Comparison Between Time Series DataBases Prometheus & Apache Cassandra

Androni Artemis
dept. Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el18117@mail.ntua.gr

Papadostefanaki Marianna
dept. Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el18099@mail.ntua.gr

Tsertou Eleni
dept. Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el18165@mail.ntua.gr

***Abstract*—Time-series databases play a critical role in modern data management, providing efficient storage and retrieval of timestamped data for a wide range of applications, from monitoring and analytics to IoT and financial systems. In this project, we conduct a comprehensive comparison between two prominent time-series databases, Prometheus and Cassandra, addressing crucial aspects of their deployment and performance. We start by setting up both databases, including the distributed edition for Cassandra, providing valuable insights into their ease of installation and scalability. Our experiments extend to data generation, loading, and compression efficiency assessment. We design standardized queries encompassing point, range, and complex operations, enabling us to evaluate their querying capabilities comprehensively. Employing client-like processes, we measure key performance metrics, including query latency, throughput, and CPU load, facilitating an informed comparison of their strengths and weaknesses. In conclusion, this analysis not only enhances our understanding of these critical data management tools but also serves as a valuable educational resource in the field of database systems.**

Keywords—time series database, Prometheus, Apache Cassandra, TSBS, NoSQL, Cassandra Query Language (CQL), nodetool, Prometheus Query Language (PromQL), Pushgateway, monitoring, data generation, pull-based approach

I. Introduction

Time series data, a fundamental component of various fields, comprises a series of measurements collected from the same source over a specific time span. These data sets are ubiquitous and find application in diverse domains such as meteorology (for weather records), finance (for financial records), IoT (sensor data), networking (network data), IT infrastructure (CPU and server metrics), system optimization (performance monitoring), and healthcare (patient health metrics). The nature of time series data presents unique challenges due to their massive volume and distinct query patterns. To efficiently handle such data, time series databases are specifically designed and optimized for this purpose.

Choosing the right database system is critical, as it directly impacts system performance. Two prominent open-source time series databases, Prometheus and Apache Cassandra, are the focus of our research. Our objective is to conduct a comparative analysis of their performance and highlight their respective strengths and weaknesses.

To conduct our experiments, we generated synthetic time series data using the Time Series Benchmark Suite and loaded it into both Prometheus and Apache Cassandra. We carefully recorded various metrics, including the time required to load data at different scales (small, medium, and the final dataset). Additionally, we evaluated the performance of both databases by executing a set of predefined queries and assessing query latency, throughput and CPU usage.

This study aims to provide valuable insights into the suitability of Prometheus and Apache Cassandra for different use cases involving time series data, helping users make informed decisions about which database best aligns with their specific performance requirements and data management needs.

The paper is structured as follows. Section 2 provides a brief overview of Prometheus and Apache Cassandra. Section 3 presents the steps we followed to set up the two databases and the essential tools. Section 4 presents the data genertation, preprocessing and loading processes. Section 5 presents the query generation and execution. Section 6 gives evaluation and comparative results. Finally, section 7 summarizes the results and concludes the paper.

II. Time-Series Databases

A. Prometheus

Prometheus is an open-source monitoring and alerting system, specifically designed for time series data, providing reliability and availability. It was originally created at SoundCloud in 2012 and now it is a part of the Cloud Native Computing Foundation (CNCF) project [1].

Prometheus collects time-series data from various sources and stores it in a local Time Series Database (TSDB). The local TSDB is a fundamental component of Prometheus, responsible for storing and querying the collected metrics data efficiently.

The local TSDB in Prometheus is designed to provide fast and reliable storage for time-series data. It uses a compressed and indexed log-structured storage engine to achieve efficient storage and retrieval of metrics. The TSDB organizes data into chunks and indexes them for quick access, allowing

Prometheus to handle large volumes of time-series data with low latency.

One of the key benefits of the local TSDB is its ability to handle high write and query throughput. It efficiently stores incoming time-series data and provides fast querying capabilities using its own query language called PromQL. With PromQL, users can perform a wide range of queries, including aggregations, filtering, and mathematical operations, to gain insights from the collected metrics data.

The local TSDB in Prometheus also supports data retention policies, allowing users to define how long the metrics data should be retained. This feature is particularly useful for managing storage space and ensuring that only relevant data is retained for analysis and monitoring purposes.

