DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Project Final Paper 2022

Title: Cassandra and MySQL: A Query Performance Comparison

Author: Bradley Culligan

Project Abbreviation: DBCOMP

Supervisor(s): Sonia Berman

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	10
Theoretical Analysis	*0*	*25*	0
Experiment Design and Execution	*0*	*20*	5
System Development and Implementation	*0*	*20*	20
Results, Findings and Conclusions	*10*	*20*	15
Aim Formulation and Background Work	*10*	*15*	10
Quality of Paper Writing and Presentation	*10*		10
Quality of Deliverables	*10*		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	*0*	*10*	
Total marks		**80**	

Cassandra and MySQL: A Query Performance Comparison

DBCOMP

Bradley Culligan
cllbra005@myuct.ac.za
University of Cape Town
Cape Town, South Africa

ABSTRACT

Previous database benchmarks have performed comparisons using basic CRUD operations, applying the same data model in each database. However, NoSQL databases were designed to only be used with specific, appropriately matched data models and queries. Furthermore, NoSQL stems from big data needs, causing a lack of research on performance in modest resource environments. This paper compares the query performance of Cassandra and MySQL in a modest resource context when used with data from the Internet of Things (IoT).

KEYWORDS

Database, SQL, NoSQL, Cassandra, MySQL, Benchmark, IoT

1 INTRODUCTION

NoSQL technology is increasingly being utilised as modern data requires the specialised performance benefits that each NoSQL database offers. As a result, database technology is rapidly evolving while independent benchmark research fails to maintain the same pace [15]. Thus, developers may choose their database systems focusing on technology familiarity and other push/pull factors, rather than currently achievable query performance.

The relational data model was designed in the 1970s [6] to guarantee ACID properties (Atomicity, Consistency, Isolation, Durability). Its design rules eliminate data redundancy which can lead to inconsistencies and, thus, provide the ACID properties. In contrast, NoSQL databases were developed around the BASE properties (Basically Available, Soft state, Eventual consistency) to prioritise speed over the benefits of ACID. It also exploits the abundance of data storage available by storing denormalized data models that target anticipated queries to more quickly retrieve such data.

Overall, our research aims to measure the extent to which latency and throughput performance of Cassandra is better than MySQL when applied to queries of IoT data in a modest resource setting. We target Cassandra as the top ranked columnar NoSQL database[1] and MySQL as a popular SQL database.

The IoT dataset allows for modelling choices to be made that target each database's purported strength and weaknesses. Loading a custom dataset and custom queries required further system design because most third-party benchmark tools focus on generating their own byte data. Similarly, most available benchmark tools were not appropriate for this style of research and the final tool used required a rewrite of some of the source code to be viable for this research. Modest resources in the context of this research refers to the machine used in obtaining the results. This machine had 8GB of RAM, an SSD, and an i5 quad core processor with hyperthreading which implied an effective 8 cores available for processing. A single Cassandra node was used.

[1] db-engines.com/en/ranking

2 BACKGROUND

NoSQL (Not Only SQL) encompasses a large class of databases to target big data concerns [14]. Big data was originally characterised by three traits: Volume (The quantity of data); Velocity (The rate at which data is created); Variety (The structured, semi-structured and unstructured notion of data) [17]. The definition has since been extended but these traits are sufficient to highlight the problems that NoSQL attempted to solve. These problems were highlighted by the Web 2.0 era which brought increased computing power and more data. Thus, in 2009, NoSQL began its popularity growth as the solution to increasing concerns about the relational model [8].

We restrict this research to the columnar NoSQL database. This is most similar to the relational model but data is stored by column rather than by row. Apache Cassandra is a NoSQL columnar database [19] which physically stores the columns of data together in memory [16]. When data is retrieved, a *block* of data is retrieved. Thus, when the data items are physically stored together, we perform fewer block retrievals and, therefore, fewer I/O requests because the amount of data required is stored on fewer blocks. Further, data is stored in partitions which Cassandra can quickly access by applying a hash function to the partition columns of the primary key. This ensures that the whole database does not need to be accessed when seeking data. Storing similar data together like this is advantageous to compression algorithms to reduce the database file size [4]. This is further enhanced by Cassandra's treatment of null values in which there is an absence of data instead of using storage for nulls [4].

The storage techniques mentioned assist to reduce the time when accessing column-based data but can negatively impact performance on other operations such as short scans [7], so it is important to design the data model around the queries in Cassandra [21]. This is further enforced by the Cassandra Query Language (CQL) which is a limited language that restricts writeable queries for a data model [1]. Whilst the tables (column families) are defined around this data model, the schema itself is non-rigid [3]. The tables can be multi-dimensional allowing column families to have a dynamic number of columns which is well suited towards data requiring a flexible schema.

NoSQL was developed with the focus of solving big data problems with highly performant, distributed systems, but little attention has been focused on its performance when used with modest computing resources. This requires further research because the focused development on solving big data problems may cause too much overhead when applying NoSQL on more modestly resourced machines. This could reduce the performance benefits achievable with NoSQL and imply that a relational model is better suited. SQL has also been implemented for longer [5] which implies that more developers are accustomed to SQL and systems have previously been built with SQL databases in mind. Thus, a noticeable query performance increase is required to adjust systems from SQL to NoSQL.

The rate of new NoSQL implementations and database upgrades surpasses the pace of independent benchmark research on these databases [15]. Database upgrades or new implementations are usually released with internal benchmarking results but this is often for marketing purposes highlighting specific strengths. Independent third-party benchmark research assists in achieving results without marketing bias but is slower to publish results.

This research targeted Apache Cassandra version 4.0.5 for the Column store NoSQL database, and MySQL version 8.0.30 for the MySQL relational database. These versions have not been sufficiently benchmarked by independent research because of their age. Cassandra version 4 was released in the latter half of 2021 and version 4.0.5 was only released in July 2022 [2]. MySQL version 8 was released in 2018 and version 8.0.30 was released in July 2022 [3].

Prior database benchmarks performed the comparisons using basic CRUD operations [22]. This aids in comparing how the latency and throughput were affected with varying data and request sizes but limits the process by benchmarking all databases with the same access operations [22]. Such benchmarks fail to take cognizance of the varying performance levels that are achievable when exploiting the design strengths of specific databases. Other, database benchmarks would focus on a specific database and make use of the varying modelling and data access methods available [18]. However, such research restricts the results to a designed system and is more appropriate when seeking performance limits of the system.

A gap in existing research is the lack of benchmarks comparing the performance of systems across database technologies (such as SQL and columnar NoSQL) whilst taking advantage of the techniques that each database provides to optimise performance. Such a benchmarking scenario requires explicit awareness of the dataset used, because it directly affects the modelling choices made and, thus, the performance that the database is able to achieve.

This research benchmarks Cassandra and MySQL when applied to an Internet of Things (IoT) dataset in order to exploit performance through design choices. This dataset was chosen based on its applicability to Cassandra, further discussed in the *Data Selection* subsection. Simultaneously, this research will use modern implementations of MySQL and Cassandra which will assist in providing up-to-date, independent benchmark research on Cassandra version 4 and MySQL version 8 in the context of modest resources.

3 DESIGN

3.1 Data Selection

Seeking an appropriate use case to benchmark Cassandra, we surveyed where Cassandra is implemented in industry. DataStax [4], a real-time data company leveraging the power of Cassandra in their cloud service, have indicated that Cassandra is a good fit for use in the Internet of Things (IoT) and edge computing. This claim is made because [11]:

- Cassandra can ingest concurrent data from any node in the cluster, since all have read/write capacity.
- Ability to handle a large volume of high-velocity, time-series data.
- High availability.
- Supports continuous, real-time analysis.

Beyond DataStax's claims, notice that IoT data can be modelled to take advantage of Cassandra's columnar design to optimise read performance of sensor data. Cassandra has presented its applicability to IoT and shown that it can present a unique performance benefit when modelling IoT data for read queries. Thus, IoT data is an ideal use case to benchmark Cassandra. MySQL is implemented as a baseline when comparing query performance of the relational SQL design and columnar NoSQL design.

IoT data is typically a stream of time series data. A streaming database typically receives data at extremely high velocity and volume. However, in focusing our research on the applicability of Cassandra and MySQL using modest resources, we instead focus on a static database scenario in which past data that is already stored is being analysed.

The IoT data used was sourced from online repositories. Originally, data was sought from sites such as Kaggle [5], Open Data Registry on AWS [6], and GitHub repositories [7] offering links to open datasets. However, most of the datasets were too small to sufficiently benchmark the databases, or the larger datasets were protected by various data protection acts and company systems. Furthermore, some datasets would focus on too large a scale (such as smart cities) [8] to be viable for modest resource analysis.

A UK company, OpenSensors [9], and research from 2004 at the Intel Berkeley Research Lab [10], provided an ideal IoT use case. The Intel Lab data provided measurements for sensors setup in the lab for little over a month, but its number of rows and columns were too limiting to be benchmarked. Separately, OpenSensors' data was company protected but provided insight into the use case of IoT data in a modern context. OpenSensors was using IoT sensors in a workspace environment to provide an optimised and sustainable workspace with dashboarding and analytics. It provided a springboard for researching occupancy detection mechanisms, HVAC (heating, ventilation, and air conditioning) technology and performance optimisation of workers based on office conditions. Small-to-medium workspaces also provide the ideal amount of data to implement this research with modest resources.

An open dataset was subsequently found in a Nature publication titled, "CU-BEMS, Smart Building Electricity Consumption and Indoor Environmental Sensor Datasets" [20]. This would provide the dataset used in this research. The paper provides details about the recorded data — how it was recorded, the incentive behind

[2]archive.apache.org/dist/cassandra
[3]dev.mysql.com/doc/relnotes/mysql/8.0/en/news-8-0-30.html
[4]datastax.com

[5]kaggle.com
[6]registry.opendata.aws
[7]github.com/awesomedata/awesome-public-datasets
[8]arrayofthings.github.io
[9]opensensors.com
[10]db.csail.mit.edu/labdata/labdata.html

the process and some exploratory data analysis. The data supplies electricity consumption and indoor environmental measurements of the seven-story, 11 700 m^2 office building located in Bangkok, Thailand. The data was adjusted to better suit this research.

