2024

COM617 – Indus

Group 3 – Database Benchmarkir

Group Members: Danial Agha, Iona Pitt

GITHUB: HTTPS://GITHUB.COM/NICKTHORNE123/DB_BENCH

Contents

Introduction

The purpose of this COM617 project, is to compare different time-series database technologies and benchmark them against each other, for the customer, a company called Rockstone Data Ltd. This is because, they have recently adopted a new column orientated database (C-Store) called ClickhouseDB and they would like a demonstrator on the companies' website, showing how a C-Store compares to a 'traditional' database. These being PostgreSQL, TimescaleDB and ArcticDB with the aim of showing ClickhouseDB as a more effective database solution to previous, more 'traditional' databases.

Project Objectives

The key objective of this project was to create a one page 'dashboard' web application, which plots time series graphs, chosen by the user using a start and end datetime picker, on database the user has chosen to benchmark the results. Upon a user pressing a submit button, this faces the data for the plot and displays the time taken to fetch the data, the disk space taken up retrieving the data, the number of rows in the table and the GB of disk storage per million rules. Finally, there will also be a down-sampling toggle, allowing a user to input a down-sampling count between 1-5000. The definition for down-sampling is in Appendix A.

Requirements

As you can see from Appendix I, a requirements table is provided and includes a complete list of the main requirements that where needed to mark the project as complete in the eyes of the team and the customer - detailing things such as UI, database, Docker, documentation, and testing requirements.

Scope and Exclusions

The main work streams that aided development of the project, are in-built features on our GitHub repository. The work streams, where issues, set on the project's kanban board in the actions section of our repository.

The kanban board helped establish order with our tasks, enabling better collaboration and maximising our efficiency. This meant allowing the team to visualise the tasks that needed to be completed, without completing unnecessary work.

These columns were: "Backlog" (items that have been started), "Ready" (ready to be picked up), "In-Progress" (actively being worked on), "In-Review" (completed and being checked by a team member) and "Done' (the item has been completed).

The tasks on the kanban board where dynamically reviewed every week and refined towards the end of each sprint, to keep the scope of the project under control and on track to meet the project requirements. This also allowed the team to remove and add tasks, meaning we can better complete the objectives of the sprint and increase customer satisfaction by the end-result. This is because the customer had the ability to view the board and see our progress in delivering their product.

Finally, to enable collaboration within the team, we opted to use Microsoft (MS) Teams, which offered the team file storage, video conferencing and a work messaging chat, meaning the team could keep in touch on a regular basis and get the tasks completed, within the scope of each sprint.

Project Organisation and Structure

The work was split amongst the 5 group members, based on our strengths, in-order to complete the project as effectively as possible - assigning each member job roles. Because the team was quite small it meant one person picked up multiple job roles.

First, the Customer, as mentioned above, was the company Rockstone Data Ltd, which provided the set of requirements to the team for what they envisage for the product.

The project manager and scrum master were combined, undertaken by Josh Clarke. They are also the team's full stack developer as they have the most experience with Python and the Database side of the project.

Iona Pitt assumed the mantle of product owner and backend developer, because they have the most experience with SCRUM methodologies, whilst also having some experience in working with databases.

Danial Agha's responsibility was the software tester, this involved completing the UI testing and getting the CI/CD pipeline up and running for the repository. The reason for this is because they have previous experience on projects testing the software.

Kyle Roberts took on the role of UI developer. Again, this is because their strengths lie in creating UIs as it is one of their main job roles. They are also involved in generating documentation for the project.

Finally, Luke Wood oversaw customer relations, meaning they were the point of contact for the customer and most of the questions from the team to the customer go through them. Not only this but they also worked on documentation too.

Project Milestones and Management

The project was broken down into three sprints, detailed in Appendix G. These lasted 3 weeks, 4 weeks and 8 weeks respectively.

To manage the project, the project manager Josh implemented two weekly meetings. The first one, on Mondays at 5pm, was a quick 15 minute catch up, to discuss what each team member has been doing and what they will be doing during the week. The next meeting was Thursdays at 3pm, this lasted from anywhere between 30 minutes to 2 hours and was the main weekly meeting. This allowed us to help each other if needed, a catch up with our support tutor, another catch up meeting to dynamically review the work within the scope of the sprint.

As detailed in appendix J, we created a project plan to describe the strategy for the project. It included an outline of the scope and lifecycle of the project, the tools and languages we would be using, the roles of the team members, timescales, and project deliverables – ideas which were expanded on in later documentation.

As shown in Appendix H, the project milestones were written for us, giving the project structure, providing goals for the end of each sprint. The first sprint, was 3 weeks long and the main goal to the gather the projects requirements, detailed in the requirements section and was completed by 9th February.

The next sprint was the initial proof of concept (PoC), as agreed with the customer, this meant a very basic application with the required functionality, on only one database. Because this was a PoC, it meant the way it looked wasn't as refined as the minimal viable product (MVP). This ended with a demonstration to the support tutor and customer, where they signed off that they were happy.

The final sprint was creating the MVP, this involved a functional application that provides enough features for the customer to build upon and implement themselves. This meant supplying them with the accompanying documentation.

Monitoring and Evaluation

Evaluation was given by the customer at the end of sprint two, this is because, as mentioned in the project milestones and management section, our PoC was demonstrated to the customer, where they gave their feedback on what they liked about the product so far and how we can improve it.

Not only this, but because we could easily contact the customer, this allowed us to also gather constant feedback from them about different requirements and how it relates to the work that needed completing. This really helped the team as they gave us feedback on multiple issues such as, feedback on our UI design, issue with setting up our work environments and issues to overcome with each database.

On top of this, when new code/functionality was created for the programme, it had to be checked by another developer to make sure that it operated correctly, which in turn provided a fresh set of eyes to catch any errors that could have been missed by the developer creating the new code. This meant the new code was both monitored and evaluated before being merged into our master branch.

The final way the project was monitored, is through Appendix H, the test plan document. As detailed in Appendix H, these are automatic unit tests that make up part of the CI/CD pipeline and prevent new code being merged into the repository. This shows to the customer the tests that have either passed or failed and demonstrates to the customer that the team has produced a high-quality product for them, by identify any bugs.

Implementation

At the start of the project, a standards document was created to give the team guidance on how to name their branch, with coding and documentation conventions. This was to increase speed and efficiency in development by improving code readability and how it is maintained. Also, the 'Joel Test' was applied, to the quality of your software team by providing 12 yes-or-no questions to determine whether your programming team is up to par (Amaresan, 2021). Although some of the principles didn't apply, such as, do new candidates write code during their interviews.

The implementation of the product changed quite heavily from the initial designs to the MVP. This was due to conversations with the customer on features to keep/remove and how best to make the UI more user friendly.

As shown in Appendix A, compared to Appendix B, the design of the original UI changed drastically. With the most notable change being the alteration from a slider to get the downsampling number, to a text input (with plus and minus buttons) becoming more user friendly, as it would have been hard to get an exact number of values to reduce it by. Not only this but the removal of unnecessary buttons made the UI feel less cluttered.

One thing that didn't change much from Appendix a to the MVP, was the class structure on how to load each data into the application. It followed a simple structure of creating a class for each database we were planning on using, using a function to fetch the data and inputting that into their respective graphs and displaying of each calculation.

Finally, some databases only allowed a user to create a certain amount of data points before running into issue such as taking up too much disk space. A work around for this, which was agreed with the customer, was to take a three-month sample size (as opposed to a year) and multiple each benchmarking calculation by four to show it as if it was a years' worth.

Results

To make sure we were producing results for the customer, as previously eluded too, we had regular contact with the customer providing opportunities and feedback on what they want to get out of the product we will deliver to them. Not only this, but as part of the set up for the project, the customer was kind enough to lend us a repository for code allowing use to enhance our workflows and provide version control capabilities.

Because the repository was owned by the customer, it meant that, they had direct oversight over pull requests, the kanban board and documentation, allowing to raise issues faster and checkout the codes functionality for themselves.

Another way results where shown was through the requirements table detailed in Appendix I. As you can see, there are two main columns, the 'Requirement' column, this provides a description of what needs to be implemented to show the requirement as complete and the 'Completed?' column. This has a simple yes/no structure to show if the requirement has been met. There is also a non-applicable column, to show if the original requirement has been removed or the customer has given permission to not complete that requirement.

Finally, results can be measured with the stats produced by the benchmarking data, seen in Appendix C through to F. As shown, from the stats, ClickhouseDB comes out on top in almost all metrics when compared to more 'traditional' databases, except RAM usage when trying to read the data, with it coming in third behind TimescaleDB and PostgreSQL. Given the data for each database was over the same amount of time, it shows how effective a C-store is compared to a traditional database, taking up less disk usage and the least time to read and write data, to and from the database.

Conclusion

In conclusion, the team believes this project to have been a success in not only proving, for the customer, that ClickhouseDB is a more effective database solution compared to 'traditional' databases but also the quality of work that was produced by the team. This is because, the team stuck to the project plan and the SCRUM methodology, keeping communication clear between not only the team but also the customer. We also met almost all the requirement set out by the customer with exceptions to a few of the stretch goals because, we ran out of time to implement them.

However, I think a big place we could improve on in the future is when gathering the requirements, keeping them as close as possible to the original requirements set out and not allowing the customer to dictate requirements mid-development. Not only this, but due to other workloads, it meant we missed events the customer invited us too. As a team we feel these could have been beneficial to allowing closer ties with the customer and that more effort could have been made to be free during these events.

Finally, as team, this was a project we enjoyed working on and felt the team 'gelled' well. This allowed every team member to have a role that suited their strengths but also allowed each member an opportunity to work on their weaknesses and improve overall, as software engineers.

WC – 2199(08/05/24 @ 1440)

2200 – 2199 = max words left: 1

References

AMARESAN, S., 2021. The 2021 Guide to The Joel Test for Programming In: *Swetha Amaresan's weblog.* 04 February 2021 [viewed 29 April 2024]. Available from: https://blog.hubspot.com/service/joel-test

Appendices

Appendix A – Design Document

Abstract / Executive Summary

Rockstone Data is a software application development company founded in 2018 by Nick Thorne.

The company has recently utilised a new column-orientated database called ClickHouseDB to host very large time series data tables. This class of database outperforms traditional row-orientated databases in both speed and storage.

The project aim is to create an interactive demonstrator running on Rockstone Data's website clearly demonstrating these benefits vs 'traditional' or row-orientated databases.

List of contents

Introduction

Purpose of the document

This document describes the architectural design and detailed design for the project.

This Design Document is to be a living document for the duration of the project. It will be updated and expanded upon, at least at the start of each stage or iteration, and as required throughout the project's software development lifecycle.

Scope of the document

This document covers all aspects of the software design.

Definitions, acronyms and abbreviations

The Project Plan Document [1] contains the definitions, acronyms and abbreviations for this project. The glossary will also be the data dictionary, so there is a common understanding of terms and consistent naming of objects within the development.

Project Initiation Document: Summary of important information about the project, including its context, scope, milestones, objectives and requirements.

Proof of Concept: Gathering evidence to gauge the feasibility of a project.

ClickHouseDB: An open-source database which is often used for queries involving real-time and historical data.

Postgres (PostgreSQL): An open-source relational database often used for data storage for web, mobile, geospatial and analytics applications.

TimescaleDB: A database engine that uses Postgres for applications requiring time series, vector, events and analytics data.

MongoDB: A document database often used for scaling throughout application development.

Docker / Docker Compose: A tool used to run multi-container applications. Each container provides different back-end functionality for the application.

DBMS (Database Management System): Software systems used to store and fetch data.

AWS (Amazon Web Services): An infrastructure platform which provides on-demand cloud computing and APIs for deploying applications.

EC2 (Amazon Elastic Compute Cloud): A compute platform used for deploying applications on the cloud.

OLAP (Online Analytical Processing): A form of computing whereby the user can extract and analyse data from different perspectives.

OLTP (Online Transaction Processing): A form of data processing whereby the user can execute many transactions at the same time.

Snowflake: Software as a service which can be used for data warehouses, lakes, sharing of data, data science and engineering, and data application management.

Data Dog: Software as a service which can be used for monitoring of servers, databases, tools and services.

DigitalOcean: Software and infrastructure as a service which can be used for deploying web applications.

ArcticDB: An embedded and serverless database used for storing and processing Pandas DataFrames.

Conventions

The Standards Document [2] contains naming conventions and terminology.

Branch Naming Conventions:

The GitHub development flow is used so as to allow individuals to be able to create a branch for issues they are currently working on.

The branches are named like the following example: nt_123_short_desc.

Breaking this down,

- nt will be the developer's initials.
- 123 will be the Issue Number.
- There will then be a one- or two-word description of the issue.

Joel Test:

The project will try and adhere to the 'Joel Test' of good software practice.

The Joel Test is a very simple and quick test that rates the quality of your software team. Rather than including open-ended responses, this test has 12 yes-or-no questions that determine whether your programming team is up to par. A score of 12 is perfect. 11 is considered tolerable, and 10 or below is unacceptable. These 12 questions are as follows:

1. Do you use source control?
2. Can you make a build in one step?
3. Do you make daily builds?
4. Do you have a bug database?
5. Do you fix bugs before writing new code?
6. Do you have an up-to-date schedule?
7. Do you have a spec?
8. Do programmers have quiet working conditions?
9. Do you use the best tools money can buy?

10. Do you have testers?

11. Do new candidates write code during their interview?

12. Do you do hallway usability testing?

Coding Conventions:

The coding conventions for this project will follow the PEP 8 Conventions - This is documentation that provides guidelines and best practices on how to write Python code. This can be viewed here: PEP 8 Documentation.

When coding standards are properly defined and implemented, developers, even those who have just joined the team, can easily find their way around the code base. Ideally, wanting our source code to look like a single developer has written, debugged, and maintained it.

Code Layout

File Header - File headers will be at the top of every file and include the following information:

- FileName - Will be the name of the file.
- FileType - Will be the type of file i.e., py (python), cs (c#), js (JavaScript)
- Created By - Who it was created by. Formatted as Surname, First Name
- Created On - The date and time it was created on in a dd/mm/yyyy hh:mm:ss AM/PM format
- Last Modified - The date and time it was last modified on in a dd/mm/yyyy hh:mm:ss AM/PM format
- Description - Brief description on what the file does

Stakeholders

The list of stakeholders is contained within the Project Plan [1].

Role	Stakeholder Name	Details
Project Sponsor	Nick Thorne	Director and Consultant Software Engineer at Rockstone Data Primary contact for the project
Support Tutor	Martin Reid	Lecturer and Apprenticeship Advisor at Solent University Must be provided with project progress update after each Sprint

Architectural Design

System Overview

The Requirements Document contains information on an overview of the final product [3].

The system will be a Dashboard web application hosted on Docker containers, featuring a line plot depicting comparison data from ClickHouseDB. Our group will design, develop, test and deploy the software according to the requested requirements supplied by the project sponsor, Rockstone Data.

Design Method

The information for the design method can be found within the Project Plan [1].

Python 3.11 was utilised for the Application back-end and front-end code.

An agile methodology utilising three iterative sprints was incorporated.

It will adopt some elements from Scrum, including the roles of Product Owner and Scrum Master, and the existence of Scrum ceremonies such as Sprint Planning, Sprint Review and Sprint Retrospective.

GitHub Project Board, GitHub Issues Board and Microsoft Word were used for Software Project Management

A GitHub Code Repository was used for Version Control

Microsoft Word and Excel were used for tracking Requirements.

Microsoft Word and Wireframe.cc were used for Analysis and Design

Implementation was ensured via the use of; Microsoft Visual Studio Code, Streamlit OR Plotly Dash, Python pip, Python virtual environments (venv), ClickHouseDB, TimescaleDB, PostgreSQL, MongoDB, Snowflake and Data Dog

Testing was done via Microsoft Visual Studio Code

Deployment carried out via Docker Desktop and Docker Compose

Support and Maintenance was carried out through GitHub Wiki Pages

Component Overview

The system will consist of 4 databases (PostgresqlDB, ClickhouseDB, ArcticDB, TimeScaleDB)

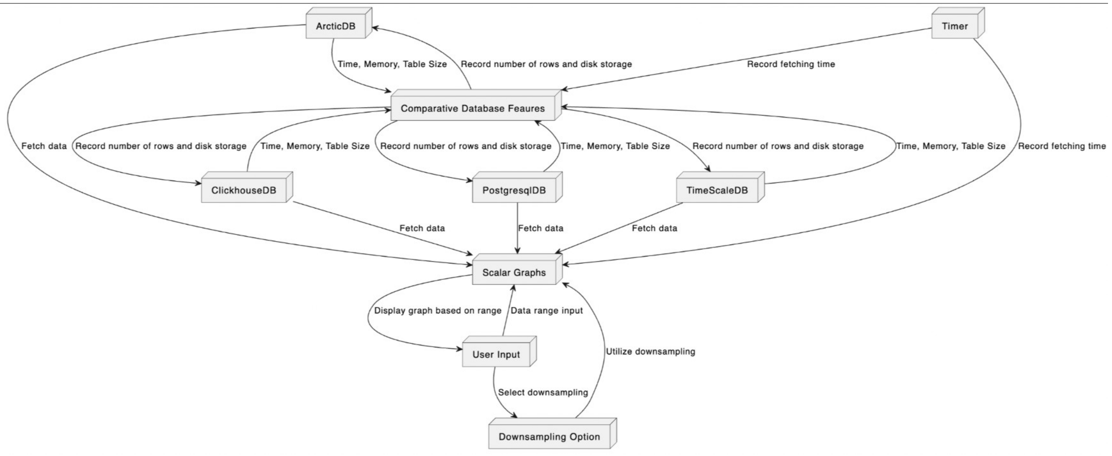
Data from the databases will be fetched and plotted on a scaler vector.

A timer will record how long the fetching process takes for each database.

The data range will be inputted by a user and the graph will reflect that range.

A downsampling option is available and will be utilised when selected.

The number of rows and disk storage required for each database will be recorded and displayed along with the timer.



Deployment View

Deployment is covered within the Project Plan [1].

Deployment will be completed via the use of Docker Desktop and Docker Compose and once deployed will run on all devices that can utilise streamlit.

It may later be deployed using Amazon Web Services EC2 or DigitalOcean

Scenarios

The system is able to display the differences between ClickHouseDB's column-orientated format and a traditional database's row-orientated format using a line plot. In addition, the application can also display this information through geo-heatmaps or data analysis using Snowflake and Data Dog.

Large data sets (for example stock market data) are fetched and then displayed as a scalar plot, this information has the option to be downsampled and can be displayed over specific time instances (24 hours, 7 days, 1 year). This is carried out through various databases which are then compared through key metrics such as 'Time Elapsed', 'Disk Storage Used' and 'Number of Rows Required', this information is then displayed to the user. Stock Market Data is a suitable data set for this due to its large and continuous nature.

Development Environment

Environment information can be found within the Project Plan [1].

The final product shall be able to run according to the advised minimum specifications for running Python 3.11, therefore, the product will not be able to run on Windows 7 or earlier. It will run better on computers with larger RAM and processing power due to the Machine Learning techniques used.

Once deployed, the application will run on all devices which can run Streamlit.