Data is organized into chunks for efficient storage and querying. Chunks are a fundamental unit of the TSDB and hold a collection of samples for a single time series. These chunks are stored in a custom, highly efficient format on local storage.

Finally, Prometheus constitutes a single-server, non-distributed system that uses a pull-based model to actively scrape metrics from targets via an HTTP endpoint at regular intervals. However, some cases require pushing data, which is supported via an intermediary gateway, where the targets send metrics instead of waiting to be scraped. Although the single server Prometheus nodes are autonomous and independent of distributed storage, they are able to share and aggregate data.

B. Apache Cassandra

Apache Cassandra is an open-source, distributed NoSQL database, designed to handle massive amount of data across many commodity servers or on cloud infrastructures, providing high availability, linear scalability and fault tolerance. Cassandra was originally designed at Facebook but now it is an Apache Software Foundation project.

Cassandra has a masterless and peer-to-peer architecture. The data is organized in clusters and in each cluster it is distributed across rings of nodes that are arranged in racks. Racks can be installed in more than one datacenters which can be spread across different geographical regions. Each node has a single instance of Cassandra and all nodes have the same capabilities and responsibilities. In each node, data are stored in column families, that are collections of rows, each consisting of a key-value pair. Moreover, Cassandra supports data replication, ensuring reliability and fault tolerance. A piece of data, chosen according to the replication strategy, can be stored on multiple nodes. The number of data replicas is indicated by the replication factor [2, 3, 4, 5].

Cassandra has its own query language, the Cassandra Query Language (CQL), which is quite similar to SQL. CQL is used to create database schemas and access and organize data within a cluster of nodes. The key concepts of the database structure are outlined bellow:

- Keyspace: It is the outermost object that defines how data is replicated on nodes per datacenter. Keyspaces consist of tables.
- Table: Determines the typed schema for a collection of partitions. Tables consist of partitions, which consist of rows and columns.
- Partition key: Determines how data is distributed across multiple nodes in the cluster. It is the first part of the primary key, used to identify the node where a row is stored. A function called partitioner hashes the partition key to generate a token, which represents a row and identifies the partition range it belongs to in a node.
- Clustering key: It is a part of the primary key, used for data sorting within the partition.
- Primary Key: Uniquely identifies a row and consists of one or more partition keys and zero or more clustering keys.
- Row: Contains a collection of columns identified by the primary key.
- Column: The basic data structure.

The distributed architecture of Cassandra as well as its ability to support time-based partitioning, make it a suitable option for storing and efficiently managing time-series data.

III. Systems Set Up

For the development of the systems and our experiments, we used resources available by the GRNET's cloud service ~okeanos-knossos. We were provided with 2 virtual machines with the following specifications respectively:

- CPUs: 8/4
- RAM: 16/8GB
- Disk: 30GB
- Operating System: Ubuntu Server 16.04.3 LTS

The machines were used to set up a cluster consisting of 2 nodes for Cassandras' distributed edition. Additionally, we set up a local IPv4 network (192.168.0.0/24) for the nodes to communicate. After the set up, we used SSH to connect to our machines.

Before installing Prometheus and Cassandra we installed the latest version of Java 8 and Python 3.8.0 according to the official guides [6, 7].

A. Installation and Configuration of Prometheus

We downloaded the latest release of Prometheus from the official Prometheus website [8] and extracted the downloaded package.

Once extracted, we navigated to the Prometheus directory and created a configuration file named `prometheus.yml`. In this basic configuration, we specified the global scrape interval, which determines how often Prometheus collects metrics, and defined scrape jobs that target specific endpoints.

The Prometheus Pushgateway is a vital component in the Prometheus ecosystem that allows the collection and storage of metrics from batch jobs, service-level exports, or other applications that cannot be directly scraped by Prometheus. As before, we downloaded the appropriate Pushgateway version and extracted the tarball package to create a directory containing the Pushgateway binaries [9].

We set the scrape interval to 2 seconds and configured Prometheus to scrape metrics from a Pushgateway instance running locally on port 9091. We also enabled the `honor_labels` parameter. Enabling this means that Prometheus will use data from the Pushgateway instead of adding certain special labels or custom labels as specified in the scraping configuration when collecting data.