The original data provided seven floors of data for the period of 18 months from July 1, 2018, to December 31, 2019. However, the volume of data requires a significant amount of loading time, so the time range was reduced to 2019. This data volume required loading times of 30 minutes for Cassandra and one hour for MySQL. Further, floor six had incomplete data so it was ignored under the assumption that it was an unrecorded floor. This has no effect on the research because the data size and variety is still sufficient for the benchmarking purposes. At this size, the smallest floor (floor one) has a CSV file size of roughly 40MB, the largest floor (floor two) has a CSV file size of roughly 95MB, and the remaining CSV files each have a size of approximately 75MB. All files have a measurement for a specific sensor, polled at one-minute intervals for a period of one year. This results in 525 600 rows for a sensor and the variation in CSV file size is caused by the differing number of sensors on each floor. The data was recorded in floor-zone subareas as depicted in figure 1.

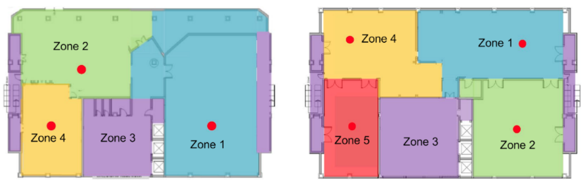


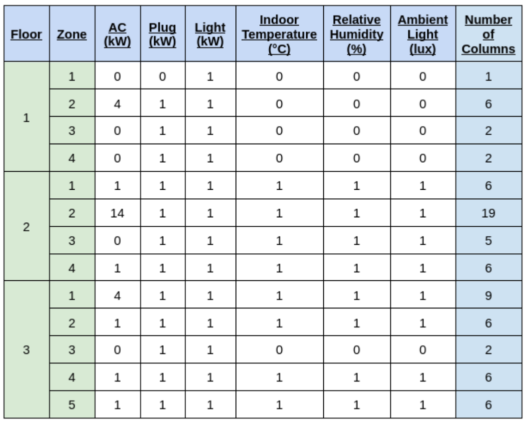
Figure 1: Floor-zone combinations set up for sensor recording (red dots indicate multi-sensor positions) [20]

The electricity consumption data (kW) includes individual air conditioning units, lighting, and plug loads in each zone. Various zones had multi-sensors for the indoor environmental sensor data which consisted of temperature (°C), relative humidity (%) and ambient light (lux). A subset of the sensor recordings in each floor-zone can be seen in figure 2. Of particular importance is the lack of structure in that each floor-zone area (location) does not have the same sensors which our data model should account for efficiently.

Each sensor may require maintenance or have dropped network packets. This dataset had a collective period when many of the sensors were taken down for maintenance. Such events lead to measurements stored as nulls which can affect the size of the database. MySQL still supplies space for the null data which would increase the database size but Cassandra is able to handle this more efficiently. Cassandra is also able to better compress the data size due to the common nature of the data stored in its columns leading to further database size benefits. For a more detailed analysis of the dataset, please see the referenced paper [20].

3.2 Database Design

Modelling the data requires different approaches in Cassandra and MySQL. For MySQL, the modelling method is traditionally targeted at entity-relationship diagrams, choosing appropriate data types to minimise data storage and retrieval, and normalisation of the resulting tables. However, under the assumption that the IoT data would likely be used for analytics of some sort, a model more focused towards OLAP was used [12]. A star schema was used for the MySQL data model with the fact table holding the sensor measurements, and three dimension tables for filtering the fact table. Dimension tables consisted of sensors, locations and measurement times. The model is extendable to store further information in each dimension, should this research be extended. Such an extension might include manufacturer information, running requirements, etc. in the *sensors* table and the *locations* table might include size, department, etc. The modelling extensions may alter the model into a snowflake schema or require denormalized dimension tables — the effects of which were not measured in this research.



Floor	Zone	AC (kW)	Plug (kW)	Light (kW)	Indoor Temperature (°C)	Relative Humidity (%)	Ambient Light (lux)	Number of Columns
1	1	0	0	1	0	0	0	1
	2	4	1	1	0	0	0	6
	3	0	1	1	0	0	0	2
	4	0	1	1	0	0	0	2
2	1	1	1	1	1	1	1	6
	2	14	1	1	1	1	1	19
	3	0	1	1	1	1	1	5
	4	1	1	1	1	1	1	6
3	1	4	1	1	1	1	1	9
	2	1	1	1	1	1	1	6
	3	0	1	1	0	0	0	2
	4	1	1	1	1	1	1	6
	5	1	1	1	1	1	1	6

Figure 2: Subset of sensor recordings available to be queried

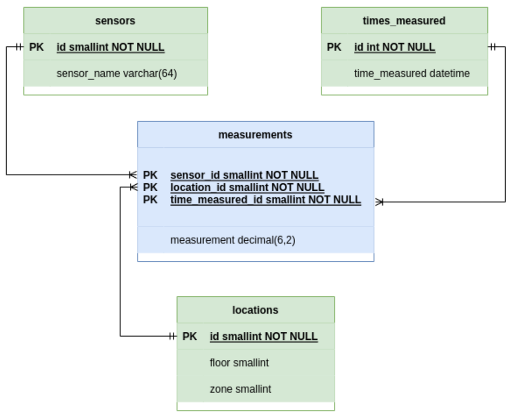


Figure 3: MySQL star schema used in modelling IoT data

For Cassandra, the modelling method should be query focused [23]. This is due to Cassandra targeting high write speeds at the expense of lower read performance, and the limitations on its query language (CQL). Cassandra requires that only one table can be used in a query (i.e. there are no joins allowed) which enforces data should be stored together in a table if it is intended to be retrieved together, leading to a denormalized database. Further, any *where* or *group by* clauses can only be applied with primary keys (or index conditions for the *where* clause), and the *order by* clause can only be used with the clustering columns. This restricts their use and, thus, the data model should be designed with this in mind.

The four-step process to design the Cassandra data model in figure 4 is depicted:

(1) Understand the data. A conceptual data model, with the aid of an entity-relationship (ER) diagram, can provide a high-level overview of what is being recorded. The entities in this research include the sensors, the locations, the measurements and the times when measurements are taken.
(2) Identify access patterns. The IoT data could be used for a machine learning scenario, dashboarding and data analytics, simulations, load predictions, or for management analysis. A large combination of access patterns could convolute the research. Instead, our access patterns are based on how a query might affect the achieved performance. This is elaborated on in the section below on *Query Choice*.
(3) We use the access patterns to design a logical data model with a query-first approach to reflect the model in Cassandra.
(4) The physical data model provides consideration for efficiency, validation and optimisation of the schema by considering the physical limitations of the compute resources and Cassandra. Our specific scenario will be elaborated on in the section below on *Query Choice*.

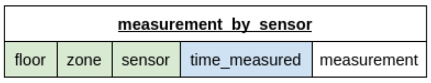


Figure 4: Cassandra IoT data model for retrieving sensor measurements

In figure 4, the green columns constitute the partition key columns of the primary key where the combination of floor, zone and sensor name allow us to target a specific sensor, and the time_measured (blue column) is the clustering column of the primary key. The remaining column stores the actual measurements obtained. The clustering column choice means time series data is stored in order, allowing for efficient retrieval of data in order. The partition key columns are explained below.

The original data model had an extra 'month' column for bucketing. This technique limits the size of partitions to maintain optimal performance. The choice of 'month' for bucketing was based on Cassandra's recommended partition size of 100MB, or approximately 100 000 rows which should equate to 100MB. A single sensor produces 44 640 rows in a month (using 31 days as an upper bound). This would imply two months of data in a partition, but a single month is logical in its access pattern. The Cassandra utility *Nodetool* was used to get metrics on the physical tables. Nodetool presented much lower partition sizes than expected when using bucketing. Further, bucketing by month required multiple partition accesses for operations extending beyond a single month. Thus, the partition keys were adjusted to the model currently used which Nodetool calculates to be roughly 17.5MB per partition.

A single partition access is required when accessing a single sensor but incurs the problem of unbounded partitions. To solve this, we can use Cassandra's time-to-live (TTL) mechanism which allows for data to automatically be removed after a certain amount of time. Based on the current data sizes obtained from Nodetool, we can predict that the database will remain efficient with up to five years' worth of data using our model. Thus, we bound our partition data size by setting a TTL for all stored data which can be stored in an archive, if necessary, or simply removed on expiry.

3.3 Query Choice

Queries were chosen to target strengths and weaknesses in Cassandra and MySQL. Each query provides insight to answer the overarching research aim by first answering it with reagards to the targeted queries. The queries can be found in appendix A and B and explained here:

1. **GetSensorData** Get all time series data for a specific sensor within a given time range. A specific sensor might be the *Light* electrical consumption on floor two at zone three. This would be the basic functionality when retrieving time series data. This query targets the performance strength of Cassandra because it retrieves data from a single partition. Such data retrieval means that Cassandra can target its data access to the partition (which is fast due to the mapping processes utilised by Cassandra) and use information from the clustering column to quickly retrieve the necessary data with minimal data interaction. Comparatively, the MySQL implementation will be required to join all tables together which is an expensive operation and should thus highlight the performance differences.
2. **GetPeakData** As an extension of simply retrieving the raw sensor data, a comparison of differences in performance when an aggregation function is applied to the retrieved data will help to compare performance of function application on data retrieval versus applying such aggregation in application logic. Thus, this query performs similarly to query one but applies an aggregation function, namely *max*, to the sensor data retrieved. The query is of interest because it would increase the amount of data accessed in Cassandra which might reduce the performance benefit achievable in Cassandra. Further, Cassandra version 4, the version used in this research, is the first version to supply aggregation queries which provides further incentive to include this query.
3. **GetAvailableData** This query requires no joins in the MySQL implementation because all queried data comes from one of the dimension tables — specifically, getting all possible floor-zone combinations. This requires no expensive join operations for MySQL and requires that the Cassandra implementation uses the *distinct* keyword and *filtering*. Filtering requires Cassandra to access every partition which can be expensive. Thus, this operation should be performant for MySQL and not for Cassandra. While it is valid to argue that the Cassandra implementation should have a new table for storing such a query, this was not done in the interest of comparing performance where filtering is used. Such a query may be applied to a dashboard or analysis where users may seek the available information to work with.

