Tarea Corta 1 - IC4302 Bases de Datos II

Introduction

Following you can find the documentation for the Tarea Corta 1 of the course IC4302 - Databases II. This homework assignment for the Databases II course involves developing an application that interacts with multiple database systems, including MariaDB, PostgreSQL, and Elasticsearch. The application is designed to load a dataset into these databases and expose consistent API endpoints for querying the data, regardless of the underlying database technology. To optimize performance, the application includes caching capabilities using Memcached or Redis, and it is instrumented with Prometheus to monitor key metrics such as HTTP request counts, query response times, and cache efficiency.

The application is deployed in a Kubernetes environment using Helm Charts, ensuring scalability and manageability. Docker images are created to package only the necessary components, depending on the specific database interactions and caching mechanisms. To evaluate the performance under different conditions, load tests are conducted using Gatling, which measures the system's behavior in terms of resource usage and query performance. The results of these tests and the metris provided can be visualized using Grafana dashboards, which display key performance indicators and help identify bottlenecks or areas for optimization. In conclusion, this project provides hands-on experience in integrating diverse technologies, deploying containerized applications, and optimizing performance in a real-world scenario.

Team Members

- Victor Aymerich
- Anthony Barrantes
- Fabricio Solis
- Melanie Wong
- Pavel Zamora

Next up, you will find the requirements to run the application, the steps to execute it, testing examples and the recommendations and conclusions we have gathered from the homework.

Requirements

The requirements for the project are the following:

- Create an user in DockerHub
- Install Docker Desktop, if you are using MacOS, please make sure you select the right installer for your CPU architecture.
- Open Docker Desktop, go to **Settings > Kubernetes** and enable Kubernetes.

K8s

- Install Kubectl
- Install Helm
- Install Visual Studio Code
- Install Lens

Building the docker images

There is a script to build the docker images, to execute it in a bash shell execute:

```
cd ./TC1/docker
./build.sh nereo08
```

Change **nereo08** to your DockerHub username

Please take a look on the script contents to make sure you understand what is done under the hood. This script will build the images for the components of the homework, including the database, the API, the cache, and the monitoring components.

Helm Charts

Configure

- Open the file **TC1/charts/app/values.yaml**
- Replace **nereo08** by your DockerHub username

```
config:
  docker_registry: nereo08
```

Install

Execute:

```
cd ./TC1/charts
./install.sh
```

Here we are installing the components in the Kubernetes cluster. The script will install the database, the API, the cache, and the monitoring components. It is important to have the DockerHub username set correctly in the script to ensure the correct components are installed.

Uninstall

Execute:

```
cd ./TC1/charts
./uninstall.sh
```

This script will uninstall the components from the Kubernetes cluster. It is important to have the DockerHub username set correctly in the script to ensure the correct components are removed.

Access Debug Pod

```
# copy the name that says debug from the following command
kubectl get pods
# then replace debug-844bb45d6f-9jt45 by that name
kubectl exec --stdin --tty debug-844bb45d6f-9jt45 -- /bin/bash
```

In case you need to access the debug pod to check the logs or execute some commands, you can use the previous command to access it.

How to Test

To test the whole project together, you can follow the next steps: First you need to do the build and install steps, in this time you have to choose the flasks that will be used for this you have to change in TC1\charts\app\values.yaml , after that to verify that the project is working correctly you can follow the next steps:

- Check Docker images are running, you can check this in the Docker Desktop application. In this area it is important to check the different images, you can check the logs of the images to see if there are any errors. You should check the logs of the FlaskApp, the MariaDB, PostgreSQL, Redis, Memcached, Prometheus, and Grafana images.

Atter that you can check in Lens that the pods are running correctly, here you can check the logs of the pods to see if there are any errors and the status of the pods.

Once you have checked all the logs and the data is being processed correctly, you can consider to go to Grafana to see the metrics of the project. To do this you should forward the port of the Grafana service called **grafana-deployment-6d4544597-stfnn** to you local machine. From here you can access the Grafana dashboards by going to http://localhost: (Port you decide) and logging in with the credentials **admin** and the password found in the GF_SECURITY_ADMIN_PASSWORD variable which can be found while observing the logs of the Grafana pod.