After saving the configuration, we started Prometheus and Pushgateway by running the following commands in their respective directories:

```
./prometheus --config.file=prometheus.yml
./pushgateway
```

With Prometheus up and running, we could access its web-based user interface at `http://localhost:9090/` to explore collected metrics, build queries, and set up alerting rules.

We could access Pushgateway at its default address `http://localhost:9091/`. Metrics can be pushed to the Pushgateway via HTTP POST requests.

B. *Installation and Configuration of Cassandra*

We downloaded the Cassandra tarball package from the official Apache Cassandra website [10] on each machine. After downloading the tarball, we extracted its contents and proceeded to customize the Cassandra configuration file, `cassandra.yaml`, for each node.

To configure Cassandra to work as a cluster, we kept the cluster name to the same value for both nodes in our cluster.

We specified the IP address that the node will use to listen for incoming connections from other nodes in the cluster and defined the initial contact points (seed nodes) that other nodes can use to join the cluster.

After making these changes in the `cassandra.yaml` files, we started the Cassandra service on each node, specifically with the command `./bin/cassandra`, ensuring that the seed node(s) is started first, and then the other node(s) can join the cluster. This configuration will allow the Cassandra nodes to operate as a cluster, with data distribution and replication across the nodes.

We also used nodetool, a command-line interface for managing and monitoring Cassandra's clusters. This utility ships with Cassandra and appears in the Cassandra's bin directory, providing various commands for viewing detailed metrics for tables, server metrics and compaction statistics [11].

We checked the health status of our cluster by executing the following command:

```
./bin/nodetool/ status
```

Also, for safe termination of our nodes we used the following command:

```
./bin/nodetool/ stopdaemon
```

For data loading purposes in Cassandra we needed to install the DataStax Bulk Loader (dsbulk), a versatile command-line tool designed for efficiently handling data loading, unloading, and counting tasks.

Once more, we downloaded the appropriate dsbulk distribution package from the official DataStax website [12], ensuring that we selected the version compatible with our Cassandra deployment and system architecture.

IV. DATASET

A. *Data Generation*

Each database was loaded with identical data. For the generation of our data we used the Time Series Benchmark Suite (TSBS), which is a collection of Go programs that generate time-series datasets and benchmark read and write performance of several databases [13, 14]. TSBS supports three use cases, the "CPU only", the "Dev ops" and the "Internet of Things (IoT)". We chose the cpu-only use case in the form that produces 10 CPU metrics per reading interval. Each reading consists of the following elements:

- The table name it belongs to according to the datatype, which is "series_bigint" in our case.
- The data source, which is cpu for the "cpu-only" use case.
- Several comma-separated tags on the form of <label> = <value> that identify the host, including its location, datacenter, operating system, team, etc.
- The field label that indicates the metric that is the usage type in our case.
- The date of the reading in YYYY-MM-DD form.
- The timestamp for the record in nanoseconds.
- The reading itself i.e. the value of the metric.

An example reading is the following:

```
series_bigint, cpu, hostname=host_0, region=us-west-
1, datacenter=us-west 1b, rack=90, os=Ubuntu15.10,
arch=x64, team=CHI, service=4, service_version=1,
service_environment=test, usage_user, 2023-01-01,
1672531200000000000,37
```

We installed the TSBS Go programs and all the required binaries with the following commands:

```
go get github.com/timescale/tsbs
cd $GOPATH/src/github.com/timescale/tsbs
make
```

TSBS provides a data generation tool that is configurable so that we can specify the number of simulated devices, the time interval and overall timespan they are generating data and the database we aim to develop. We generated our dataset with the following command in the `tsbs-master/bin/` directory:

```
./tsbs_generate_data --use-case="cpu-only" --seed=42
--scale=100 --timestamp-start="2023-01-01T00:00:00Z" --
timestamp-end="2023-01-04T00:00:00Z" --log-
interval="60s" --format="cassandra" | gzip >
/home/user/info_sys/large-data.gz
```

This configuration produces a dataset in "pseudo-CSV" format for Cassandra as we specified in the options of the command.