4. **GetFloorZoneData** Similar to query three, this query also retrieves all floor-zone combinations but with the restriction of limiting the combinations to a specific sensor name. For example, seeking all floor-zone combinations where an AC1 (air conditioning unit 1) can be found. This query affects the MySQL implementation by requiring a 3-way join between the sensor and location dimension tables and the measurements fact table. This reduces the inefficiency of the join operation because the result is limited by the *where* clause which invokes less work for the join operation. The Cassandra implementation still requires filtering, thus, there is no expected performance increase. It also uses the *distinct* keyword in Cassandra which requires all partition columns to be included in the query. However, the query has already targeted a specific sensor in the *where* clause, so the returned result of distinct locations will also have the repeated sensor name which could be ignored in any application logic. Again, modelling a specific table for this use case is possible but purposefully excluded.

3.4 Experiment Design

The experiment was designed to compare the latency and throughput performance of Cassandra and MySQL when utilised with IoT data. The queries used to perform this comparison were described above and highlight the importance of each query. This section aims to provide insight into the design of the experiment to ensure credible results.

Experimental research should provide a framework which allows for the results to be reproduced. To ensure reproducibility of the results for this research, Docker was used to bundle all tools used and provide an external environment to perform the benchmark. This ensures that the same tools, with the same versions and configurations, are used whenever the benchmark results are to be tested.

Docker containers allowed us to achieve good benchmark hygiene. This was achieved with separate containers for each database, containers for preparing the data and databases, as well as a container for the benchmarking tool used. Thus, each benchmark is performed from a cold start with a new database instance loaded with fresh data. This avoids any caching which could have occurred between benchmark runs as well as any other performance enhancements performed in the background should the same system be continually used.

Docker also allows us to provide resource control limits, such as limiting the accessible CPU count or memory available. This prevents more performant computer systems from using more resource power and, thus, achieve inconsistent results with this research. Further, the containers are easy to launch on server systems to perform the research with scaled resources.

One could argue that containerisation limits the research results to systems only implemented with containers [13]. However, benchmark results were obtained under the same conditions (each database runs in a container) so the results of this research remain valid in their comparison. Cassandra's official guidelines introduce Cassandra with their Docker image. Thus, it is not infeasible that many implemented systems would be using containers — providing greater alignment to the results obtained.

This research focused on comparing the performance of custom queries and data between Cassandra and MySQL. The benchmark systems shipped with each database would be an inappropriate benchmarking tool because of the differences in how each benchmark system is implemented. This would not provide results which could be fairly compared. In-house development of a database benchmarking tool for this research would also not be ideal. This is because the various implementation guidelines and tools already designed can take different approaches to attaining benchmark results. Such differences can affect the interpretation of results and we should therefore strive to use a benchmarking tool which has already proven its confidence in the community. There were three major options.

Firstly, we could use an implementation of one of the TPC benchmarks [11]. However, these typically target a specific use case, such as e-commerce, and adjust various properties of the workload against the database to mimic the use case in action. Properties such as the access patterns, open connections, network latency and availability are some characteristics which could be implicated. This is not ideal for our research where the dataset is used to guide modelling choices in the database design and not to provide replication of an implemented system.

NoSQLBench [12] is another popular option and fits the flexibility of custom queries and data modelling choices required for our research. However, this tool is limited to NoSQL and, thus, would create difficulty in a cross-comparison when looking at the SQL benchmark results.

Finally, Yahoo! Cloud Serving Benchmark (YCSB) [7] is a popular, open-source NoSQL benchmarking tool. Notably, it is extensible in allowing new databases to be benchmarked with the suite of templated workloads and allows for custom workloads to be implemented. Thus, we can extend the system to perform our custom queries on our custom datasets and still take advantage of the backend implementation of how the benchmark metrics are tracked. The database also makes provision for use with the JDBC driver which allows us to use YCSB with MySQL via the JDBC implementation. Thus, we can achieve a comparison between NoSQL and SQL databases using a common benchmark system.

In general, YCSB would launch with a load phase to generate the data according to the workload parameter file and store it in the desired database, then a run phase to perform benchmarking. Our YCSB implementation had a custom dataset so the loading phase was avoided. Instead, data loading would have to be done separately as described in the *Implementation* section. The run phase would execute the workload using various threads making connections to the database and collecting and storing the performance data achieved. Thus, YCSB acts independently of the chosen database and stores performance data of simulated connections and queries as seen in figure 5 [13].

The metrics gathered by YCSB have various output forms. This research used raw output, storing timestamped and labelled latencies (microseconds) for each query performed. Python scripts then

[11] tpc.org/information/benchmarks5.asp
[12] docs.nosqlbench.io
[13] benchant.com/blog/ycsb

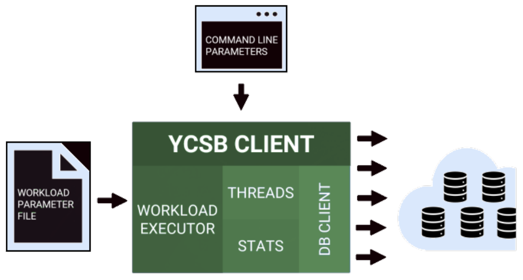


Figure 5: Interaction between YCSB, workload parameters and the database

read the raw output and generate graphical representations of the latency and throughputs obtained. These graphics make it simple to compare the performance of each database.

Overall, the entire benchmark process aligns with figure 6. The process is intended to be fully automated so the user simply needs to launch the 'make' target, passing in a chosen environment file to set the variables for a specific run. This will launch a network of containers and begin tracking Docker metrics in a background process. The raw dataset is read and processed, then loaded into the database of choice. YCSB's run mode is launched with one thousand queries. The four queries are randomly interspersed in equal proportions. The tracked metrics are available in CSV files which the user can use to generate graphs for further analysis by launching the appropriate 'make' targets. Independently, Python scripts can be executed against the database to compare output from the Cassandra and MySQL queries to validate the returned results.

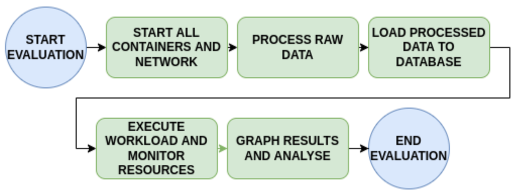

START EVALUATION
START ALL CONTAINERS AND NETWORK
PROCESS RAW DATA
LOAD PROCESSED DATA TO DATABASE
EXECUTE WORKLOAD AND MONITOR RESOURCES
GRAPH RESULTS AND ANALYSE
END EVALUATION

Figure 6: Benchmark process executed for a full, valid run

4 IMPLEMENTATION

4.1 Docker Usage

Multiple containers with individual setup procedures were required to communicate and provide operations for each other. A Docker Compose YAML file was the ideal approach for this. The Compose file would set up the internal Docker network used for communication between containers, as well as build and launch the containers with the appropriate configuration parameters provided via an environment file.

The database containers were launched from the official Docker images released by each database vendor. The name of the services running the containers would act as domain names for communication within the network and internal ports were mapped to external ports, using the default ports for each application. Resource limits were placed on the containers to ensure consistent results when benchmarking on more powerful machines. We limited the CPU count to 8 processes and 8GB of RAM.

The images were version controlled, ensuring the benchmarked database versions were 4.0.5 for Cassandra and 8.0.30 for MySQL. This ensures the benchmarks are not run with different database versions which could create inconsistent results and instabilities in the system. Docker also allows us to version control the remaining tools used to ensure stability. The other containers ran the YCSB service and the setup container which replaced YCSB's load phase. Unit tests and graphing the results also used Docker to avoid installing any tools for the sole purpose of running these functions.

The setup container is built from the Ubuntu base image and primarily makes use of Java and Python. Python is used to read and transform the raw CSV dataset. The same Python script was used to create the database schemas via the database drivers. Python is ideal for this type of data manipulation and script work due to its lightweight nature. Cassandra's bulk data loading utility, called DSBulk, requires Java's runtime environment in the setup container. Similarly, MySQL provides a MySQL shell tool which also provides bulk data loading. The remaining tools installed are various dependencies for the highlighted tools, and the Python packages required for data manipulation and database communication. The setup image also copied the setup scripts for internal use and has the CSV data files mounted via a volume for data communication between the container and host system. A volume mount allows Docker to control the data storage, making the system more platform independent, and provides the host researcher with access to the processed files.

The setup script cannot start the setup process until the database container is able to facilitate communications. A bash script, recommended by Docker (wait-for-it.sh), is used to ping the database until it is able to successfully make a connection. The setup Python scripts are then launched for the remaining processes to proceed. The YCSB container waits for the setup container to complete its processes before launching its own container run from a custom Dockerfile image. This image is also created from the Ubuntu (22.04) base image. YCSB is developed in Java and, therefore, requires the JDK, and Maven for its dependency management. The script used to initiate YCSB is written in Python version 2 which is also installed in this image. Finally, the YCSB files are copied over to the image and a volume with the outputs is mounted to the container.

Various variables need to be set for each container in the Compose file to allow for specific database-dataset combinations, such as MySQL-IoT or Cassandra-IoT, to be executed. This is managed with the environment variables files. These provide the necessary YCSB run arguments to launch an appropriate workload file to be benchmarked and the database connection properties. It also provides JVM runtime arguments which were necessary to adapt for the MySQL implementation which would run out of Java heap memory. However, this could be used for any benchmark scenario where the researcher intends to adjust the JVM runtime arguments. The output raw data files are also designed to be append-only. However, the design of this research would then have multiple, different benchmark runs appended to the same file. Instead we clean the results obtained from a previous run so that each benchmark call has its own output file for results.

Finally, a Makefile provides the necessary targets to build and run the benchmark suite as well as perform unit tests and generate the graphical representation of the results. It ensures that prior containers included in the benchmark are stopped and removed before and after benchmarking. This ensures that a clean instance of all targets is always launched for every benchmark run. It also launches a background process to track the Docker metrics which can be analysed in tandem with the benchmark results for a greater depth of understanding in the results. Because each line in the Makefile is executed in its own subshell, the work is performed with a bash script. This allows the Docker stats call to occur in the background and its process ID stored for termination at the end of a benchmark run.

