# Database Benchmarking Suite For Survival Analysis Data


Presented by -

Aarsh Patel

Department of Computer Science

SJSU SAN JOSÉ STATE UNIVERSITY

**Committee**:

Dr. Chris Pollett (Advisor)

Dr. Robert Chun

Dr. William Andreopoulos

# Agenda

- ❖ Problem Statement
- ❖ Database and Benchmarking
- ❖ Time-Series Data & Background
- ❖ Survival Analysis Data
- ❖ Design & Metrics of Suite
- ❖ Implementation of Suite
- ❖ Experiments
- ❖ Results & Conclusion
- ❖ Future work

# Problem Statement

- Survival analysis data is used for analyzing till the event occurs and is crucial for predicting future events and making informed decisions.
- This type of data is stored in various databases, and it's important to select the right database for each use case.
- Benchmarking is used for database selection. Many suites already exist developed for particular data or databases

  TPCC – OLTP systems

  TSBS – Time-series databases
- Development of a benchmarking suite specifically designed for survival analysis data.
- Encompasses performance metrics for both read and write operations and has been applied to various databases.
- Specialized topics related to survival analysis, such as Log-Rank, Cox Proportional Hazards, and Kaplan-Meier, were given significant attention.
- Comparison of NoSQL databases with time-series databases for storing and retrieving survival analysis data.

# Databases

- The word DATA is Latin for FACTS.
- A database is a place or thing that stores facts.
- Used for storing, managing, and retrieving information.

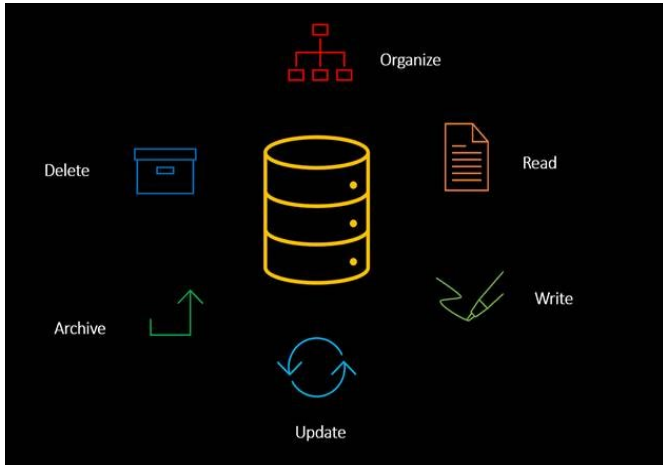

**Types of Databases:**

Relational Databases (RDBMS):

- Tables with predefined relationships (e.g., MySQL, PostgreSQL).

NoSQL Databases:

- Document-oriented, Key-Value, Column-family, Graph (e.g., MongoDB, Redis, Cassandra, Neo4j).




Time-Series Databases:

- Specialized for time-stamped data (e.g., InfluxDB, Prometheus).

SJSU SAN JOSÉ STATE UNIVERSITY

# Benchmarking

**What is Benchmarking?**

- Evaluation of system performance against defined standards or criteria.

**Importance of Benchmarking:**

- Performance & Scalability Assessment
- Technology Selection & Cost-Efficiency

**Database Benchmarking Process:**

- Define Objectives: Clearly define benchmarking goals.
- Select Workloads: Choose representative workloads.
- Design Scenarios: Develop scenarios for write and read operations.
- Execute Benchmarks: Run workloads, collecting performance metrics.
- Analyze Results: Identify strengths, weaknesses, and areas for improvement.