Information about the different sizes of generated data is presented in the table below:

TABLE I. GENERATED DATA INFORMATION

Data Size	Generated Data Information			
	Rows	*Host Devices*	*Metrics Interval*	*Total Time*
Small	60,000	100	60sec	1h
Medium	180,000	100	60sec	3h
Large	360,000	100	60sec	6h

B. Data Preprocessing

Prometheus primarily deals with time-series data. Its data model is centered around metric data that is organized into time-series, where each time-series consists of a metric name, a set of key-value labels (metadata), and a time stamp.

Cassandra is a NoSQL database that utilizes a wide-column store data model based on tables and columns.

Each dataset was loaded both in Cassandra and in Prometheus databases. Before loading the data into Cassandra database, we first modified the label-value pairs format so that the value string did not contain the label name. An example of the modified data for Cassandra is shown below:

```
series_bigint cpu host_0,us-west-1,us-west-1b,90,Ubuntu15.10,x64,CHI,4,1,test,usage_user,2023-01-01,1672531200000000000,37
```

Respectively, before we pushed the data to Prometheus, we transformed the label-value pairs into the Prometheus' simple text-based exposition format, as gauge type metrics i.e. numerical values that can arbitrarily increase or decrease.

```
usage_user{host="host_0", region="us-west-1", availability_zone="us-west-1b", os="Ubuntu15.10", architecture="x64", provider="CHI", test="test"} 37 1672531200000000000
```

C. Data Loading

In order to load the data into Prometheus database, we used the Prometheus Pushgateway. Subsequently, we used the curl command to push each metric into the Pushgateway specifying the URL of the Prometheus Pushgateway and the name of the corresponding job, which consists of the metric's name and the value of the counter that records the number of metrics.

We implemented a script file to streamline the process of pushing Prometheus metrics to the Pushgateway. This script dynamically generated a curl command for each Prometheus metric in our dataset. Each curl command was tailored to the specific metric, including the metric name, labels, and associated values. Additionally, within the same script, we incorporated a timing mechanism to measure the duration it took to load the entire dataset, consisting of all the metrics, to both the Pushgateway and subsequently to Prometheus.

In order to load the data into Apache Cassandra database, we used the dsbulk command-line tool [15]. Having created the required tables based on our queries, as described in the section above, we implemented a script file to run the dsbulk load command specifying the keyspace, the table and the URL for the dataset source csv file.

We monitored the data loading process recording the total time required to load the entire dataset in each database and the results are shown in the table below:

TABLE II. LOADING TIME

Data Size	Loading Time		
	Rows	*Prometheus*	*Apache Cassandra (average for all tables)*
Small	60,000	1521.94sec	7.6sec
Medium	180,000	4857.84sec	10.91sec
Large	360,000	9850.72sec	13.75sec

Upon analyzing the table above, a significant disparity in loading times between Prometheus and Cassandra becomes evident. This notable difference is primarily attributed to the data flow process. In Prometheus, data is pushed through the Pushgateway, introducing an additional layer of data transfer. This intermediary step inherently results in extended loading times, especially noticeable with larger dataset sizes. Conversely, Cassandra demonstrates commendable performance across all dataset sizes, consistently achieving notably lower loading times. This efficiency can be attributed to Cassandra's distributed architecture and streamlined data management capabilities. Moreover, loading times for Cassandra remain relatively stable even as the dataset size increases, underscoring its scalability and ability to handle substantial data volumes with efficiency.

V. QUERY GENERATION AND EXECUTION

In order to get a comprehensive view of our databases' performance, we created a set of queries, mainly targeting the time dimension and executed them on both databases, recording relevant performance metrics. We focused on queries that use aggregate or grouping functions such as count, average and maximum. We created the following 10 queries:

1. Get the values of a specific usage type for each host in the last timestamp.
2. Get the values of all usage types for a specific host in the last timestamp.
3. Get the average value of a specific usage type over all hosts in the last timestamp.
4. Get the average value of a specific usage type for a specific host (host_3).
5. Get the maximum value of a specific usage type for each host.
6. Count how many times the value of a specific usage type of a specific host is grater or equal to a specific value.

7. Get the maximum value of a specific usage type for a specific datacenter in the last timestamp.
8. Get the values of all usage types for a specific datacenter in the last timestamp.
9. Get the maximum value in a specific datacenter.
10. Get the average value of a specific usage type for a specific datacenter and a specific service.