Testing Environment

Test Environment can be found within the Project Plan [1].

Test Progress [5] demonstrates implementation of testing.

1 - Reviews and Inspections

All code must be placed into a pull request on GitHub before being merged into the master branch. The pull request must be reviewed and approved by at least one other team member. Commit messages that summarise the code changes must be added to branch commits and to the resulting pull request.

2 - Testing

The software development team will perform unit testing using pytest, system testing and acceptance testing. All software components must undergo unit testing, and the test results, harnesses and input data must be documented in the test document. The test document will also include the system and acceptance test strategies and test scripts.

3 - Lessons Learned

Lessons learned will be discussed throughout the project and documented as part of the Sprint Retrospective.

The Requirements for testing are found in the Requirements Document [3].

Test data shall contain visible results on a 24-hour basis

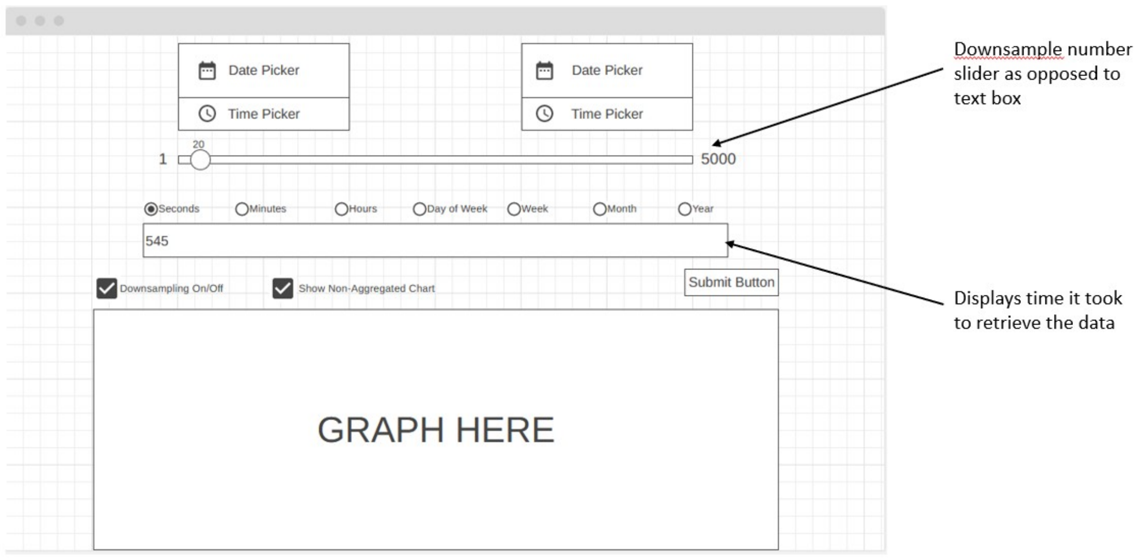
Test data shall contain visible results on a 1-month basis

Test data shall contain visible results on a 1-year basis

Detailed Design

User Interface

The User Interface Requirements can be found within the Requirements Document [3].



- The application shall be a multi-page Dashboard web application of at least 2 pages.

- The user interface shall display a line plot of a scalar value over time.

- The user interface shall display a line plot of a start / end datetime picker.

- The user interface shall display a line plot of a database source picker.

- The user interface shall have a ‘Submit’ button which fetches the data for the line plot.

- The user interface shall have a ‘Downsampling On/Off’ toggle.

- The user interface shall have a ‘Downsampling’ text entry form.

- The user interface may have a multidimensional class of data displayed as a geo-heatmap or spectrograph or surface plot.

- The graphs may have additional animations or orientation controls.

- The multidimensional data may be shown below the line chart.

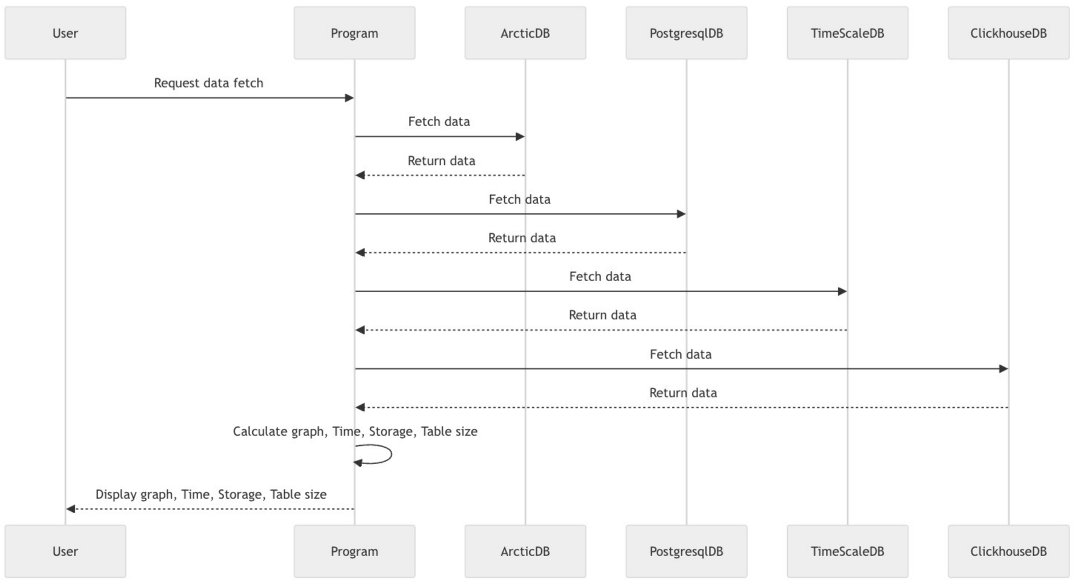
- The user interface may compare costs using Snowflake or Data Dog.

- The application may benchmark writing data to large tables.

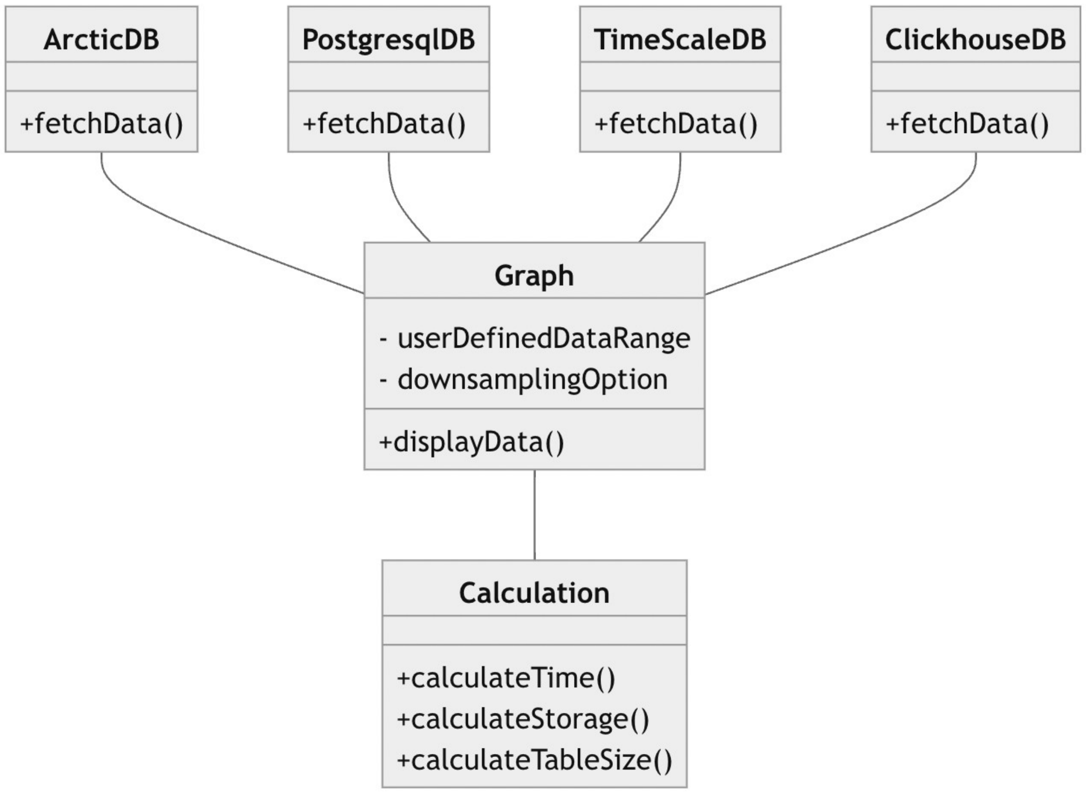
- The application may benchmark the ArcticDB time-series collections

Class Model

Sequence diagram:



Class diagram:



Database Design

Database Benchmarking [4] contains details of the database design.

In the Design phase, the organisation will determine what aspects of the database need to be measured - such as performance, scalability, availability, elasticity or cost – and how to

measure them. After this, the organisation will choose an appropriate benchmark and workload.

Lastly, test runs of the benchmarking process will occur in several iterations to obtain an idea on what reliable results should look like.

Databases used:

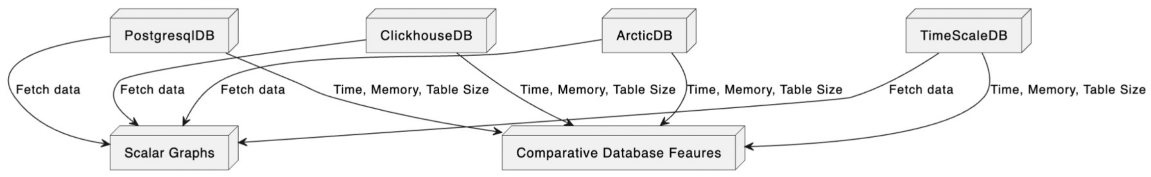
ArcticDB

ClickhouseDB

PostgresqlDB

TimescaleDB

The databases are not relational, they are accessed via fetching before the data is displayed on a graph. The relationship between the database and the graph is 1-1.



Design Document References / Bibliography

docs/RequirementsDocument.docx

[1] GitHub. (n.d.). db_bench/docs/Standards.md at master · NickThorne123/db_bench. https://github.com/NickThorne123/db_bench/blob/master/docs/RequirementsDocument.docx

[2] GitHub. (n.d.). db_bench/docs/Standards.md at master · NickThorne123/db_bench. https://github.com/NickThorne123/db_bench/blob/master/docs/Standards.md

[3] GitHub. (n.d.). db_bench/docs/Standards.md at master · NickThorne123/db_bench. https://github.com/NickThorne123/db_bench/blob/master/docs/RequirementsDocument.docx

[4] GitHub. (n.d.). db_bench/docs/Standards.md at master · NickThorne123/db_bench. https://github.com/NickThorne123/db_bench/blob/master/docs/Research/Database Benchmarking.docx

[5] GitHub. (n.d.). db_bench/docs/Standards.md at master · NickThorne123/db_bench. https://github.com/NickThorne123/db_bench/blob/master/tests/test_postgress.py

Appendix B – User Guide

List of contents

1 Setup [1]

1.1 Installing and Using this Repo

This project has four main components initially,

1. the Flask/Plotly/Dash python module.
2. the Postgres Docker container
3. the two Clickhouse DB containers

The steps to install and run it are:
Git clone this repo then create the virtual environment and install the packages:

```
cd db_bench

python -m venv .venv

pip install -r requirements.txt
```

Create local versions of the config files

```
cp .vscode/launch.json.example .vscode/launch.json
cp .env_example .env

cp postgres.env_example postgres.env
```

Enable all user permissions

```
cd etc\clickhouse-server
```

In *chuser.xml*, add the grant to the user profile:

```
<chuser>
  <profile>ch_profile</profile>
  <networks>
      <ip>::/0</ip>
  </networks>
  <password>chuser_pwd</password>
  <quota>ch_quota</quota>
  <grants>
```

```
    <query>GRANT ALL ON *.*</query>
    </grants>
  </chuser>
```

This will create a *subdirectory .venv* containing a virtual Python environment isolating the project from other projects on your computer. You may want to move across to using the poetry package manager as one of your deliverables. It handles dependencies in a more intelligent way than venv and pip.

If you're using VS Code, note the *.vscode* directory which contains an entry allowing you to start and debug the project.

1.2 Configuring ClickHouseDB

You can try this now but will likely get errors about not being able to connect to the database. So the next step is to run up the Docker containers for Clickhouse and configure them. You will need Docker Desktop installed on your machine.

(The following commands are to be entered in your machines terminal)

cd db_bench

docker-compose up ch_server ch_client

This will build your containers and run them locally. You can see their status with docker container ls -a.

Now we need to check that the clickhouse database is running locally, choose your preferred SQL client. I like to use DBeaver. Create a connection of type Clickhouse on localhost, port 8124 (specified in *docker-compose.yml*), user *chuser* and password *chuser_pwd* (specified in */etc/clickhouse-server/users.d/chuser.xml* and *.env*) and we start with database default.

You should now be able to connect to your locally running Clickhouse docker container. When you are connected, open an SQL terminal and create the database. Disconnect and reconnect as this will refresh DBeaver - the new database will not show up on the GUI if you don't do this.

Create a new SQL script and run the below command;

CREATE DATABASE ts_db;

Now create the demo timeseries table with the following SQL command. This only creates a small table. Once you're sure of the installation, change all the t*oDate(2021* to *toDate(2022* to generate a year and 10 minute's worth of 1 second time series data. Once again, refresh DBeaver.

```
CREATE TABLE ts_db.demo_ts

ENGINE = MergeTree

ORDER BY tuple()
AS

SELECT  toDateTime(arrayJoin(range(toUInt32(toDateTime('2021-01-01  00:00:00')),

toUInt32(toDateTime('2022-01-01 00:10:00')), 1) )) as cdatetime,

    toSecond(toDateTime(arrayJoin(range(toUInt32(toDateTime('2021-01-01   00:00:00')),

toUInt32(toDateTime('2022-01-01 00:10:00')), 1) ))) +

    toMinute(toDateTime(arrayJoin(range(toUInt32(toDateTime('2021-01-01   00:00:00')),

toUInt32(toDateTime('2022-01-01 00:10:00')), 1) ))) +

    2  *  toHour(toDateTime(arrayJoin(range(toUInt32(toDateTime('2021-01-01  00:00:00')),

toUInt32(toDateTime('2022-01-01 00:10:00')), 1) ))) +

    5  *  toDayOfWeek(toDateTime(arrayJoin(range(toUInt32(toDateTime('2021-01-01
```

00:00:00')), toUInt32(toDateTime('2022-01-01 00:10:00')), 1)))) +

*8 * toWeek(toDateTime(arrayJoin(range(toUInt32(toDateTime('2021-01-01 00:00:00')), toUInt32(toDateTime('2022-01-01 00:10:00')), 1)))) +*

*12 * toMonth(toDateTime(arrayJoin(range(toUInt32(toDateTime('2021-01-01 00:00:00')), toUInt32(toDateTime('2022-01-01 00:10:00')), 1)))) +*

*20 * (toYear(toDateTime(arrayJoin(range(toUInt32(toDateTime('2021-01-01 00:00:00')), toUInt32(toDateTime('2022-01-01 00:10:00')), 1))))-2021) as ts_values*

Make sure all the packages in chdemoapp.py have been installed, and then you can start the app and it should connect to the ClickHouse database and show some data. This can now also be done with the db_bench.py application.

1.3 Configuring PostgreSQL

To configure Postgres, run the command docker compose up db. This will create the *psql_db* container. Go to DBeaver and create a new connection to a Postgres database on port 5432 with the username *postgres* and password *postgres*.

Once connected, create a table with the SQL command

```
CREATE TABLE demo_ts (  cdatetime DATE,  ts_values
INTEGER

);
```

and generate some data with

```
WITH time_series AS (

        SELECT * FROM
          generate_series( '2021-01-01
          00:00:00'::timestamp,

          '2022-01-01 00:10:00'::timestamp,

          '1 second'::interval

        ) as cdatetime

),

random_values AS (

   SELECT random() * 100 AS ts_values -- Adjust range as needed

   FROM generate_series(1, 5) -- Generate 5 random values

)

INSERT INTO demo_ts (cdatetime, ts_values)

SELECT time_series.cdatetime, random_values.ts_values
FROM time_series
```

```
CROSS JOIN random_values;
```

Lastly, in order to display the data on the Streamlit app, navigate to your *.streamlit* folder (default is at *C:\Users\Username\.streamlit)* and create a *secrets.toml* file. Add the following code:

```
CREATE TABLE demo_ts (
  cdatetime DATE,
  ts_values INTEGER

);
```

And generate some data with

```
 WITH time_series AS (
     SELECT * FROM
       generate_series( '2021-01-01
       00:00:00'::timestamp,

       '2022-01-01 00:10:00'::timestamp,
       '1 second'::interval
     ) as cdatetime
),
random_values AS (
    SELECT random() * 100 AS ts_values -- Adjust range as needed
    FROM generate_series(1, 5) -- Generate 5 random values

)
INSERT INTO demo_ts (cdatetime, ts_values)
SELECT time_series.cdatetime, random_values.ts_values
FROM time_series

CROSS JOIN random_values;
```

1.4 Configuring TimescaleDB

To configure Timescale, run the command docker compose up timescaledb. This will create the *tmscl_db* container. Go to DBeaver and create a new connection to a Timescale database on *port 5433* with the username *postgres* and password *postgres*. (Timescale uses Postgres)

Once connected, create a table with the SQL command.

```
CREATE TABLE demo_ts (
     cdatetime DATE,
     ts_values INTEGER

);
```

and generate some data with

```
WITH time_series AS (
          SELECT * FROM
           generate_series( '2021-01-01
           00:00:00'::timestamp,

           '2021-06-01 00:10:00'::timestamp,

           '1 second'::interval

          ) as cdatetime

),

random_values AS (

   SELECT random() * 100 AS ts_values -- Adjust range as needed

   FROM generate_series(1, 5) -- Generate 5 random values

)

INSERT INTO demo_ts (cdatetime, ts_values)

SELECT time_series.cdatetime, random_values.ts_values
FROM time_series

CROSS JOIN random_values;
```

1.5 Configuring ArcticDB

Make sure Clickhouse DB is set up before configuring the ArcticDB database To first install ArcticDB locally, run the command *pip install arcticdb*.

Create an Amazon AWS Account and set up an S3 bucket. Within the project *.env* file, add the URL for the S3 bucket '*s3s://s3.<REGION_NAME>.amazonaws.com:<BUCKET_NAME>?aws_auth=tru e*' to ARCTIC_URL.

To access the S3 bucket, IAM User Access can be set up on AWS. See https://docs.arcticdb.io/4.4.1/#getting-started for more information. The other option which is less recommended is making the bucket publically available by disabling the 'Block public access' settings and adding a statement to the 'Bucket Display'. These options can be found within the AWS bucket options.

Public Bucket Policy:

{ "Id": "BucketPolicy", "Version": "2012-10-17", "Statement": [{ "Sid":
"AllAccess", "Action": "s3:", "Effect": "Allow", "Resource":
["arn:aws:s3:::<BUCKET_NAME>",
"arn:aws:s3:::<BUCKET_NAME>/"],

"Principal": "" }] }*

Run the *arcticdb_setup.py* file by running *python .\arcticdb_setup.py* from the root folder (This may take some time). This sends the same dataset from the Clickhouse database to the Arctic storage.