SJSU SAN JOSÉ STATE UNIVERSITY

# Time-Series Data

- Time-series data refers to a series of data points collected or recorded chronologically, typically at regular intervals.
- Each data point is associated with a specific timestamp

**Example:** Stock Prices, Weather Data

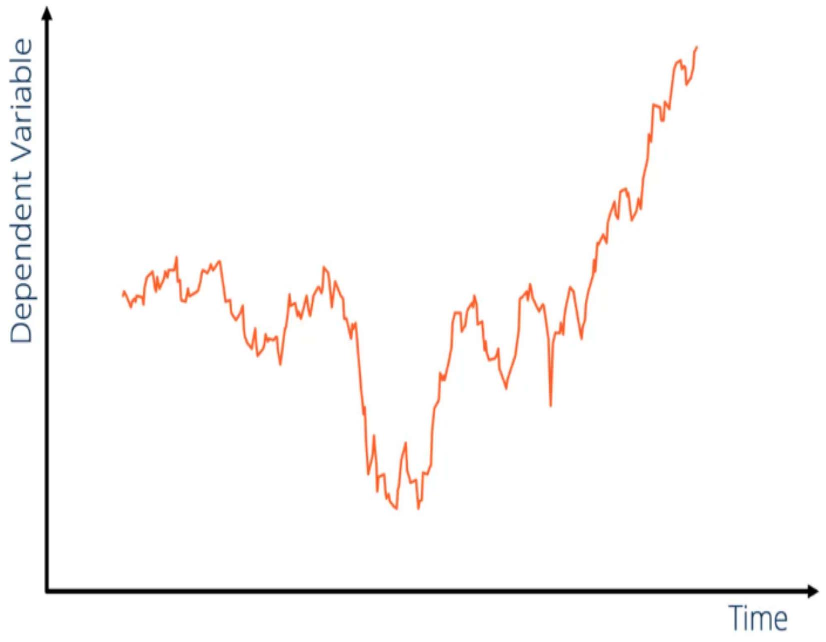
**Benefits of Time-Series Data Analysis:**

**Trend Analysis:** Long-term trends, such as increasing or decreasing patterns over time.

**Anomaly Detection:** Unusual events or outliers can be detected by analyzing deviations from the expected patterns.

**Forecasting:** Used to predict future values based on historical data

**Decision Making:** Make informed decisions, especially in areas like finance, marketing, and operations.



# Background Work

**Time Series Benchmarking Suite (TSBS)**

A tool for benchmarking time-series databases.

Collection of Go programs that generate datasets and benchmark various databases' read and write performance.

Supports many Timeseries and NoSQL Databases

Link to the suite: [https://github.com/timescale/tsbs](https://github.com/timescale/tsbs)

**CS297 Work:**

- Benchmarked three time-series databases (TimeScaleDB, InfluxDB, and QuestDB) and MongoDB as a NoSQL database.
- 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.

# Survival Analysis

- A statistical method used to analyze the time until an event of interest occurs.
- Example: A machine's failure, a disease's occurrence, or a patient's death.

**Benefits of Survival Analysis:**

**Time-to-Event Analysis:** Survival analysis provides a comprehensive way to analyze the time until an event, accounting for censored data where the event might not have occurred for some individuals.

**Understanding Risk Factors:** It helps identify and quantify factors that may influence the time until an event occurs.

**Comparing Groups:** Researchers can compare survival curves for different groups to understand if significant differences exist in the time until the event.


SJSU SAN JOSÉ STATE UNIVERSITY

# Design & Metrics

- The design of the suite is kept simple. There are 3 folders: data generation, data load, and query execution.
- The data generation folder contains the scripts for generating data.
- The data loading folder contains scripts for each database for loading the data.
- The query execution folder includes many sub-folders specifying each query I used, and the query folder has a script for performing the query on each database.
- docker compose yml file includes configuration of the databases I used for implementation.
- Readme file explains how to use docker compose and contains the  example scripts for each part
- It’s very easy to modify the scripts of code for each part and add other databases by modifying the yml file.
- I used 4 databases which can be classified as Time-Series and NoSQL Databases.

| | | | |
|---|---|---|---|
| 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 |

SJSU SAN JOSÉ STATE UNIVERSITY

# Timeseries Databases

- Specialized databases designed for handling time-series data efficiently.
- Manages data points associated with specific timestamps, making them ideal for applications that track and analyze changes over time.
- Provide optimized storage, indexing, and query capabilities for time-ordered data, allowing for high-performance retrieval and analysis.