Once you enter Grafana you can see the different dashboards that are available, you can see the metrics of the different monitored databases, these databases are MariaDB, PostgreSQL, and Elasticsearch, plus Memcached and Redis. These metrics are important to see the performance of the databases and the cache systems. You can also see the metrics of the Flask API, which is the main component of the project. The metrics are all available due to the Prometheus component that is monitoring the different components of the project. Prometheus is scraping the metrics of the different components and storing them in its database, which is then used by Grafana to display the metrics in the dashboards. This is a very important part of the project as it allows you to see the performance of the different components and see if there are any bottlenecks or areas that need to be optimized. Together all these components allow you to have a complete overview of the project and see how it is performing in real time.

To test correctly all this databases we use gatling, a load testing tool that is used to simulate user behavior and measure the performance of the application under different conditions, in this case we have 4 different tests, check the gatling README in case you want to learn more about them. To start this gatling tests you just need to go to the gatling folder, open src\test\java and then run the engine.java aplication, for this you need to especifically have the java version 17 which you can download in the following link https://www.oracle.com/uk/java/technologies/downloads/. After running it the test will automatically start since there are no other configurations inside the folder.

Recommendations and Conclusions

Recommendations

To successfully complete the project, the following recommendations are provided to help you navigate the different components and technologies involved:

1. **Learn How to Use a Kubernetes Cluster**
   - **Recommendation**: It is very important to get a basic understanding of Kubernetes, what is their function and how it works, as the whole project is designed to run in a Kubernetes cluster.
   - **Reason**: Knowing how to deploy and manage the application in Kubernetes will help you run the project in a real-world environment.
2. **Learn How to Use Helm Charts**
   - **Recommendation**: Learn how to use Helm charts, which are used to deploy the application in Kubernetes. It is relevant to understand how they work and what it their function so you can deploy the application correctly.
   - **Reason**: The project uses Helm charts to deploy the application in Kubernetes. Knowing how to use Helm charts will help you manage the deployment of the application.

3 **Get Familiar with MariaDB**

- **Recommendation**: Learn how to access and use MariaDB, which the the main database used. You should learn the basics, including knowing how to connect to it, view tables, and check data for example.
- **Reason**: You will need to know how to connect to MariaDB to ensure certain components are working correctly, as well as to verify that data is being updated and stored properly.

4 **Get Familiar with Elasticsearch Indexing**

- **Recommendation**: Study the basics of Elasticsearch, focusing on how to create and manage indices where data will be stored.
- **Reason**: Understanding how to create and manage indices in Elasticsearch is crucial for storing and retrieving data efficiently.

5 **Get Familiar with PostgreSQL**

- **Recommendation**: Learn how to access and use PostgreSQL, which is another database used in the project. You should know how to connect to it, view tables, and check data.
- **Reason**: You will need to know how to connect to PostgreSQL to ensure certain components are working correctly, as well as to verify that data is being inserted and stored properly.

6 **Monitor Resource Usage**

- **Recommendation**: Keep an eye on resource usage (CPU, memory) while running the script to ensure it operates within acceptable limits.
- **Reason**: Monitoring helps prevent performance bottlenecks and ensures that the script runs efficiently, particularly for large datasets.

7 **Document Configuration and Steps**

- **Recommendation**: Document the configuration parameters and execution steps clearly for future reference and ease of use.
- **Reason**: Clear documentation helps users understand the setup and execution process, making it easier to troubleshoot and replicate the environment.

8 **Understand the Data Schema**

- **Recommendation**: Familiarize yourself with the data schema and relationships between tables to write efficient queries.
- **Reason**: Understanding the data schema helps optimize queries and ensures accurate results when fetching data from the databases.

9 **Understand the Endpoints**

- **Recommendation**: Learn how to use the endpoints to query data from the databases and understand the responses.
- **Reason**: Understanding the endpoints helps you retrieve specific data from the databases and analyze the results effectively.

10 **Familizarize with Gatling**

- **Recommendation**: Learn how to use Gatling to run load tests and analyze the performance of the application under different conditions.
- **Reason**: Gatling helps evaluate the performance of the application and identify potential bottlenecks or areas for optimization, by simulating real-world scenarios

with varying loads.

11 **Understand Prometheus and how to use it**

- **Recommendation**: Learn how to use Prometheus to monitor the different components of the project and how to scrape the metrics.
- **Reason**: Prometheus is a key component of the project as it captures the metrics of the different components and stores them in its database.

12 **Understand Grafana and the consequent Dashboards**

- **Recommendation**: Learn how to use Grafana to visualize the metrics of the different components of the project.
- **Reason**: Grafana is a key component of the project as it allows you to see the metrics of the different components and see how they are performing in real time.