1.6 Troubleshooting

ImportError: cannot import name *'load_dotenv'* from '*dotenv'*

If you get the error message shown above, install the package *python_dotenv* instead of *dotenv*. You do not need to change the import name, as dotenv will automatically be installed with *python_dotenv*.

toml.decoder.TomlDecodeError: Key group not on a line by itself. (Line 1 column 1 char 0)

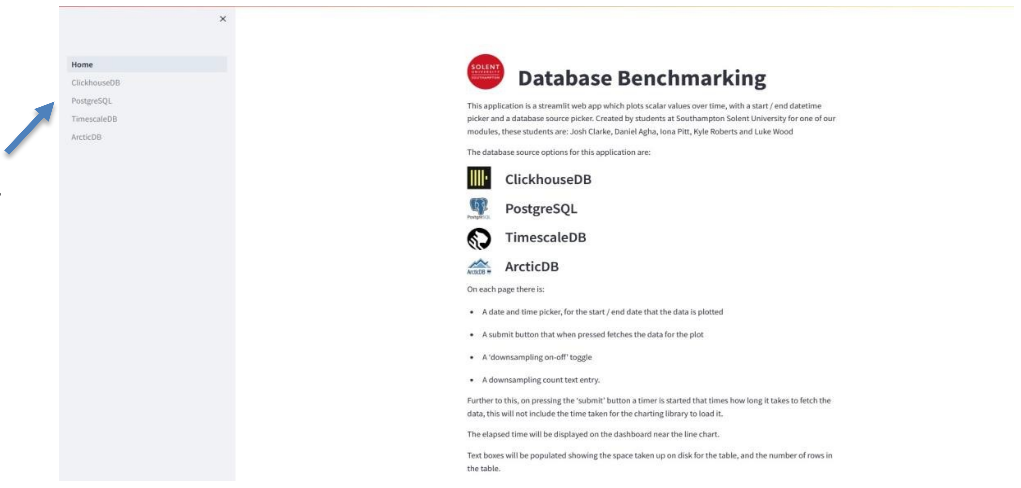
If you get the error message shown above, go to your *.streamlit* folder on your computer (*default is at C:\Users\Username\.streamlit*) and delete the *config.toml* file.

2 Page Interface

2.1 Home Page

The home page has a brief introduction and run down of the project as well as navigation buttons for the 4 other pages that allow you access each database benchmarking page

Navigation Options



Database Benchmarking

This application is a streamlit web app which plots scalar values over time, with a start / end datetime picker and a database source picker. Created by students at Southampton Solent University for one of our modules, these students are: Josh Clarke, Daniel Agha, Iona Pitt, Kyle Roberts and Luke Wood

The database source options for this application are:

ClickhouseDB
PostgreSQL
TimescaleDB
ArcticDB

On each page there is:

- A date and time picker, for the start / end date that the data is plotted
- A submit button that when pressed fetches the data for the plot
- A 'downsampling on-off' toggle
- A downsampling count text entry.

Further to this, on pressing the 'submit' button a timer is started that times how long it takes to fetch the data, this will not include the time taken for the charting library to load it.

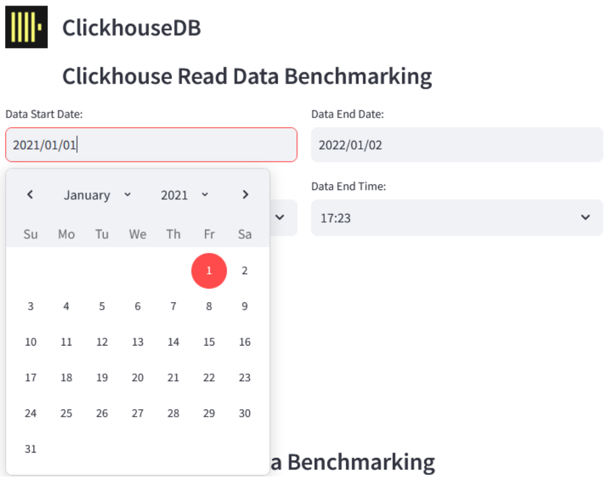
The elapsed time will be displayed on the dashboard near the line chart.

Text boxes will be populated showing the space taken up on disk for the table, and the number of rows in the table.

2.2 Database Pages

Each database page will consist of:

- Date and time pickers (start and end of the database sample)


ClickhouseDB

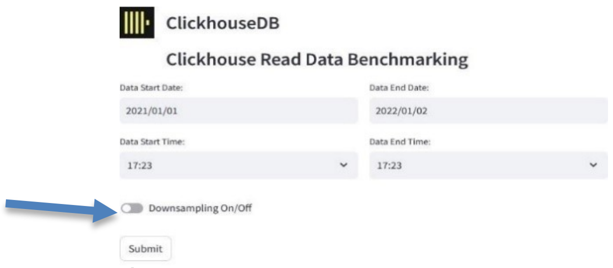
Clickhouse Read Data Benchmarking

Data Start Date: 2021/01/01

Data End Date: 2022/01/02

Data End Time: 17:23

- The downsampling on/off toggle (this will show the downsampling count text entry)
  Off:


ClickhouseDB

Clickhouse Read Data Benchmarking

Data Start Date: 2021/01/01

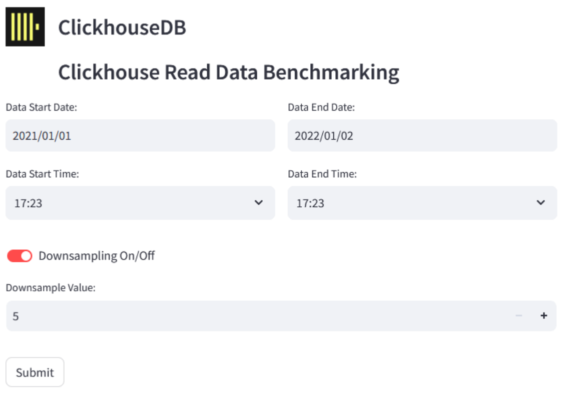
Data End Date: 2022/01/02

Data Start Time: 17:23

Data End Time: 17:23

Downsampling On/Off

On:


ClickhouseDB

Clickhouse Read Data Benchmarking

Data Start Date: 2021/01/01

Data End Date: 2022/01/02

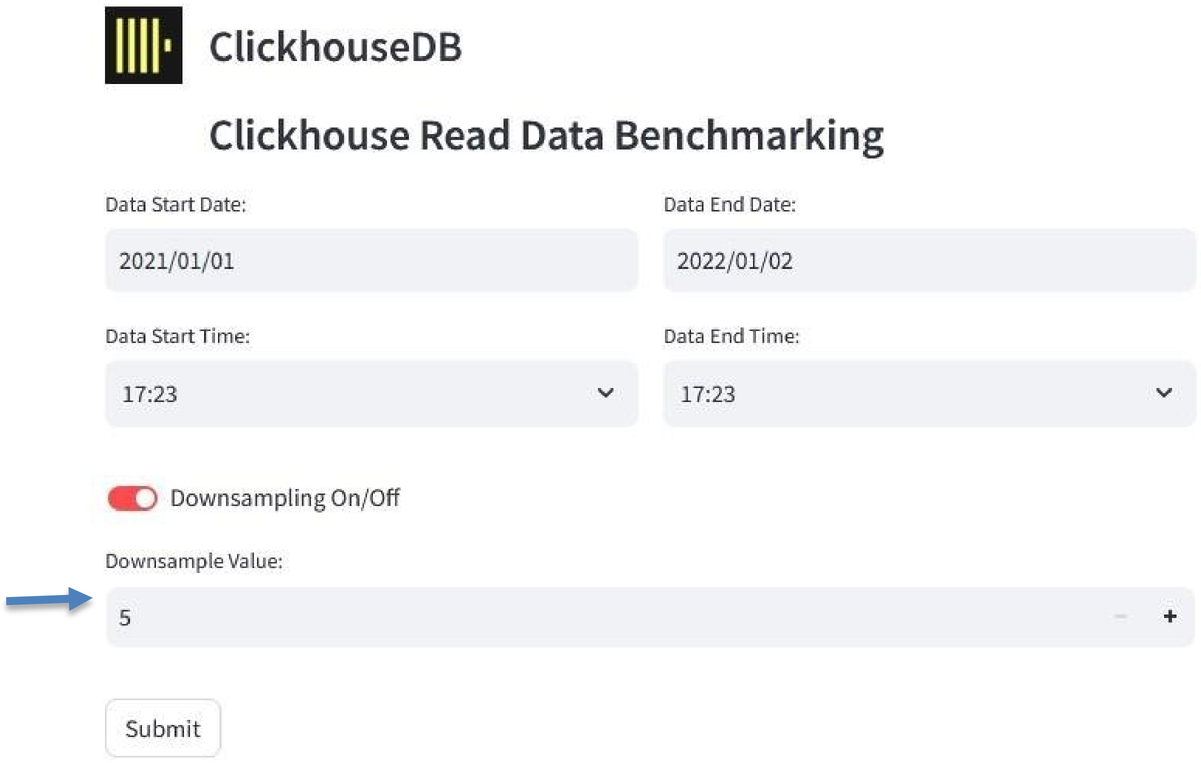
Data Start Time: 17:23

Data End Time: 17:23

Downsampling On/Off

Downsample Value: 5

- The downsampling count text entry. (Limited to 5000 samples)


ClickhouseDB

Clickhouse Read Data Benchmarking

Data Start Date: 2021/01/01

Data End Date: 2022/01/02

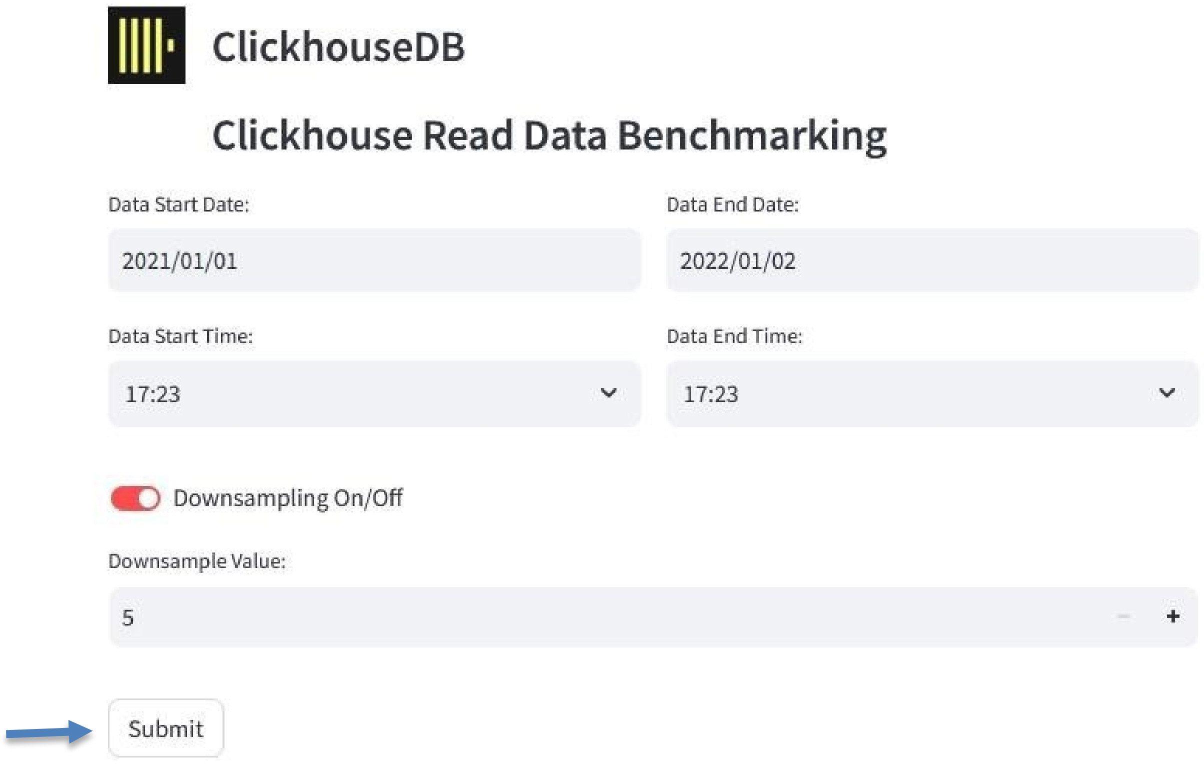
Data Start Time: 17:23

Data End Time: 17:23

Downsampling On/Off

Downsample Value: 5

- The submit option that when pressed, will begin the process of fetching the data


ClickhouseDB

Clickhouse Read Data Benchmarking

Data Start Date: 2021/01/01

Data End Date: 2022/01/02

Data Start Time: 17:23

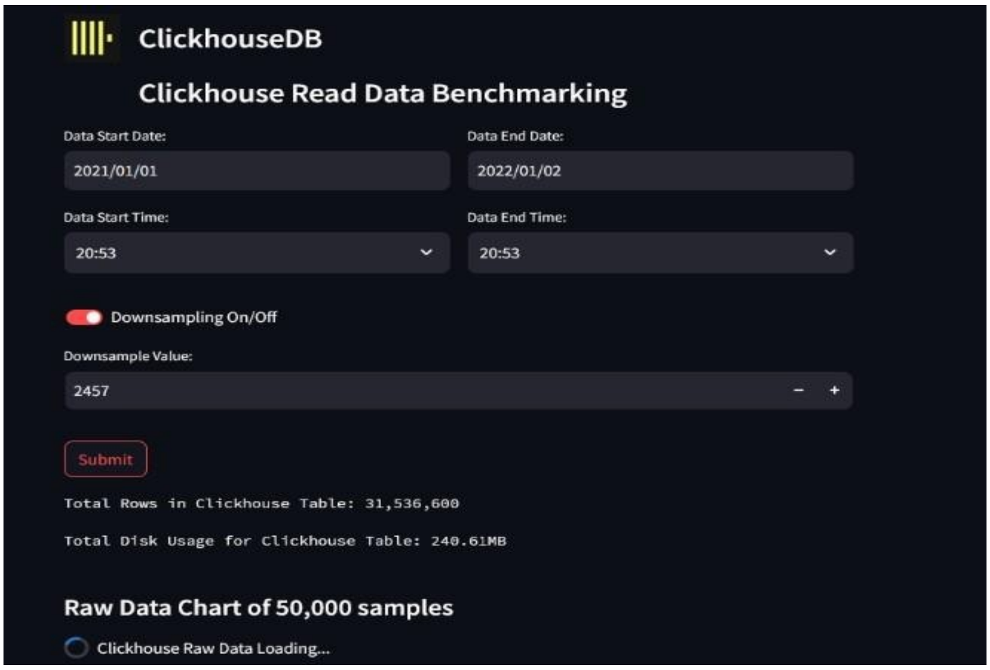
Data End Time: 17:23

Downsampling On/Off

Downsample Value: 5

Once the submit button has been pressed:

- A timer starts, that times how long it takes to fetch the data (this does not include the time taken for the charting library to display the data on the graph).
- The elapsed time which is displayed on the dashboard near the line chart.
- Text boxes populated, showing the space taken up on disk for the table and the number of rows in the table.
- A text box showing the total database size on disk in MB..
- A graph of the fetched data



ClickhouseDB

Clickhouse Read Data Benchmarking

Data Start Date: 2021/01/01

Data End Date: 2022/01/02

Data Start Time: 20:53

Data End Time: 20:53

Downsampling On/Off

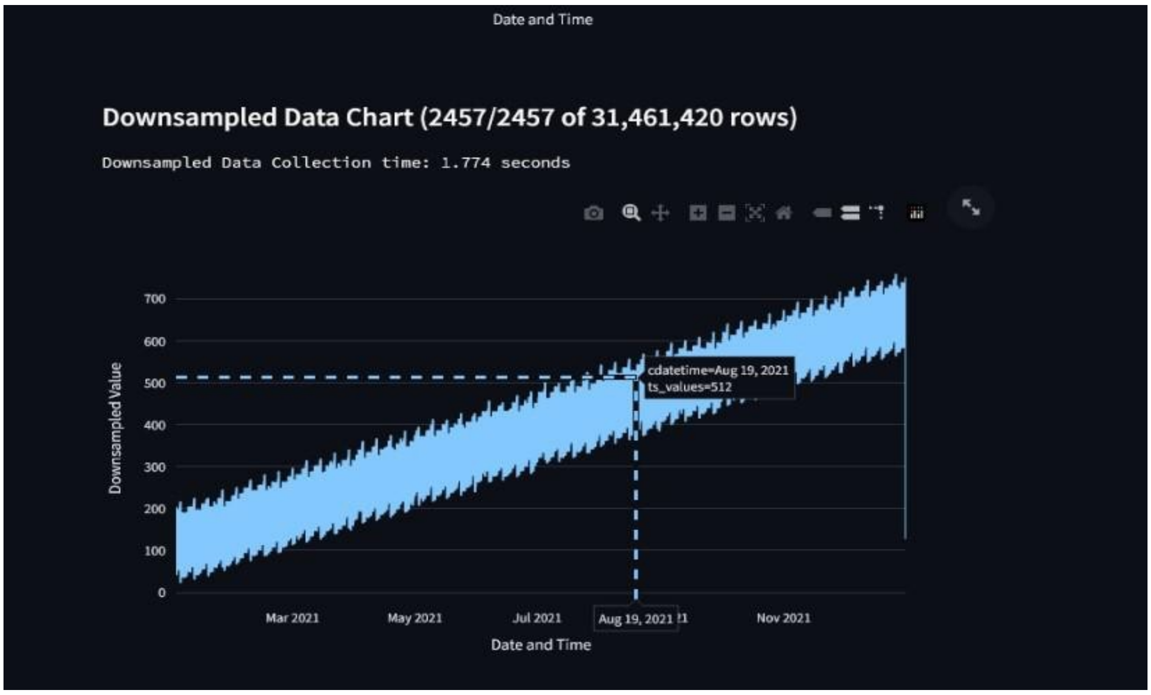
Downsample Value: 2457

Submit

```
Total Rows in Clickhouse Table: 31,536,600

Total Disk Usage for Clickhouse Table: 240.61MB
```

Raw Data Chart of 50,000 samples

Clickhouse Raw Data Loading...

