases, including QuestDB, TimescaleDB, Cassandra, and MongoDB. Specialized topics related to survival analysis, such as Log-Rank, Cox Proportional Hazards, and Kaplan-Meier, were given significant attention. Using the suite, I compared NoSQL databases with time-series databases for storing and retrieving survival analysis data. The project's findings reveal differences as NoSQL databases don't perform as well as time series databases. Although NoSQL databases are generally useful, certain survival analysis queries are unresponsive. TimescaleDB performs exceptionally well across various queries, indicating its suitability for time-dependent data scenarios. The comparative analysis highlights the importance of selecting databases tailored to the specific data needs of survival analysis. It recognizes that specialized time-series databases have an advantage in this area.

**Keywords:** Survival analysis, Time-Series Data, Benchmarking Suite, Time-Series Databases, NoSQL Databases

## **ACKNOWLEDGEMENT**

I would like to thank Dr. Christopher Pollett, my project advisor, for his unwavering advice and assistance over the last year as I worked on this master's research. I owe him a great deal for the abilities and information I acquired from working with him. Additionally, I would like to thank the other faculty members in the San José State University Department of Computer Science for their assistance and efforts in teaching the advanced computer science courses. Finally, I want to express my gratitude to my friends and family for their unwavering support and encouragement as I worked toward earning my Master of Science in Computer Science.

## Contents

<b>ABSTRACT.....</b>	<b>2</b>
<b>Introduction.....</b>	<b>8</b>
<b>I. Background.....</b>	<b>11</b>
<b>II. Design and Architecture.....</b>	<b>13</b>
2.1 Design of Suite .....	13
2.2 Databases.....	14
2.2.1 Time Series Databases.....	14
2.2.2 NoSQL Databases .....	15
2.3 Metrics .....	16
2.3.1 Write Performance .....	16
2.3.2 Read Performance .....	16
<b>III. Implementation .....</b>	<b>21</b>
3.1 Basic Requirements.....	21
3.2 Database Setup .....	21
3.3 Data Generation:.....	22
3.4 Data Loading .....	23
3.5 Query Execution.....	26
<b>IV. Experiments .....</b>	<b>33</b>
<b>Conclusion.....</b>	<b>46</b>

<b>REFERENCES .....</b>	<b>48</b>
-------------------------	-----------

## **Introduction**

Survival analysis data is essential because it helps us understand the time it takes for an event of interest, such as a system failure or a disease's occurrence. This type of data is commonly used in medical research, engineering, and other fields to make predictions and to inform decision-making. By storing survival analysis data in databases, researchers and analysts can easily access and analyze the data, making more accurate predictions and ultimately making better decisions based on the data. To store such essential data, having knowledge and understanding of databases is necessary, and choosing a suitable database is crucial. However, selecting an appropriate database can be challenging, especially regarding survival analysis data. Companies often use benchmarking suites to make informed decisions. Although many benchmarking suites are available, none focus on evaluating survival analysis data. In my previous project, I used TSBS (Time Series Benchmarking Suite) to compare various time-series and NoSQL databases against time-series data. This research led to the idea of creating a benchmarking suite for survival analysis data. The suite includes write and read workloads, with various statistical queries useful for survival analysis in the read performance section. Another part of the project was to compare NoSQL and time-series databases for survival analysis data, so I used two time-series and two NoSQL databases for the project. This Introduction section provides background on time series and survival analysis, benchmarking suites, and their importance. It also provides a summary of the various sections of the report.

Time-series data refers to a sequence of information collected at fixed intervals. Such data usually consists of one or more values associated with a specific timestamp [1]. Examples of time-series data include temperature readings, sensor data, market prices, and medical records. While time-series data can be used in forecasting and decision-making, a specific type known as survival analysis data is widely employed in medical research. Survival analysis is a statistical method used to analyze time-to-



event data. Survival analysis points out how long it will take for an event to happen. It helps to evaluate durations and forecast future events [2], taking censoring into account, especially in cases where the event is yet to occur or is unknown.

Although the benefits and significance of time-series data are well known, storing such data in large quantities is necessary for analyzing and carrying out calculations to support various use cases. As a result, efficient data storage and management are crucial for making fact-based decisions. Traditional relational databases may not be the best option for handling this type of time-based data, leading to the development of customized databases and systems for specific use cases. Unique databases such as TimescaleDB and QuestDB have been created to store time-series data, which can also be used to store survival analysis data. Additionally, NoSQL databases like MongoDB offer more general-purpose storage, which can also be used for storing survival analysis data.

Choosing the appropriate database for survival analysis data is crucial to ensure optimal performance and scalability. The selection of the wrong database can significantly impact data analysis and application performance. Benchmarking is an effective way to assess the performance of various databases and help enterprises make informed decisions about their data architecture. To carry out benchmarking, a benchmarking suite is used, which usually includes scripts to run and metrics like data loading and query performance that can assess the performance of databases. My project focuses on developing a benchmarking suite that generates synthetic survival analysis datasets, loads them into databases, and performs various aggregation and survival analysis queries. The time it takes to load the data and execute the queries will serve as a basis for assessing the performance of databases. In this project, I will compare the performance of two time-series databases (TimescaleDB and QuestDB) and two NoSQL databases (MongoDB and Cassandra) for storing and querying survival analysis data. This will help me analyze how NoSQL databases perform against specialized time-series databases. By

exploring the write performance during data loading and analyzing the read performance with various aggregation and statistical queries, I aim to understand the performance of databases better.

In the next section of my report, I will provide some background information on benchmarking and share the results of the benchmarking study I did for my CS297. I will then move on to the Design section, explaining how I designed the suite and presenting the dataset I used. In the Architecture section, I will describe how I created scripts for various functions, programming languages, tools, and the queries I employed for benchmarking. I will also focus on statistical queries typically used in survival analysis to make informed decisions regarding the probability of survival. In my experiment and results section, I will present my findings, including screenshots of the queries I performed and a table showing the execution time of all my results. The final part of my report will be the conclusion, including a general analysis of my findings and a comparison of the performance of time-series and NoSQL databases when storing survival analysis data.

## **I. Background**

This section introduces benchmarking and benchmarking suites and their importance. I will discuss current benchmarking suites and the typical metrics used in database benchmarking. This section will also include the results from my preliminary work for CS297, where I used TSBS (Time Series Benchmarking Suite) to evaluate the performance of various databases. These things together will explain my goal and idea of my project to develop a benchmarking suite for survival analysis data and the metrics I used for my benchmarking.

Measuring a system's or component's ability to complete a task under specific conditions is called benchmarking. It involves evaluating the performance of a system and comparing it with other systems to find areas for improvement. This information can then be used to make decisions to enhance overall performance. A benchmarking suite consisting of tasks or tests is developed and used to evaluate the system's performance. For database systems, benchmarking includes a series of metrics and queries to evaluate the performance of different databases. The results can help select the appropriate database for a specific use case and workload. Response time, throughput, and resource management are some metrics that can be used for benchmarking.

Several database benchmarking suites are available today, each designed for specific data types or databases to meet particular needs. Some of the standard benchmarking suites include HammerDB, YCSB (Yahoo Cloud Serving Benchmark), TPCC, and TSBS (Time Series Benchmarking Suite). HammerDB is a popular benchmarking tool that supports both relational and non-relational databases. YCSB is specifically designed for measuring the performance of distributed key-value store databases. TPCC is a standard benchmark for

OLTP (Online Transaction Processing) systems and supports relational databases. Lastly, TSBS is a benchmarking suite designed for time-series data and supports various time-series and NoSQL databases.

Before developing this suite, I worked on benchmarking various databases for time-series data using TSBS. I benchmarked three time-series databases (TimeScaleDB, InfluxDB, and QuestDB) and MongoDB as a NoSQL database. My main research interest was time-series data and different databases, so I used TSBS to evaluate the performance of these databases. I benchmarked the write performance (data loading time) and four aggregation queries for read performance. The results showed that specialized time-series databases performed better than MongoDB, with InfluxDB being the best overall performer.

Through benchmarking various databases for time-series data, I realized the need for a specialized suite designed for survival analysis data. As no such benchmarking suite existed, I developed one focused on survival analysis use cases. I wanted to use similar metrics to those used for time-series data and provide support for statistical queries to measure read performance. To achieve this, I compared two time series and two NoSQL databases against two survival analysis datasets of different sizes. I conducted write and read time measurements for six queries and evaluated the results.