13 **Understand the Cache Systems**

- **Recommendation**: Learn how to use Memcached and Redis, which are used as cache systems in the project.
- **Reason**: Understanding how to use Memcached and Redis will help you cache data and understand how it impacts the performance of the application.

14 **Understand the loading scripts**

- **Recommendation**: Understand how the loading scripts work and how they are used to load data into the databases.
- **Reason**: The loading scripts are essential for the way the databases are loaded with data, and understanding how they work will help you manage the data effectively and understand the structure of the databases. Please note that the information is loaded from CSV files.

Conclusions

1 The structure of the relational database is designed to make queries efficient, with tables separated by categories. This allows searches using indexes and foreign keys.
2 The use of PostgreSQL and MariaDB allows handling large volumes of data. Additionally, integrating Elasticsearch into the project provides options for fast and scalable searches in even larger or distributed databases.
3 The API offers direct and simple access to data through well-structured SQL queries. The relationships between tables allow obtaining details of races, times, and rankings, which is essential for analyzing the performance of drivers and constructors.
4 The database and API are designed in a way that facilitates the incorporation of new data (future drivers, additional circuits, etc.) without interrupting current operations.
5 The data source is configured to connect to Prometheus using an internal URL and is set as the default and editable data source, making it easy to adjust in the environment.
6 The dashboard-loader.yaml configuration allows the automatic creation of dashboards for multiple services (Elasticsearch, Redis, MariaDB, PostgreSQL and Memcached).
7 Each of these dashboards is enabled or disabled through the values.yaml file, allowing for flexible deployment based on monitoring needs.
8 The dashboards are based on pre-existing Grafana configurations, making it easier to adopt best practices for monitoring.
9 The dashboards follow a standard Grafana structure, using panels like Graph and Singlestat, which are essential for metric visualizations and alerts.
10 Each dashboard's JSON file defines inputs such as DS_PROMETHEUS to connect to the Prometheus data source.

11 The project is designed to provide a comprehensive monitoring solution for the different components, allowing users to visualize key metrics and identify performance bottlenecks or areas for optimization.
12 The integration of diverse technologies, helps create a application that uses optimal solutions for different tasks, such as data storage, retrieval, caching, and monitoring.

Components

API

The API developed in Flask allows users to access Formula 1 data stored in PostgreSQL and MariaDB databases. Routes are built that execute SQL queries on various tables related to races, drivers, teams, circuits, and Formula 1 events. The main routes include:

1 Routes for obtaining drivers and constructors:

/drivers: Returns a list of all drivers. /constructors: Returns a list of all constructors.

2 Routes for obtaining laps and times:

/circuit/int:id/laps: Provides the laps recorded at a specific circuit. /drivers/int:id/laps: Provides the laps completed by a driver in all races.

3 Other routes:

/driver/int:id/total_races: Returns the total number of races a specific driver has participated in. /driver/int:id /pitstops: Details the pit stops of a driver in different races.

Databases:

The databases used in the project are MariaDB and PostgreSQL and ElasticSearch. These databases are widely used in the industry and offer features for storing and managing data.

MariaDB is a popular open-source relational database management system that is compatible with MySQL. It provides excellent performance, scalability, and reliability. MariaDB ensures data integrity and consistency. It also offers advanced features such as replication, clustering, and high availability, making it suitable for handling large volumes of data.

PostgreSQL, on the other hand, is a powerful open-source object-relational database management system. It is known for its extensibility, flexibility, and support for advanced SQL features. PostgreSQL offers a wide range of data types, indexing options, and query optimization techniques, allowing for efficient data retrieval and manipulation. It also provides support for transactions and offers features like replication, partitioning, and full-text search.

Elasticsearch is also a popular NoSQL database that is used for full-text search and analytics. It is designed for real-time search and analysis of large volumes of data. It offers features like full-text search, aggregations, and spatial search, making it suitable for handling complex data structures and performing advanced queries.

Both MariaDB and PostgreSQL are well-suited for handling complex data structures and performing complex queries. They offer strong data consistency, reliability, and security. The choice between the two databases depends on specific project requirements, familiarity with the technology, and the need for specific features or compatibility with existing systems. On the other hand, Elasticsearch is ideal for full-text search and analytics, providing fast and scalable search capabilities for large volumes of data.