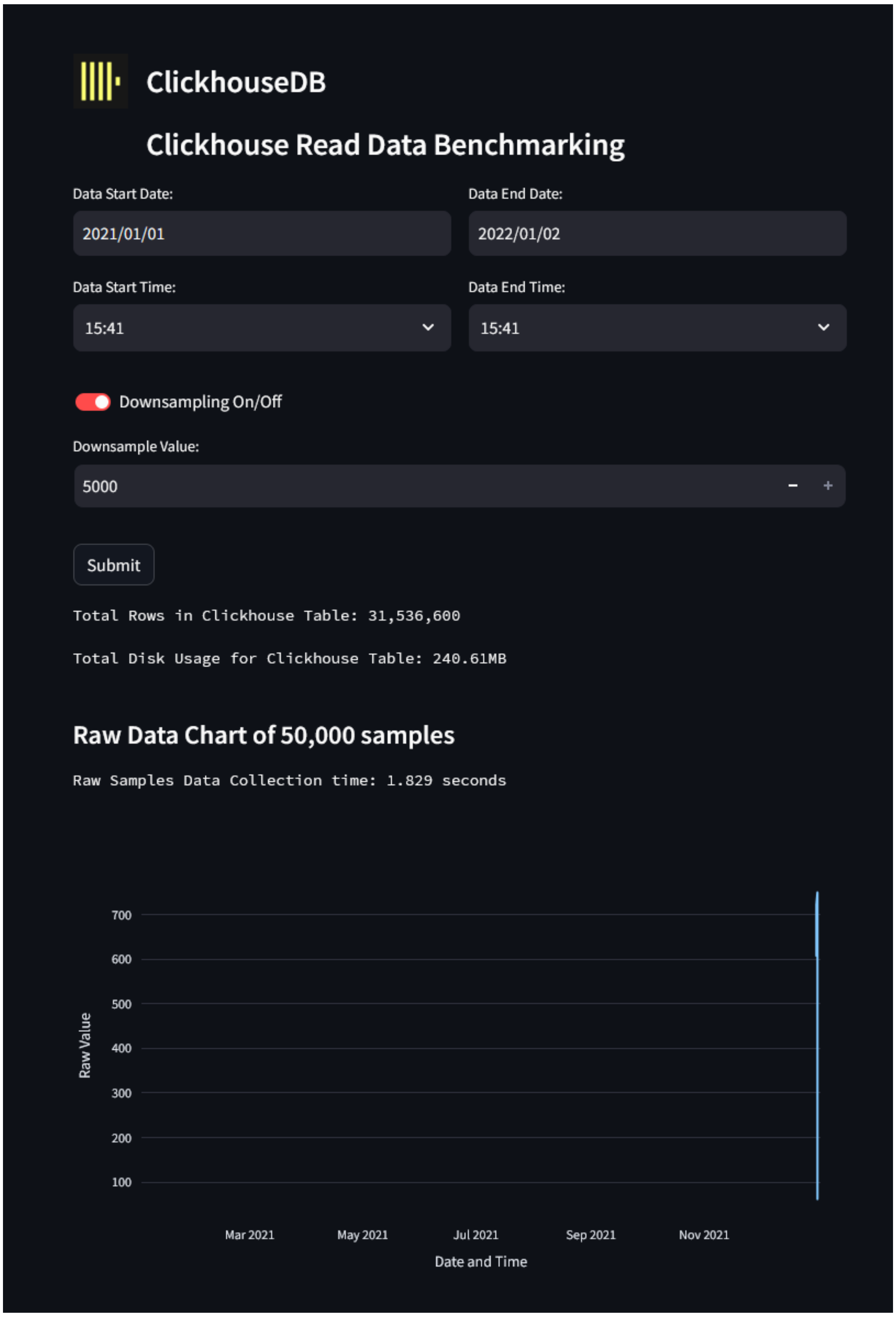


Date and Time

Downsampled Data Chart (2457/2457 of 31,461,420 rows)

```
Downsampled Data Collection time: 1.774 seconds
```

Appendix C – ClickHouseDB Stats


ClickhouseDB

Clickhouse Read Data Benchmarking

Data Start Date: 2021/01/01

Data End Date: 2022/01/02

Data Start Time: 15:41

Data End Time: 15:41

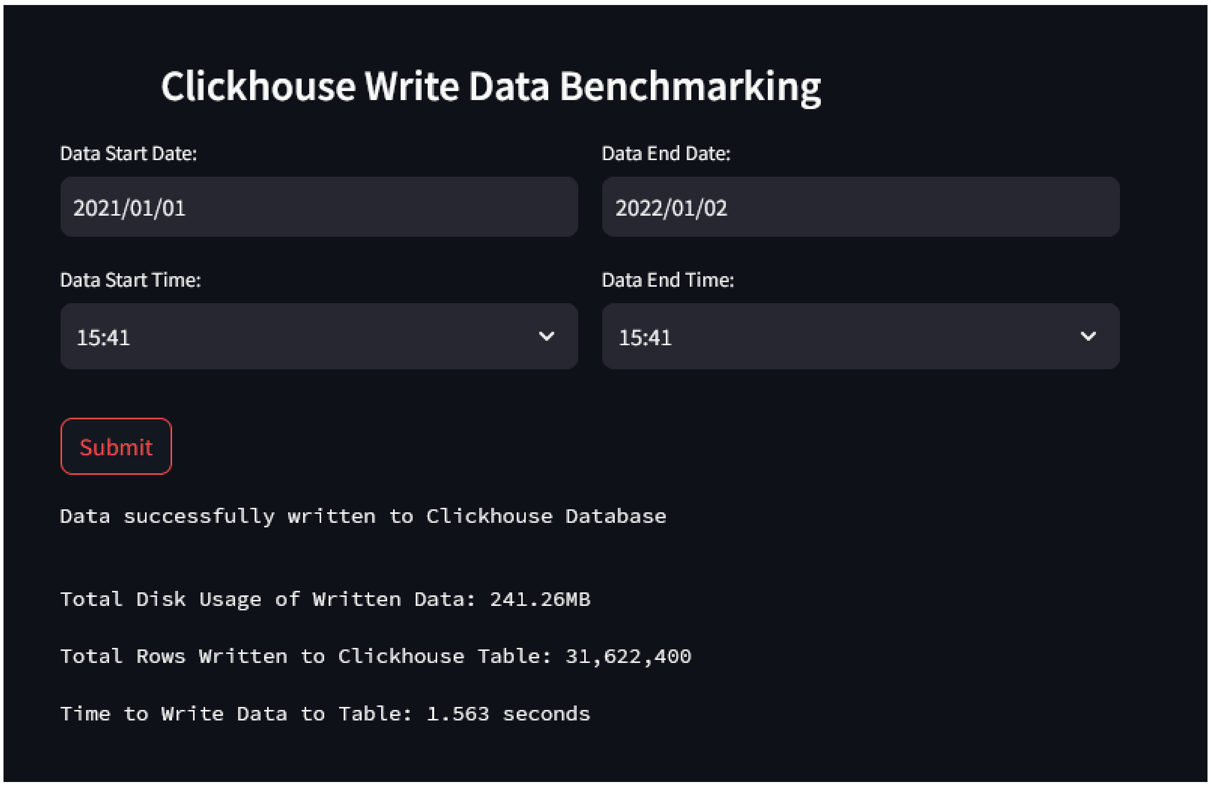
Downsampling On/Off

Downsample Value: 5000

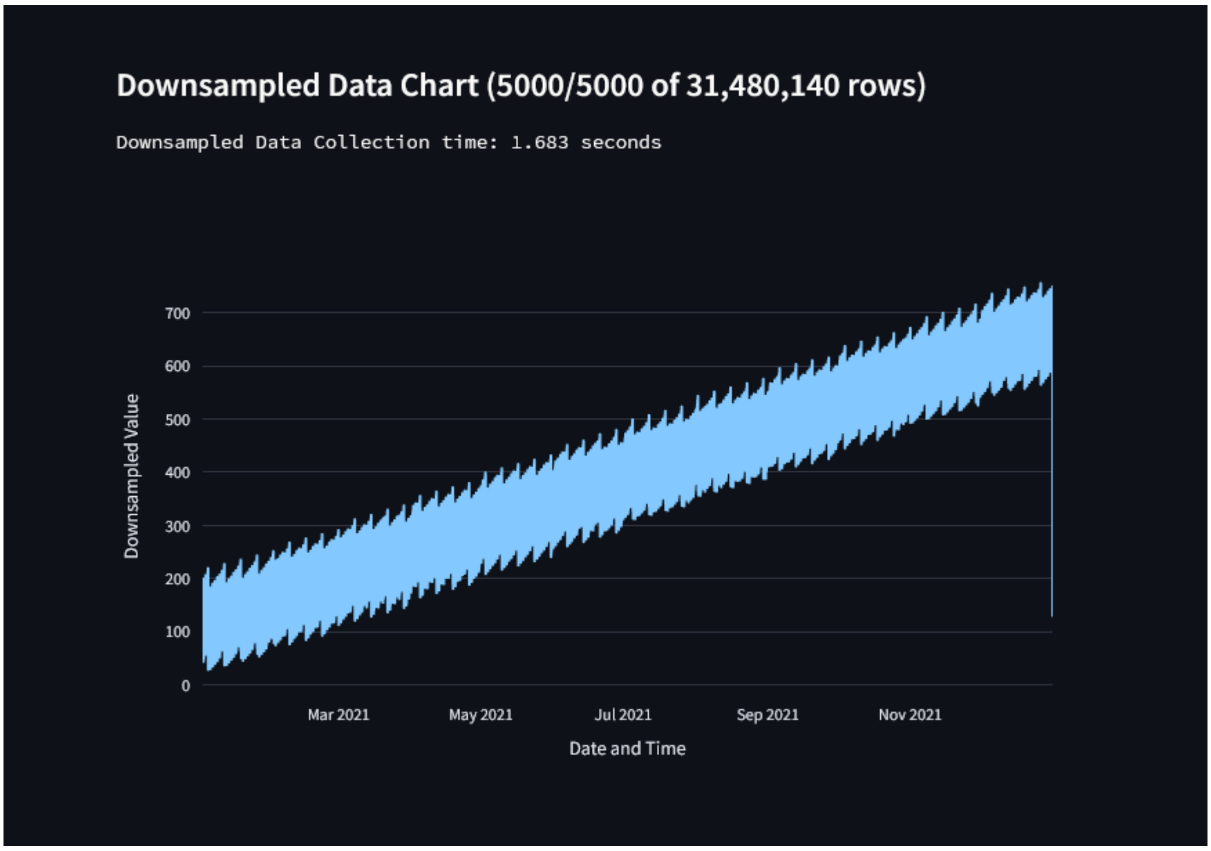
```
Total Rows in Clickhouse Table: 31,536,600

Total Disk Usage for Clickhouse Table: 240.61MB
```

Raw Data Chart of 50,000 samples

```
Raw Samples Data Collection time: 1.829 seconds
```

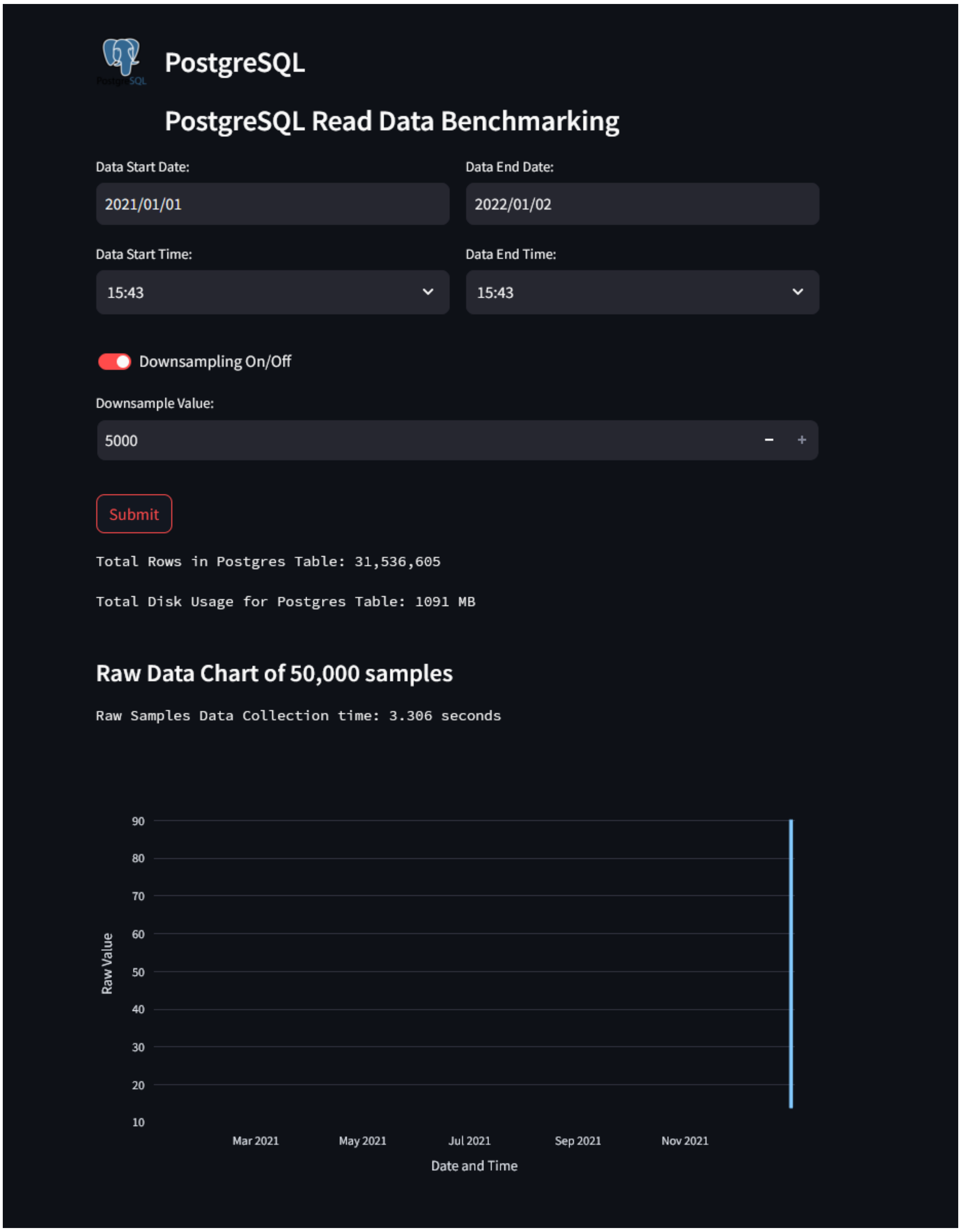


```
Data successfully written to Clickhouse Database

Total Disk Usage of Written Data: 241.26MB

Total Rows Written to Clickhouse Table: 31,622,400

Time to Write Data to Table: 1.563 seconds
```



```
Downsampled Data Collection time: 1.683 seconds
```

Appendix D – PostgreSQL Stats



PostgreSQL

PostgreSQL Read Data Benchmarking

Data Start Date: 2021/01/01

Data End Date: 2022/01/02

Data Start Time: 15:43

Data End Time: 15:43

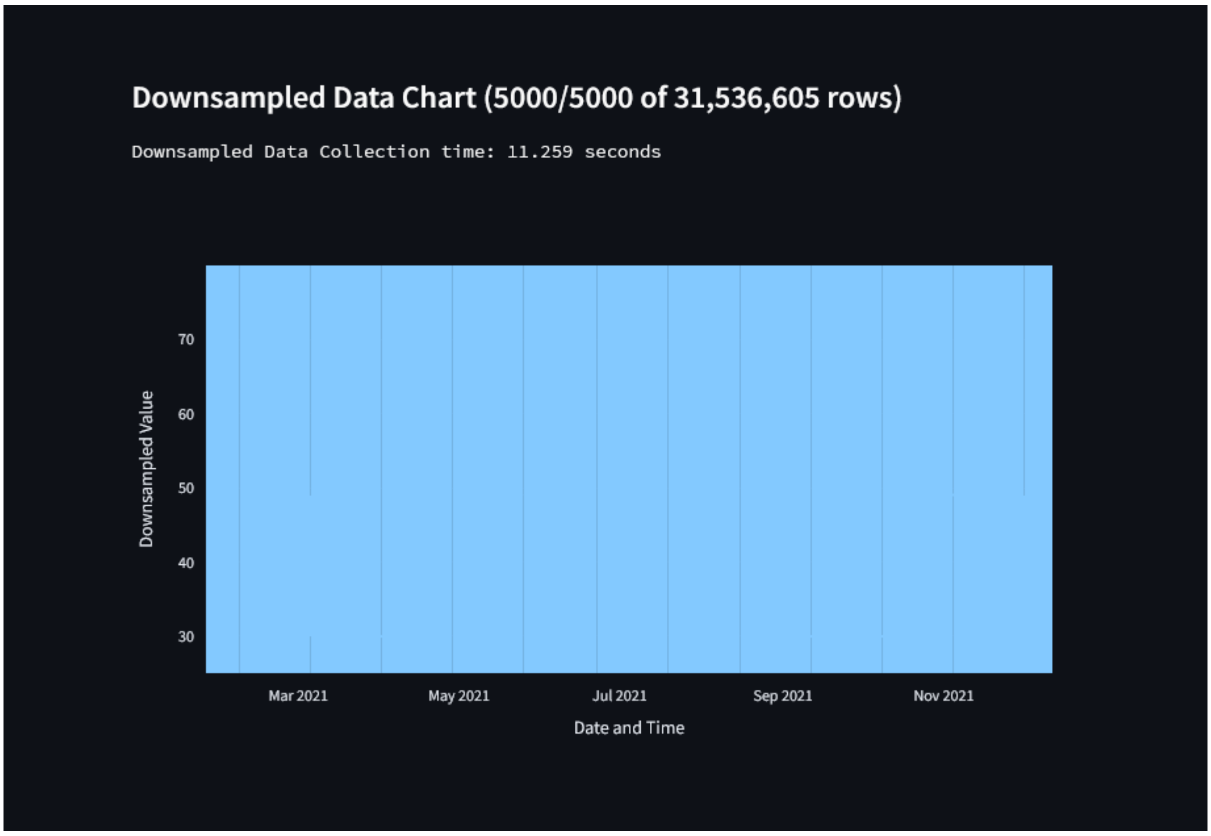
Downsampling On/Off

Downsample Value: 5000

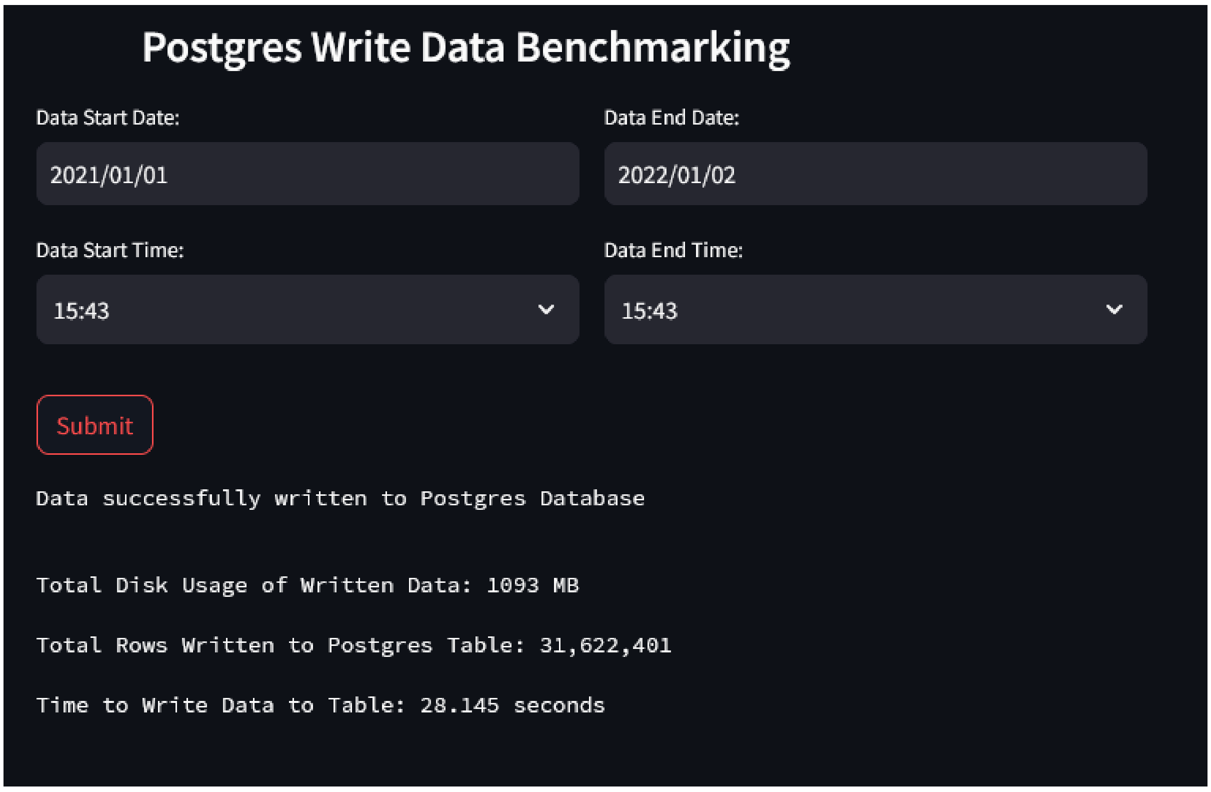
```
Total Rows in Postgres Table: 31,536,605

Total Disk Usage for Postgres Table: 1091 MB
```