A. *Prometheus Queries*

In Prometheus, we expressed the above queries using the Prometheus Query Language (PromQL) as follows:

Query 1: `usage_idle`

Query 2: `{hostname="host_2"}`

Query 3: `avg(usage_user)`

Query 4:
```
max(max_over_time(usage_user{hostname=
"host_3"}[5h]))
```

Query 5:
```
avg(avg_over_time(usage_irq[5h]))
by(hostname)
```

Query 6: `sum(usage_iowait)`

Query 7:
```
max(max_over_time(usage_guest{datacenter=
"us-west-2b"}[5h]))
```

Query 8: `{datacenter="us-west-1b"}`

Query 9:
```
max(max_over_time({datacenter=
"apsoutheast-1a"}[5h]))
```

Query 10:
```
avg(avg_over_time(usage_iowait{
datacenter="us-west-1b", service="4"}[5h]))
```

We entered these expressions in the Prometheus's expression browser which is available on the Prometheus server at /graph and viewed their result both in a tabular format and as time-series graphs.

B. *Cassandra Queries*

Cassandra uses a query-based data modeling approach in which data is organized around the queries. This means that we had to model our queries first and then construct the appropriate table structures to achieve an efficient query execution. Efficiency is gained when data are appropriately grouped together on nodes to improve the performance of reads and writes. The way data are partitioned and distributed across a cluster is indicated by the partition key, which is the first component of the table's primary key. Moreover, within a partition, rows are clustered by the remaining columns of the primary key that constitute the clustering key. Therefore, the primary key should be specified carefully, also taking into consideration that since it uniquely identifies each row in the table, inserting data with a particular primary key will overwrite the existing data with the same primary key.

In Cassandra, we executed our queries using the Cassandra Query Language (CQL). The tables we created for each query and the corresponding query expressions in CQL are presented below.

For Query 1 we created the table `query_1_table` which consists of the columns `hostname`, `usage_type`, `timestamp` and `value`, with `usage_type` as partition key and `timestamp` and `hostname` as clustering keys.

Query 1:
```
SELECT hostname, value FROM query_1_table WHERE
usage_type='usage_idle' ORDER BY timestamp DESC
LIMIT 100
```

For Query 2 we created the table `query_2_table` which consists of the same columns as the `query_1_table` with the difference that it has `hostname` as partition key and `timestamp` and `usage_type` as clustering key.

Query 2:
```
SELECT usage_type, value FROM query_2_table WHERE
hostname='host_2' ORDER BY timestamp DESC LIMIT
10
```

For Query 3 we created the `query_3_table`, which has the same columns as the previous tables with the difference that it has `usage_type` as partition key and `hostname` as clustering key.

Query 3:
```
SELECT AVG(value) as avg_value FROM query_3_table
WHERE usage_type='usage_user'
```

For Query 4 we created the table `query_4_table`, which consists of the same columns as the previous tables with the difference that it has both `hostname` and `usage_type` as partition keys and `timestamp` as clustering key. This way, `usage_types` of the same host are stored in the same node.

Query 4:
```
SELECT MAX(value) as max_value FROM query_4_table
WHERE hostname='host_3' AND
usage_type='usage_user'
```

For Query 5 we created the table `query_5_table`, which consists of the same columns as the previous tables with the difference that it has the `usage_type` as partition key and `hostname` and `timestamp` as clustering keys.

Query 5:
```
SELECT hostname, AVG(value) as max_value FROM
query_5_table WHERE usage_type='usage_irq' GROUP
BY hostname
```

For Query 6 we created the `query_6_table`, which is practically identical to `query_5_table`. In addition, this query required to create an index on this table for the `hostname` column.

Query 6:
```
SELECT SUM(value) as summary FROM query_6_table
WHERE usage_type='usage_iowait' AND
timestamp=1672552740000000000
```

For Query 7 we created the `query_7_table`, which consists of the columns `datacenter`, `usage_type`, `timestamp` and `value`, with `datacenter` and `usage_type` as partition keys and `timestamp` as clustering key.

Query 7:

```
SELECT MAX(value) as max_value FROM query_7_table
WHERE datacenter='us-west-2b' AND
usage_type = 'usage_guest'
```

For Query 8 we created the `query_8_table`, which consists of the columns `hostname`, `datacenter`, `usage_type`, `timestamp` and `value`, with `datacenter` as partition key and `timestamp`, `usage_type` and `hostname` as clustering keys.