4.2 YCSB Customisation

YCSB was developed in Java, thus all extensions to the tool for our custom requirements were also implemented in Java. YCSB made use of Maven as its build tool which was modified to make use of specific drivers which this research required. The Cassandra driver connection for Java in use by YCSB was the version 3 driver. When trying to upgrade the driver to better match the Cassandra version being benchmarked, multiple errors would occur because the new driver changed the package and file structure of the tools, as well as the methods to access the database. A full rewrite of the Cassandra code in YCSB would have been necessary to update the driver. Instead, the old driver remained to avoid excessive code rewrite. The benchmark would still be executed against the database running Cassandra version 4 which was deemed acceptable. The MySQL driver was able to be upgraded to version 8.0.30 without errors and a database properties file was set with the properties used in the database container. Code for databases not implemented in this research was removed from the YCSB source files to reduce the project size.

YCSB's extensibility is achieved by the various classes that handle the benchmark operations at different levels of generality. At the top level, the core YCSB package supplies a *DB* abstract class outlining the basic methods and CRUD operations where we define methods for our custom queries. These methods perform a query which is benchmarked and return the result of the operation as a *Status* object. To avoid unnecessary implementation in other databases, these methods were made to return a *NOT_IMPLEMENTED* status. The *DB* class essentially holds the method headers.

DBWrapper wraps around a "real" database that measures latencies and counts return codes. This supplies scope strings for the TraceScope object for tracing. The headers defined in *DB* are also implemented here. These begin time measurements for the database queries and call the database to perform the query. The recorded timing is then further processed by measurement classes that YCSB provides, and the status is reported.

Actual database implementations of the query functions are written in the *Client* classes of each database package. The queries are generated and passed to a prepared statement object which has its query executed. Tracing, logging and exception handling is also managed in these methods. The query methods accept a set of arguments which are dynamically updated to set dynamic queries at each method call. Choosing the arguments to be passed at each method call is handled in the workload class files.

4.3 Data Loading

The majority of the benchmark time is spent loading data into the database so it is further discussed here. Python scripts were used to set up any schemas, read the original CSV data, and used Pandas to manipulate the data into an appropriately modelled form — replicating the form of the database models. The aim was to use the data, now in an appropriate form and in memory, and pass it to the respective databases via their drivers. This was expected to be an efficient approach because the data was already in memory but it would prove not to be an ideal data loading technique.

Aiming for performance with this technique, list comprehension and mapped lambda functions were used to create a batch of values which could then be passed to a prepared batch insert statement. The prepared statement allows the statement to be passed to the database once to be validated and only the necessary list of values needs to be sent later. The batching was done in the size of one month's worth of data for a specific sensor to avoid overloading the coordinator node in Cassandra. It was also performed with the old month bucket partition model so batching the data by month meant that a single batch insert would only implicate a single partition, thus, helping to keep the query performant. Batching also helped reduce the number of individual statements being processed. This technique achieved an insert rate of roughly two minutes and forty seconds for the data from floor one — the smallest data size. However, it was primarily single threaded and entirely run from a single Python process. It also made no provision for failed insert operations.

This loading method was later optimised by using chunking from *more_itertools* in Python which allowed us to avoid the slow process of list comprehension when extracting the data from the Pandas dataframe. Part of the reason why the list comprehension was slow, was because this method of accessing data in the dataframe is an anti-pattern for the way Pandas has optimised data retrieval in their dataframes. The chunking allowed for the data retrieval to reduce the number of operations required to retrieve the data and the iterative chunking data structure allowed for further optimisations by applying lazy loading techniques — particularly assisting optimisations in the memory usage of the program.

The month bucket model had been removed at this stage so the batch sizes were increased to 64 000 which is slightly below the maximum batch size for Cassandra. However, large batch sizes can put significant strain on the coordinator node and, instead, small batch sizes with concurrent statements being passed to the database should be used. Further, the batch size of 64 000 meant that multiple partitions would likely be accessed when performing the inserts. This would slow the insert process and cause a backlog for the coordinator node, thus, applying further strain. This data loading method could then lead to increased chances of failed operations and, without appropriate error checking on the inserts, it could lead to erroneous data loads for the database. Note that the batch size of 64 000 was most appropriate in this instance but holds no guarantee when used with a different database or different version of Cassandra. This data loading method is, therefore, not maintainable, nor portable enough to apply to alternative scenarios.

Nonetheless, this technique was suitable at the time for a prototype, despite the lack of multiprocessing which could allow further performance increases. Python is limited by its Global Interpreter Lock (GIL) which prevents race conditions and ensures thread safety [2]. To overcome this performance barrier, the manipulation and loading of data into a database was separated by floor, as the original CSV files are read, using multiprocessing. Each floor of data could then make a separate connection to the database to insert its data. Python made use of a pool of CPUs whereby the main program launched individual Python processes running the operations to process and insert a single floor of data. This does not increase the performance of a single floor's loading operations, but it does increase the throughput of the data loading scripts overall because each floor's data can be handled by a separate CPU core.

This data loading technique did not work when applying it to MySQL. Various alternative techniques were attempted to load the data directly from the dataframe in memory to avoid writing the data to disk and then reading the data from disk into the database of choice. However, saving the processed data as a CSV file to disk, proved to be the appropriate option. External CSV loading tools can provide a higher performance gain to validate writing the data to disk and provide error handling. The CSV files are also not required to be recreated for every benchmark run. Instead, a directory was supplied for storing processed data in CSV format and new runs can make use of the data that has already been processed. The setup scripts check if these files are available and will spend the time processing the raw data and saving new, processed CSV files for loading if they cannot be found. *Make* targets are supplied to quickly remove any processed CSV files in order to trigger their re-creation.

This approach with MySQL led to Out of Memory Exceptions (OOME) which would kill containers. To solve this, the number of Python processes generated was limited. While this negatively affects the overall performance, it increases system stability. Foreign key checks in the MySQL schema were turned off for data loading to avoid hindering load performance. After loading, the foreign key restrictions were re-enabled.

Various security issues were also brought to attention by the MySQL implementation. This focused on various security parameter combinations when loading data from an external source such as the setup scripts. The implemented solution was to mount a volume of the processed CSV files to the database container. This way, the database stored the CSV files locally and could make use of the *load_local_infile* parameter to allow MySQL to load the locally stored data files.

Finally, external data loading tools were used to handle the data loading operations from the processed CSV files. The internal MySQL loading commands had no multithreading options which hindered performance. MySQL made use of the *MySQL Shell* which performs the same internal data loading command, but performs extra processes to analyse the data and optimally load it. Cassandra made use of *DSBulk* — the industry standard for loading bulk CSV files to Cassandra. Both tools are multithreaded and use multiple processes to achieve a high write rate. These tools also manage all failed queries and report loading statistics for monitoring purposes. The external loading tools assist with portability and maintainability concerns.

4.4 Query Execution

Workload class files are used to interpret the workload key-value files which define all properties set in a YCSB session. Separate workload files are defined for the Cassandra and MySQL implementations and include definitions for the measurement type, output files, number of executions for statistical confidence and proportion of operations per session. It is a way to succinctly pass a file to the YCSB run operation holding all parameters which would normally be defined via the terminal.

These key-values are then read by the Workload Java classes and the properties for a YCSB session are set. To implement the dynamic query generation, both the Cassandra and MySQL workload files will use the same data and functions. Thus, these were set as child classes of a core IoT workload class. The core IoT workload class inherited the core YCSB workload class and then held a subset of the total data stored. The data required to be stored was the available time ranges that could be queried and valid combinations of floor, zone and sensor names.

To obtain a random date, the minimum and maximum dates stored in the database (the year of 2019) were stored in the core IoT class. These were stored in epoch format to ensure a uniform distribution when selecting a random day in the time frame. At each call to get a random date range, Java's thread local random object was called to choose two dates. These dates were then ordered and a capped number of days was applied to the length of time between the two dates. This ensures queries will not be performed against too long a period of time so as to cause the Java heap memory to become full, and queries to timeout. Note that the thread local random object had to be accessed because YCSB can have multiple threads calling the workloads and, thus, the thread local random object should be used to ensure performant and thread-safe code. The time ranges are also limited to dates (not including time values as stored in the databases) but this is sufficient because the net result is still a dynamically chosen time range for each query to use.

The floor, zone and sensor name combinations required more work than simply choosing each value at random. This could cause invalid combinations to be chosen because of the inconsistency of floor, zone and sensor name combinations in the data (every floor does not have the same zones, and every location does not have the same sensors). Invalid combinations would then return empty result sets when queried which could skew the benchmark results with invalid measurements. Instead, a hashmap of a valid set of combinations was stored. The hashmap allows for efficient retrieval of the values desired. A full set of data that has been loaded into the database could be utilised but this would be unnecessary and inefficient work. Instead, a random subset of the loaded data can be hard coded into the class. In conjunction with the time ranges, this would sufficiently cover a range of dynamic queries to avoid targeting the same data in the database which may incur unwanted background optimisations.

The Cassandra and MySQL IoT workload classes retrieve the data from the workload files and use default properties when a custom value is not set. The total number of queries are distributed by the set proportion amounts for each query. In this research, we targeted four queries so the proportions were set to 25% for each query. This allows YCSB to randomly distribute the queries, providing a further mix of database queries.

A YCSB session executes the benchmark, as initialised, and outputs the raw data to the output file specified. Every ten seconds, a status update is printed for the user and, at the end of a run, summary statistics are reported to standard out. For further analysis of these results, we make use of Python scripts to graph the outputted data.

A thorough explanation of the graphing scripts can be found in David Court's paper on database performance comparison of MongoDB and MySQL [9]. In general, the scripts are written in Python and use Pandas and Matplotlib to manipulate the raw data results into buckets for graphing. Matplotlib then graphs the processed data for further analysis. As with the other processes in the benchmark system, this can be launched from a Makefile target and is handled within a container to control the necessary dependencies used.