Raw Data Chart of 50,000 samples

```
Raw Samples Data Collection time: 3.306 seconds
```


Downsampled Data Chart (5000/5000 of 31,536,605 rows)

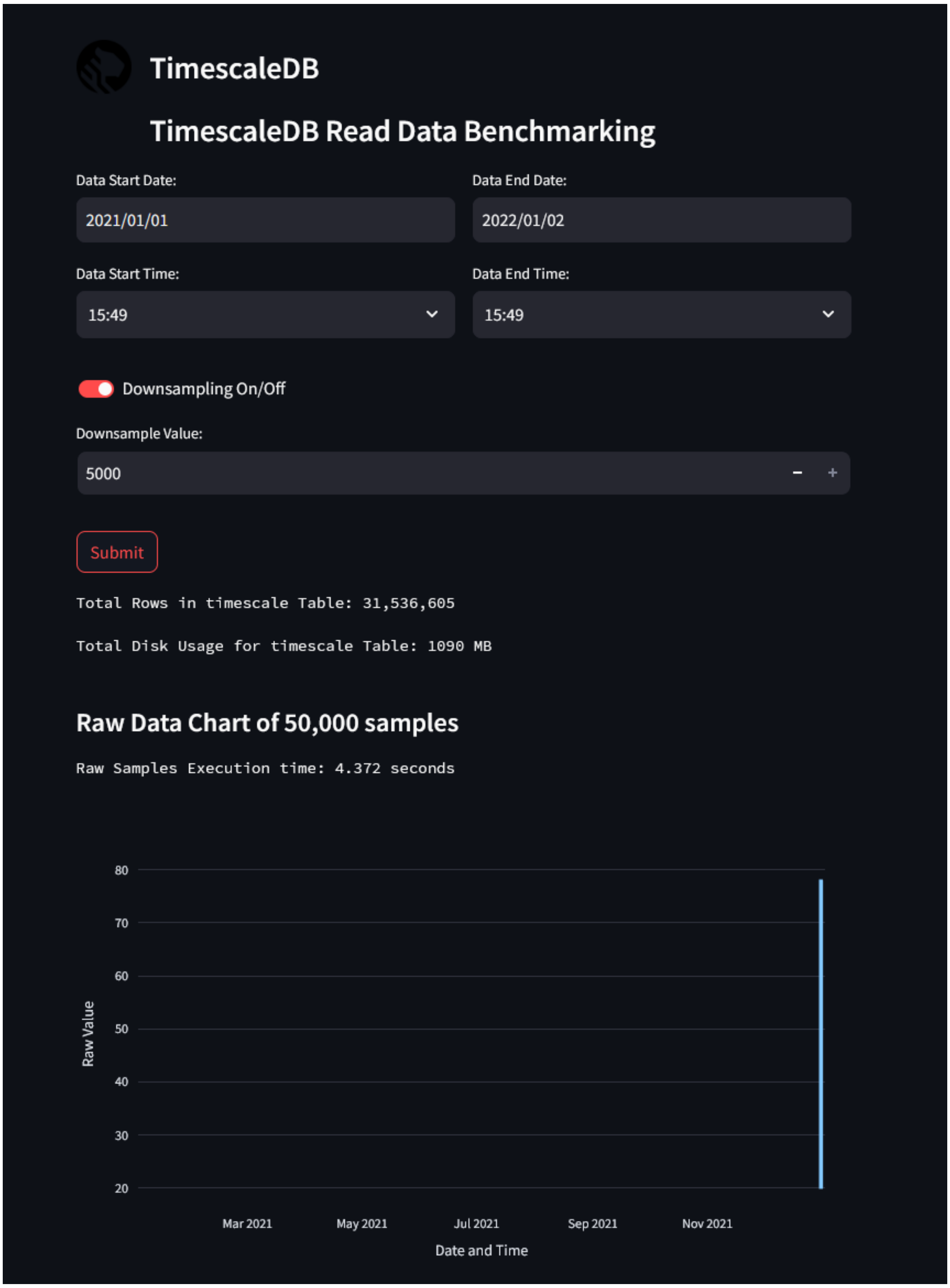
```
Downsampled Data Collection time: 11.259 seconds
```


Postgres Write Data Benchmarking

Data Start Date: 2021/01/01	Data End Date: 2022/01/02
Data Start Time: 15:43	Data End Time: 15:43

```
Data successfully written to Postgres Database

Total Disk Usage of Written Data: 1093 MB

Total Rows Written to Postgres Table: 31,622,401

Time to Write Data to Table: 28.145 seconds
```

Appendix E – TimeseriesDB Stats



TimescaleDB

TimescaleDB Read Data Benchmarking

Data Start Date: 2021/01/01

Data End Date: 2022/01/02

Data Start Time: 15:49

Data End Time: 15:49

Downsampling On/Off

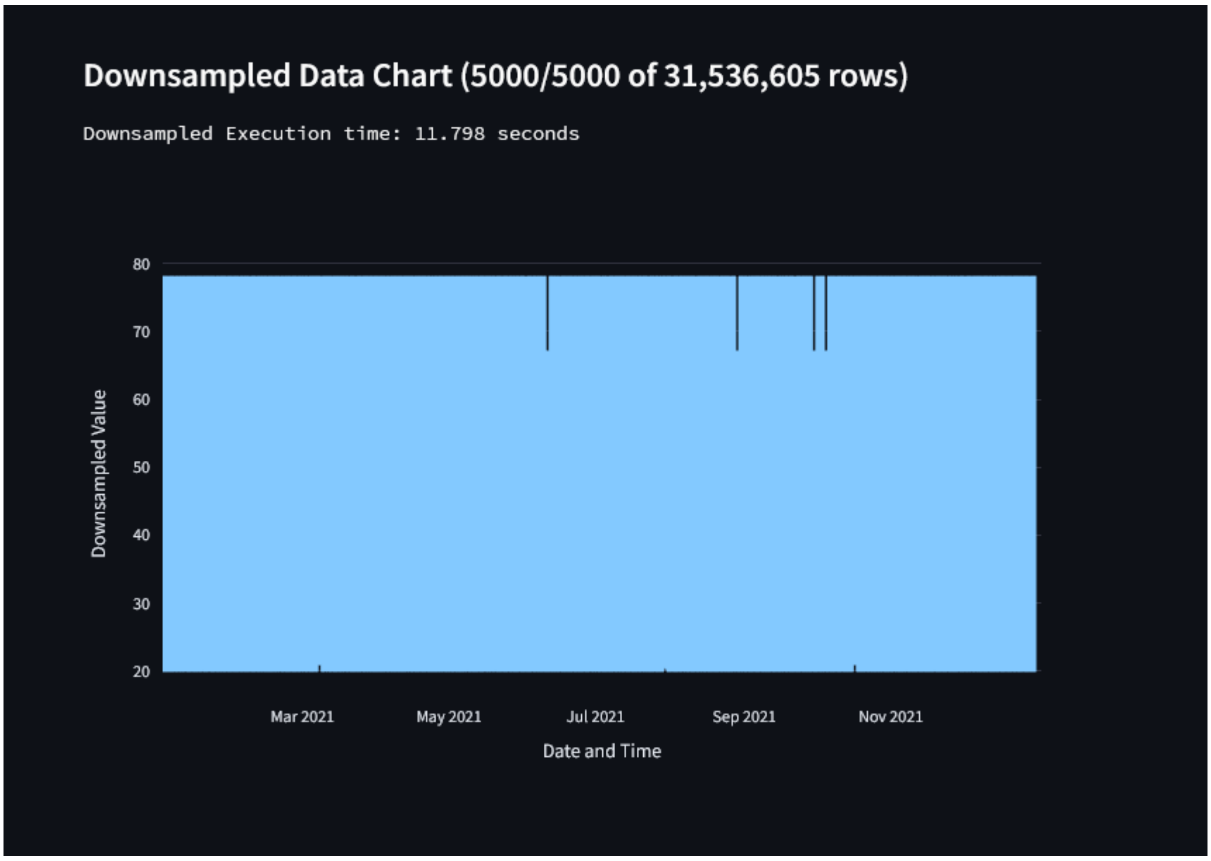
Downsample Value: 5000

Submit

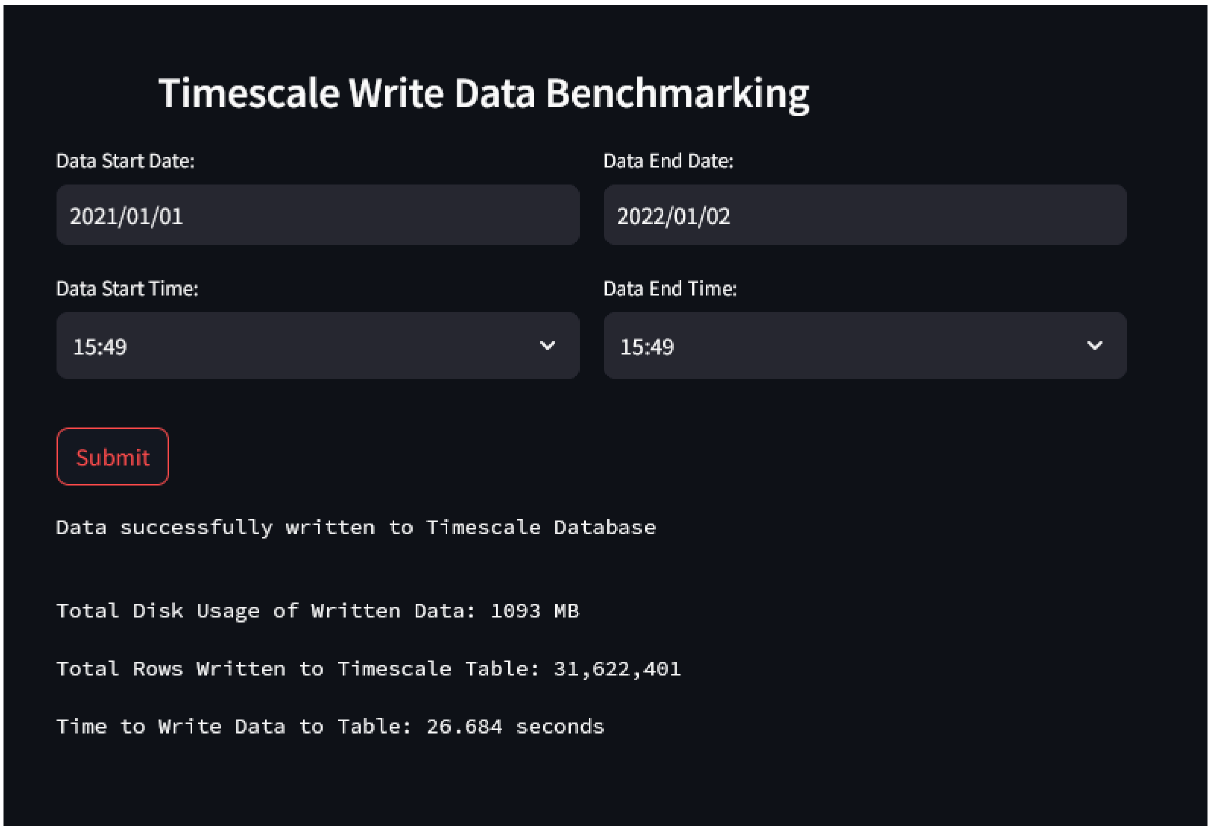
```
Total Rows in timescale Table: 31,536,605

Total Disk Usage for timescale Table: 1090 MB
```

Raw Data Chart of 50,000 samples

```
Raw Samples Execution time: 4.372 seconds
```


Downsampled Data Chart (5000/5000 of 31,536,605 rows)

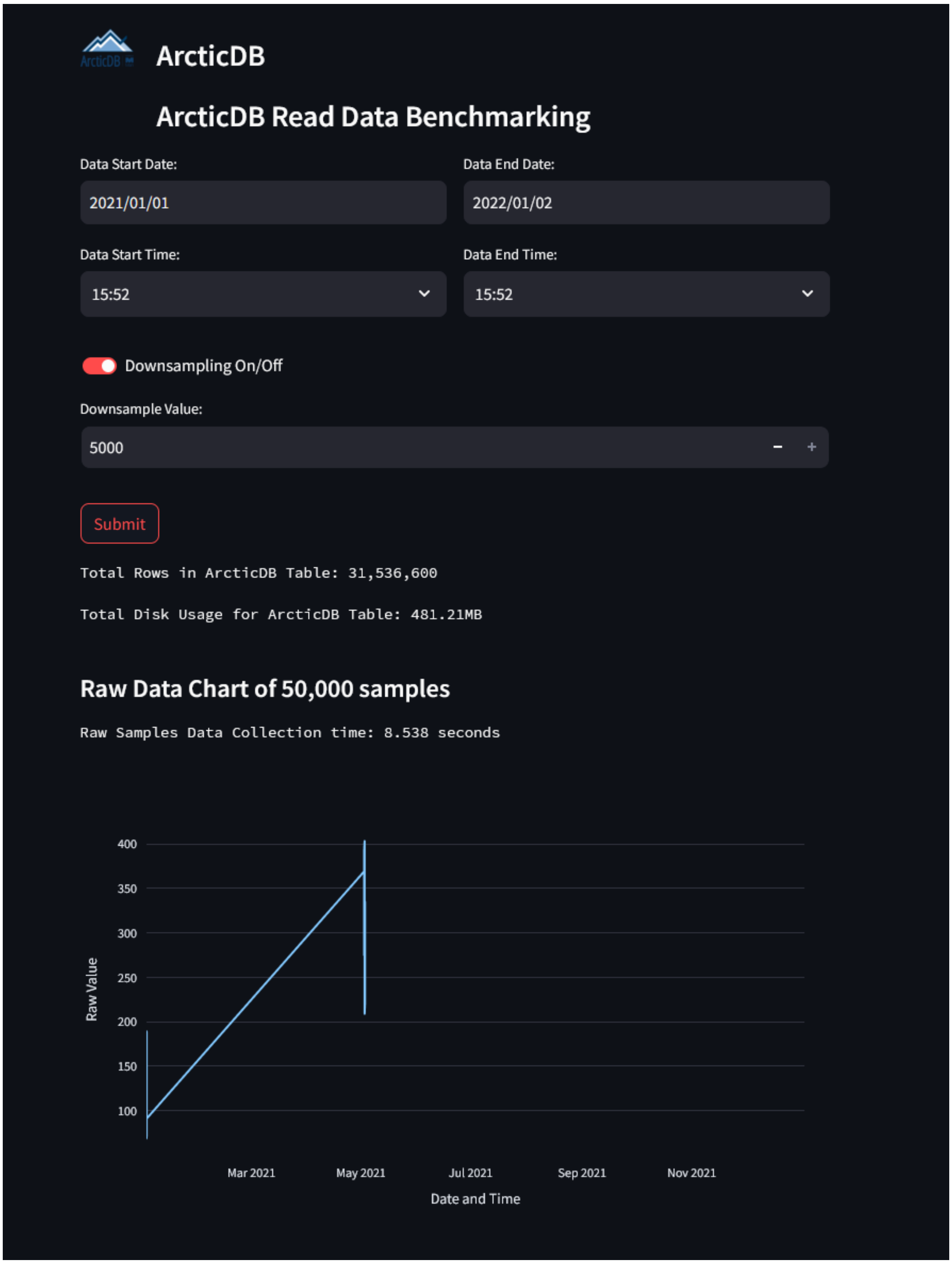
```
Downsampled Execution time: 11.798 seconds
```


Timescale Write Data Benchmarking

Data Start Date: 2021/01/01	Data End Date: 2022/01/02
Data Start Time: 15:49	Data End Time: 15:49

```
Data successfully written to Timescale Database

Total Disk Usage of Written Data: 1093 MB

Total Rows Written to Timescale Table: 31,622,401

Time to Write Data to Table: 26.684 seconds
```