# TimescaleDB & QuestDB

**TimescaleDB:**

An open-source time-series constructed as a PostgreSQL plugin, so it supports SQL.

**Hyper tables:** Optimized for storing and querying large amounts of time-series data.

**Chunking:** Chunks data into small pieces, making it query faster.

**Compression:** Compresses data, reducing the required storage space.

**Rollups:** Can roll up data into aggregates, making summarizing and analyzing large datasets easier

**QuestDB:**

A lightweight, high-performance time-series database written in Java, designed to be easy to use and scale large volumes of data.

**High performance:** Process millions of events per second.

**Scalability:** Scale to handle large volumes of data.

**Durability:** Designed to be durable and reliable, even during a power failure.

# NoSQL Databases

- NoSQL databases do not adhere to the traditional relational database management system (RDBMS) model.
- Designed to handle large volumes of unstructured or semi-structured data and provide flexible data models.

**CAP Theorem:**

It is impossible to achieve all three of the following guarantees simultaneously:

- Consistency (C): All nodes in the system see the same data simultaneously. This implies that a read request will always return the most recent write.
- Availability (A): Every request made to a non-failing node in the system receives a response
- Partition Tolerance (P): The system continues to operate despite network partitions that may cause node communication failures.

# Types of NoSQL Databases

Key-Value Stores:

- Data is stored as a collection of key-value pairs.
- Examples: Redis, Amazon DynamoDB

Column-Family Stores:

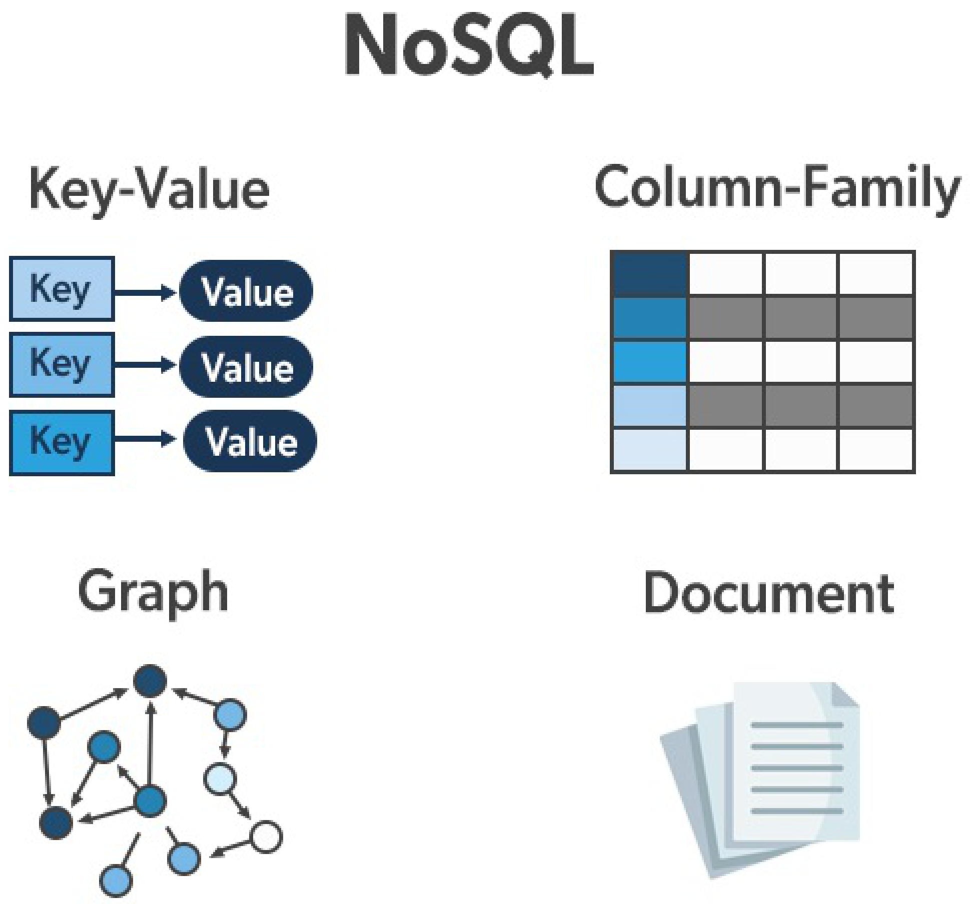
- Data is organized as columns rather than rows.
- Examples: Apache Cassandra, HBase