Overall, the combination of MariaDB, PostgreSQL and ElasticSearch in this homework ensures efficient and

reliable data storage and retrieval, enabling the application to handle large amounts of data effectively.

The structure of the databases is designed to store historical Formula 1 data. The database consists of several tables related to each other through foreign keys, allowing for queries between the data. The main tables are:

1. CIRCUIT: Contains information about the circuits where the races take place, such as geographical location and name.
2. CONSTRUCTOR, DRIVER: Store detailed information about the teams and drivers who have participated in F1 seasons, including their nationality and name.
3. RACE, RESULT: Include details of each race, such as date, circuit, and results for each driver, along with the corresponding constructor, rankings, times, and points earned.
4. LAP_TIME, PIT_STOP: The tables store information about the lap times of each driver and the details of pit stops during races.
5. DRIVER_STANDING, CONSTRUCTOR_STANDING: The standings tables record the points earned by drivers and constructors during each race and season.

Data Loading Scripts:

Each table in the database is loaded with historical data through CSV files that contain information about Formula 1.

Gatling

Gatling is an open-source load testing tool that is used to simulate user behavior and measure the performance of the application under different conditions. Gatling allows users to create scenarios that simulate real-world user interactions, such as browsing web pages, submitting forms, and making API requests. It generates load on the system by sending multiple requests concurrently and measures the response times, throughput, and error rates. Gatling provides detailed reports and metrics that help users analyze the performance of the application and identify bottlenecks or areas for optimization. In this project, Gatling is used to test the performance of the API under varying loads and analyze the system's behavior in terms of resource usage and query performance. The load tests are designed to simulate different scenarios, such as multiple users accessing the API simultaneously, querying large datasets, and performing complex operations. The results of the load tests are used to evaluate the performance of the application and identify areas that need improvement. Gatling is configured to run load tests against the API and measure key metrics, such as response times, throughput, and error rates. The load tests are executed in a controlled environment, allowing users to analyze the impact of different loads on the system and make informed decisions based on the data.

If you want to know more about the gatling configuration of this project it is recommended to check the README inside the gatling folder.

Prometheus

Prometheus is an open-source monitoring and alerting software that is used to collect and store metrics from various components of the project. Prometheus scrapes metrics from the different components, such as databases, API, cache systems, and monitoring tools, and stores them in a time-series database. It provides a specific query language, called PromQL, to retrieve and analyze metrics, enabling users to monitor the performance and health of the system. Prometheus offers features like service discovery, multi-dimensional data model, and powerful queries, making it suitable for monitoring complex environments. In this case Prometheus is used to monitor the different components of the project and store the metrics in its database. It is configured to scrape the metrics with the goal of visualizing them in Grafana. It is configured with automatic service discovery due to the use of technologies like MariaDB or PostgreSQL which for example count with an integrated exporter that allows Prometheus to scrape the metrics of the databases automatically. In the route TC1/charts/databases/values.yaml you can see the configuration of the databases to be scraped by Prometheus. The servicemonitor resource is used to configure the scraping of the metrics of the different components of the project. As this is enabled in the Helm chart, Prometheus will automatically scrape the metrics of the different components of the project.

Grafana

Grafana is an open-source analytics and monitoring platform that is used to visualize the metrics collected by Prometheus. Grafana provides a user-friendly interface to create dashboards and panels that display the metrics in a visually appealing way. It offers a wide range of visualization options, such as graphs, tables, and gauges, allowing users to customize the dashboards according to their needs. Grafana supports various data sources, including Prometheus, Elasticsearch, and InfluxDB, making it versatile for monitoring different systems. In this project, Grafana is used to create dashboards that display the metrics of the different components, such as databases, API and cache systems. The dashboards provide insights into the performance and health of the system, allowing users to identify bottlenecks, anomalies, or areas for optimization. Grafana is configured to connect to Prometheus as a data source, enabling it to retrieve the metrics stored in the Prometheus database and visualize them in the dashboards. The dashboards are designed to display key performance indicators, such as HTTP request counts, query response times, cache efficiency, and resource usage, helping users monitor the system in real time and make informed decisions based on the data.