Appendix F – ArcticDB Stats



ArcticDB

ArcticDB Read Data Benchmarking

Data Start Date: 2021/01/01

Data End Date: 2022/01/02

Data Start Time: 15:52

Data End Time: 15:52

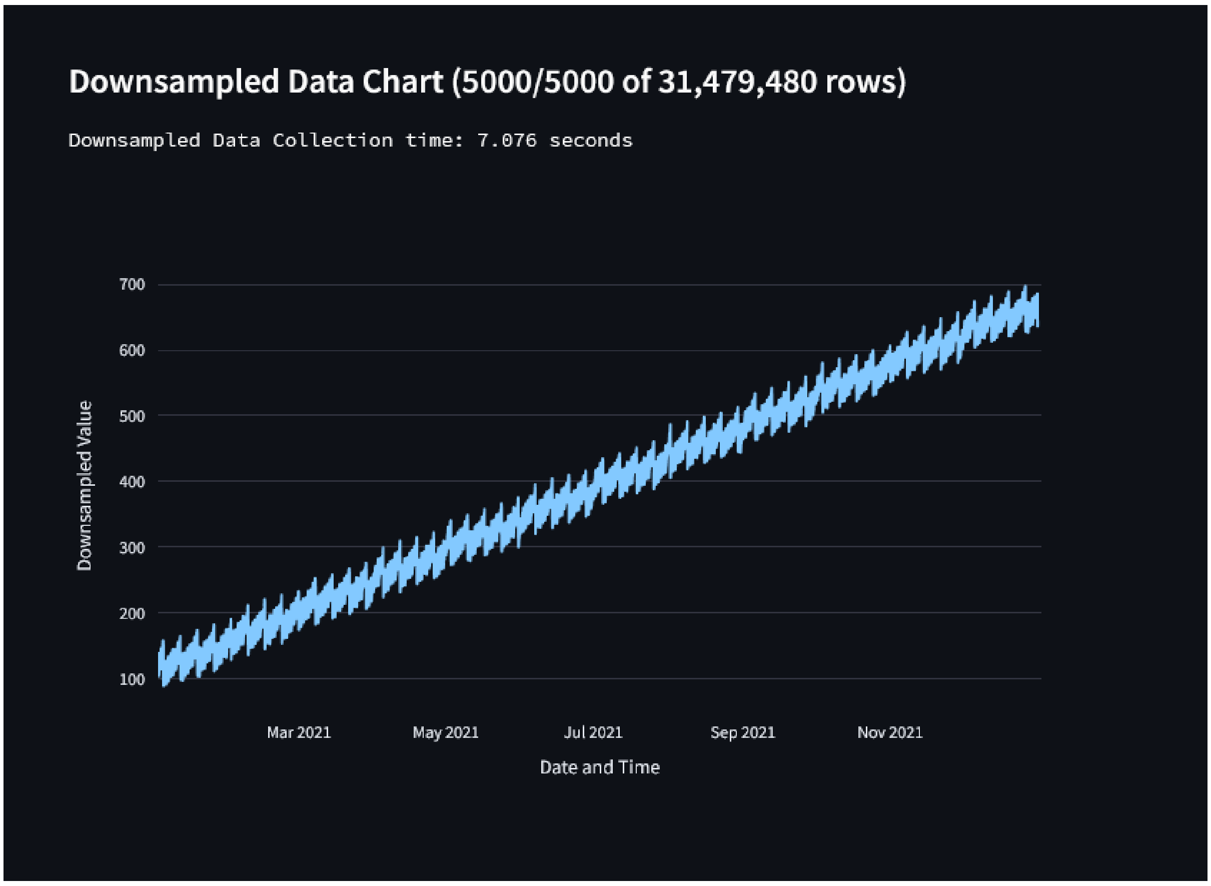
Downsampling On/Off

Downsample Value: 5000

```
Total Rows in ArcticDB Table: 31,536,600

Total Disk Usage for ArcticDB Table: 481.21MB
```

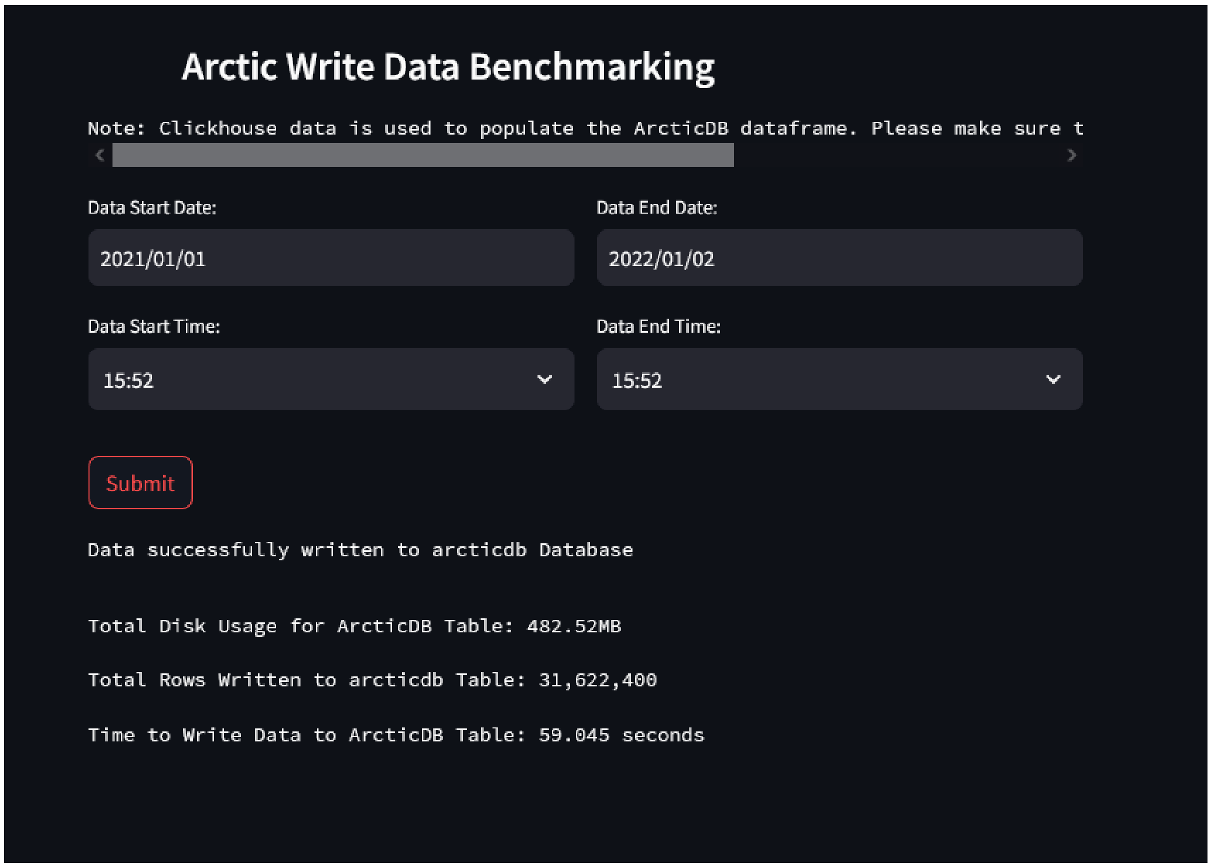
Raw Data Chart of 50,000 samples

```
Raw Samples Data Collection time: 8.538 seconds
```



Downsampled Data Chart (5000/5000 of 31,479,480 rows)

```
Downsampled Data Collection time: 7.076 seconds
```



Arctic Write Data Benchmarking

```
Note: Clickhouse data is used to populate the ArcticDB dataframe. Please make sure t
```

Data Start Date: 2021/01/01

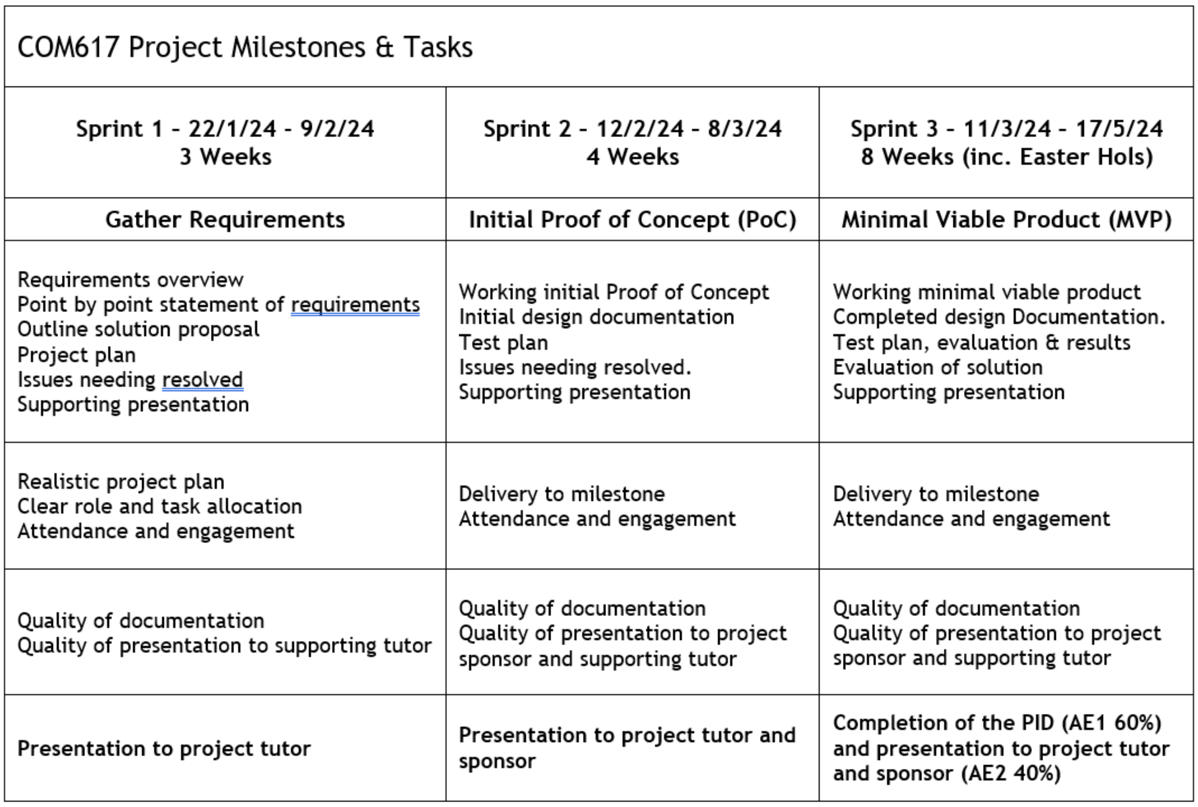
Data End Date: 2022/01/02

Data Start Time: 15:52

Data End Time: 15:52

```
Data successfully written to arcticdb Database

Total Disk Usage for ArcticDB Table: 482.52MB

Total Rows Written to arcticdb Table: 31,622,400

Time to Write Data to ArcticDB Table: 59.045 seconds
```

Appendix G – Sprint Overview



COM617 Project Milestones & Tasks

Sprint 1 - 22/1/24 - 9/2/24 3 Weeks	**Sprint 2 - 12/2/24 - 8/3/24 4 Weeks**	**Sprint 3 - 11/3/24 - 17/5/24 8 Weeks (inc. Easter Hols)**
Gather Requirements	**Initial Proof of Concept (PoC)**	**Minimal Viable Product (MVP)**
Requirements overview Point by point statement of requirements Outline solution proposal Project plan Issues needing resolved Supporting presentation	Working initial Proof of Concept Initial design documentation Test plan Issues needing resolved. Supporting presentation	Working minimal viable product Completed design Documentation. Test plan, evaluation & results Evaluation of solution Supporting presentation
Realistic project plan Clear role and task allocation Attendance and engagement	Delivery to milestone Attendance and engagement	Delivery to milestone Attendance and engagement
Quality of documentation Quality of presentation to supporting tutor	Quality of documentation Quality of presentation to project sponsor and supporting tutor	Quality of documentation Quality of presentation to project sponsor and supporting tutor
Presentation to project tutor	**Presentation to project tutor and sponsor**	**Completion of the PID (AE1 60%) and presentation to project tutor and sponsor (AE2 40%)**

Appendix H – Test Plan Document

TEST TITLE			
COM617 DB Benchmark Test Plan			
TEST DESCRIPTION			
Unit tests to be created alongside the code and run via Github Actions			
TEST DESCRIPTION	**TEST DEPENDENCIES**	**TEST CONDITIONS**	**TEST CONTROL**
Unit tests to be created alongside the code and run via Github Actions after each push to the repository. This test plan should make up a part of the CI/CD Pipeline and should be updated accordingly when features are added. The tests can also be run manually by the user using Pytest in the IDE.	The dependencies will vary as the software becomes more complex as some tests will require different databases to be run.	Tests are to be run automatically through Github actions. Tests should also be able to be run manually on branch using Pytest.	Test to be run automatically after each push by Github so should be in a controlled environment. To test manually, first close application then run through.

STEP ID	**STEP DESCRIPTION**	**TEST DATE**	**EXPECTED RESULTS**	**ACTUAL RESULTS**	**PASS / FAIL**	**ADDITIONAL NOTES**
1	A test to check the Home Page opens	03/24/2024	Test Pass- The home page can be opened	Test Pass- The home page can be opened	PASS	
2	Downsampling Toggle can be selected for DB1 – (Clickhouse)	03/24/2024	Test Pass – The downsampling toggles can be selected for each database type	Test Pass – The downsampling toggles can be selected for each database type	PASS	
3	Increment the Downsample Number Input for each Database type – DB 1 (Clickhouse)	03/24/2024	Test Pass – The downsampling value is incremented by 1 to a value of 6 (min is 5)	Test Pass – The downsampling value is incremented by 1 to a value of 6 (min is 5)	PASS	
4	Set the Downsample Number Input for each Database type – DB 1 (Clickhouse)	03/24/2024	Test Pass – The downsampling value is set to a value of 5000	Test Pass – The downsampling value is set to a value of 5000	PASS	
5	Change the start date of	03/24/2024	Test Pass – The start date	Test Pass – The start date	PASS	

	each Database type – DB 1 (Clickhouse)		can be changed	can be changed		
6	Change the end date of each Database type – DB 1 (Clickhouse)	03/24/2024	Test Pass – The end date can be changed	Test Pass – The end date can be changed	PASS	
7	Change the start time of each Database type – DB 1 (Clickhouse)	03/24/2024	Test Pass – The start time can be changed	Test Pass – The start time can be changed	PASS	
8	Change the end time of each Database type – DB 1 (Clickhouse)	03/24/2024	Test Pass – The end time can be changed	Test Pass – The end time can be changed	PASS	
9	Submit Button can be selected and data ingressed for each Database type – DB 1 (Clickhouse)	03/24/2024	Test Pass – The submit button can be selected and the data is imported	Test Expected Fail (Due to pytest timeout) - The submit button can be selected and the data is imported	XFAIL	Test times out after 3 seconds, set as XFAIL when it times out passes as it shows data is being imported
10	Downsamplin g Toggle can be selected for DB 2 – (PostgreSQL)	03/24/2024	Test Pass – The downsampling toggles can be selected for each database type	Test Pass – The downsampling toggles can be selected for each database type	PASS	
11	Increment the Downsample Number Input for each Database type – DB 2 (PostgreSQL)	03/24/2024	Test Pass – The downsampling value is incremented by 1 to a value of 6 (min is 5)	Test Pass – The downsampling value is incremented by 1 to a value of 6 (min is 5)	PASS	
12	Set the Downsample Number Input for each Database type – DB 2 (PostgreSQL)	03/24/2024	Test Pass – The downsampling value is set to a value of 5000	Test Pass – The downsampling value is set to a value of 5000	PASS	
13	Change the start date of each Database type – DB 2 (PostgreSQL)	03/24/2024	Test Pass – The start date can be changed	Test Pass – The start date can be changed	PASS	
14	Change the end date of each Database type – DB 2 (PostgreSQL)	03/24/2024	Test Pass – The end date can be changed	Test Pass – The end date can be changed	PASS	
15	Change the	03/24/2024	Test Pass –	Test Pass –	PASS	

	start time of each Database type – DB 2 (PostgreSQL)		The start time can be changed	The start time can be changed		
16	Change the end time of each Database type – DB 2 (PostgreSQL)	03/24/2024	Test Pass – The end time can be changed	Test Pass – The end time can be changed	PASS	
17	Submit Button can be selected and data ingressed for each Database type – DB 2 (PostgreSQL)	03/24/2024	Test Pass – The submit button can be selected and the data is imported	Test Expected Fail (Due to pytest timeout) - The submit button can be selected and the data is imported	XFAIL	Test times out after 3 seconds, set as XFAIL when it times out passes as it shows data is being imported
18	Downsampling Toggle can be selected for DB 3 – (TimescaleDB)	03/24/2024	Test Pass – The downsampling toggles can be selected for each database type	Test Pass – The downsampling toggles can be selected for each database type	PASS	
19	Increment the Downsample Number Input for each Database type – DB 3 (TimescaleDB)	03/24/2024	Test Pass – The downsampling value is incremented by 1 to a value of 6 (min is 5)	Test Pass – The downsampling value is incremented by 1 to a value of 6 (min is 5)	PASS	
20	Set the Downsample Number Input for each Database type – DB 3 (TimescaleDB)	03/24/2024	Test Pass – The downsampling value is set to a value of 5000	Test Pass – The downsampling value is set to a value of 5000	PASS	
21	Change the start date of each Database type – DB 3 (TimescaleDB)	03/24/2024	Test Pass – The start date can be changed	Test Pass – The start date can be changed	PASS	
22	Change the end date of each Database type – DB 3 (TimescaleDB)	03/24/2024	Test Pass – The end date can be changed	Test Pass – The end date can be changed	PASS	
23	Change the start time of each Database type – DB 3 (TimescaleDB)	03/24/2024	Test Pass – The start time can be changed	Test Pass – The start time can be changed	PASS	

24	Change the end time of each Database type – DB 3 (TimescaleDB)	03/24/2024	Test Pass – The end time can be changed	Test Pass – The end time can be changed	PASS	
25	Submit Button can be selected and data ingressed for each Database type – DB 3 (TimescaleDB)	03/24/2024	Test Pass – The submit button can be selected and the data is imported	Test Pass – The submit button can be selected	PASS	Currently Passes as Timescale has not got data yet
26	Downsampling Toggle can be selected for DB 4 – (ArcticDB)	03/24/2024	Test Pass – The downsampling toggles can be selected for each database type	Test Pass – The downsampling toggles can be selected for each database type	PASS	
27	Increment the Downsample Number Input for each Database type – DB 4 (ArcticDB)	03/24/2024	Test Pass – The downsampling value is incremented by 1 to a value of 6 (min is 5)	Test Pass – The downsampling value is incremented by 1 to a value of 6 (min is 5)	PASS	
28	Set the Downsample Number Input for each Database type – DB 4 (ArcticDB)	03/24/2024	Test Pass – The downsampling value is set to a value of 5000	Test Pass – The downsampling value is set to a value of 5000	PASS	
29	Change the start date of each Database type – DB 4 (ArcticDB)	03/24/2024	Test Pass – The start date can be changed	Test Pass – The start date can be changed	PASS	
30	Change the end date of each Database type – DB 4 (ArcticDB)	03/24/2024	Test Pass – The end date can be changed	Test Pass – The end date can be changed	PASS	
31	Change the start time of each Database type – DB 4 (ArcticDB)	03/24/2024	Test Pass – The start time can be changed	Test Pass – The start time can be changed	PASS	
32	Change the end time of each Database type – DB 4 (ArcticDB)	03/24/2024	Test Pass – The end time can be changed	Test Pass – The end time can be changed	PASS	
33	Submit Button can be selected and	03/24/2024	Test Pass – The submit button can be	Test Expected Fail (Due to pytest	PASS	Test times out after 3 seconds, set as

	data ingressed for each Database type – DB 4 (ArcticDB)		selected and the data is imported	timeout) - The submit button can be selected and the data is imported		XFAIL when it times out passes as it shows data is being imported
34	All values can be edited and stay the selected values when the submit button is selected – DB 1 (Clickhouse)	03/24/2024	Test Pass – The submit button can be selected and all the changed fields remain changed	Test Expected Fail (Due to pytest timeout) - The submit button can be selected but times out due to data import	XFAIL	Test times out after 3 seconds, set as XFAIL when it times out passes as it shows data is being imported
35	All values can be edited and stay the selected values when the submit button is selected – DB 2 (PostgreSQL)	03/24/2024	Test Pass – The submit button can be selected and all the changed fields remain changed	Test Expected Fail (Due to pytest timeout) - The submit button can be selected but times out due to data import	XFAIL	Test times out after 3 seconds, set as XFAIL when it times out passes as it shows data is being imported
36	All values can be edited and stay the selected values when the submit button is selected – DB 3 (TimescaleDB)	03/24/2024	Test Pass – The submit button can be selected and all the changed fields remain changed	Test Pass – The submit button can be selected	PASS	Currently Passes as Timescale has not got data yet
37	All values can be edited and stay the selected values when the submit button is selected – DB 4 (ArcticDB)	03/24/2024	Test Pass – The submit button can be selected and all the changed fields remain changed	Test Expected Fail (Due to pytest timeout) - The submit button can be selected but times out due to data import	XFAIL	Test times out after 3 seconds, set as XFAIL when it times out passes as it shows data is being imported
38	Change the start date of each Database type (Writing Data Benchmark) – DB 1 (ClickhouseD B)	04/06/2024	Test Pass – The start date can be changed	Test Pass – The start date can be changed	PASS	
39	Change the end date of each Database type (Writing Data Benchmark) –	04/06/2024	Test Pass – The end date can be changed	Test Pass – The end date can be changed	PASS	