5 RESULTS

The benchmark results are graphed and can be found in full in appendix C. Note that the figures in appendix C provide a page of varying views for a single query's achieved benchmark values. This allows us to see all of the resulting data points, but also focus on subsets of the results for further insight and clarity. The first data points provide insight into initial database requests, but after a few requests, the database has warmed-up and the results more closely resemble expected performance values. Thus, results have also been graphed ignoring these initial data points to provide clarity on the achieved performance. The latencies are supplied against the number of queries but these results have a large variance, so an exponentially smoothed average line is fitted to better interpret the trend in our results. We notice that throughput is inversely proportional to the latency achieved. The throughput values are supplied by percentile which highlights that the achieved throughput values for a query are limited to the n^{th} percentile.

Mean Latency at 95th Percentile (ms)		
Query	**Cassandra**	**MySQL**
Floor-Zone	5.3275	**3.2989**
Sensor	**16.5589**	1313.3975
Peak	**155.2567**	1189.9008
Available	5.7207	**0.6246**

Figure 7: Average comparison for Cassandra and MySQL latencies on IoT data over all queries at the 95th percentile — more performant values are in bold

Cassandra was expected to perform best when applying queries that targeted single partitions. Indeed, Cassandra performed best for queries one (GetSensorData) and two (GetPeakData) which targeted specific partitions in Cassandra to retrieve the data. The Cassandra implementation of query one performed, on average, with a throughput rate more than 79X faster than the MySQL equivalent. The average latency retrieval, at the 95th percentile, in MySQL was approximately 1313.3975 ms and only 16.5589 ms in Cassandra. Such a large difference in performance can assist with training time in machine learning applications. Cassandra's low latency can help prevent backlog in the system resulting in sustained performance and stability. Noticeably, the graphed latency measurements for Cassandra are consistent over time unlike the MySQL graphs which typically have spiked latencies particularly towards the end of runs where the system may have become backlogged and caused subsequent queries to be more heavily impacted. Further, the Cassandra data is stored in order and, therefore, retrieved in descending time order. The MySQL data is not stored in this way and the query does not require the retrieved data to be ordered in the same way as Cassandra. This would likely further slow the query for MySQL and increase the performance gap.

Query two focused on applying the newly added aggregate functions in Cassandra to the data retrieval. Whilst Cassandra has an average 7.6X throughput improvement over MySQL, the change in performance from query one to query two was more indicative of a performance hit in Cassandra than MySQL. Query one and two have a similar design except for applying a function to the retrieved column. Cassandra's performance degraded by little over 9X from query one to two. MySQL's performance difference between query one and two shows little difference. Nonetheless, Cassandra is still more performant but the stark difference in performance achieved between query one and two may indicate that it is more efficient to apply aggregation of any sort to data at the application level — after it has been retrieved from the database in a raw format.

Our graphed Docker metrics show that Cassandra used minimal CPU and memory resources when compared with the MySQL implementation for performing these queries. Thus, in spite of Cassandra's design choice in targeting large, distributed systems, we can see that its data access methods allow it to use computing resources more efficiently than MySQL. Cassandra is then a more applicable database when limiting computing resources to modest amounts.

The results indicate that the modelling choice made for Cassandra assisted its performance benefit. This claim is made on the basis that less CPU and memory was utilised — indicating less data access. This was expected as Cassandra utilises its ability to retrieve only the necessary data columns for efficient data retrieval.

Query three (GetAvailableData) and four (GetFloorZoneData) were predicted to be performant in MySQL and this was also proven true. MySQL's implementation of query three had little over a 9X average throughput increase compared to Cassandra. This focused on retrieving dimension data which requires no joins. The results indicate that this line of thinking indeed helped with performance. However, the Cassandra query could have been more performant with a table designed specifically for this request. Further, if the dimensions were extended with more columns, the performance difference could be reduced due to the increased amount of data accessed.

Query four made use of a 3-way join optimisation and supplied a *where* clause to reduce the data retrieval in MySQL. It also focused on smaller dimension tables. Indeed, these led to a performance increase in MySQL with regards to latency at the 95th percentile — 1313.3975 ms when retrieving sensor data, but 3.2989 ms when retrieving the floor-zone data for a sensor name. However, this was only 1.6X more performant than the Cassandra implementation. The MySQL implementation had a greater performance reduction when comparing queries three and four, whilst the Cassandra implementation had fairly similar results returned for both queries.

Overall, the Cassandra implementation performed more efficiently — both with regards to time and resource usage. The MySQL implementation had a much larger final database size at approximately 7.7GB in size, whilst the Cassandra database only used 600MB of data in total. The time ranges being queried also had to be limited to avoid not having sufficient Java heap memory when querying MySQL with the modest resources used. This is a further indication of poor usability for the MySQL implementation with modest resources.

The figures in appendix C highlight the differences in performance for most of the query results, except for query four. Query four's figures indicate the similarity in achieved performance between both databases. This query required extra work from both databases — a three-way join for MySQL and filtering for Cassandra. The extra work required, reduced the performance benefits that are achievable for each database when performing queries that better align with the implemented data model. This indicates the importance of using the appropriate database and data model for the queries being performed.

The focus has been on general applicability of the results but, when applied, further study should be focused on the results per percentile. Stark latency differences in the 99th percentile might have minimal effect on a dashboarding application where a sudden hiccup in performance can likely be ignored. But these latency spikes in performant systems, like machine learning, can cause a backlog in the system causing further reduction in stability and performance and, thus, usability degradation. This would need to be dealt with by control mechanisms such as load balancing.

6 CONCLUSIONS

The results indicate that Cassandra is the more performant system with regards to latency and throughput when applied with a modest data size and resource allocation for IoT data. Cassandra's established strengths exhibit a significant improvement over MySQL performance. In contrast, MySQL's known strengths are only mildly more performant than Cassandra, and additional dimension attributes could further reduce this gap. We conclude with lessons learned from this research and suggestions for future work.

Finding appropriate data, cleaning and loading it proved to be a much longer process than expected. The loading phase of the benchmark is also the longest part of the benchmark. Optimisations to the research should focus on live generating the data used in the benchmark whilst aiming to mimic the patterns of the required dataset. For example, the IoT data could mimic one-minute read intervals of multiple sensors where sensor readings represent expected data values.

Alternatively, the data already used could be extended with more data in the dimension tables. More columns and rows could provide a clearer performance difference. Similarly, more queries could be used to target more areas of performance difference. Alternative NoSQL databases, such as document or key-value databases could also be compared with a similar style of analysis which would involve new query designs.

Finally, this research had the intention of comparing database performance when utilising modest resources. A natural extension would be to scale the benchmark either vertically (more powerful machines) or horizontally (adding additional nodes). Horizontally scaling the research would better focus its results on typical NoSQL implementations in the real-world where NoSQL is usually implemented on multiple nodes.

IoT has become more prevalent in modern society and continues to grow [10] so this research provided insight towards bettering its use. Occupancy patterns and utilisation rates can provide insight to assist in optimising workspace utilisation. The same data can also be used to monitor workspace health, including metrics such as noise levels, temperatures and average CO_2 emissions. This data could be provided in dashboard applications or utilised in machine learning applications and other applied optimisation scenarios.

Modern engineering principles are typically used to reduce energy usage. Better efficiency levels can be reached by using data-driven results from IoT sensors. The data can be used in machine learning to train models which could optimise HVAC energy usage to indoor conditions (temperature, humidity and CO_2 levels) and ambient lighting conditions to light usage. An optimised usage of these utilities could reduce expenditure on these utilities, maintenance, and their carbon footprint. Further, anomaly detection could be used to detect early warning signs of maintenance requirements or possible attacks on the operational technology (OT).

Cassandra's query performance and resource efficiency makes it the ideal candidate to cover all aspects when applied to IoT. This indicates that the relational model is a poor fit for utilisation in the IoT framework and systems which could follow a similar set of requirements and modelling choices. Particularly, this research has highlighted Cassandra's applicability to modestly resourced systems and the performance gains achievable. It has also provided a methodology for extending the YCSB suite towards benchmarking further databases in the same manner — focusing on performance through data model alignment to the structural strengths of the database.

7 ACKNOWLEDGEMENTS

This work is based on the research supported wholly/in part by the National Research Foundation of South Africa (Grant Number 144718).

NRF National Research Foundation

REFERENCES

[1] Gabriel Bassett. 2014. *Examination of the Cassandra Distributed Storage System.* Technical Report. Information Security Analytics.

[2] David Beazley. 2010. Understanding the python gil. In *PyCON Python Conference. Atlanta, Georgia.*

[3] Carlos J. Fernández Candel, Diego Sevilla Ruiz, and Jesús J. García-Molina. 2021. A Unified Metamodel for NoSQL and Relational Databases. (2021). https://doi.org/10.1016/j.is.2021.101898 arXiv:arXiv:2105.06494

[4] Jeff Carpenter and Eben Hewitt. 2022. *Cassandra: The definitive guide: Distributed data at web scale* (3 ed.). O'Reilly.

[5] Rick Cattell. 2011. Scalable SQL and NoSQL Data Stores. *ACM SIGMOD Record* 39, 4 (2011), 12–27. https://doi.org/10.1145/1978915.1978919

[6] E. F. Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387. https://doi.org/10.1145/362384.362685

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10* (2010). https://doi.org/10.1145/1807128.1807152

[8] Alejandro Corbellini, Cristian Mateos, Alejandro Zunino, Daniela Godoy, and Silvia Schiaffino. 2017. Persisting big-data: The nosql landscape. *Information Systems* 63 (2017), 1–23. https://doi.org/10.1016/j.is.2016.07.009

[9] David Court. 2022. *MongoDB and MySQL: A Query Performance Comparison.* Technical Report. University of Cape Town.

[10] M. Dachyar, Teuku Yuri M. Zagloel, and L. Ranjaliba Saragih. 2019. Knowledge growth and development: internet of things (IoT) research, 2006-2018. *Heliyon* 5, 8 (2019), e02264. https://doi.org/10.1016/j.heliyon.2019.e02264

[11] Rich Edwards. 2021. What are People Using Cassandra for Anyway? | DataStax. https://www.datastax.com/blog/exploring-common-apache-cassandra-use-cases.

[12] M. Golfarelli and S. Rizzi. 2017. From Star Schemas to Big Data: 20+ Years of Data Warehouse Research. *Studies in Big Data* 1, 93 (2017).

[13] Martin Grambow, Jonathan Hasenburg, Tobias Pfandzelter, and David Bermbach. 2018. Dockerization Impacts in Database Performance Benchmarking. (12 2018).