In this report, I will explain the design and implementation of my benchmarking suite. I will cover the dataset, the databases, languages, frameworks, and tools used for the implementation and include code snippets to help readers understand and use the suite. In the experiments section, I will provide screenshots of my results and conclude my findings.

## II. Design and Architecture

In this section, I will describe the architecture design of my suite, the databases I chose, and the metrics my suite supports in detail. This section is divided into design, databases, and metrics.

### 2.1 Design of Suite

I have ensured that my suite implementation is user-friendly and easy to understand. The root folder of my suite consists of several folders, including those for data generation, loading, query execution, docker configuration, and readme files. The docker configuration file contains details about different databases. The comprehensive readme file provides step-by-step instructions and example scripts to help users carry out the benchmarking process from scratch.

 data_generation	11/9/2023 3:43 PM	File folder	
 data_load	10/30/2023 10:17 PM	File folder	
 query_execution	11/3/2023 2:09 PM	File folder	
 docker-compose.yml	10/15/2023 8:25 PM	Yaml Source File	1 KB
 readme.txt	10/30/2023 10:37 PM	Text Document	2 KB

**Fig.1: Root Folder of Benchmarking Suite**

In the data generation folder, you'll find scripts for generating synthetic survival analysis data taken from Kaggle. The link to the dataset is as follows:

<https://www.kaggle.com/datasets/louise2001/survival-analysis-synthetic-data/>

This dataset contains information about fictional clients of a life insurance company, including their entry and departure dates, age at entry and exit, and the reason for their departure (either death or withdrawal). Additionally, the dataset contains columns for the start and end date of the insurance. I have changed the dataset's code to allow users to pass the number of rows required as a parameter in the data generation script. This alteration makes it easier for users to generate datasets of different sizes as

per their requirements. Once generated, the code produces a CSV file with the specified rows. In addition to the data generation scripts, the folder contains "addColumn" and "tojson" files used for Cassandra and MongoDB, respectively. A data load folder with data loading scripts for each database is also included, making it easy to load the generated data. Finally, you'll find a query execution folder that contains different subfolders with query scripts. Each subfolder has scripts to perform various queries, and a readme file briefly explains each query so users can easily understand and modify them as needed.

## **2.2 Databases**

My suite currently supports four databases that I used for this project. They can be classified as Time Series and NoSQL databases.

### **2.2.1 Time Series Databases**

The purpose of a time-series database is to handle and manage time-series data effectively. Data points referenced in time order and usually collected regularly make up time-series data.[3] Time-series databases are designed to execute queries and analyses on data with a temporal component. They frequently offer functions including effective storage, rapid write throughput, and customized query capabilities. The two time series databases I used for the project are TimescaleDB and QuestDB.

#### **TimescaleDB:**

An open-source time-series database called TimescaleDB was constructed as a PostgreSQL plugin, so it supports SQL. It blends specific time-series functionality with the advantages of a relational database.

TimescaleDB has many features, making it favorable for storing time-series data.

As per TimescaleDB docs [4], the following are the key features of TimescaleDB:

- **Hyper tables:** Hyper tables are optimized for storing and querying large amounts of time-series data. They can store trillions of rows of data and be queried quickly and efficiently.
- **Chunking:** TimescaleDB chunks data into small pieces, making it query faster.
- **Compression:** TimescaleDB compresses data, reducing the required storage space.
- **Rollups:** TimescaleDB can roll up data into aggregates, making summarizing and analyzing large datasets easier.

**QuestDB:**

QuestDB is a lightweight, high-performance time-series database. It is designed to be easy to use and scale large volumes of data. QuestDB is written in Java and can run on any platform that supports Java.

QuestDB docs [5] explain the primary features of QuestDB, which are as follows:

- High performance: QuestDB can process millions of events per second.
- Scalability: QuestDB can scale to handle large volumes of data.
- Durability: QuestDB is designed to be durable and reliable, even during a power failure.
- Ease of use: QuestDB is easy to use and manage.

**2.2.2 NoSQL Databases**

NoSQL databases, often known as "Not Only SQL" databases, are database systems that offer an alternative to conventional relational databases [6] for storing and retrieving data. Large volumes of unstructured or semi-structured data can be handled using NoSQL databases, providing greater scalability and flexibility regarding data models. There are four types of NoSQL databases: document, key-value, column, and graph, which are explained in [6][7]. First is the document type, which contains information in semi-structured formats like JSON or BSON. CouchDB and MongoDB are two examples. Second is a key-value store where data is kept as key-value pairs, where each key corresponds to a distinct value. Redis and DynamoDB are two such examples. The third type is column-based, which stores data in columns instead of rows. HBase and Apache Cassandra are two examples. The last type is graph databases, which are beneficial for applications such as fraud detection and social networks where data can be represented as graphs. Amazon Neptune and Neo4j are two examples. The two NoSQL databases I implemented for this project are MongoDB and Cassandra.

**MongoDB:**

MongoDB is a document-oriented NoSQL database that stores documents as JSON objects. As per MongoDB Docs. [8], key characteristics of MongoDB are:

- Flexible schema: Since MongoDB does not need a schema, adding and removing fields from documents is simple.
- Horizontal scalability: Adding extra servers allows MongoDB to be expanded horizontally.
- High performance: The performance of MongoDB is well-known.

## **Cassandra:**

Cassandra is a distributed NoSQL database. Large-scale data processing and storing applications frequently employ Cassandra because of its scalable and dependable nature [9]. Among the main characteristics of Cassandra are:

- High availability: Even in the case of a server failure, Cassandra is built to be highly available.
- Cassandra has linear scalability, meaning it can manage massive data.

### **2.3 Metrics**

The metrics I chose for my benchmarking are read and write performance. A brief about writing and reading performance, how it is achieved, and what queries are used for my implementation is described in the following sub-sections.

#### **2.3.1 Write Performance**

Write is loading the data in the database, and that performance is measured in terms of time, such as how much time it took to load the data. I calculated the time for the four databases. For write performance, I use different Python drivers for each database to connect to their respective ports and then load the data. The support for a new database can easily be added. We can write a script for connecting to the database using a port number or authentication and then write a function that loads the data in the appropriate data format.

#### **2.3.2 Read Performance**

Read performance is to retrieve the data from the database, perform queries like aggregation, and measure the time it takes for the database to complete it. For read performance, I have created six queries, three aggregation queries related to the data, and three statistical queries used explicitly for survival analysis. For read performance, if there is a need to support a new query for the same database, then the existing query code can be taken and modified accordingly. For adding the support of a new database, the script can be written that connects to the database and the specific table, and then the query can retrieve the results and print them out, and execution time can be measured.

I will briefly explain the different queries below.



### **Query 1:**

Calculate the number of dead people whose start date is greater than '1991-09-10' and whose end date is less than '2010-03-07'. Query in SQL:

```
SELECT COUNT (*) FROM data1 WHERE date_start_observed > '1991-09-10' AND  
date_end_observed < '2010-03-07' AND is_dead = true
```

### **Query 2:**

Calculate the percentage of censored data (individuals for whom the exact death time is unknown).

Query in SQL:

```
True count: SELECT COUNT (*) FROM data1 WHERE is_dead = TRUE;
```

```
Total count: SELECT COUNT (*) FROM data1;
```

```
Percentage = (true/total) * 100
```

### **Query 3:**

Calculate the average duration of observations for uncensored individuals (i.e., those who completed the observation period). Query in SQL:

```
SELECT AVG (age_end - age_start_observed) AS average_duration FROM data1 WHERE  
is_censored = False
```

### **Kaplan-Meier Estimate:**

The Kaplan-Meier estimator is a statistical method that calculates the probability of an event not occurring by a particular time. It considers the times when events, such as death, occur to adjust the survival odds. This estimator is handy when dealing with censored data, where the exact timing of an event is unknown, but it is known to have happened after a specific time [10]. A survival curve can be generated using the Kaplan-Meier estimator, predicting the likelihood of surviving past a particular point. Two things are needed for estimating: time to the event and event status. For my dataset, the

columns I chose are described below.

*Time-to-Event:*

"time-to-event" is the difference between "age\_end" and "age\_start\_observed."

*Event Status:*

If "is\_dead" is True, it means an event occurred (death), so set the event status to 1.

If "is\_censored" is True and "is\_dead" is False, it means the observation was censored (the event did not occur within the observation period), so set the event status to 0.

**Log-Rank Test:**

The log-rank test is a statistical method used to compare the survival curves of two or more groups and determine if there are any statistically significant differences in the survival durations between them.

Being a non-parametric test, it does not make any assumptions about the distribution of survival times.