In this homework, talking specifically about Grafana we can find six main dashboards, which are:

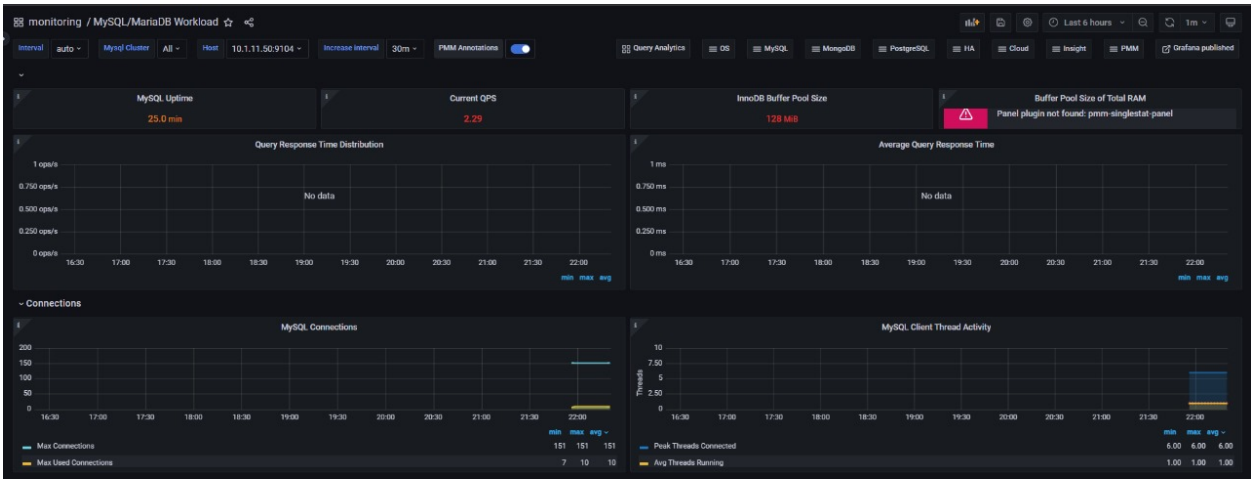
- **MariaDB Dashboard:** This dashboard provides a comprehensive view of MariaDB's performance and health. It tracks various metrics, helping users monitor database activity, identify potential bottlenecks, and optimize overall efficiency.
- **PostgreSQL Dashboard:** Offering a detailed look into PostgreSQL's performance, this dashboard enables users to monitor several key indicators. It assists in identifying performance trends and areas that may need attention for optimization.
- **Elasticsearch Dashboard:** Designed to give insights into Elasticsearch·s health, this dashboard presents a range of performance metrics. Users can easily track the system's efficiency and address potential issues.
- **Memcached Dashboard:** Focused on the health of the Memcached system, this dashboard offers a variety of performance metrics, enabling users to monitor and enhance caching effectiveness.
- **Redis Dashboard:** With this dashboard, users can monitor Redis' performance through several metrics. It highlights areas for improvement and provides an overview of the cache system·s health and efficiency.
- **Flask API Dashboard:** This dashboard gives an overview of the Flask API's performance by tracking multiple key metrics like average response time, request count, and cache efficiency by collecting cache misses and hits. This helps users identify performance bottlenecks and optimize API response times.

Tests

For the project we did the necessary tests to the databases now we are gonna show the results based on grafana dashboards:

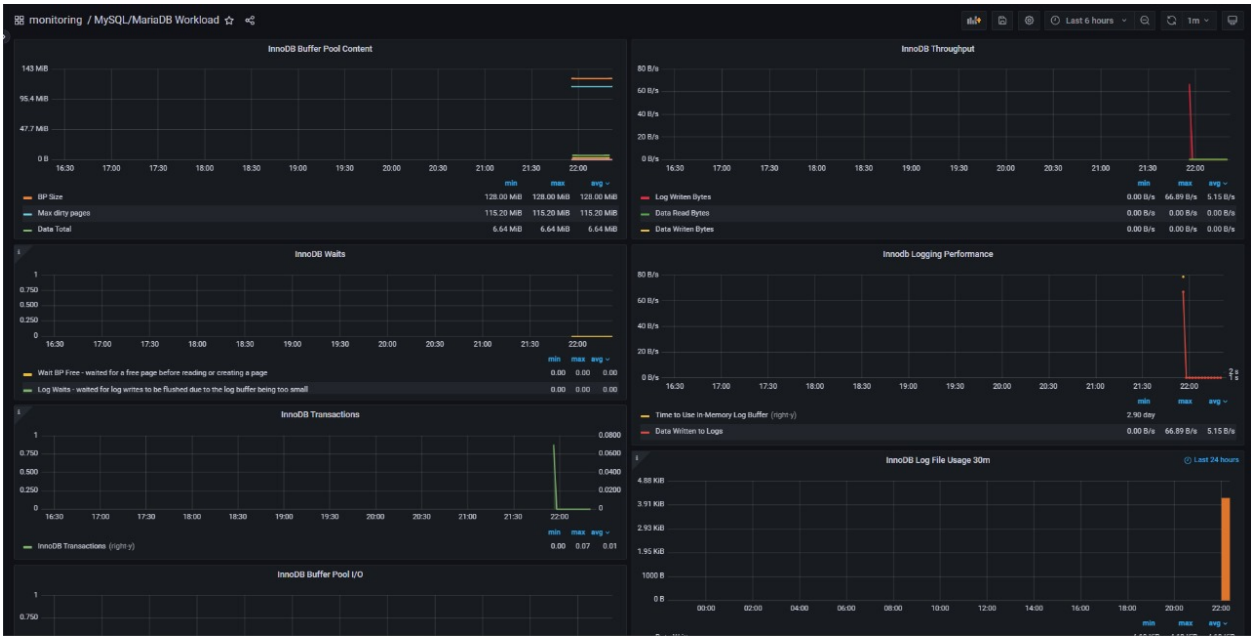
MariaDB

Without cache



Prueba

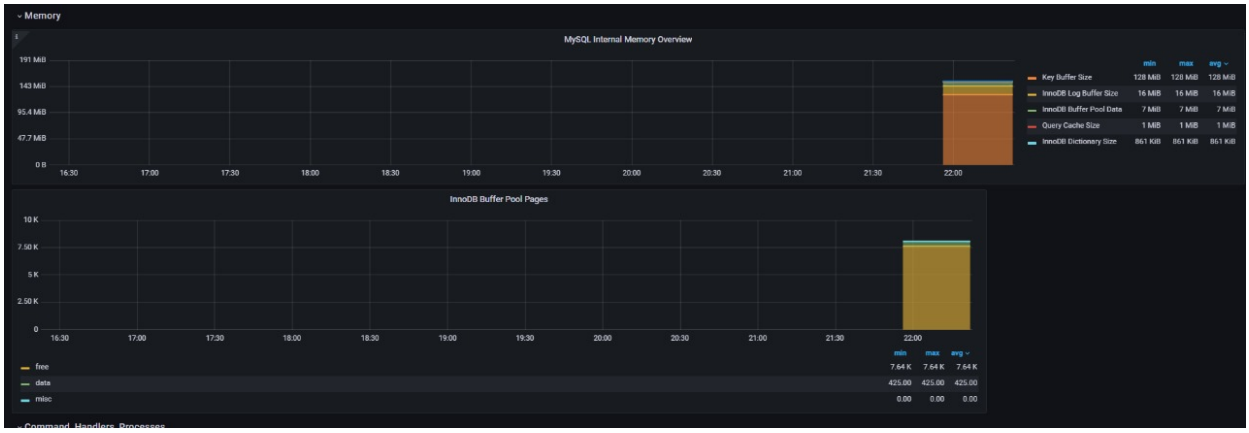
Prueba



Prueba

Prueba

Prueba



Prueba

PostgreSQl

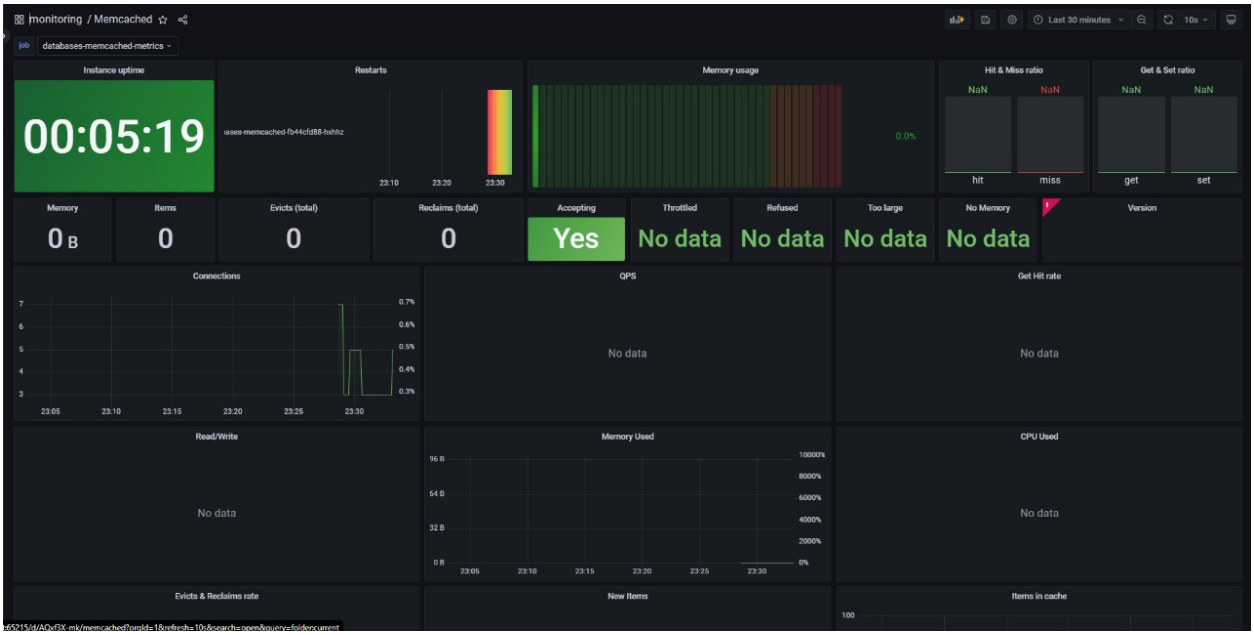
```
Simulation DatabasePerformanceSimulation completed in 900 seconds
Parsing log file(s)...
Parsing log file(s) done
Generating reports...

================================================================================
---- Global Information --------------------------------------------------------
> request count                                       9080 (OK=9079   KO=1     )
> min response time                                     26 (OK=26     KO=44    )
> max response time                                   5641 (OK=5641   KO=44    )
> mean response time                                   348 (OK=348    KO=44    )
> std deviation                                        777 (OK=777    KO=0     )
> response time 50th percentile                         48 (OK=48     KO=44    )