[14] Venkat N. Gudivada, Dhana Rao, and Vijay V. Raghavan. 2014. NoSQL systems for Big Data Management. *2014 IEEE World Congress on Services* (2014). https://doi.org/10.1109/services.2014.42

[15] T. Ivanov, T. Rabl, M. Poess, A. Queralt, J. Poelman, N. Poggi, and J. Buell. 2016. Big Data Benchmark Compendium. *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things* (2016), 135–155. https://doi.org/10.1007/978-3-319-31409-9_9

[16] Michael Kaminsky. 2020. Episode 2: Row vs Column Store. https://youtu.be/XNrsRVMfj1c

[17] Rob Kitchin and Gavin McArdle. 2016. What makes Big Data, Big Data? Exploring the ontological characteristics of 26 datasets. *Big Data & Society* 3, 1 (2016). https://doi.org/10.1177/2053951716631130

[18] John Klein, Ian Gorton, Neil Ernst, Patrick Donohoe, Kim Pham, and Chrisjan Matser. 2015. Performance evaluation of nosql databases. *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems* (2015). https://doi.org/10.1145/2694730.2694731

[19] Avinash Lakshman and Prashant Malik. 2010. Cassandra. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40. https://doi.org/10.1145/1773912.1773922

[20] M. Pipattanasomporn, G. Chitalia, J. Songsiri, C. Aswakul, W. Pora, S. Suwankawin, K. Audomvongseree, and N. Hoonchareon. 2020. CU-BEMS, smart building electricity consumption and indoor environmental sensor datasets. *Scientific Data* 7, 1 (2020).

[21] Jialin Qiao, Xiangdong Huang, Lei Rui, and Jianmin Wang. 2018. Heterogeneous Replica for Query on Cassandra. (2018). arXiv:arXiv:1810.01037

[22] Vincent Reniers, Dimitri Van Landuyt, Ansar Rafique, and Wouter Joosen. 2017. On the state of nosql benchmarks. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion.* 107–112.

[23] Guoxi Wang and Jianfeng Tang. 2012. The nosql principles and basic application of Cassandra Model. *2012 International Conference on Computer Science and Service System* (2012). https://doi.org/10.1109/csss.2012.336

APPENDICES

A SQL QUERIES

1. **GetSensorData** SELECT m.measurement, t.time_measured FROM measurements m INNER JOIN sensors s ON s.id = m.sensor_id INNER JOIN times_measured t ON t.id = m.time_measured_id INNER JOIN locations l ON l.id = m.location_id WHERE l.floor = 1 AND l.zone = 1 AND s.sensor_name = 'Light' AND t.time_measured >= '2019-03-05' and t.time_measured < '2019-05-22'

2. **GetPeakData** SELECT max(m.measurement) FROM measurements m INNER JOIN sensors s ON s.id = m.sensor_id INNER JOIN times_measured t ON t.id = m.time_measured_id INNER JOIN locations l ON l.id = m.location_id WHERE l.floor = 1 AND l.zone = 1 AND s.sensor_name = 'Light' AND t.time_measured >= '2019-03-05' and t.time_measured < '2019-05-22'

3. **GetAvailableData** SELECT floor, zone FROM locations

4. **GetFloorZoneData** SELECT DISTINCT s.sensor_name, l.floor, l.zone FROM measurements m INNER JOIN sensors s ON s.id = m.sensor_id INNER JOIN locations l ON l.id = m.location_id WHERE s.sensor_name = 'AC1'

B CASSANDRA QUERIES

1. **GetSensorData** SELECT measurement, time_measured FROM measurement_by_sensor WHERE floor = 1 AND zone = 1 AND sensor = 'Light' AND time_measured >= '2019-03-05' AND time_measured < '2019-05-22'

2. **GetPeakData** SELECT max(measurement) FROM measurement_by_sensor WHERE floor = 1 AND zone = 1 AND sensor = 'Light' AND time_measured >= '2019-03-05' AND time_measured < '2019-05-22'

3. **GetAvailableData** SELECT DISTINCT floor, zone, sensor FROM measurement_by_sensor ALLOW FILTERING

4. **GetFloorZoneData** SELECT DISTINCT floor, zone, sensor FROM measurement_by_sensor WHERE sensor = 'AC1' ALLOW FILTERING

C RESULT FIGURES

C.1 Cassandra, Query 1: GetSensorData

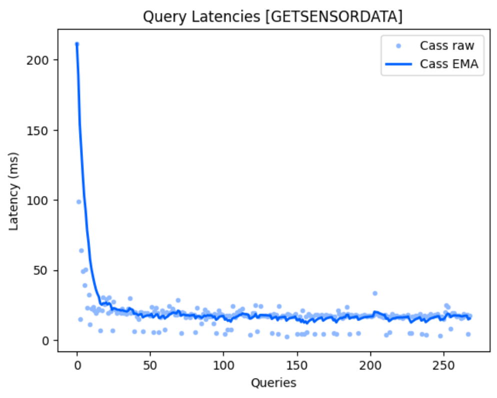


Figure 8: Latency graphed for all requests of query 1

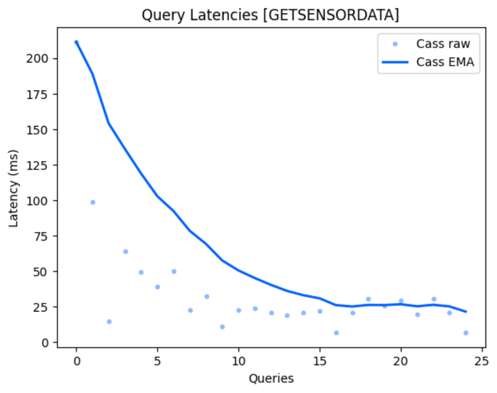


Figure 9: Latency graphed for the first 25 requests of query 1

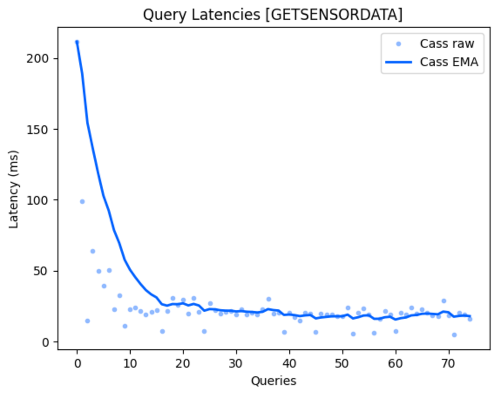


Figure 10: Latency graphed for the first 75 requests of query 1

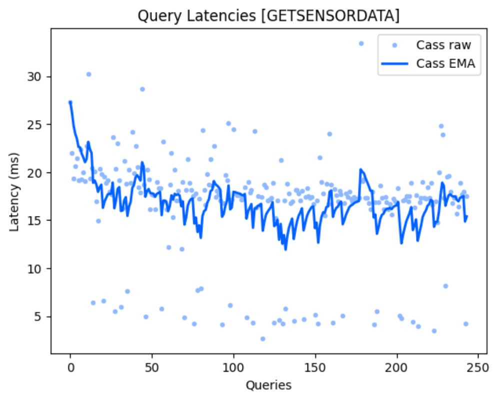


Figure 11: Latency graphed for requests of query 1 excluding the first 25 requests

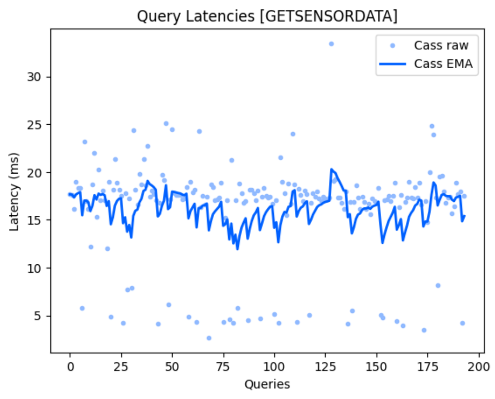


Figure 12: Latency graphed for requests of query 1 excluding the first 75 requests

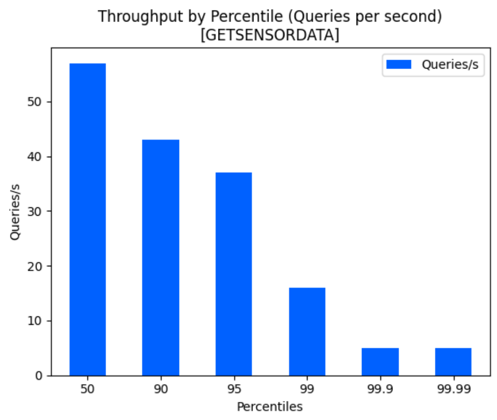


Figure 13: Throughput (queries per second) by percentile

C.2 Cassandra, Query 2: GetPeakData

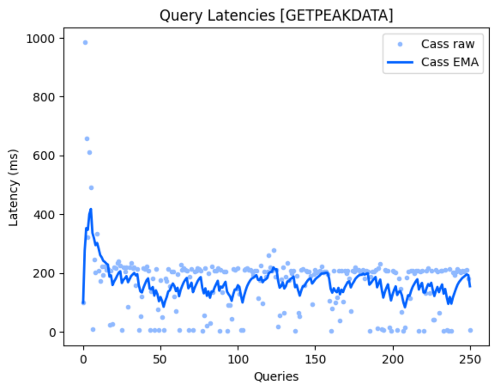


Figure 14: Latency graphed for all requests of query 2

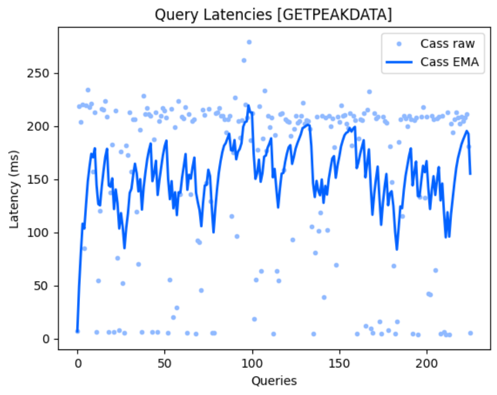


Figure 17: Latency graphed for requests of query 2 excluding the first 25 requests