Graph Databases:

- Data is represented as nodes, edges, and properties.
- Examples:Neo4j.Amazon Neptune

Document Stores:

- Data is stored in flexible, schema-less documents
- Examples: MongoDB. CouchDB




SJSU SAN JOSÉ STATE UNIVERSITY

# MongoDB & Cassandra

**MongoDB:**

**Data Model:** Document-oriented NoSQL database that stores data in flexible, JSON-like BSON (Binary JSON) documents.

**Query Language:** MongoDB uses a rich query language that supports dynamic queries, indexing, and secondary indexes.

**Scalability:** MongoDB supports horizontal scalability through sharding. It can distribute data across multiple nodes to handle large datasets and high write and read loads.

**Consistency:** MongoDB provides strong consistency by default.

**Apache Cassandra:**

**Data Model:** A wide-column store NoSQL database. It uses a table structure with rows and columns.

**Query Language:** CQL provides a SQL-like interface for querying data in Cassandra.

**Scalability:** Designed for horizontal scalability and high availability.

**Consistency:** Supports tunable consistency, allowing users to choose the level of consistency based on their requirements.

# Dataset

- Kaggle: A subsidiary of Google and an online community of data scientists that allows users to find datasets they want to use.
- Found an example of a synthetic survival analysis dataset that is developed in Python.
- I reworked the code, allowing the users to specify the number of rows as parameters, allowing creating the dataset as per the dataset.
- The link to the dataset: [https://www.kaggle.com/datasets/louise2001/survival-analysis-synthetic-data/](https://www.kaggle.com/datasets/louise2001/survival-analysis-synthetic-data/)
- This dataset represents entry dates, departure dates and other information about fictional clients of a life insurance company.
- You have the age at which the insured entered the contract, the age at which he left, and the reason: either death or withdrawal

| age_start_observed | age_end | date_end_observed | date_start_observed | is_censored | is_dead | is_truncated |
|---|---|---|---|---|---|---|
| 23 | 94 | 1950-01-01 | 2020-07-04 | True | False | True |
| 33 | 97 | 1950-01-01 | 2013-05-14 | True | True | True |
| 5 | 92 | 1950-01-01 | 2020-12-31 | True | True | True |
| 28 | 63 | 1950-01-01 | 1984-02-17 | True | True | True |
| 42 | 68 | 1950-01-01 | 1975-11-07 | True | False | True |
| 49 | 85 | 1950-01-01 | 1985-04-11 | True | True | True |
| 10 | 97 | 1950-01-01 | 2020-12-31 | True | True | True |
| 16 | 84 | 1950-01-01 | 2017-06-08 | True | True | True |

# Metrics

Write Performance (Data Loading Time)

- Metric: Elapsed time for loading a specified amount of data into the database.
- Calculation: Measure the time it takes to insert a dataset into the database.

Read Performance (Query Response Time)

- Metric: Elapsed time for executing a specific read query against the database.
- Calculation: Measure the time it takes to retrieve results for a representative read query.
- Considerations: Assess the efficiency of the database in handling complex read queries.

# Aggregation Queries

Based on the column names and descriptions, these are the first three queries I used for evaluating the read performance.

**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 2:**

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

**Query 3:**

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

# Kaplan Meier Estimator

- A non-parametric method used to estimate the survival function from censored data.
- Censoring occurs when the event of interest is not observed for all subjects in the study.
- The Kaplan-Meier estimator provides a step-function estimate of the survival probability over time.