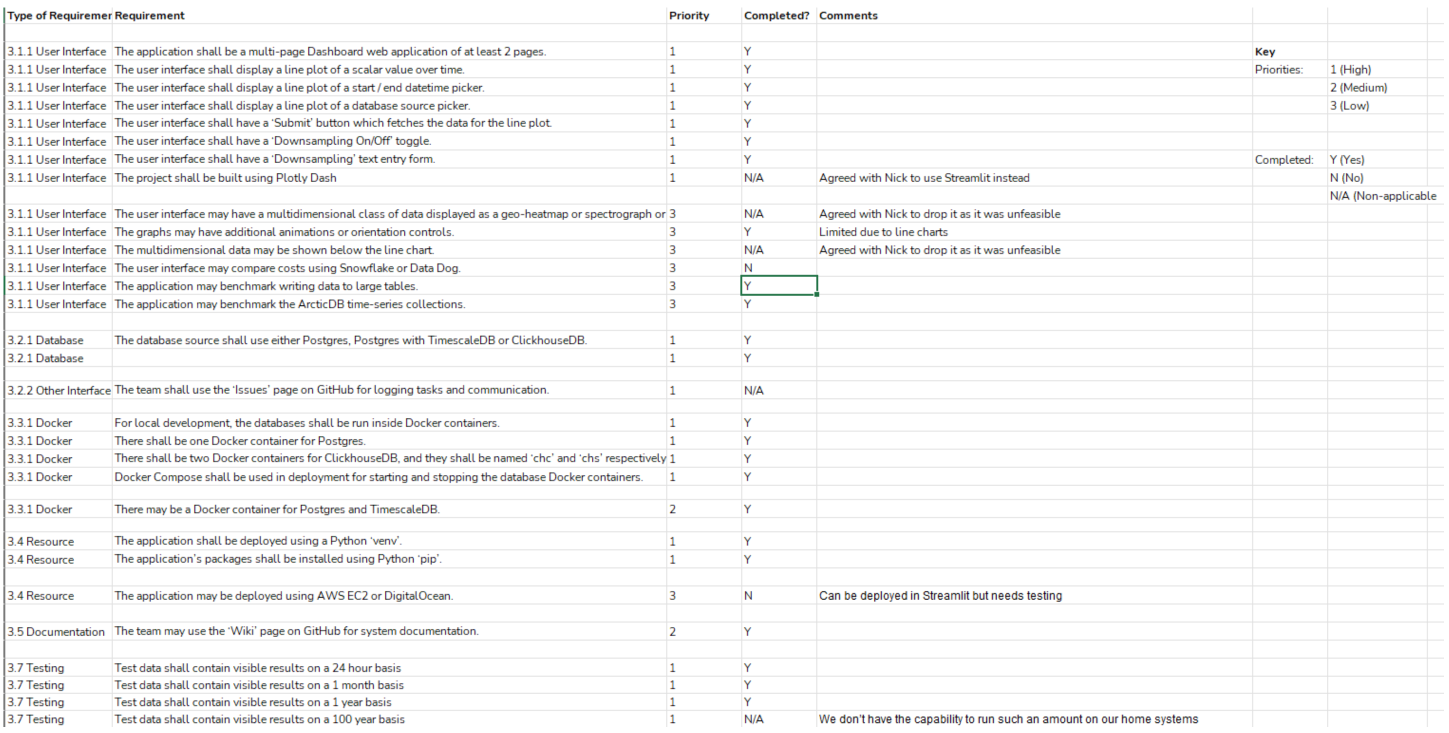
	DB 1 (ClickhouseDB)					
40	Change the start time of each Database type (Writing Data Benchmark) – DB 1 (ClickhouseDB)	04/06/2024	Test Pass – The start time can be changed	Test Pass – The start time can be changed	PASS	
41	Change the end time of each Database type (Writing Data Benchmark) – DB 1 (ClickhouseDB)	04/06/2024	Test Pass – The end time can be changed	Test Pass – The end time can be changed	PASS	
42	Submit Button can be selected and data ingressed for each Database type (Writing Data Benchmark) – DB 1 (ClickhouseDB)	04/06/2024	Test Pass – The submit button can be selected and the data is imported	Test Expected Fail (Due to pytest timeout) - The submit button can be selected but times out due to data import	XFAIL	Test times out after 3 seconds, set as XFAIL when it times out passes as it shows data is being imported
43	Change the start date of each Database type (Writing Data Benchmark) – DB 2 (PostgresSQL)	04/06/2024	Test Pass – The start date can be changed	Test Pass – The start date can be changed	PASS	
44	Change the end date of each Database type (Writing Data Benchmark) – DB 2 (PostgresSQL)	04/06/2024	Test Pass – The end date can be changed	Test Pass – The end date can be changed	PASS	
45	Change the start time of each Database type (Writing Data Benchmark) – DB 2 (PostgresSQL)	04/06/2024	Test Pass – The start time can be changed	Test Pass – The start time can be changed	PASS	
46	Change the end time of each Database type (Writing	04/06/2024	Test Pass – The end time can be changed	Test Pass – The end time can be changed	PASS	

	Data Benchmark) – DB 2 (PostgresSQL)					
47	Submit Button can be selected and data ingressed for each Database type (Writing Data Benchmark) – DB 2 (PostgresSQL)	04/06/2024	Test Pass – The submit button can be selected and the data is imported	Test Expected Fail (Due to pytest timeout) - The submit button can be selected but times out due to data import	XFAIL	Test times out after 3 seconds, set as XFAIL when it times out passes as it shows data is being imported
48	Change the start date of each Database type (Writing Data Benchmark) – DB 3 (TimescaleDB)	04/19/2024	Test Pass – The start date can be changed	Test Pass – The start date can be changed	PASS	
49	Change the end date of each Database type (Writing Data Benchmark) – DB 3 (TimescaleDB)	04/19/2024	Test Pass – The end date can be changed	Test Pass – The end date can be changed	PASS	
50	Change the start time of each Database type (Writing Data Benchmark) – DB 3 (TimescaleDB)	04/19/2024	Test Pass – The start time can be changed	Test Pass – The start time can be changed	PASS	
51	Change the end time of each Database type (Writing Data Benchmark) – DB 3 (TimescaleDB)	04/19/2024	Test Pass – The end time can be changed	Test Pass – The end time can be changed	PASS	
52	Submit Button can be selected and data ingressed for each Database type (Writing Data Benchmark) – DB 3 (TimescaleDB	04/19/2024	Test Pass – The submit button can be selected and the data is imported	Test Expected Fail (Due to pytest timeout) - The submit button can be selected but times out due to data import	XFAIL	Test times out after 3 seconds, set as XFAIL when it times out passes as it shows data is being imported

	)					
53	Change the start date of each Database type (Writing Data Benchmark) – DB 4 (ArcticDB)	04/06/2024	Test Pass – The start date can be changed	Test Pass – The end date can be changed	PASS	
54	Change the end date of each Database type (Writing Data Benchmark) – DB 4 (ArcticDB)	04/06/2024	Test Pass – The end date can be changed	Test Pass – The start time can be changed	PASS	
55	Change the start time of each Database type (Writing Data Benchmark) – DB 4 (ArcticDB)	04/06/2024	Test Pass – The start time can be changed	Test Pass – The end time can be changed	PASS	
56	Change the end time of each Database type (Writing Data Benchmark) – DB 4 (ArcticDB)	04/06/2024	Test Pass – The end time can be changed	Test Pass – The start date can be changed	PASS	
57	Submit Button can be selected and data ingressed for each Database type (Writing Data Benchmark) – DB 4 (ArcticDB)	04/06/2024	Test Pass – The submit button can be selected and the data is imported	Test Expected Fail (Due to pytest timeout) - The submit button can be selected but times out due to data import	XFAIL	Test times out after 3 seconds, set as XFAIL when it times out passes as it shows data is being imported
58	All values can be edited and stay the selected values when the submit button is selected (Writing Data Benchmark) – DB 1 (Clickhouse)	04/06/2024	Test Pass – The submit button can be selected and all the changed fields remain changed	Test Expected Fail (Due to pytest timeout) - The submit button can be selected but times out due to data import	XFAIL	Test times out after 3 seconds, set as XFAIL when it times out passes as it shows data is being imported
59	All values can be edited and stay the selected values when	04/06/2024	Test Pass – The submit button can be selected and all the	Test Expected Fail (Due to pytest timeout) - The submit button	XFAIL	Test times out after 3 seconds, set as XFAIL when it times out

	the submit button is selected (Writing Data Benchmark) – DB 2 (PostgreSQL)		changed fields remain changed	can be selected but times out due to data import		passes as it shows data is being imported
60	All values can be edited and stay the selected values when the submit button is selected (Writing Data Benchmark) – DB 3 (TimescaleDB)	04/19/2024	Test Pass – The submit button can be selected and all the changed fields remain changed	Test Expected Fail (Due to pytest timeout) - The submit button can be selected but times out due to data import	XFAIL	Test times out after 3 seconds, set as XFAIL when it times out passes as it shows data is being imported
61	All values can be edited and stay the selected values when the submit button is selected (Writing Data Benchmark) – DB 4 (ArcticDB)	04/06/2024	Test Pass – The submit button can be selected and all the changed fields remain changed	Test Expected Fail (Due to pytest timeout) - The submit button can be selected but times out due to data import	XFAIL	Test times out after 3 seconds, set as XFAIL when it times out passes as it shows data is being imported

Appendix I – Project Requirements Table



Type of Requiremer	Requirement	Priority	Completed?	Comments		
3.1.1 User Interface	The application shall be a multi-page Dashboard web application of at least 2 pages.	1	Y		**Key**	
3.1.1 User Interface	The user interface shall display a line plot of a scalar value over time.	1	Y		Priorities:	1 (High)
3.1.1 User Interface	The user interface shall display a line plot of a start / end datetime picker.	1	Y			2 (Medium)
3.1.1 User Interface	The user interface shall display a line plot of a database source picker.	1	Y			3 (Low)
3.1.1 User Interface	The user interface shall have a 'Submit' button which fetches the data for the line plot.	1	Y			
3.1.1 User Interface	The user interface shall have a 'Downsampling On/Off' toggle.	1	Y			
3.1.1 User Interface	The user interface shall have a 'Downsampling' text entry form.	1	Y		Completed:	Y (Yes)
3.1.1 User Interface	The project shall be built using Plotly Dash	1	N/A	Agreed with Nick to use Streamlit instead		N (No)
						N/A (Non-applicable
3.1.1 User Interface	The user interface may have a multidimensional class of data displayed as a geo-heatmap or spectrograph or	3	N/A	Agreed with Nick to drop it as it was unfeasible		
3.1.1 User Interface	The graphs may have additional animations or orientation controls.	3	Y	Limited due to line charts		
3.1.1 User Interface	The multidimensional data may be shown below the line chart.	3	N/A	Agreed with Nick to drop it as it was unfeasible		
3.1.1 User Interface	The user interface may compare costs using Snowflake or Data Dog.	3	N			
3.1.1 User Interface	The application may benchmark writing data to large tables.	3	Y			
3.1.1 User Interface	The application may benchmark the ArcticDB time-series collections.	3	Y			
3.2.1 Database	The database source shall use either Postgres, Postgres with TimescaleDB or ClickhouseDB.	1	Y			
3.2.1 Database		1	Y			
3.2.2 Other Interface	The team shall use the 'Issues' page on GitHub for logging tasks and communication.	1	N/A			
3.3.1 Docker	For local development, the databases shall be run inside Docker containers.	1	Y			
3.3.1 Docker	There shall be one Docker container for Postgres.	1	Y			
3.3.1 Docker	There shall be two Docker containers for ClickhouseDB, and they shall be named 'chc' and 'chs' respectively	1	Y			
3.3.1 Docker	Docker Compose shall be used in deployment for starting and stopping the database Docker containers.	1	Y			
3.3.1 Docker	There may be a Docker container for Postgres and TimescaleDB.	2	Y			
3.4 Resource	The application shall be deployed using a Python 'venv'.	1	Y			
3.4 Resource	The application's packages shall be installed using Python 'pip'.	1	Y			
3.4 Resource	The application may be deployed using AWS EC2 or DigitalOcean.	3	N	Can be deployed in Streamlit but needs testing		
3.5 Documentation	The team may use the 'Wiki' page on GitHub for system documentation.	2	Y			
3.7 Testing	Test data shall contain visible results on a 24 hour basis	1	Y			
3.7 Testing	Test data shall contain visible results on a 1 month basis	1	Y			
3.7 Testing	Test data shall contain visible results on a 1 year basis	1	Y			
3.7 Testing	Test data shall contain visible results on a 100 year basis	1	N/A	We don't have the capability to run such an amount on our home systems		

Appendix J – Project Plan

Table of Contents

1 Introduction

1.1 *Purpose of the Plan*

The project plan highlights the overall scope and timeline of the project, alongside the roles and responsibilities of the team and an overview of the project stakeholders. The plan also includes the flow of the project from start to end, to make sure our work is as realistic as possible in terms of time and quality assurance.

1.2 *Scope and Timeline of the Project*

The system will be a Dashboard web application hosted on Docker containers, featuring a line plot depicting comparison data from ClickHouseDB. Our group will design, develop, test and deploy the software according to the requested requirements supplied by the project sponsor, Rockstone Data.

The work will be divided into three sprints, between 22nd January and 17th May 2024, with the first sprint over three weeks, the second over four, and the third over eight. The requirements and project plan will be drafted in the first sprint, including role and task allocation.

In the second sprint, the team shall provide a working Proof of Concept, application design documentation, a test plan, and a supporting presentation to the project sponsor and course tutor. Lastly, in the third sprint, the team will conclude development and deploy the software. They will have a stand-up meeting once a week where the team will discuss project progress and any blockers.

Our group must produce a Project Initiation Document defining the project stages and the work that needs to be done.

1.3 *Project Standards*

All project standards are highlighted in the Project Standards document. The document includes best practices for software development, including coding and naming conventions, formatting documentation, and the steps to perform project quality checks through the Joel Test.

1.4 *Project Directory Structure*

All documentation must be uploaded to the `docs` directory in the GitHub repository, and each file must have its own dedicated page on the `db_bench` wiki.

The unit test file shall be placed in its own separate directory named `workflows`.

Lastly, if there is more than one file used for the Streamlit GUI, they will be placed into their own separate directory also.

2 Roles and Responsibilities

2.1 Stakeholders

Below is a list of stakeholders in the project and their relevant details:

Role	**Stakeholder Name**	**Details**
Project Sponsor	Nick Thorne	● Director and Consultant Software Engineer at Rockstone Data ● Primary contact for the project
Support Tutor	Martin Reid	● Lecturer and Apprenticeship Advisor at Solent University ● Must be provided with project progress update after each Sprint

2.2 Team Members

Below is a list of team members in the project and their roles and responsibilities:

Role	**Assigned Team Member**	**Responsibilities**
Product Owner	Iona Pitt	● Defining user stories ● Creating/maintaining project backlog ● Leading sprint review, retrospective and planning sessions ● Monitoring the project's performance and provide feedback ● Managing the project vision and strategy
Scrum Master	Josh Clarke	● Ensuring the agile process is followed correctly and conducts Scrum meetings ● Leading the Weekly Standup ● Removes any impediments that impacts the developers
Head of Stakeholder Communications	Luke Wood	● Primary contact for communication with the stakeholder - sends all queries and updates to the stakeholder ● Organising meetings with the stakeholder when required
Software Developer	Iona Pitt, Josh Clarke, Luke Wood, Danny Agha, Kyle Roberts	● Writing software code that satisfies the requirements and adheres to best practices

		● Creating and maintaining project documentation ● Testing software components ● Training application users ● Providing updates to the system or software when required

2.3 Timescales

The project began on Monday 22nd January 2024, and will end on Friday 17th May 2024.

Sprint 1 was a 3-week Sprint and took place between 22nd January and 9th February 2024, with a Sprint Review, Retrospective and Planning session on the afternoon of Thursday 15th February at 3pm.

Sprint 2 shall be a 4-week Sprint and shall take place between 12th February and 8th March 2024, with a Sprint Review, Retrospective and Planning session on the afternoon of 14th March at 3pm.

Lastly, Sprint 3 shall be an 8-week Sprint and shall take place between 11th March and 17th May 2024, with a Sprint Review, Retrospective and Planning session on the afternoon of Thursday 16th May at 3pm. The Scrum ceremonies will be led by the Product Owner Iona Pitt.

The team shall meet for a weekly stand-up meeting every Monday at 5pm for the duration of the project, where the team – lead by the Scrum Master Josh Clarke – shall discuss the progress of the project and deliverables, and any hinderances/impediments the team may have encountered.

2.4 Deliverables

The following deliverables will be produced throughout the project:

Deliverable	**Estimated Date of Completion**
Project Initiation Document	Friday 10th May 2024
Software product – Fully meets requirements	Friday 17th May 2024
User guide on GitHub wiki	Friday 17th May 2024
Video of demonstration of the project (approx. 6 min 40 sec)	Friday 17th May 2024
20 slide deck for presentation (max. 5 min)	Friday 17th May 2024

2.5 Documentation

The following documentation will be produced throughout the project:

Document	**Estimated Date of Completion**
Requirements Document	Friday 9th February 2024
Standards Document	Friday 9th February 2024

Project Plan	Friday 1st March 2024
Design Document	Friday 8th March 2024
Test Document	Friday 8th March 2024
Acceptance Test Document	Friday 8th March 2024

3 Project Planning

3.1 *Project Board*

Tasks and their assigned team members shall be organised through the GitHub repository's Issues page and Project Board, available respectively at:

https://github.com/NickThorne123/db_bench/issues

https://github.com/users/NickThorne123/projects/8

3.2 *Project Monitoring, Control and Problem Reporting*

As mentioned in section 2.3, at the end of each Sprint there shall be a Sprint Review, Retrospective and Planning session. This shall be where the team will discuss the work completed, any changes to the requirements, and what the successful aspects of the sprint were, alongside any potential for improvement.

The team will plan the tasks for the next Sprint during the Planning session, moving the relevant and upcoming tasks from the 'No Status' column to the 'To Do' column within the Project Board.