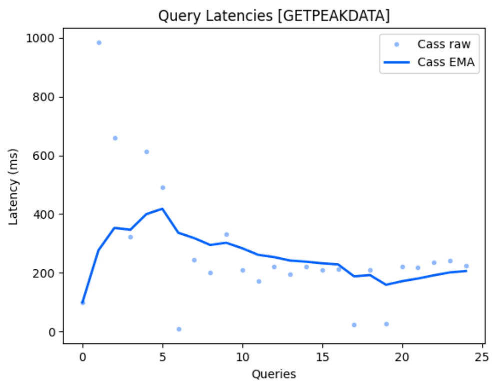


Figure 15: Latency graphed for the first 25 requests of query 2

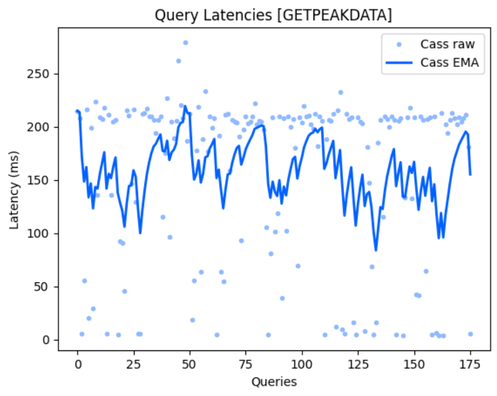


Figure 18: Latency graphed for requests of query 2 excluding the first 75 requests

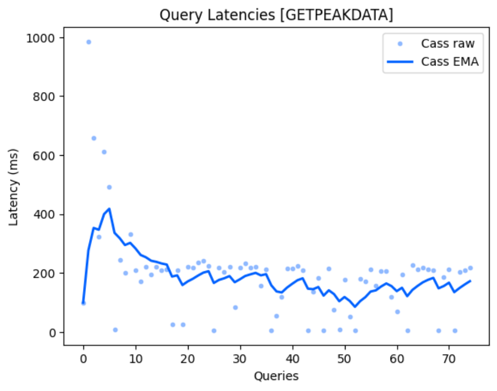


Figure 16: Latency graphed for the first 75 requests of query 2

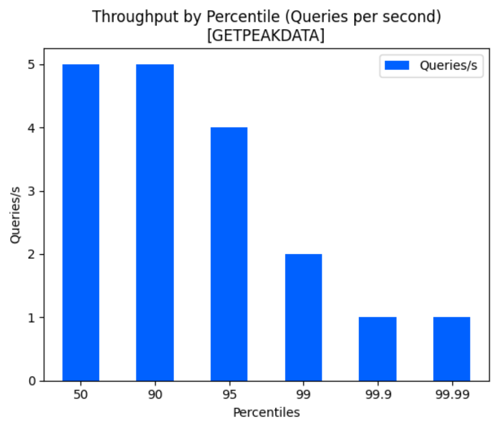


Figure 19: Throughput (queries per second) by percentile

C.3 Cassandra, Query 3: GetAvailableData

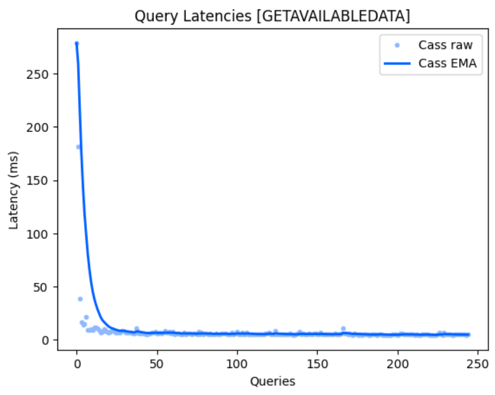


Figure 20: Latency graphed for all requests of query 3

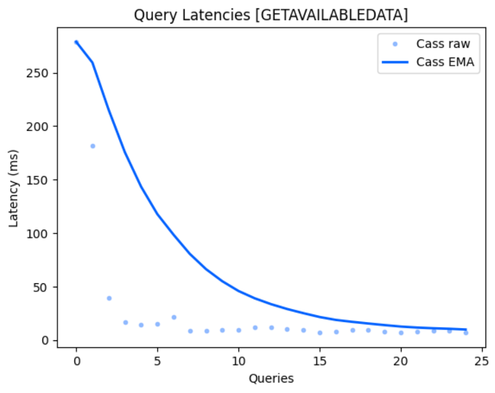


Figure 21: Latency graphed for the first 25 requests of query 3

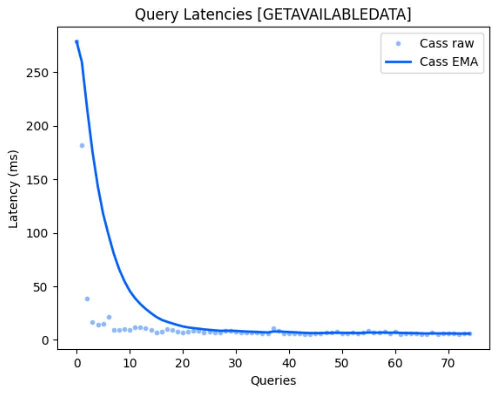


Figure 22: Latency graphed for the first 75 requests of query 3

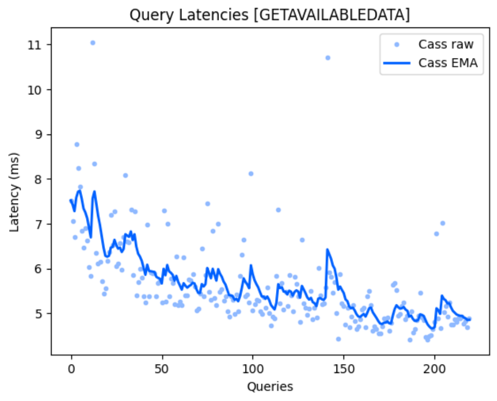


Figure 23: Latency graphed for requests of query 3 excluding the first 25 requests

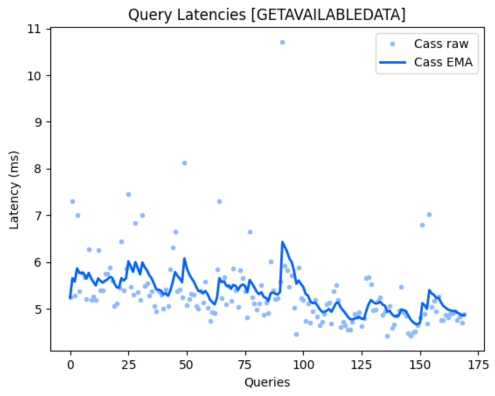


Figure 24: Latency graphed for requests of query 3 excluding the first 75 requests

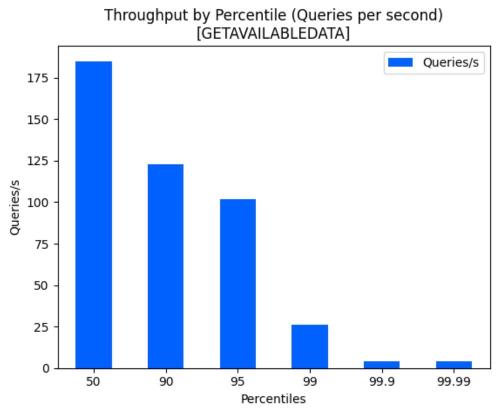


Figure 25: Throughput (queries per second) by percentile

C.4 Cassandra, Query 4: GetFloorZoneData

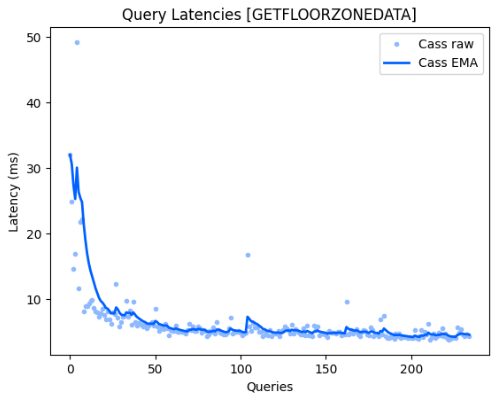


Figure 26: Latency graphed for all requests of query 4

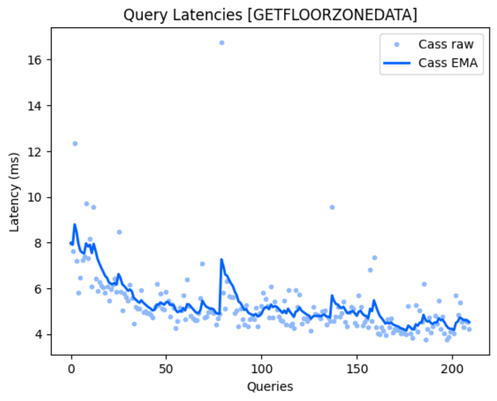


Figure 29: Latency graphed for requests of query 4 excluding the first 25 requests

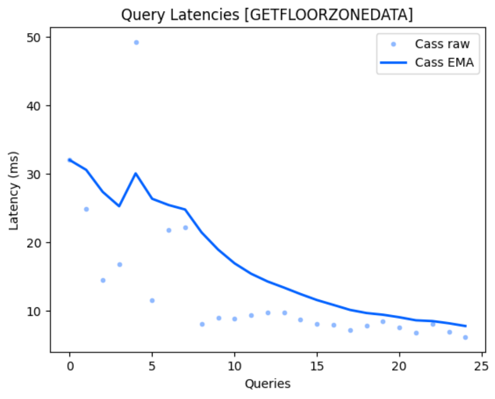


Figure 27: Latency graphed for the first 25 requests of query 4

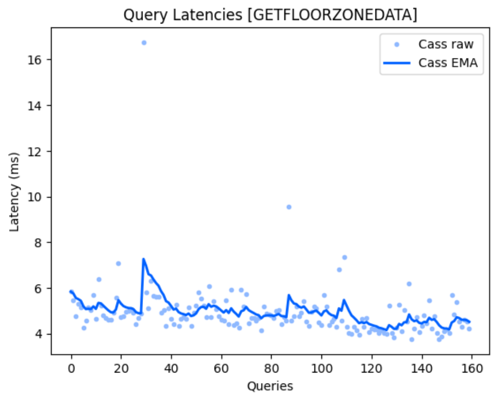


Figure 30: Latency graphed for requests of query 4 excluding the first 75 requests