Query 8:

```
SELECT hostname, usage_type, value FROM
query_8_table WHERE datacenter='us-west-1b' ORDER
BY timestamp DESC LIMIT 80
```

For Query 9 we created the `query_9_table`, which consists of the same columns as `query_7_table`, with the difference that it has `datacenter` as partition key and `timestamp` and `usage_type` as clustering keys.

Query 9:

```
SELECT MAX(value) as max_value FROM query_9_table
WHERE datacenter='ap-southeast-1a'
```

For Query 10, we created the `query_10_table`, which consists of the columns `hostname`, `datacenter`, `service`, `usage_type`, `timestamp` and `value`. The partition keys are `datacenter` and `usage_type` and the clustering keys are `hostname`, `service` and `timestamp`. In addition, we created an index on this table for the column service.

Query 10:

```
SELECT AVG(value) as avg_value FROM
query_10_table WHERE datacenter='us-west-1b' AND
usage_type='usage_iowait' AND service='4'
```

Cqlsh (Cassandra Query Language Shell) is a command-line interface tool provided by Cassandra for interacting with its databases using the CQL.

We utilized the cqlsh command-line interface to execute each query by using the following commands inside the bin directory of each node:

```
./cqlsh <node IP>

cqlsh> USE mykeyspace;

cqlsh:mykeyspace> SELECT hostname, value FROM
query_1_table WHERE usage_type='usage_idle' ORDER
BY timestamp DESC LIMIT 100;
```

Despite the differences in query languages and database structures, our testing showed that both consistently provided the same results for all queries.

VI. Evaluation

A. Performance Evaluation

In our performance evaluation, we conducted an in-depth assessment of the two databases. The evaluation process involved executing the predefined queries on both databases and monitoring their performance by utilizing the following key performance metrics:

Latency: Latency measures the time taken for a single query to be processed and receive a response from the database. It reflects the responsiveness of the system when retrieving data. Lower latency values indicate faster query performance.

Total Latency: Total latency represents the cumulative time taken for all queries to be processed by the database system. It offers an overview of the overall time spent on query execution.

Throughput: Throughput measures how many queries can be processed per unit of time, typically expressed as queries per second (QPS). It assesses the database's overall processing capacity and responsiveness.

Total CPU Usage: Total CPU usage reflects the change in CPU utilization during the entire query execution process. It indicates the impact of query execution on the system's CPU resources.

By utilizing these key performance metrics, our evaluation aimed to provide a holistic understanding of how both Prometheus and Cassandra perform in terms of query processing, resource utilization, and responsiveness. These insights are crucial for making informed decisions about database selection, optimization, and resource allocation, tailored to specific time series data management requirements and use cases.

We initiated the experimentation by scripting the execution of queries on both Prometheus and Cassandra. Each query's execution was carefully timed, and we captured the key performance metrics. The results of our experimentation are summarized in the provided tables.

TABLE III. QUERY LATENCY

Query No.	**Query Latency (sec)**	
	Prometheus	***Cassandra***
1	0.0371	0.0118
2	0.0358	0.0072
3	0.0359	0.0063
4	0.0366	0.0071
5	0.0360	0.1698
6	0.0359	0.0138
7	0.0363	0.0051
8	0.0364	0.0063
9	0.0366	0.0159
10	0.0693	0.0166

TABLE IV. TOTAL METRICS

Metric	**Total Metrics**	
	Prometheus	***Cassandra***
Latency	0.39sec	0.26sec
Throughput	25.25QPS	38.42QPS
CPU Usage	0.09	1.6%

The query latency values in Table III indicate the time taken by both databases to process each individual query. Lower latency values are generally desirable, as they signify faster query performance. We observed that Cassandra consistently exhibited lower latency compared to Prometheus for most queries. This suggests that Cassandra's query processing efficiency is notably higher.

Table IV presents the total metrics aggregated over all queries for both databases. The total latency for Prometheus was measured at 0.39 seconds, whereas Cassandra recorded a total latency of 0.26 seconds. This implies that, in aggregate, Cassandra completed the queries more swiftly. Moreover, the throughput metric reveals that Cassandra handled a higher number of queries per second (38.42 QPS) compared to Prometheus (25.25 QPS). Additionally, the CPU usage metric illustrates resource utilization, and here we observed that Prometheus exhibited significantly lower CPU usage (0.09%) compared to Cassandra (0.09%), suggesting a more efficient utilization of system resources.