*Time-to-Event:*

"time-to-event" is the difference between "age_end" and "age_start_observed."

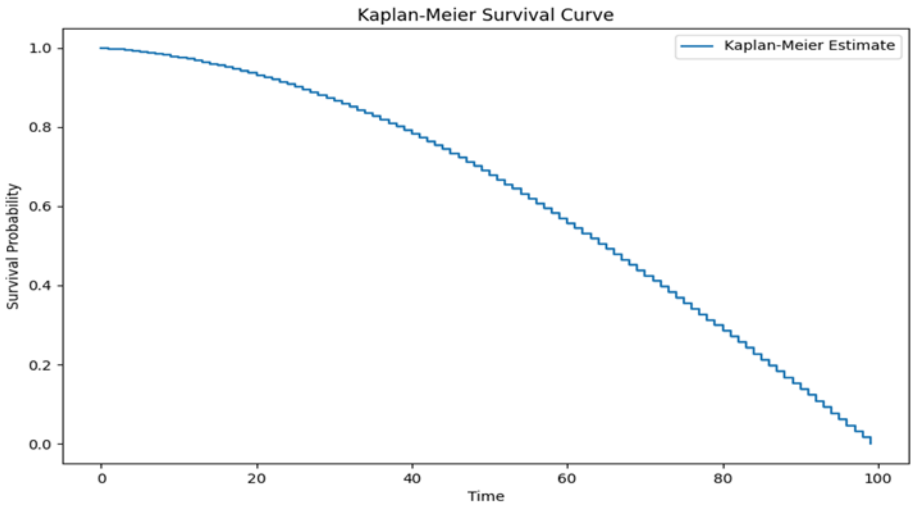
*Event Status:*

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.

**Steps:**

1. Calculate the survival probability based on the number of subjects at risk and the number experiencing the event.
2. Multiply the survival probabilities across time points to obtain the overall survival function.




**Output:**

A curve representing the estimated survival probability over time.

SJSU SAN JOSÉ STATE UNIVERSITY

# Log Rank Test

- A statistical test used to compare the survival curves of two or more groups to determine if there are significant differences in survival times.

**Steps:**

1. Calculate each group's observed and expected number of events for each time point.
2. The test statistic is based on the difference between observed and expected events, standardized by the variance.
3. To determine statistical significance, compare the test statistic to a chi-squared distribution.

**Output:**

A p-value is obtained, indicating whether there are significant differences in survival times between groups.

```
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: 2.50 seconds
```

The two groups are age_end and is_dead. The p-value is less than 0.005, so we can reject the null hypothesis.

# CoX Proportional Hazard

- A semi-parametric model assesses the relationship between survival time and one or more predictor variables.
- The hazard of an event is proportional across different levels of the predictor variables.

```
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
```

**Key Concepts:**

The Cox model estimates a hazard ratio for each predictor variable, indicating how the hazard changes relative to a reference level.

**Output:**

The Cox model estimates each predictor variable's hazard ratios, confidence intervals, and p-values.

"age_start_observed" has a negative coefficient, suggesting that as these variables increase, the hazard decreases.

"is_truncated" has a positive coefficient, indicating that individuals with truncated observations have a higher risk of death than those without truncated observations

# Implementation

**My Configuration:**

- OS: Windows 11 Home
- Processor:  Intel Core i7-8550U CPU
- Ram: 16 GB

**Language/Tools:**

- Used Docker for Database setup and installation.
- Docker is a software platform that allows you to build, test, and deploy applications quickly using containers.
- Every script is in Python programming language.
- Used different Python drivers for each database to make a connection
- I used pip (Python package manager) to install and use any dependency.
- Used Pandas for data generation and statistical query parts
- Pandas is a Python library used for working with data sets. It has functions for analyzing, cleaning, exploring, and manipulating data.