> response time 75th percentile                        202 (OK=202    KO=44    )
> response time 95th percentile                       2194 (OK=2194   KO=44    )
> response time 99th percentile                       3980 (OK=3980   KO=44    )
> mean requests/sec                                 10.089 (OK=10.088 KO=0.001 )
---- Response Time Distribution ------------------------------------------------
> t < 800 ms                                          8095 ( 89%)
> 800 ms <= t < 1200 ms                                255 (  3%)
> t ? 1200 ms                                          729 (  8%)
> failed                                                 1 (  0%)
---- Errors --------------------------------------------------------------------
> status.find.is(200), but actually found 500                    1 (100.0%)
================================================================================
```

Prueba

Prueba

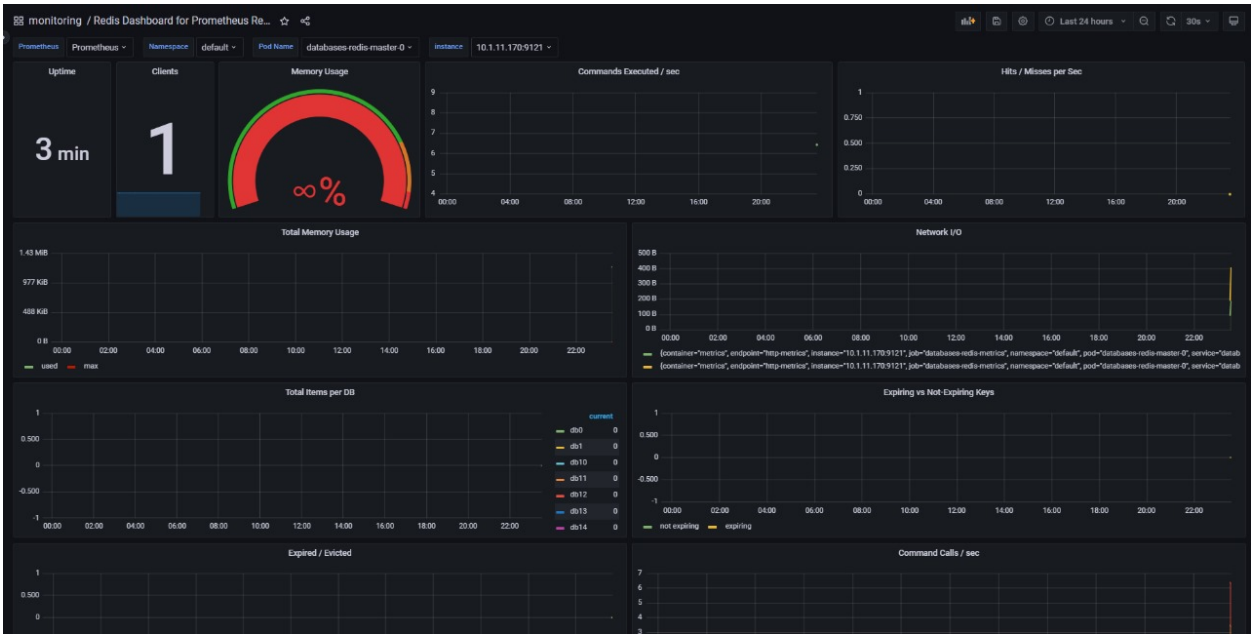
The following pictures are proof the dashboards implementation in grafana