Our evaluation suggests that Cassandra is better suited for handling time series data workloads, as it consistently demonstrated lower query latency and higher throughput. The lower total latency for Cassandra indicates its efficiency in processing a larger volume of queries within a given time frame. However, the higher CPU usage observed in Cassandra could imply a potential trade-off between query performance and resource utilization efficiency. In contrast, Prometheus exhibited lower CPU usage, suggesting more efficient resource management, but at the cost of slightly higher query latency and lower throughput. Therefore, the choice between Prometheus and Cassandra should consider specific use case requirements, with Cassandra excelling in scenarios prioritizing query performance and throughput, while Prometheus may be preferred for resource-constrained environments where efficient resource utilization is crucial.

B. *JConsole*

In order to measure the query performance of Cassandra we also experimented with an external monitoring system, the JConsole application.

Cassandra exposes a number of statistics and management operations using MBeans. This information can be accessed via Java Management Extensions (JMX), which is a Java technology that supplies tools for manipulating and monitoring Java applications and services [16]. JConsole is a JMX compliant tool, which can collect the Cassandra's metrics per node through JMX and then filter, aggregate, and render them in the desired format.

JConsole accepts the metrics on availability and performance and the operations that Cassandra exposes and displays them in a Graphical User Interface (GUI) to be explored by the user [17, 18]. It consumes a significant amount of system resources and for this reason, we decided to run it on our host machine rather that the remote VMs. Since by default Cassandra allows JMX accessible only from local host, we had to enable the JMX remote connection by setting the following system properties on the configuration bash script file cassandra-env.sh:

```
LOCAL_JMX=no

Dcom.sun.management.jmxremote.rmi.port=
$JMX_PORT"
```

where `JMX_PORT=7199` is the number of the port through which we want to enable JMX RMI connections. Moreover, we set the public IP address of the VM as the server's hostname and disabled password authentication and ssl by setting the following system properties:

```
Djava.rmi.server.hostname=83.212.80.117

Dcom.sun.management.jmxremote.authenticate=false"

Dcom.sun.management.jmxremote.ssl=false
```

To launch JConsole we opened a terminal from our local machine and entered `jconsole`. Then, we selected the remote process option and provided the node's public address and the port Cassandra uses for JMX, that is 7199 by default. For each Cassandra node, JConsole provides six separate tab views: Overview, Memory, Threads, Classes, VM Summary and MBeans [19].

The Overview tab displays the overall information about the monitored JVM, consisting of Memory, Classes loaded, Threads Count and CPU usage graphs. We captured the CPU usage over time.

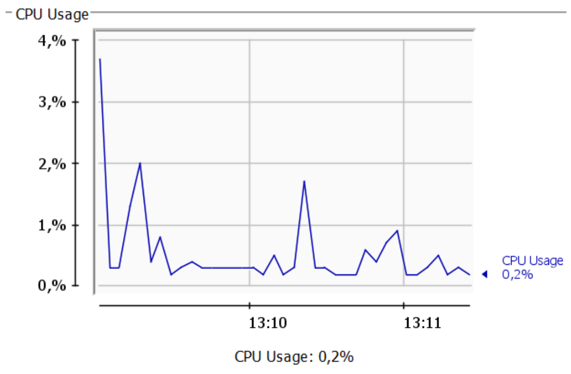

Fig. 1. CPU usage over time

MBeans is the most important area of JConsole. It provides a tree-like structure of MBeans, displaying all available JMX paths. We expanded the org.apache.cassandra.metrics of the MBeans tab, which contains almost all metrics needed to monitor a Cassandra Cluster, including metrics on CQL, clients, keyspaces, read repair, storage, threadpools and other topics. In subsequence, we measured the read latency and throughput both at the overall cluster level and for each individual table. In order to capture the read latency and throughput of each individual table we selected the Table folder and then the folder corresponding to the table we wanted to monitor. Below we present the JMX paths which mirror the JConsole interface's folder structure, for the metrics that we recorded:

Read Latency:

```
org.apache.cassandra.metrics:
type=ClientRequest,scope=(Read),name=TotalLatency
Attribute: Count
```