# Database Setup

- 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 start the containers using the terminal’s ‘docker-compose –up’ script.
- I have used docker for QuestDB, MongoDB, and Cassandra. For TimescaleDB, I installed PostgreSQL from the official website and added the TimescaleDB extension.

```
PS D:\Aarsh\SJSU\CS298\Project> docker-compose up -d
[+] Building 0.0s (0/0)                                                                    docker:default
[+] Running 3/3
 ✔ Container mongodb                 Started                                                         0.0s
 ✔ Container questdb                 Started                                                         0.0s
 ✔ Container my-cassandra-container  Started                                                         0.0s
```

pgAdmin
Management Tools for PostgreSQL

SJSU SAN JOSÉ STATE UNIVERSITY

# Data Generation

- Updated the Python script for the Kaggle dataset with parameters for a number of rows and database name.
- Created a dataset of 1 and 10 million in CSV format to load in TimescaleDB and QuestDB.

```
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.
```

- For MongoDB, I wrote another script to change the data format to JSON.
- The script that takes the CSV file using the 'read_csv' method, converts it in Pandas framework and uses 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 require a unique identifier column that helps in querying the data as it is filtered using the unique identifier.
- While MongoDB automatically includes a unique identifier column when the data is loaded, Cassandra doesn’t.
- I wrote another script that takes the CSV file, loads the data, adds a UIUD data column as the first column, and returns the updated CSV that can be uploaded in Cassandra.

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

SJSU SAN JOSÉ STATE UNIVERSITY

# Data Loading - MongoDB

- Used pymongo as my Python driver, which helped me connect to the database using Mongo Client.

- 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

Example script:

python mongo_load.py --database project1 --collection data --json_file data1.json

```python
# 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()
```

SJSU SAN JOSÉ STATE UNIVERSITY

# Data Loading - Cassandra

- Used Cassandra cluster as my Python driver
- I had first to create a table with the names of the columns, and after that, I could load data in the database.
- 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()
```

# Data Loading - TimescaleDB

- Used psycopg2 as my Python driver.
- 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.

Example script:

python timescale_load.py --database aarsh --table data1 --csv_file data1.csv --username postgres --password aarsh

```python
# 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()
```

# Data Loading - QuestDB

- Used HTTP REST API for database connection
- Used request as python library and requests.post method to send a POST request to the QuestDB server with the specified CSV file
- The POST response is then outputted as text
- 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.

```
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)
```

Example script:

python questdb_load.py http://localhost:9000 project1 data1.csv

# Query Execution : Aggregation Queries

```python
# 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)
```

```python
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)
```

```python
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)
```

```python
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)
```

# Query Execution: Statistical Queries

```python
# 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()
```

```python
# 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)
```

```python
# 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()
```

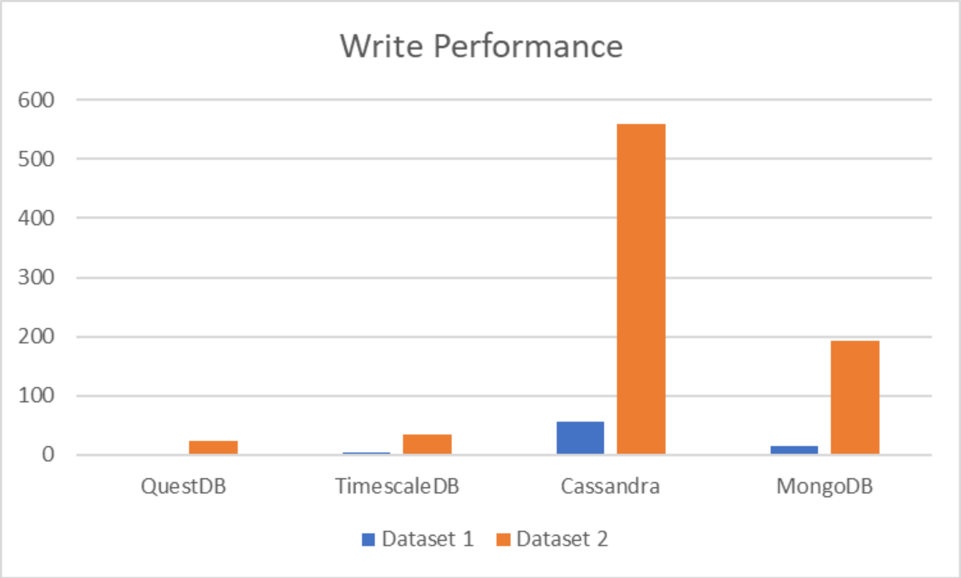
# Experiments: Write Performance