The test calculates the observed and expected values to determine if there are any significant differences between the observed and expected number of events in each group [2]. For my benchmarking and dataset, I selected the age at which the patient died and whether the patient is deceased as my two groups.

**Cox Proportional Hazard:**

The Cox proportional hazards model is a semi-parametric regression model that examines the relationship between predictor factors and an individual's survival time [2]. It assumes that the chance of an event happening to each person is constantly multiplied by a baseline risk that changes over time.

Unlike parametric models, the Cox model is more adaptable because it makes no assumptions regarding the shape of the baseline hazard [10]. The model estimates hazard ratios (HR) for each predictor variable, which shows the relative probability of experiencing the event of interest. For my implementation, I used age\_end as the duration column and is\_dead as the event column. I used the

summary () function that prints out a detailed model overview. Below, I am providing a screenshot of the Cox Proportional Hazard implementation result and how we can interpret it.

```
<lifelines.CoxPHFitter: fitted with 890155 total observations, 499477 right-censored observations>
    duration col = 'age_end'
    event col = 'is_dead'
    baseline estimation = breslow
    number of observations = 890155
    number of events observed = 390678
    partial log-likelihood = -4761971.72
    time fit was run = 2023-11-06 20:42:13 UTC

----
              coef exp(coef)  se(coef)  coef lower 95%  coef upper 95%  exp(coef) lower 95%  exp(coef) upper 95%
covariate
age_start_observed -0.01      0.99      0.00      -0.01      -0.01      0.99      0.99
is_truncated       0.04      1.04      0.01       0.03       0.05      1.03      1.05

              cmp to      z      p  -log2(p)
covariate
age_start_observed  0.00 -36.43 <0.005   962.60
is_truncated       0.00  7.19 <0.005    40.48

----
Concordance = 0.54
Partial AIC = 9523947.44
log-likelihood ratio test = 2663.53 on 2 df
-log2(p) of ll-ratio test = inf
```

**Fig. 2: Cox Implementation sample output**

### Interpretation of Cox Proportional Hazard:

The CoxPH model was fitted with a total of 890,155 observations. Among those, 390,678 events were observed, and 499,477 observations were right-censored. The baseline estimation method used was "breslow," one of the methods for baseline survival estimation.

#### *Coefficient Table:*

The coefficient table shows the estimated coefficients for each covariate in the model. "coef" is the estimated coefficient for each covariate. "exp(coef)" is the estimated hazard ratio (HR) for each covariate, which represents the multiplicative effect on the hazard. "se(coef)" is the standard error of the coefficient estimate. "coef lower 95%" and "coef upper 95%" represent the lower and upper bounds of the 95% confidence interval for the coefficient. "exp(coef) lower 95%" and "exp(coef) upper 95%" are the lower and upper bounds of the 95% confidence interval for the hazard ratio.

#### *Comparisons and p-values:*

The "cmp to" column indicates the reference level for categorical covariates (if applicable). In the above

output, all covariates are continuous or binary, so there is no comparison. "z" is the z-score for each covariate's coefficient, indicating how many standard errors the coefficient estimate is from zero. "p" is the p-value associated with each covariate, showing the statistical significance of the variable. In the output above, all p-values are very close to zero, suggesting statistical significance. " $\log_2(p)$ " represents the negative logarithm base 2 of the p-value, which is often used to assess effectiveness. Large values of  $-\log_2(p)$  indicate high significance.

#### *Model Fit Statistics:*

Concordance is a measure of predictive accuracy. The above case is 0.54, indicating the model's ability to discriminate between subjects with different survival times. Partial AIC (Akaike Information Criterion) is a measure of model goodness-of-fit. The log-likelihood ratio test and  $-\log_2(p)$  of the ll-ratio test provides information about the model's overall fit.

#### *Interpretation:*

In the above case, "age\_start\_observed" has a negative coefficient, suggesting that as these variables increase, the hazard decreases. This means that older ages may be associated with a lower risk of death. "is\_truncated" has a positive coefficient, indicating that individuals with truncated observations have a higher risk of death than those without truncated observations.

### **III. Implementation**

This section discusses in detail the various things I used to implement the suite, including different languages and frameworks I used to build this suite. I will briefly discuss the basic requirements for running my suite and the device configurations on which I developed my suite.

#### **3.1 Basic Requirements**

##### **My Configuration:**

OS: Windows 11 Home

Processor: Intel Core i7-8550U CPU

Ram: 16 GB

##### **Basic Requirements:**

The user must install Python on the device as the complete suite is in Python. If the user uses Docker Compose or Docker for database setup, he should install Docker Desktop. I installed the different drivers and frameworks using pip (python package manager). So, if a user needs to install the missing frameworks, he can use pip. For example, to install pymongo (MongoDB driver), we can do 'pip install pymongo' to install pymongo, and then the scripts will work.

#### **3.2 Database Setup**

I used a Docker container as a form of data storage. To create and start the Docker containers, I used Docker Compose. My suite has a compose.yml file, which has configurations for my databases. I can use the 'docker-compose -up' script in the terminal to start the containers. I have used docker for QuestDB, MongoDB, and Cassandra. For TimescaleDB, I installed PostgreSQL from the official website and added the TimescaleDB extension.

### 3.3 Data Generation:

To generate the data, I wrote scripts in Python and gave parameters to pass the number of rows and database names. Depending on the number of rows I want, I can pass N and database as parameters; it will generate the data file and print the time. For TimescaleDB, I am using data in CSV format. So, for this project, I created 1 and 10 million rows in TimescaleDB format, and it took 10.94 and 109 seconds, as shown in the picture below.

```
PS D:\Aarsh\SJSU\CS298\Project\data_generation> python generate_data.py --n 1000000 --database timescaledb
Dataset creation took 10.94 seconds.
Data saved in timescaledb format.
```

```
PS D:\Aarsh\SJSU\CS298\Project\data_generation> python generate_data.py --n 10000000 --database timescaledb
Dataset creation took 109.00 seconds.
Data saved in timescaledb format.
```

After creating the CSV data, I can use it for TimescaleDB and QuestDB. But for MongoDB and Cassandra, I had to modify the above CSV file. For MongoDB, the data format is JSON, so I have another script that converts CSV to JSON. I loaded the CSV file using the 'read\_csv' method, converted it in Pandas framework, and used the 'to\_json' method to convert the CSV file to JSON to load in MongoDB.

```
PS D:\Aarsh\SJSU\CS298\Project\data_generation> python .\mongodb_json.py
CSV file "data1.csv" has been converted to JSON: "data1.json"
```

NoSQL databases have a unique index as one of the columns in their data that helps simplify the aggregation and querying process. While I loaded the MongoDB JSON file, the unique identifier was automatically created by MongoDB. But, for Cassandra, I had to add the unique identifier column manually. So, I wrote another script that loads the generated CSV file and adds the UIUD column as the first column in the new CSV file, thus making it ready to load the data in Cassandra.

```
PS D:\Aarsh\SJSU\CS298\Project\data_generation> python .\cassandra_addColumn.py
UUIDs added and saved to cassandra_data1.csv
```

### 3.4 Data Loading

The data loading part involves loading the data file into the database. Each database has different methods, so I wrote scripts for each database. I provided parameters like database name, table name, and path to the data file, which the user can specify. I will briefly explain the methods I implemented for each database and the Python drivers I used.