All problems will be listed on the Issues page, with an appropriate tag (e.g. Bugs for software problems) for the ticket.

4 Technical Plan

4.1 *Lifecycle*

The project will use an Agile lifecycle, and the team will work in iterative Sprints. The Sprints will be of different lengths:

- Sprint 1 – 3 weeks
- Sprint 2 – 4 weeks
- Sprint 3 – 8 weeks

It will adopt some elements from Scrum, including the roles of Product Owner and Scrum Master, and the existence of Scrum ceremonies such as Sprint Planning, Sprint Review and Sprint Retrospective.

4.2 *Languages*

Language	**Use**	**Coding Standard**
Python 3.11	Application back-end and front-end	PEP 8 coding practices

	code	

4.3 Tools

Process Area	Tool
Software Project Management	● GitHub Project Board ● GitHub Issues board ● Microsoft Word
Version Control	● GitHub code repository
Requirements	● Microsoft Word ● Microsoft Excel
Analysis and Design	● Microsoft Word ● Wireframe.cc
Implementation	● Microsoft Visual Studio Code ● Streamlit OR Plotly Dash ● Python pip ● Python virtual environments (venv) ● ClickHouseDB ● TimescaleDB ● PostgreSQL ● MongoDB ● Snowflake ● Data Dog
Testing	● Microsoft Visual Studio Code ● Github Workflows
Deployment	● Docker Desktop ● Docker Compose
Support & Maintenance	● GitHub Wiki pages

4.5.1 Development and Target Environments

The application will be produced on both Windows and Apple Mac iOS operating systems with varying hardware and networks, as the individual specifications of each team member's device is different.

The final product shall be able to run according to the advised minimum specifications for running Python 3.11, therefore, the product will not be able to run on Windows 7 or earlier. It will run better on computers with larger RAM and processing power due to the Machine Learning techniques used.

Once deployed, the application will run on all devices which can run Streamlit.

5 Quality Plan

5.1 Reviews and Inspections

All code must be placed into a pull request on GitHub before being merged into the master branch. The pull request must be reviewed and approved by at least one other team member. Commit messages that summarise the code changes must be added to branch commits and to the resulting pull request.

5.2 Testing

The software development team have implemented automated testing using GitHub workflows which tests each commit that the software development team pushes to the project's GitHub repository. Utilising GitHub workflows removes the need of manual human intervention to check and test each commit that gets pushed to the project repository. All commits within the project's repo undergo unit tests that have been written by the testers using pytest, all software components are automatically unit tested through GitHub workflows, which then proceed to document the tests and results in the test plan document. The test document will also include the system and acceptance test strategies and tests scripts.

5.3 Lessons Learned

Lessons learned will be discussed throughout the project, and documented as part of the Sprint Retrospective.

Appendix K – Project Standards Document

Overview

This piece of documentation outlines the rules and conventions that developers must follow when writing code or engaging in the software required for this project. These rules below define the characteristics necessary to maintain a uniform software solution and to facilitate collaboration among developers maintaining clean, readable, secure, and efficient practices.

This document is split into 4 main components, these being the branch naming conventions, the Joel Test (used to implement good software practices), coding conventions and documentation standards that members of the team must follow.

Branch Naming Conventions

The GitHub development flow will be used so individuals can create a branch for issues they are working on.

The branch will be named like the following example: `nt_123_short_desc`

Breaking this down,

1. `nt` will be the developer's initials
2. `123` will be the Issue Number
3. There will then be a one or two word description of the issue.

Projects will move from In-Progress to In-Review Upon pushing changes up to a Pull Request. Which will then run through CI/CD pipeline and be reviewed by another member of the group.

Joel Test

The project will try and adhere to the 'Joel Test' of good software practice.

The Joel Test is a very simple and quick test that rates the quality of your software team. Rather than including open-ended responses, this test has 12 yes-or-no questions that determine whether your programming team is up to par. A score of 12 is perfect. 11 is considered tolerable, and 10 or below is unacceptable. These 12 questions are as follows:

1. Do you use source control?

Source control (or version control) tracks and manages changes made to code. Changes are marked by a tag, known as the "revision number." The original code is deemed "revision 1," and, after the first round of edits, it becomes known as "revision 2," and so on.

Source control is important because programmers can work together on code and track changes over time. This makes it easier to highlight mistakes and correct them before they cause major problems. And, since the source code gets uploaded into every programmer's hard drive, it's much harder to lose revisions.

2. Can you make a build in one step?

A "build in one step" is the ability to combine multiple sections of code, written by various programmers, into one top-level program or tool. As you build, the source code of various tools or features should combine into a single program that can run on its own.

It's important to have a "build in one step" because large programs take more time to complete if they're written by a single person. Instead, it's faster to divide it up into multiple sections, so work can be distributed between several developers. In the end, all the sections and sub-sections are combined into an all-inclusive program.

3. Do you make daily builds?

When using source control, it's easy for programmers to break the build when writing new code. What's even worse is when they don't realize it's broken. A broken build can stall production until the issue is recognized and resolved. To safeguard against this, your team should be conducting daily builds to verify if anything has broken.

If something shows up during the daily build, programmers can check the code that got added or modified to figure out what broke it. Then, fixing the code becomes the responsibility of the person who added that change. This system ensures that errors are noted and fixed as soon as possible and that each programmer can navigate the build.

4. Do you have a bug database?

It's impossible to remember every bug in the code -- especially when there can be so many. For instance, if you click a link and realize it's dead, that action may be automatically saved into a bug database. Since bugs can be reported in a variety of ways, it's important to collect all reported bugs into one database because, if not, they may get forgotten and never fixed.

Additionally, bug databases can work as knowledge bases for programmers. You can use the bug database to see if a similar problem has been reported and if there are steps on how to fix the issue.

5. Do you fix bugs before writing new code?

When writing, you may prefer to write first and edit mistakes later. Many programmers have a similar mindset when it comes to writing code and would prefer to fix bugs after the original code is written. However, this can lead to major flaws, making it more important to prioritize bug-fixing than writing new code.

The longer you wait to fix a bug, the harder it will be to remember where the bug occurred. Fixing it the same day will take minutes while fixing a bug that was written weeks or months ago will be stressful. And, if the product has already shipped, you'll have to recall it and waste copious funds.

Additionally, it's much easier to predict how long it will take to write new code than it will take to fix a bug. This is because fixing a bug depends on when that code was written and how long it will take to track it down. Instead, if bugs are fixed immediately, you can schedule more time to write new code.

6. Do you have an up-to-date schedule?

It's essential to know when code is going to be complete. It's simply unacceptable to leave this to the programmers' leisure. Advertisements, shipments, and even the general satisfaction of customers all ride on code being written in an efficient timeframe.

Having a schedule makes your programmers' work easier. With a schedule, programmers can sort through the list of features to add and weed out the unnecessary ones that will take up too much time. This will ensure your programmers are hitting deadlines and working on code that's essential to your product's success.

7. Do you have a spec?

Product specs are blueprints that describe exactly what the product will do, what it should look like, and how it will perform. Additionally, these guidelines can help product management optimize the product's features for your target audience. They'll outline all the information about the product, so every member of the product development team will know exactly what to create.

When products aren't built from specs, they're more likely to have bugs, be poorly designed, or take too long to build. This wastes time and money, making it risky to write code without having a spec. Whether it's written by your programmers or external writers who are hired for the job, specs should be written and approved before programming begins.

8. Do programmers have quiet working conditions?

According to Spolsky, it's important for software workers -- among other knowledge workers -- to get "in the zone" to get productive work done. Being intently focused is great, but it's a difficult state to attain. After all, it takes the average employee about 15 minutes to start working at one's maximum productivity level.

The simplest distractions can lead to a severe lack of productivity. Overhearing conservations, taking phone calls, and being interrupted by coworkers can result in 15 to 30 minutes of wasted time. So, giving programmers quiet offices, rather than open cubicles, can lead to more production.

9. Do you use the best tools money can buy?

The best companies provide their employees with the best tools. Without them, your employees won't be able to produce a high-quality product. If you want your software development team to succeed, then it's important to invest in the best software development tools.

10. Do you have testers?

It's vital that your software team has testers -- about one per every two or three programmers. Without testers, you may be sending out defective products or wasting money by having programmers. Additionally, testers help refocus your programmers', so they can spend more time coding and troubleshooting.

11. Do new candidates write code during their interview?

You probably wouldn't hire someone for a job without seeing if they can perform that job. For instance, would you ever hire a graphic designer without seeing their creative portfolio? Or, would you hire a baker to make your wedding cake without first tasting their samples?

The sentiment should be applied to programmers. Of course, you can still select candidates based on their resumes, references, personalities, or answers to interview questions. However, their ability to write code during the interview should be the most important. After all, that's what they'll be doing all day.

12. Do you do hallway usability testing?

Hallway usability testing is when you stop someone who walks by you in the hallway and ask them to use the code you just finished writing. Having five random people test out your code will show you 95% of the usability problems with your code.

User interfaces are hard to assess when you're the one who has been staring at it for hours on end. It's important to get fresh eyes on your code to point out issues you may have overlooked. And, since hallway usability testing is both quick and free, you can continuously test your code until it's perfect.

More Info Here

Coding Conventions

The coding conventions for this project will follow is the PEP 8 Conventions - This is documentation that provides guidelines and best practices on how to write Python code. This can be viewed here: PEP 8 Documentation. However I will pull out the main features within this document.

When coding standards are properly defined and implemented, developers, even those who have just joined the team, can easily find their way around the code base. Ideally, wanting our source code to look like a single developer has written, debugged, and maintained it.

Code Layout

1. File Header - File headers will be at the top of every file and include the following information:

    - FileName - Will be the name of the file
    - FileType - Will be the type of file i.e. py (python), cs (c#), js (JavaScript)
    - Created By - Who it was created by. Formatted as Surname, First Name
    - Created On - The date and time it was created on in a dd/mm/yyyy hh:mm:ss AM/PM format
    - Last Modified - The date and time it was last modified on in a dd/mm/yyyy hh:mm:ss AM/PM format
    - Description - Brief description on what the file does

An example can be seen below:

```
# File Header --------------------------------------
# FileName:      main.py
# FileType:      Python File
# Created By:    Roberts, Kyle
# Created On:    01/02/2024 6:00:00 PM
# Last Modified: 01/02/2024 6:30:16 PM
# Description:   This is a file TODO
# End Header ---------------------------------------
```

2. Imports - Imports will be put at the top of the file, putting a blank line between each group of imports. Imports should be grouped in the following order. Standard library imports, related third party imports, local application/library specific imports.An example of this can be seen below:

```
import os
import warnings

import pandas as pd
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.cluster import KMeans

from local_module import local_class
from local_package import local_function
```

3. Indentation - Indentation refers to the spaces at the beginning of a code line. Where in other programming languages the indentation in code is for readability only, the indentation in

Python is very important. Python uses indentation to indicate a blocks of code. Use 4 spaces per indentation level. There needs to be at least 2 blank spaces between methods, with the new method starting on the 3rd line.

Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations) and the use of blank lines in functions, needs to also be done sparingly, but should be there to indicate logical sections.

Finally, a line break needs to be done before a binary operator called Knuth's style an example of this can be seen below:

```
# Easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

Naming Conventions

Type	Naming Convention	Examples
Functions	Use a lowercase word or words. Separate words by underscores to improve readability	function, my_function
Variable	Use a lowercase single letter, word, or words. Separate words with underscores to improve readability. Needs to be a clear and concise explanation of what the variable does or stores	x, var, my_variable, clickhouse_dataset
Class	Start each word with a capital letter. Do not separate words with underscores. This style is called camel case or pascal case	Model, MyClass
Method	Use a lowercase word or words. Separate words with underscores to improve readability. Needs to be a clear and concise explanation of what the method does (don't worry about character length)	class_method, method, loads_postgres_database
Constant	Use an uppercase single letter, word, or words. Separate words with underscores to improve readability	CONSTANT, MY_CONSTANT, MY_LONG_CONSTANT

Type	Naming Convention	Examples
Module	Use a short, lowercase word or words. Separate words with underscores to improve readability.	module.py, my_module.py
Package	Use a short, lowercase word or words. Do not separate words with underscores	package, mypackage

Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes. The first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).A comment must be short and straightforward, and sensible adding value to your code. You should add comments to give code overviews and provide additional information that is not readily available in the code itself, containing only information that is relevant to reading and understanding the program.

1. Docstrings - Docstrings or triple-quoted string literals will be used to provide descriptions of classes, functions(Methods) and constructors. These should be a brief description of what each method does. These can be a single line or a multi-line and should begin with a capital letter and end in a period. Examples can be seen below:

```
""" Description of method here. """
def method():
    print("Hello there")
"""
Bigger Description of
a method is placed here.
"""
def method():
    print("Hello there")
```

2. Single Line Comments - Single Line Comments will primarily be comments that are written over Python statements to clarify what they are doing. An example be seen below:

```
# Program to print the user's name
name = input('Enter your name')
print(name)
```

3. Inline Comments - An Inline comment should be a concise comment on the same line as the code they describe and are useful when you are using any formula or any want to explain the code line in short. They should be separated by at least two spaces. An Example can be seen below:

```
   plt.plot(xtest, ytest, color='green', linewidth=1, label='Actual Price')
# plotting the initial datapoints
```

4. Block Comments - Blocks comments will be used to create a file header at the top of every new python file that is created and when it was last modified. An example of this can be seen below:

```
# FileName:      main.py
# FileType:      Python File
# Created By:    Roberts, Kyle
# Created On:    01/02/2024 6:00:00 PM
# Last Modified: 01/02/2024 6:30:16 PM
# Description:   This file in the main python file
```

Strings

This application will follow the convention of using double quoted strings as opposed to single quoted strings. An example of this can be seen below:

```
print("The quick brown fox jumps over the lazy dog")
```

Whitespace in Expressions and Statements

Immediately inside parentheses, brackets or braces:

```
spam(ham[1], {eggs: 2})
```

Between a trailing comma and a following close parenthesis:

```
foo = (0,)
```

Immediately before a comma, semicolon, or colon:

```
if x == 4: print(x, y); x, y = y, x
```

Immediately before the open parenthesis that starts the argument list of a function call:

```
spam(1)
```

Immediately before the open parenthesis that starts an indexing or slicing:

```
dct['key'] = lst[index]
```

More than one space around an assignment (or other) operator to align it with another:

```
x = 1
y = 2
long_variable = 3
```

Don’t use spaces around the = sign when used to indicate a keyword argument, or when used to indicate a default value for an unannotated function parameter:

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

Example Python File

Overall what a python file could look like using the PEP8 conventions below:

```
# FileName:      main.py
# FileType:      Python File
# Created By:    Roberts, Kyle
# Created On:    01/02/2024 6:00:00 PM
# Last Modified: 01/02/2024 6:30:16 PM
# Description:   This is a file TODO

import os
import warnings

from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

from local_module import local_class
from local_package import local_function

""" Plots a line graph for the user """
def plot_a_line_graph():
    dataset = pd.read_csv("somecsv.csv")

    # gets x and y values specific columns
    x = dataset.iloc[:, 0:260].values
    y = dataset.iloc[:, 100].values

    xplot = []
    for i in range(1, len(x)):
        xplot.append(i)

    plt.plot(xplot, y, color='green', linewidth=1, label='Actual Price')
# plotting the initial datapoints
    plt.show()


"""
Allows a user to view their income based off
the passed in parameters used to calculate it.
"""
def view_and_calculate_income(gross_wages, taxable_interest, dividends,
ira_deduction):
    # Calculates income based on the below features
    income = (gross_wages
             + taxable_interest
             + dividends
             - ira_deduction)
    print(income)
```

Final Remarks to keep in mind

Code Line Length: Limit the length of lines to help improve readability. PEP8 recommends no more then 79, however up to 120 is acceptable if needed.

Function/Method Length: Try to keep functions/methods short and focused on a specific task. Long functions should be broken down into smaller, more manageable ones.

Error Handling: Implement proper error handling and communicate errors clearly. The goal is to identify and properly respond to errors and exceptions. When there are clear guidelines for reporting, logging, and handling errors, your code becomes more reliable and, over time, more error-proof.

File Organization: Organize code into logical sections or modules. Follow a consistent file and directory structure.

Documentation Standards

The documentation standards are the guidelines and best practices for creating and maintaining clear, consistent, and useful documentation for this project. This includes how to organize and structure information, how to format and style content. Furthermore, this document should be used as an example of how to format documentation for this project.

Styling Headers

For main headers such as the examples in this document; Overall, Coding Standards and Documentation Standards. The heading size will be the 'h3' designation or in text `###`. For sub-headers these will be 'h4' or in text `####`. Some examples in this document include; styling headers, comments and example python file.

Styling Code

To style code of different lines, a user needs too use 3 apostrophes ``` at the start and end of the code block. With the code being styled and formatted in the way it is present on an IDE with the correct indentations and spacing's.

Styling Bullet Points

Bullet points can be styled in 1 of two ways, if the user intends to make a numbered list, like in the branch naming conventions. It should be as follows:

1. Example 1
2. Example 2

If the user intended to use classic bullet points they need to use the * character in order to achieve this. An example of how this would look can be seen below:

- Bullet Point 1
- Bullet Point 2

Adding in Links

In an effort to make URL/URI links more readable for a developer, the link will be given link text, followed by the URL/URI. To do this a user would need to place the link text in square brackets `[]` proceeded by round brackets or parentheses `()`. Combined they will look like this `[]()`.

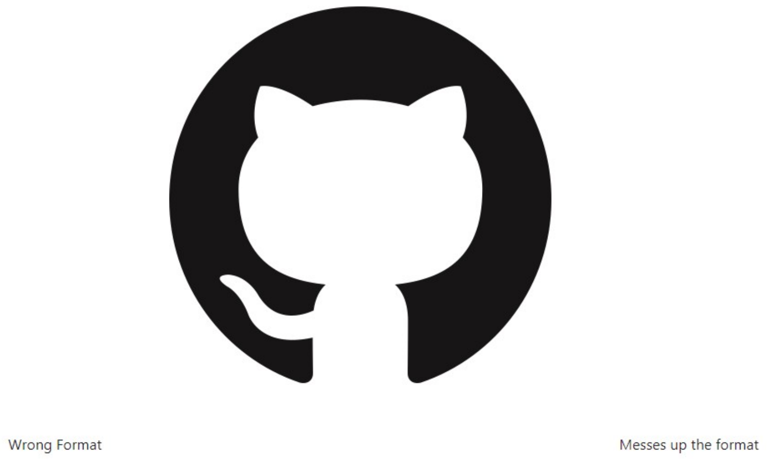
If a developer opts to name the link in the square bracket, it will be displayed as blue hyperlink, which is underlined. As seen above, in an example linking to PEP8 documentation.

Adding Photos

To add a photo into markdown, it follows the same pattern as adding a link, with the only exception being an exclamation mark the brackets. An example looks like this `![]()`.

A user can choose to add a name to the photo or leave it blank. But to add a photo I would suggest a user leaves a line spacing before and after the picture to allow it to be formatted correctly. As seen in the example below:

Because if not, then the formatting can looks off as seen below:



Wrong Format Messes up the format