```
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 [uiud, 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 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 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 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.
```




QuestDB performs best, while Cassandra is the slowest for loading the data. Dataset 2 takes more time for NoSQL databases, explaining the indexing needed when inserting data as it is later used for querying.

SJSU SAN JOSÉ STATE UNIVERSITY

# Experiments: Read Performance

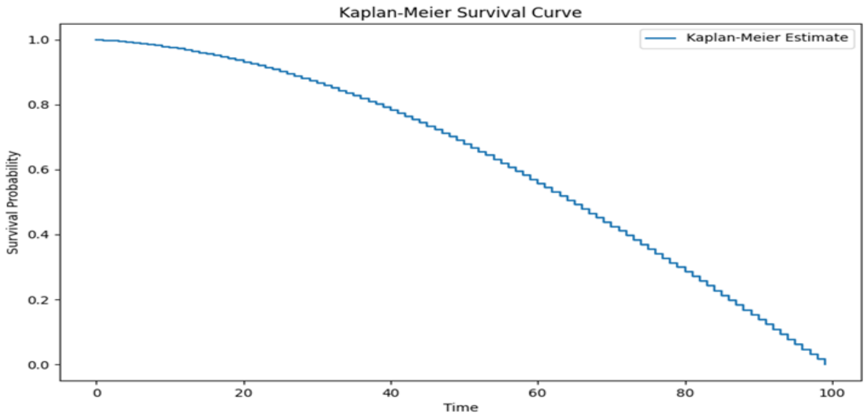
```
PS D:\Aarsh\SJSU\CS298\Project\query_execution\query1> python .\query1_questdb.py
"count"
1254

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
```

```
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
```

```
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
```

# Experiments: Read Performance

```
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 SQLAlch
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: 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
```

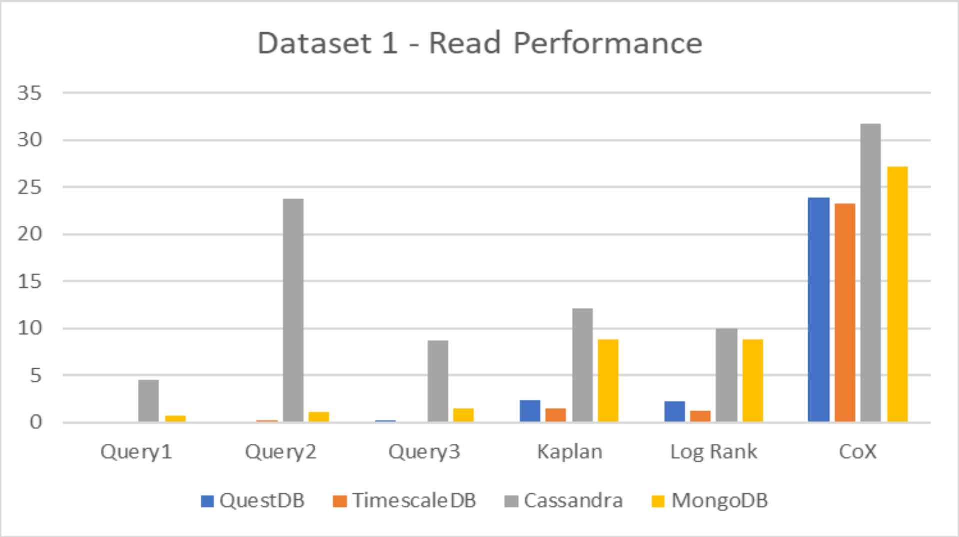


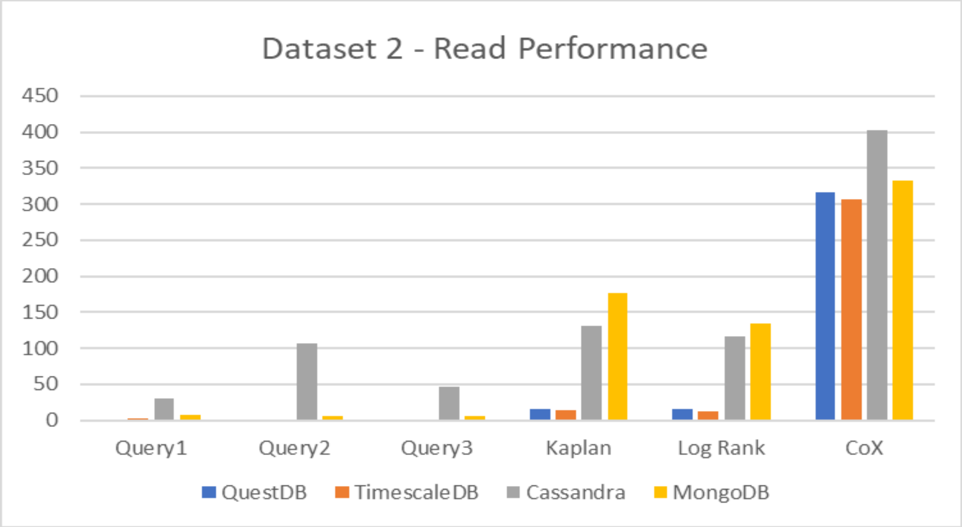

SJSU SAN JOSÉ STATE UNIVERSITY