Starting with MongoDB, I used pymongo as my Python driver, which helped me connect to the database using Mongo Client. After a successful connection, I used the 'json.load' method to load the data in the database with a function to check if the data is in a list or dictionary, and the data will be loaded as per that. So, the code to run the script will be like this, and we can change the parameters per the database's design. Example script:

**python mongo\_load.py --database project1 --collection data --json\_file data1.json**

```
# Connect to MongoDB
client = pymongo.MongoClient(args.host, args.port)
db = client[args.database]
collection = db[args.collection]

# Record the start time
start_time = time.time()

# Load JSON data into MongoDB
with open(args.json_file, 'r') as json_file:
    data = json.load(json_file)
    if isinstance(data, list):
        collection.insert_many(data)
        print(f'{len(data)} documents inserted into {args.collection} in {args.database}.')
    elif isinstance(data, dict):
        collection.insert_one(data)
        print(f'1 document inserted into {args.collection} in {args.database}.')
    else:
        print('Invalid JSON data format.')

# Record the end time
end_time = time.time()
```

**Fig. 3: MongoDB data loading code**

For QuestDB, I used requests.post method to send a POST request to the QuestDB server with the

specified CSV file attached as part of the files parameter. A dictionary csv is created with a single key 'data' pointing to a tuple (table\_name, data\_file). So, my script for loading the data looks like this:

project1 is the table name,9000 is the port, and data1 is the CSV file. Example script:

**python questdb\_load.py http://localhost:9000 project1 data1.csv**

```
csv = {'data': ('my_table', open('./survival_data.csv', 'r'))}
host = 'http://localhost:9000'

try:
    response = requests.post(host + '/imp', files=csv)
    print(response.text)
except requests.exceptions.RequestException as e:
    print(f'Error: {e}', file=sys.stderr)
```

**Fig. 4: QuestDB data loading code**

For TimescaleDB, I used psycopg2 as my Python driver. At first, my script used the psycopg2 library to connect to a PostgreSQL database and load data from a CSV file into a specified table. Then, it establishes a connection to the PostgreSQL database using the psycopg2.connect method. The script opens the CSV file specified in the argument and reads the header to get the column names. It dynamically generates a CREATE TABLE query based on the column names obtained from the CSV file. The script then loads the data from CSV using the COPY command. As PostgreSQL requires authentication, I had to specify my PostgreSQL username and password to make the connection and load the data. My script for TimescaleDB is as follows.

**python timescale\_load.py --database aarsh --table data1 --csv\_file data1.csv --username postgres -  
-password aarsh**



```

# Connect to the specified database
db_params = db_params_without_db.copy()
db_params['dbname'] = args.database
connection = psycopg2.connect(**db_params)
cursor = connection.cursor()

# Read the CSV file to get the column names
with open(args.csv_file, 'r') as csv_file:
    csv_reader = csv.reader(csv_file)
    header = next(csv_reader)

# Create the table if it doesn't exist, dynamically generating the schema
create_table_query = f"CREATE TABLE IF NOT EXISTS {args.table} ("
for column_name in header:
    create_table_query += f"{column_name} TEXT, "
create_table_query = create_table_query.rstrip(', ') + ");"

cursor.execute(create_table_query)
connection.commit()

# Load data from CSV file into the table
with open(args.csv_file, 'r') as csv_file:
    cursor.copy_expert(f"COPY {args.table} FROM STDIN CSV HEADER DELIMITER ','", csv_file)
    connection.commit()

```

**Fig. 5: TimescaleDB data loading code**

Lastly, I had to create the table with column names and types for loading data in Cassandra, as Cassandra cannot automatically parse the CSV file and generate the table. I used Cassandra cluster as my Python driver, so after connecting to Cassandra, I had first to create a table with the names of the columns, and after that, I could load data in the database. At first, I used the INSERT statement to insert individual rows of data into a Cassandra table. However, it was inefficient for large datasets, so I used the COPY command, as it allows bulk data to load into a table from a CSV file. Example script:

**python cassandra\_load.py --keyspace project1 --table data --csv\_file cassandra\_data1.csv**

```

# Create the table with the specified schema
create_table_query = f"""
    CREATE TABLE IF NOT EXISTS {args.table} (
        UIUD TEXT PRIMARY KEY,
        age_start_observed INT,
        age_end INT,
        date_start_observed DATE,
        date_end_observed DATE,
        is_truncated BOOLEAN,
        is_censored BOOLEAN,
        is_dead BOOLEAN
    )
"""

session.execute(create_table_query)

# Record the start time
start_time = time.time()

# Execute the COPY command using cqlsh
copy_command = f"cqlsh -e \"COPY {args.keyspace}.{args.table} FROM '{args.csv_file}' WITH HEADER = true;\"
subprocess.run(copy_command, shell=True)

# Record the end time
end_time = time.time()

```

**Fig. 6: Cassandra data loading code**

### 3.5 Query Execution

I have used six queries overall to evaluate the read performance of the databases. In this section, I will discuss how I implemented the queries. The three aggregation queries have similar scripts, except for changes in the actual queries. So, I will talk about how I implemented the first query. To execute other queries, a user must change the query's logic in the script, and everything else should work out. For the query execution, I used the same code and driver as the data loading part to connect the database. I will discuss the part of the script after the connection to the database. There is no significant change in TimescaleDB and MongoDB, I made notable changes in QuestDB and Cassandra to make the queries work.

Below is the screenshot of MongoDB and TimescaleDB script of the query implementation and printing out the data. MongoDB query language slightly differs from other databases, so I used keywords like

"match" and "group," which serve as WHERE and GROUP BY in SQL query language.

```
# Define the aggregation pipeline
pipeline = [
    {
        "$match": {
            "date_start_observed": {"$gt": "1991-09-10T00:00:00Z"},
            "date_end_observed": {"$lt": "2010-03-07T00:00:00Z"},
            "is_dead": True
        }
    },
    {
        "$group": {
            "_id": None,
            "count": {"$sum": 1}
        }
    }
]

# Execute the aggregation pipeline and retrieve the result
result = list(collection.aggregate(pipeline))

# Extract the count from the result
count = result[0]["count"]

# Print the count
print("Count:", count)
```

**Fig. 7: MongoDB code for aggregation queries**

For TimescaleDB, I had to define the SQL query, use a cursor to make the connection, and then return the result, as shown in the figure below.

```

query = """
    SELECT COUNT(*)
    FROM data1
    WHERE date_start_observed > '1991-09-10'
        AND date_end_observed < '2010-03-07'
        AND is_dead = true
"""

cur = conn.cursor()
cur.execute(query)
result = cur.fetchall()

# Print the data (for demonstration purposes)
for row in result:
    print(row)

```

**Fig. 8: TimescaleDB code for aggregation queries**

I had to use a POST request to load the data in the data loading part for QuestDB. So now, I use GET request to get results from queries. The 'urllib.parse.quote' function is used to encode the SQL query, and the encoded query is then appended to the URL, creating the complete URL for the GET request. The script uses 'requests.get' to send a GET request and get the response from the server.

```

query = """
    SELECT COUNT(*)
    FROM project1
    WHERE date_start_observed > '1991-09-10'
        AND date_end_observed < '2010-03-07'
        AND is_dead = true
"""

encoded_query = urllib.parse.quote(query)
url = f'http://localhost:9000/exp?query={encoded_query}'

resp = requests.get(url)
print(resp.text)

```

**Fig. 9: QuestDB code for aggregation queries**

Cassandra's code is similar, but I made two significant changes. By default, Cassandra's timeout setting is of very few milliseconds. So, while executing the script, it gave an error that the query had timed out. To solve this, I used `ExecutionProfile` and imported it from `Cassandra.Cluster` class and increased the `request_timeout` before performing the query in my script. Also, I used an 'ALLOW FILTERING' keyword to allow filtering on non-indexed columns.

```

execprof = ExecutionProfile(request_timeout=100000)
profiles = {EXEC_PROFILE_DEFAULT:execprof}
cluster = Cluster(['localhost'], execution_profiles=profiles)
session = cluster.connect('project1')

query = """
    SELECT COUNT(*)
    FROM data1
    WHERE date_start_observed > '1991-09-10'
        AND date_end_observed < '2010-03-07'
        AND is_dead = true
    ALLOW FILTERING
"""
result = session.execute(query)

# Print the data (for demonstration purposes)
for row in result:
    print(row)

```

**Fig. 10: Cassandra code for aggregation queries**

I performed three statistical queries for survival analysis: Kaplan-Meier survival curve, Log-Rank test, and Cox Proportional Hazard Regression. I used the lifelines library, a complete survival-analysis library in Python, to implement these queries. So, for each query, I need to import the model I am implementing from the lifelines library. For example, I used 'from lifelines import KaplanMeierFitter' for Kaplan Meier. So, the connection to the database is made like the other queries, but to perform these statistical queries, I had to retrieve the data, put it in a Pandas data frame, and then perform the function. I have explained briefly about these statistical queries and what column names I chose from each model in the Design section, so I will briefly explain TimeScaleDB's implementation of queries.