Memcached



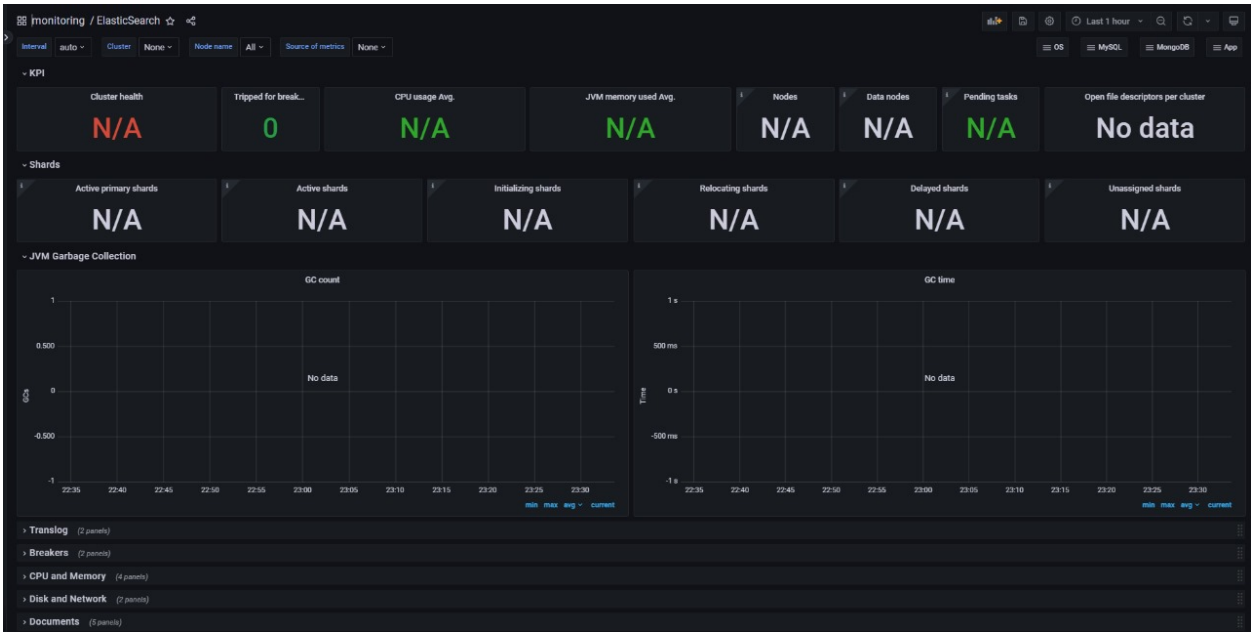
Prueba

Prometheus



Prueba

Elastic search



Prueba

References

- [1] "Dockerfile reference," Docker Documentation. [Online]. Available: https://docs.docker.com/reference/dockerfile/#overview. [Accessed: Sep. 4, 2024].
- [2] "kubectl commands," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands. [Accessed: Sep. 4, 2024].
- [3] "MariaDB Documentation," MariaDB Knowledge Base. [Online]. Available: https://mariadb.com/kb/en/documentation/. [Accessed: Sep. 4, 2024].
- [4] "PostgreSQL Documentation," PostgreSQL Documentation. [Online]. Available: https://www.postgresql.org/docs/. [Accessed: Sep. 4, 2024].
- [5] "Grafana Documentation," Grafana Documentation. [Online]. Available: https://grafana.com/docs/. [Accessed: Sep. 4, 2024].