```
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
```

```
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
```

# Experiments: Read Performance




**Dataset 1:** QuestDB performs better than TimescaleDB for the aggregation queries, but TimescaleDB wins for statistical queries. MongoDB performs slowly overall but is faster than Cassandra.

**Dataset 2:** QuestDB wins overall, but TimescaleDB gives an edge in statistical queries. MongoDB performs better than Cassandra for the aggregation queries but lags behind for some of the statistical queries.

# GitHub Source Code and Docker Image

**GitHub Repository:**

The git hub repository for my project is as follows :

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

It contains the folders and readme with the scripts to perform the benchmarking.

**Docker Image:**

This is the docker image of my project

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

The project can be pulled using docker pull. Example:

docker pull asp10/survivalbenchmark:latest

# Conclusion

- A new benchmarking suite for survival analysis data.
- The suite comprises read-and-write performance and 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 design and implementation are kept simple. So, adding new databases and metrics is made easy.
- The experiments conclude that Time series databases are better than NoSQL databases overall.
- QuestDB and TimescaleDB are column-oriented databases, so they are generally faster for analytical queries than row-oriented databases like MongoDB. Cassandra is a distributed database, which can be more scalable but also slower.
- The findings pave the way for further research and optimization efforts within the database community, like more databases and queries and looking into multi-node systems and scalability.

# Future Work

- Use real data to incorporate real-world scenarios.
- Modify configurations of supported databases to enhance the performance.
- Use more resources (RAM and storage ) and try out scalability with multi-node architecture.
- Support and evaluate new databases like NewSQL/Distributed databases.
- Include more diverse and complex query scenarios to simulate real-world use cases better.
- Extend benchmarking to real-world applications through collaborations with industry partners for practical insights.

# References

[1] 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

[2] 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

[3] TimescaleDB. (2023). TimescaleDB Documentation. Retrieved from https://docs.timescale.com/

[4] QuestDB. (2023). QuestDB Documentation. Retrieved from https://questdb.io/docs/

[5] 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.

[6] 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.

[7] MongoDB. (2023). MongoDB Documentation. Retrieved from https://www.mongodb.com/docs/

[8] Apache Software Foundation. (2023). Apache Cassandra Documentation. Retrieved from https://cassandra.apache.org/doc/latest/

# Thank you!

Questions?