For Kaplan-Meier, after retrieving the data from the databases, I stored it in a Data Frame, calculated the time-to-event and event status, plotted the curve, and returned the median survival rate. A snippet of my

implementation in TimescaleDB is as follows.

```
# Retrieve data from TimescaleDB and store it in a DataFrame
data = pd.read_sql_query(query, conn)

# Calculate time-to-event
data['time_to_event'] = data['age_end'] - data['age_start_observed']

# Set event status based on 'is_dead' and 'is_censored'
data['event_status'] = data['is_dead'].apply(lambda x: 1 if x else 0)
data.loc[data['is_censored'], 'event_status'] = 0

# Perform Kaplan-Meier analysis
kmf = KaplanMeierFitter()
kmf.fit(data['time_to_event'], event_observed=data['event_status'])

# Calculate the median survival time
median_survival_time = kmf.median_survival_time_

# Record the execution time
execution_time = time.time() - start_time

# Print the median survival time
print(f"Median Survival Time: {median_survival_time}")

# Print the execution time
print(f"Execution time: {execution_time:.2f} seconds")

# Plot the Kaplan-Meier curve
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 6))
kmf.plot(label="Kaplan-Meier Estimate")
plt.xlabel("Time")
plt.ylabel("Survival Probability")
plt.title("Kaplan-Meier Survival Curve")
plt.show()
```

**Fig. 11: Implementation of Kaplan-Meier for TimescaleDB**

I selected the required column from the database for the log-rank test and stored it in a Data frame.

Then, I fitted the age\_end and is\_dead columns in the model and printed the results.

```

# Specify your SQL query to retrieve data
query = "SELECT age_start_observed,age_end,is_truncated,is_dead FROM data1"

# Execute the SQL query and load the data into a DataFrame
data = pd.read_sql_query(query, conn)

# Create a KaplanMeierFitter object
kmf = lf.KaplanMeierFitter()

# Fit the model to the data
kmf.fit(data['age_end'], event_observed=data['is_dead'])

# Perform the log-rank test
results = logrank_test(data['is_dead'], data['age_end'])

# Print the results
print(results)

```

**Fig. 12: Implementation of Log-Rank for TimescaleDB**

To implement the Cox Proportional Hazard, I used `age_end` as my duration column and `is_dead` as my event\_col fitted it in the model and printed a summary of results.

```

# Specify your SQL query to retrieve data (replace with your query)
query = "SELECT age_start_observed,age_end,is_truncated,is_dead FROM data1"

# Execute the SQL query and load the data into a DataFrame
data = pd.read_sql_query(query, conn)

#Create a CoxPHFitter object
model = CoxPHFitter()

model.fit(data, duration_col='age_end', event_col='is_dead')

# Display the summary of the Cox regression model
model.print_summary()

```

**Fig. 13: Implementation of Cox Proportional Hazard for TimescaleDB**



## IV. Experiments

This section is divided into two sub-sections; the first is Write Performance, where I performed the data loading part of both databases in all databases as described in the Implementation section, and the second section is Read Performance, where I ran the scripts of various queries and recorded the execution time. After each section, I have provided a table with my execution times to make the comparison more visible.

### Write Performance:

#### Dataset 1: 1 million rows

```
PS D:\Aarsh\SJSU\CS298\Project\data_load> python cassandra_load.py --keyspace project1 --table data --csv_file cassandra_data1.csv
localhost
Using 7 child processes

Starting copy of project1.data with columns [uiuid, age_end, age_start_observed, date_end_observed, date_start_observed, is_censored, is_dead, is_truncated].
Processed: 890155 rows; Rate: 11465 rows/s; Avg. rate: 16820 rows/s
890155 rows imported from 1 files in 0 day, 0 hour, 0 minute, and 52.923 seconds (0 skipped).
Data loading took 55.08 seconds.
```

```
PS D:\Aarsh\SJSU\CS298\Project\data_load> python questdb_load.py http://localhost:9000 project1 data1.csv
Data loaded into project1
Data loading took 1.61 seconds.
```

```
PS D:\Aarsh\SJSU\CS298\Project\data_load> python mongo_load.py --database project1 --collection data --json_file data1.json
890155 documents inserted into data in project1.
Data loading took 14.59 seconds.
```

```
PS D:\Aarsh\SJSU\CS298\Project\data_load> python timescale_load.py --database aarsh --table data1 --csv_file data1.csv --username postgres --password aarsh
Data loaded into data1 in aarsh.
Data loading took 3.65 seconds.
```

#### Dataset 2: 10 million rows

```
PS D:\Aarsh\SJSU\CS298\Project\data_load> python mongo_load.py --database project2 --collection data --json_file data2.json
8902269 documents inserted into data in project2.
Data loading took 192.43 seconds.
```

```
PS D:\Aarsh\SJSU\CS298\Project\data_load> python questdb_load.py http://localhost:9000 project2 data2.csv
Data loaded into project2
Data loading took 22.78 seconds.
```

```
PS D:\Aarsh\SJSU\CS298\Project\data_load> python cassandra_load.py --keyspace project2 --table data --csv_file cassandra_data2.csv
localhost
Using 7 child processes

Starting copy of project2.data with columns [uiuid, age_end, age_start_observed, date_end_observed, date_start_observed, is_censored, is_dead, is_truncated].
Processed: 8902269 rows; Rate: 27956 rows/s; Avg. rate: 16030 rows/s
8902269 rows imported from 1 files in 0 day, 0 hour, 9 minutes, and 15.360 seconds (0 skipped).
Data loading took 558.61 seconds.
```

```
PS D:\Aarsh\SJSU\CS298\Project\data_load> python timescale_load.py --database aarsh --table data2 --csv_file data2.csv --username postgres --password aarsh
Data loaded into data2 in aarsh.
Data loading took 34.44 seconds.
```

**Write Performance Result Table (Time in seconds):**

	QuestDB	TimescaleDB	Cassandra	MongoDB
Dataset 1	1.61	3.65	55.08	14.59
Dataset 2	22.76	34.44	558.61	192.43

For both 1 million and 10 million rows, QuestDB continuously shows good write performance, indicating scalability and appropriateness for bigger datasets. With a minor increase in write time for the larger dataset, TimescaleDB works reasonably well. While both Cassandra and MongoDB perform well for smaller datasets, their write times for datasets with 10 million rows increase noticeably, with Cassandra exhibiting the longest write time.

**Read Performance:**

The queries for evaluating the read performance include three aggregations and three statistical queries designed explicitly for survival analysis. The query names in the screenshots are simplified for ease of use. So, I am explaining what different names in the Python files mean below.

query1: Calculate the number of people whose start date is greater than '1991-09-10' and whose end date is less than '2010-03-07' and who are dead.

query2: Calculate the percentage of censored data (individuals for whom the exact death time is unknown).

query3: Calculate the average duration of observations for uncensored individuals (i.e., those who completed the observation period).

Kaplan: Kaplan-Meier Curve

logrank: Log Rank Estimation

cox: Cox Proportional Hazard

Below are the screenshots for all queries. I will start with Dataset 1 and provide a table and then follow with Dataset 2

### Dataset 1:

Query1:

```
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query1> python .\query1_questdb.py
"count"
1254

Execution Time: 0.08 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query1> python .\query1_timescaledb.py
(1254,)
Execution Time: 0.10 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query1> python .\query1_cassandra.py
Row(count=1254)
Execution Time: 4.52 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query1> python .\query1_mongodb.py
Count: 1254
Execution Time: 0.78 seconds
```

Query2

```
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query2> python .\query2_questdb.py
Percentage of True values: 43.89%
Execution Time: 0.06 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query2> python .\query2_timescaledb.py
Percentage of True values: 43.89%
Execution Time: 0.30 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query2> python .\query2_cassandra.py
Percentage of True values: 43.89%
Execution Time: 23.78 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query2> python .\query2_mongodb.py
Percentage of True values: 43.89%
Execution Time: 1.17 seconds
```

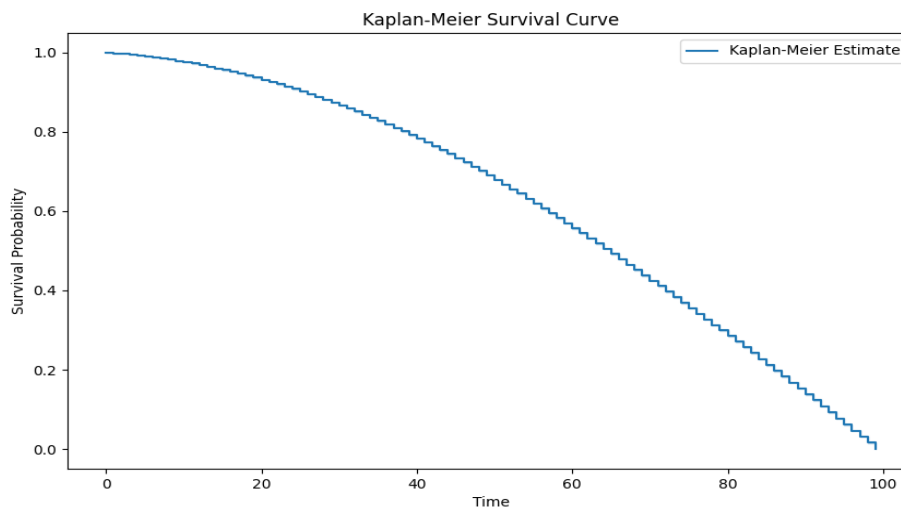
### Query3:

```
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query3> python .\query3_questdb.py
"average_duration_of_observation"
52.547430364648

Execution Time: 0.24 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query3> python .\query3_timescaledb.py
(Decimal('52.5474303646481245'),)
Execution Time: 0.14 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query3> python .\query3_cassandra.py
Row(average_duration_of_observation=-52)
Execution Time: 8.76 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query3> python .\query3_mongodb.py
average_duration_of_observation: 52.54743036464812
Execution Time: 1.56 seconds
```