Read Throughput:

```
org.apache.cassandra.metrics:
type=ClientRequest,scope=(Read),name=Latency
Attribute: OneMinuteRate
```

Read Latency of each Table:

```
org.apache.cassandra.metrics:
type=Table,mykeyspace,<table_name>,
name=ReadLatency
Attribute: Count
```

Read Throughput of each Table:

```
org.apache.cassandra.metrics:
type=Table,mykeyspace,<table_name>,
name=ReadLatency
Attribute: OneMinuteRate
```

VII. Discussion

Our extensive evaluation of Prometheus and Cassandra in the context of handling time series data has provided valuable insights into their respective strengths and considerations. Based on our findings, it is evident that both databases offer distinct advantages and considerations, and the choice between them should be driven by specific use case requirements.

It is important to note that at its core, Prometheus is primarily designed as a monitoring system. Its primary purpose is to collect, store, and query metrics and time-series data generated by various applications, services, and infrastructure components in a distributed computing environment. Prometheus excels at monitoring the health, performance, and behavior of these components. Therefore, Prometheus is not a traditional time-series database (TSDB) but it relies on a TSDB for efficient storage and retrieval of time-series data.

The choice between Prometheus and Cassandra hinges on the nature of the data and the intended use case. For our scenario involving the generation of static or pre-generated time series data, Cassandra emerges as the more suitable choice. Prometheus excels at real-time monitoring and collecting metrics from actively running systems, making it ideal for scenarios where instant visibility and responsiveness are paramount. However, Cassandra's strengths lie in its ability to store historical or static data efficiently, offering longer retention periods, flexible data modeling, and scalability advantages. Ultimately, the selection of the appropriate database should be driven by a thorough assessment of the specific data characteristics and the requirements of the use case at hand.

In our specific case, involving pre-generated time series data, Cassandra consistently exhibited lower query latency, signifying its prowess in processing individual queries swiftly. Cassandra also demonstrated decent throughput, handling a larger number of queries per second. However, it's essential to acknowledge the trade-off associated with Cassandra's higher CPU usage.

Prometheus showcased efficient resource utilization with lower CPU usage. Nonetheless, Prometheus recorded slightly higher query latency and lower throughput compared to Cassandra.

In conclusion, our evaluation underscores the importance of aligning the choice of database with specific use case requirements. Cassandra emerges as a robust solution for applications prioritizing query performance on pre-generated time-series data and high throughput.

References

[1] https://prometheus.io/docs/introduction/overview/

[2] https://cassandra.apache.org/_/cassandra-basics.html

[3] Cassandra Partition Key, Composite Key, and Clustering Key | Baeldung

[4] Cluster, Datacenters, Racks and Nodes in Cassandra | Baeldung

[5] The Cassandra Query Language (CQL) | Apache Cassandra Documentation

[6] https://www.java.com/download/help/linux_install.html

[7] https://www.python.org/downloads/release/python-380/

[8] https://prometheus.io/download/

[9] https://prometheus.io/docs/practices/pushing/

[10] https://cassandra.apache.org/_/download.html

[11] https://docs.datastax.com/en/cassandra-oss/2.1/cassandra/operations/ops_monitoring_c.html#opsMonitoringJconsole

[12] https://docs.datastax.com/en/dsbulk/docs/reference/dsbulk-cmd.html

[13] https://github.com/timescale/tsbs

[14] https://questdb.io/time-series-benchmark-suite/

[15] https://www.datastax.com/blog/introducing-datastax-bulk-loader

[16] https://docs.datastax.com/en/cassandra-oss/2.1/cassandra/operations/ops_monitoring_c.html#opsMonitoringJconsole

[17] https://www.datadoghq.com/blog/how-to-monitor-cassandra-performance-metrics/#throughput

[18] https://www.datadoghq.com/blog/how-to-collect-cassandra-metrics/#collecting-metrics-with-jconsole

[19] https://community.appdynamics.com/t5/Knowledge-Base/How-do-I-use-jConsole-to-test-and-troubleshoot-connectivity/ta-p/37217#h_42749547081576884218583

[20] E. Tsertou, “EleniTser/Information_Systems_Project: Information Systems Project NTUA 2022-2023,” Github. [online]. Available: https://github.com/EleniTser/Information_Systems_Project