### Kaplan Meier Curve:

```
PS D:\Aarsh\SJSU\CS298\Project\query_execution\Kaplan-Meier> python .\Kaplan_questdb.py
Median Survival Time: 65.0
Execution time: 2.39 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\Kaplan-Meier> python .\Kaplan_timescaledb.py
D:\Aarsh\SJSU\CS298\Project\query_execution\Kaplan-Meier\Kaplan_timescaledb.py:23: UserWarning: pandas only supports SQLAlchemy
engine connectable (engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested
. Please consider using SQLAlchemy.
  data = pd.read_sql_query(query, conn)
Median Survival Time: 65.0
Execution time: 1.56 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\Kaplan-Meier> python .\Kaplan_cassandra.py
Median Survival Time: 65.0
Execution time: 12.17 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\Kaplan-Meier> python .\Kaplan_mongo.py
Median Survival Time: 65.0
Execution time: 8.79 seconds
```



## Log Rank Test:

```
PS D:\Aarsh\SJSU\CS298\Project\query_execution\LogRank> python .\logrank_questdb.py
<lifelines.StatisticalResult: logrank_test>
      t_0 = -1
      null_distribution = chi squared
      degrees_of_freedom = 1
      test_name = logrank_test

---
      test_statistic      p      -log2(p)
      1812886.77 <0.005      inf
Execution time: 2.26 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\LogRank> python .\logrank_timescaledb.py
D:\Aarsh\SJSU\CS298\Project\query_execution\LogRank\logrank_timescaledb.py:23: UserWarning: pandas only supports SQLAlchemy connectable (engine/connection)
or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider using SQLAlchemy.
  data = pd.read_sql_query(query, conn)
<lifelines.StatisticalResult: logrank_test>
      t_0 = -1
      null_distribution = chi squared
      degrees_of_freedom = 1
      test_name = logrank_test

---
      test_statistic      p      -log2(p)
      1812886.77 <0.005      inf
Execution time: 1.20 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\LogRank> python .\logrank_cassandra.py
<lifelines.StatisticalResult: logrank_test>
      t_0 = -1
      null_distribution = chi squared
      degrees_of_freedom = 1
      test_name = logrank_test

---
      test_statistic      p      -log2(p)
      1812886.77 <0.005      inf
Execution time: 10.03 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\LogRank> python .\logrank_mongo.py
<lifelines.StatisticalResult: logrank_test>
      t_0 = -1
      null_distribution = chi squared
      degrees_of_freedom = 1
      test_name = logrank_test

---
      test_statistic      p      -log2(p)
      1812886.77 <0.005      inf
Execution time: 8.83 seconds
```

## Cox Proportional Hazard.

```
PS D:\Aarsh\SJSU\CS298\Project\query_execution\Cox> python .\cox_questdb.py
<lifelines.CoxPHFitter: fitted with 890155 total observations, 499477 right-censored observations>
      duration col = '1'
      event col = '2'
      baseline estimation = breslow
      number of observations = 890155
      number of events observed = 390678
      partial log-likelihood = -4761971.72
      time fit was run = 2023-11-06 20:44:04 UTC

---
      coef  exp(coef)  se(coef)  coef lower 95%  coef upper 95%  exp(coef) lower 95%  exp(coef) upper 95%
covariate
0      -0.01      0.99      0.00      -0.01      -0.01      0.99      0.99
3       0.04      1.04      0.01       0.03       0.05      1.03      1.05

      cmp to      z      p      -log2(p)
covariate
0       0.00 -36.43 <0.005      962.60
3       0.00  7.19 <0.005      40.48

---
Concordance = 0.54
Partial AIC = 9523947.44
log-likelihood ratio test = 2663.53 on 2 df
-log2(p) of ll-ratio test = inf

Execution time: 23.89 seconds
```

```

PS D:\Aarsh\SJSU\CS298\Project\query_execution\Cox> python .\cox_timescaledb.py
D:\Aarsh\SJSU\CS298\Project\query_execution\Cox\cox_timescaledb.py:23: UserWarning: pandas only supports SQLAlchemy connectable
(engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider
using SQLAlchemy.
  data = pd.read_sql_query(query, conn)
<lifelines.CoxPHFitter: fitted with 890155 total observations, 499477 right-censored observations>
      duration col = 'age_end'
      event col = 'is_dead'
      baseline estimation = breslow
      number of observations = 890155
      number of events observed = 390678
      partial log-likelihood = -4761971.72
      time fit was run = 2023-11-06 20:42:13 UTC

---
      coef exp(coef) se(coef) coef lower 95% coef upper 95% exp(coef) lower 95% exp(coef) upper 95%
covariate
age_start_observed -0.01 0.99 0.00 -0.01 -0.01 0.99 0.99
is_truncated 0.04 1.04 0.01 0.03 0.05 1.03 1.05

      cmp to z p -log2(p)
covariate
age_start_observed 0.00 -36.43 <0.005 962.60
is_truncated 0.00 7.19 <0.005 40.48
---
Concordance = 0.54
Partial AIC = 9523947.44
log-likelihood ratio test = 2663.53 on 2 df
-log2(p) of ll-ratio test = inf
Execution time: 23.21 seconds

```

```

PS D:\Aarsh\SJSU\CS298\Project\query_execution\Cox> python .\cox_mongo.py
<lifelines.CoxPHFitter: fitted with 890155 total observations, 499477 right-censored observations>
      duration col = 'age_end'
      event col = 'is_dead'
      baseline estimation = breslow
      number of observations = 890155
      number of events observed = 390678
      partial log-likelihood = -4761971.72
      time fit was run = 2023-11-06 20:44:53 UTC

---
      coef exp(coef) se(coef) coef lower 95% coef upper 95% exp(coef) lower 95% exp(coef) upper 95%
covariate
age_start_observed -0.01 0.99 0.00 -0.01 -0.01 0.99 0.99
is_truncated 0.04 1.04 0.01 0.03 0.05 1.03 1.05

      cmp to z p -log2(p)
covariate
age_start_observed 0.00 -36.43 <0.005 962.60
is_truncated 0.00 7.19 <0.005 40.48
---
Concordance = 0.54
Partial AIC = 9523947.44
log-likelihood ratio test = 2663.53 on 2 df
-log2(p) of ll-ratio test = inf
Execution time: 27.17 seconds

```

```

PS D:\Aarsh\SJSU\CS298\Project\query_execution\Cox> python .\cox_cassandra.py
<lifelines.CoxPHFitter: fitted with 890155 total observations, 499477 right-censored observations>
    duration col = 'age_start_observed'
    event col = 'is_truncated'
    baseline estimation = breslow
    number of observations = 890155
    number of events observed = 390678
    partial log-likelihood = -4761971.72
    time fit was run = 2023-11-06 20:46:05 UTC

---
      coef  exp(coef)   se(coef)   coef lower 95%   coef upper 95%  exp(coef) lower 95%  exp(coef) upper 95%
covariate
age_end    -0.01      0.99      0.00        -0.01        -0.01           0.99          0.99
is_censored  0.04      1.04      0.01         0.03         0.05           1.03          1.05

      cmp to      z      p   -log2(p)
covariate
age_end      0.00 -36.43 <0.005   962.60
is_censored  0.00  7.19 <0.005    40.48
---
Concordance = 0.54
Partial AIC = 9523947.44
log-likelihood ratio test = 2663.53 on 2 df
-log2(p) of ll-ratio test = inf

Execution time: 31.77 seconds

```

#### Read Performance Result Table for Dataset 1 (Time in seconds):

Dataset1	Query1	Query2	Query3	Kaplan	Log Rank	CoX
QuestDB	0.08	0.06	0.24	2.39	2.26	23.89
TimescaleDB	0.10	0.30	0.14	1.56	1.20	23.21
Cassandra	4.52	23.78	8.76	12.17	10.03	31.77
MongoDB	0.78	1.17	1.56	8.79	8.83	27.17

## Dataset 2:

### Query1:

```
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query1> python .\query1_questdb.py
"count"
12253

Execution Time: 1.70 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query1> python .\query1_timescaledb.py
(12253,)
Execution Time: 1.83 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query1> python .\query1_cassandra.py
Row(count=12253)
Execution Time: 29.80 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query1> python .\query1_mongodb.py
Count: 12260
Execution Time: 7.99 seconds
```

### Query2:

```
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query2> python .\query2_questdb.py
Percentage of True values: 43.95%
Execution Time: 0.16 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query2> python .\query2_timescaledb.py
Percentage of True values: 43.95%
Execution Time: 0.95 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query2> python .\query2_cassandra.py
Percentage of True values: 43.95%
Execution Time: 106.73 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query2> python .\query2_mongodb.py
Percentage of True values: 43.95%
Execution Time: 5.61 seconds
```

### Query3:

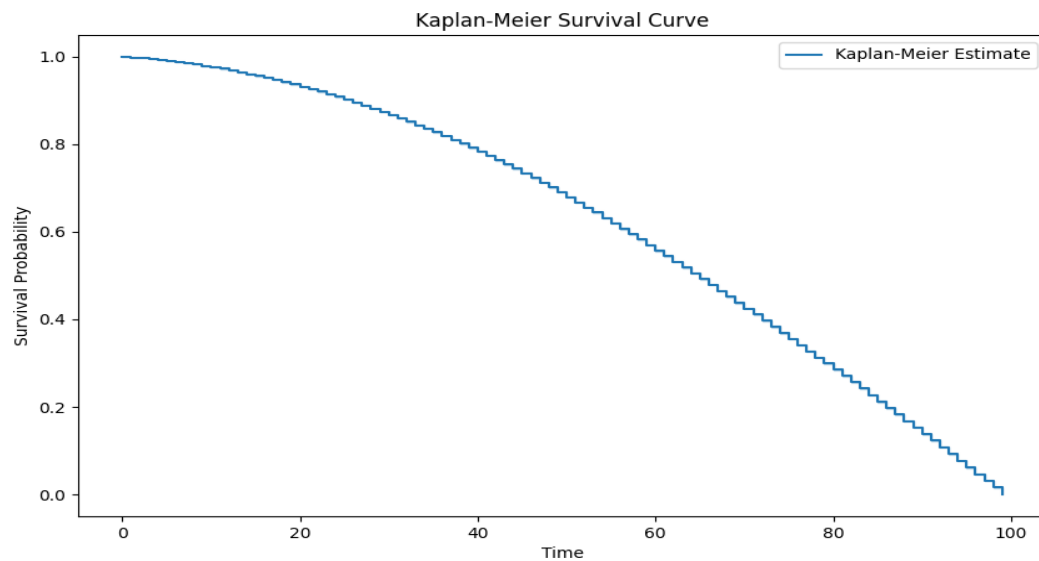
```
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query3> python .\query3_questdb.py
"average_duration_of_observation"
52.619327898403

Execution Time: 0.52 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query3> python .\query3_timescaledb.py
(Decimal('52.6193278984039485'),)
Execution Time: 0.67 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query3> python .\query3_cassandra.py
Row(average_duration_of_observation=-52)
Execution Time: 46.04 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query3> python .\query3_mongodb.py
average_duration_of_observation: 52.619327898403945
Execution Time: 5.17 seconds
```



## Kaplan Meier Curve:

```
PS D:\Aarsh\SJSU\CS298\Project\query_execution\Kaplan-Meier> python .\Kaplan_questdb.py
Median Survival Time: 65.0
Execution time: 16.20 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\Kaplan-Meier> python .\Kaplan_timescaledb.py
D:\Aarsh\SJSU\CS298\Project\query_execution\Kaplan-Meier\Kaplan_timescaledb.py:23: UserWarning: pandas only supports SQLAlchemy
emy connectable (engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested
. Please consider using SQLAlchemy.
  data = pd.read_sql_query(query, conn)
Median Survival Time: 65.0
Execution time: 13.80 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\Kaplan-Meier> python .\Kaplan_cassandra.py
Median Survival Time: 65.0
Execution time: 131.38 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\Kaplan-Meier> python .\Kaplan_mongo.py
Median Survival Time: 65.0
Execution time: 176.33 seconds
```



## Log Rank Test:

```
PS D:\Aarsh\SJSU\CS298\Project\query_execution\LogRank> python .\logrank_questdb.py
<lifelines.StatisticalResult: logrank_test>
      t_0 = -1
      null_distribution = chi squared
      degrees_of_freedom = 1
      test_name = logrank_test

---
      test_statistic      p      -log2(p)
      18125791.12 <0.005      inf
Execution time: 16.14 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\LogRank> python .\logrank_timescaledb.py
D:\Aarsh\SJSU\CS298\Project\query_execution\LogRank\logrank_timescaledb.py:23: UserWarning: pandas only supports SQLAlchemy connectable (engine/connection)
or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider using SQLAlchemy.
  data = pd.read_sql_query(query, conn)
<lifelines.StatisticalResult: logrank_test>
      t_0 = -1
      null_distribution = chi squared
      degrees_of_freedom = 1
      test_name = logrank_test

---
      test_statistic      p      -log2(p)
      18125791.12 <0.005      inf
Execution time: 12.95 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\LogRank> python .\logrank_cassandra.py
<lifelines.StatisticalResult: logrank_test>
      t_0 = -1
      null_distribution = chi squared
      degrees_of_freedom = 1
      test_name = logrank_test

---
      test_statistic      p      -log2(p)
      18125791.12 <0.005      inf
Execution time: 116.05 seconds
PS D:\Aarsh\SJSU\CS298\Project\query_execution\LogRank> python .\logrank_mongo.py
<lifelines.StatisticalResult: logrank_test>
      t_0 = -1
      null_distribution = chi squared
      degrees_of_freedom = 1
      test_name = logrank_test

---
      test_statistic      p      -log2(p)
      18125791.12 <0.005      inf
Execution time: 133.78 seconds
```

## Cox Proportional Hazard.

```
PS D:\Aarsh\SJSU\CS298\Project\query_execution\Cox> python .\cox_questdb.py
<lifelines.CoxPHFitter: fitted with 8.90227e+06 total observations, 4.98968e+06 right-censored observations>
      duration col = '1'
      event col = '2'
      baseline estimation = breslow
      number of observations = 8.90227e+06
      number of events observed = 3.91259e+06
      partial log-likelihood = -56696051.25
      time fit was run = 2023-11-06 20:20:33 UTC

---
      coef  exp(coef)  se(coef)  coef lower 95%  coef upper 95%  exp(coef) lower 95%  exp(coef) upper 95%
covariate
0      -0.01      0.99      0.00      -0.01      -0.01      0.99      0.99
3       0.04      1.04      0.00       0.03       0.04      1.03      1.04

      cmp to      z      p      -log2(p)
covariate
0       0.00 -113.21 <0.005      inf
3       0.00  21.04 <0.005    323.91
---
Concordance = 0.54
Partial AIC = 113392106.49
log-likelihood ratio test = 26246.24 on 2 df
-log2(p) of ll-ratio test = inf

Execution time: 317.06 seconds
```

```

PS D:\Aarsh\SJSU\CS298\Project\query_execution\Cox> python .\cox_timescaledb.py
D:\Aarsh\SJSU\CS298\Project\query_execution\Cox\cox_timescaledb.py:23: UserWarning: pandas only supports SQLAlchemy connectable (engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider using SQLAlchemy.
  data = pd.read_sql_query(query, conn)
<lifelines.CoxPHFitter: fitted with 8.90227e+06 total observations, 4.98968e+06 right-censored observations>
  duration col = 'age_end'
  event col = 'is_dead'
  baseline estimation = breslow
  number of observations = 8.90227e+06
  number of events observed = 3.91259e+06
  partial log-likelihood = -56696051.25
  time fit was run = 2023-11-06 19:52:02 UTC

---

```

	coef	exp(coef)	se(coef)	coef lower 95%	coef upper 95%	exp(coef) lower 95%	exp(coef) upper 95%
covariate							
age_start_observed	-0.01	0.99	0.00	-0.01	-0.01	0.99	0.99
is_truncated	0.04	1.04	0.00	0.03	0.04	1.03	1.04

	cmp to	z	p	-log2(p)
covariate				
age_start_observed	0.00	-113.21	<0.005	inf
is_truncated	0.00	21.04	<0.005	323.91

```

---
Concordance = 0.54
Partial AIC = 113392106.49
log-likelihood ratio test = 26246.24 on 2 df
-log2(p) of ll-ratio test = inf

Execution time: 306.76 seconds

```

```

PS D:\Aarsh\SJSU\CS298\Project\query_execution\Cox> python .\cox_mongo.py
<lifelines.CoxPHFitter: fitted with 8.90227e+06 total observations, 4.98968e+06 right-censored observations>
  duration col = 'age_end'
  event col = 'is_dead'
  baseline estimation = breslow
  number of observations = 8.90227e+06
  number of events observed = 3.91259e+06
  partial log-likelihood = -56696051.25
  time fit was run = 2023-11-06 20:06:40 UTC

---

```

	coef	exp(coef)	se(coef)	coef lower 95%	coef upper 95%	exp(coef) lower 95%	exp(coef) upper 95%
covariate							
age_start_observed	-0.01	0.99	0.00	-0.01	-0.01	0.99	0.99
is_truncated	0.04	1.04	0.00	0.03	0.04	1.03	1.04

	cmp to	z	p	-log2(p)
covariate				
age_start_observed	0.00	-113.21	<0.005	inf
is_truncated	0.00	21.04	<0.005	323.91

```

---
Concordance = 0.54
Partial AIC = 113392106.49
log-likelihood ratio test = 26246.24 on 2 df
-log2(p) of ll-ratio test = inf

Execution time: 333.39 seconds

```

```

PS D:\Aarsh\SJSU\CS298\Project\query_execution\Cox> python .\cox_cassandra.py
<lifelines.CoxPHFitter: fitted with 8.90227e+06 total observations, 4.98968e+06 right-censored observations>
    duration col = 'age_start_observed'
    event col = 'is_truncated'
    baseline estimation = breslow
    number of observations = 8.90227e+06
    number of events observed = 3.91259e+06
    partial log-likelihood = -56696051.25
    time fit was run = 2023-11-06 20:29:51 UTC

---
      coef  exp(coef)  se(coef)  coef lower 95%  coef upper 95%  exp(coef) lower 95%  exp(coef) upper 95%
covariate
age_end      -0.01      0.99      0.00      -0.01      -0.01      0.99      0.99
is_censored   0.04      1.04      0.00      0.03      0.04      1.03      1.04

      cmp to      z      p  -log2(p)
covariate
age_end      0.00 -113.21 <0.005      inf
is_censored   0.00  21.04 <0.005    323.91
---
Concordance = 0.54
Partial AIC = 113392106.49
log-likelihood ratio test = 26246.24 on 2 df
-log2(p) of ll-ratio test = inf
Execution time: 403.29 seconds

```

#### Read Performance Result Table for Dataset 2 (Time in seconds):

Dataset2	Query1	Query2	Query3	Kaplan	Log Rank	CoX
QuestDB	1.70	0.16	0.52	16.20	16.14	317.06
TimescaleDB	1.83	0.95	0.67	13.80	12.96	306.76
Cassandra	29.80	106.73	46.04	131.38	116.05	403.29
MongoDB	7.99	5.61	5.17	176.33	133.78	333.39

QuestDB consistently surpasses competing databases in terms of performance while executing a wide range of queries, demonstrating its effectiveness in handling typical aggregate and specialized survival analysis queries. TimescaleDB shows competitive performance but with marginally longer execution times than QuestDB. MongoDB and Cassandra, although appropriate for common queries, exhibit longer execution times, particularly when doing survival analysis queries. Cassandra demonstrates

exceptional proficiency in handling survival analysis queries in the domains of Kaplan-Meier and Cox Proportional Hazards compared to MongoDB. QuestDB and TimescaleDB exhibit strong performance in survival analysis, albeit perhaps with longer execution durations when compared to Cassandra.

## Conclusion

This research has created and studied a new benchmarking suite to examine database systems that manage survival analysis data thoroughly. The code of the suite is available in a GitHub repository and as a Docker Image uploaded to the Docker Hub. The link to the GitHub repository for my project is as follows :

<https://github.com/patelaarsh/Survival-Analysis-Data-Benchmarking-Suite>

It contains the folders and readme file with the scripts to perform the benchmarking. The link to the Docker image of the suite is as follows:

<https://hub.docker.com/repository/docker/asp10/survivalbenchmark/>

The project can be pulled using docker pull. Example:

```
docker pull asp10/survivalbenchmark:latest
```

The suite consists of read and write performance measurements and has been tested with prominent databases such as QuestDB, TimeScaleDB, Cassandra, and MongoDB. The suite focuses on specialized questions related to survival analysis, including Kaplan-Meier, Cox Proportional Hazards, and Log-Rank. The goal is to provide transparency and reproducibility by explaining the datasets, custom metrics, and approach, making adding more databases and functionalities easy. This suite stands out even more for integrating read and write performance indicators catering to real-world applications' comprehensive needs, including survival analysis queries. The suite's dependability and utility are enhanced through the implementation of a transparent methodology and open-source accessibility, fostering collaboration and verification among professionals in the realm of database management.

The study findings show significant variations in different databases' read and write performance. QuestDB offers excellent scalability for larger datasets and performs well across various queries. TimeScaleDB performs competitively, particularly in write operations. On the other hand,

Cassandra performs well in survival analysis queries but struggles with write performance when handling more massive datasets. MongoDB performs poorly, but it has limitations in specific survival analysis queries. These outcomes demonstrate the trade-offs one must consider while selecting a database for a particular application, providing valuable insights for informed decision-making.

When comparing a time series to NoSQL databases, it becomes apparent that there are subtle differences. Although NoSQL databases can be helpful, they might not respond quickly to certain survival analysis queries. This is where TimeScaleDB comes into play. It is well-suited for situations where time-dependent data is present, as demonstrated by its competitive performance in various queries. TimeScaleDB is specifically designed for time-series data. By analyzing and comparing the benefits of specialized time-series databases, it becomes clear how crucial it is to select the appropriate database to match the requirements of survival analysis data.

In conclusion, the benchmarking suite is valuable for evaluating database systems that manage survival analysis data. It highlights the pros and cons of NoSQL and time-series databases and helps to make informed decisions when choosing a database for survival analysis-focused applications. Additionally, the findings pave the way for further research and optimization efforts within the database community.

## REFERENCES

- [1] Time Series Database (TSDB) guide: Influxdb. InfluxData. (2023, October 23). Retrieved from <https://www.influxdata.com/time-series-database/>
- [2] Clark, T. G., Bradburn, M. J., Love, S. B., & Altman, D. G. (2003). Survival analysis part I: Basic concepts and first analyses. *British Journal of Cancer*, 89(2), 232–238. <https://doi.org/10.1038/sj.bjc.6601118>
- [3] A. Struckov, S. Yufa, A. A. Visheratin, and D. Nasonov, "Evaluation of modern tools and techniques for storing time-series data," *Procedia Computer Science*, vol. 156, pp. 19–28, 2019. doi: 10.1016/j.procs.2019.08.125
- [4] TimescaleDB. (2023). TimescaleDB Documentation. Retrieved from <https://docs.timescale.com/>
- [5] QuestDB. (2023). QuestDB Documentation. Retrieved from <https://questdb.io/docs/>
- [6] J. Han, H. E, G. Le, and J. Du, "Survey on NoSQL database," 2011 6th International Conference on Pervasive Computing and Applications, pp. 363–366, 2011.
- [7] V. Abramova and J. Bernardino, "NoSQL databases: MongoDB vs Cassandra," *Proceedings of the International C\* Conference on Computer Science and Software Engineering*, pp. 14–22, 2013.
- [8] MongoDB. (2023). MongoDB Documentation. Retrieved from <https://www.mongodb.com/docs/>
- [9] Apache Software Foundation. (2023). Apache Cassandra Documentation. Retrieved from <https://cassandra.apache.org/doc/latest/>
- [10] Schober, P., & Vetter, T. R. (2018). Survival Analysis and Interpretation of Time-to-Event Data: The Tortoise and the Hare. *Anesthesia and analgesia*, 127(3), 792–798. <https://doi.org/10.1213/ANE.00000000000003653>