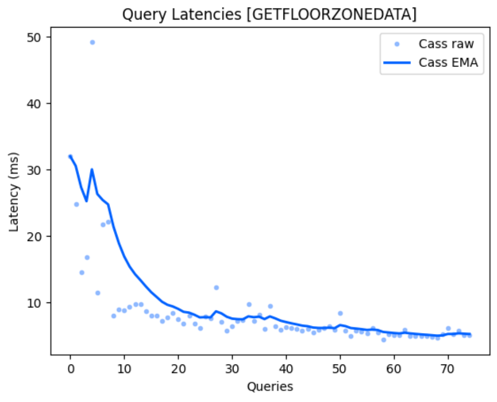


Figure 28: Latency graphed for the first 75 requests of query 4

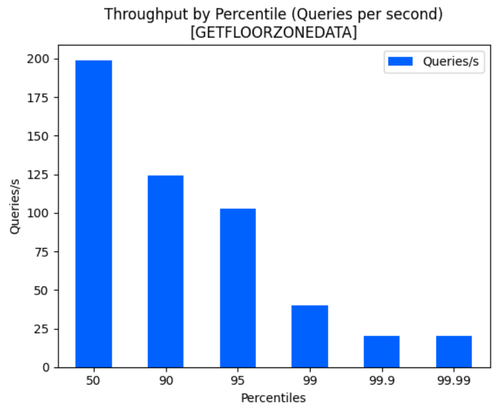


Figure 31: Throughput (queries per second) by percentile

C.5 MySQL, Query 1: GetSensorData

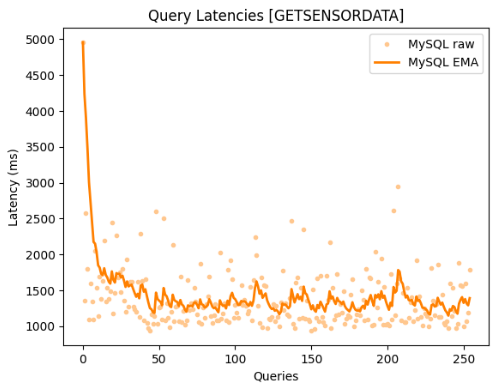


Figure 32: Latency graphed for all requests of query 1

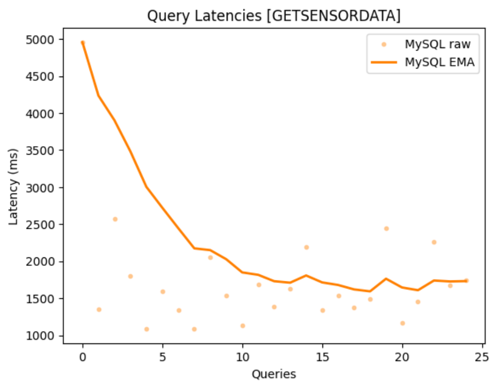


Figure 33: Latency graphed for the first 25 requests of query 1

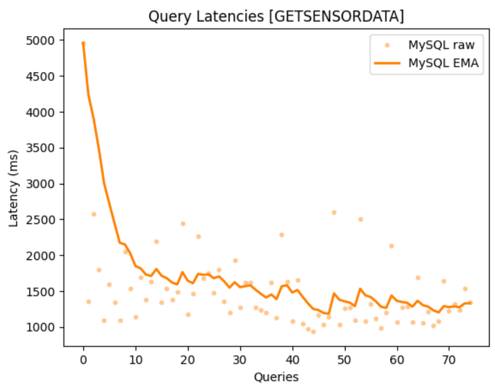


Figure 34: Latency graphed for the first 75 requests of query 1

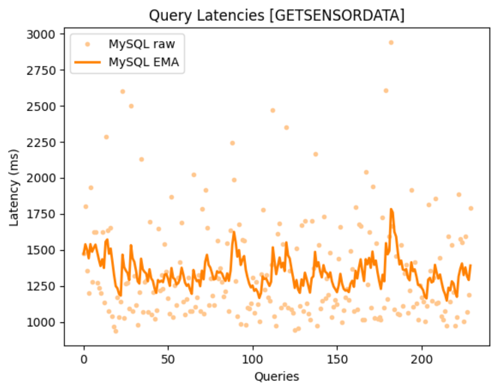


Figure 35: Latency graphed for requests of query 1 excluding the first 25 requests

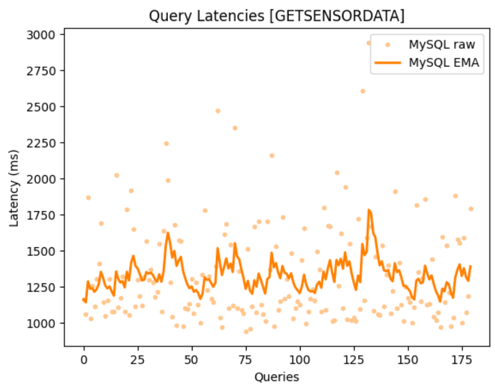


Figure 36: Latency graphed for requests of query 1 excluding the first 75 requests

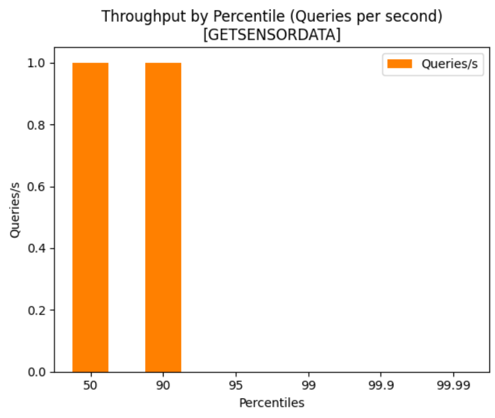


Figure 37: Throughput (queries per second) by percentile

C.6 MySQL, Query 2: GetPeakData

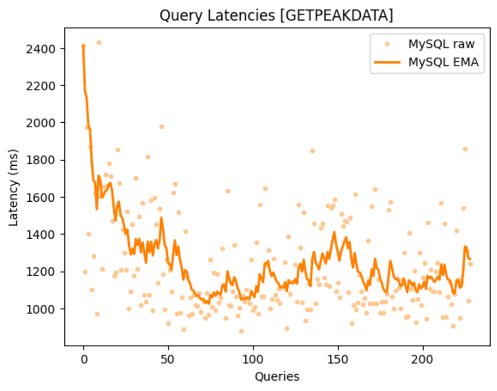


Figure 38: Latency graphed for all requests of query 2

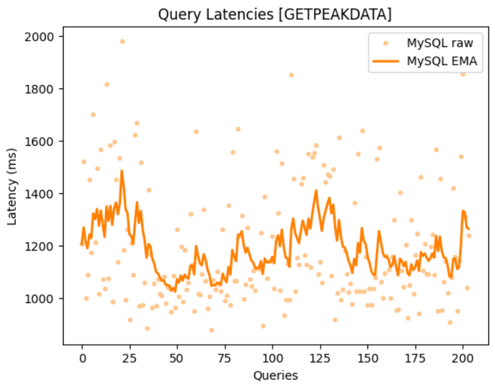


Figure 41: Latency graphed for requests of query 2 excluding the first 25 requests

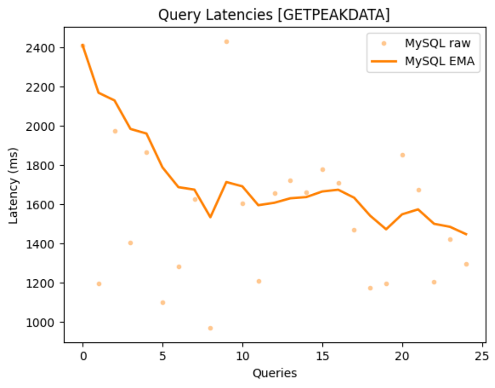


Figure 39: Latency graphed for the first 25 requests of query 2

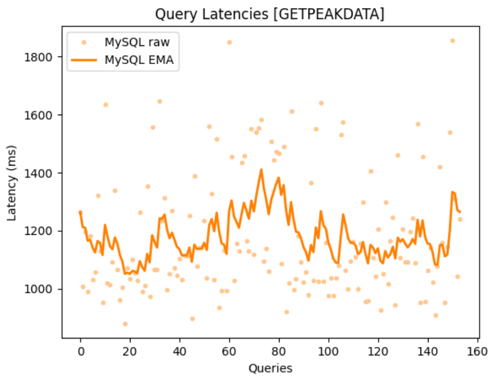


Figure 42: Latency graphed for requests of query 2 excluding the first 75 requests

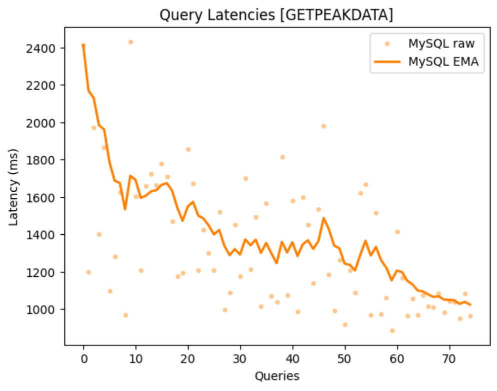


Figure 40: Latency graphed for the first 75 requests of query 2

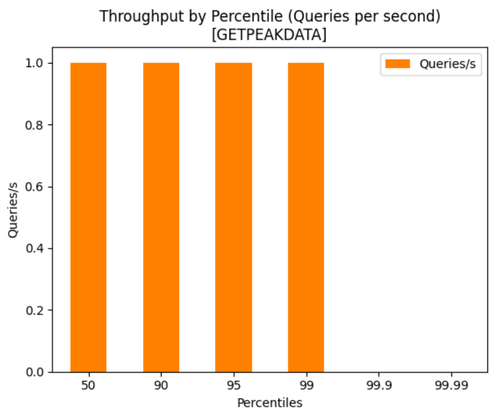


Figure 43: Throughput (queries per second) by percentile

C.7 MySQL, Query 3: GetAvailableData

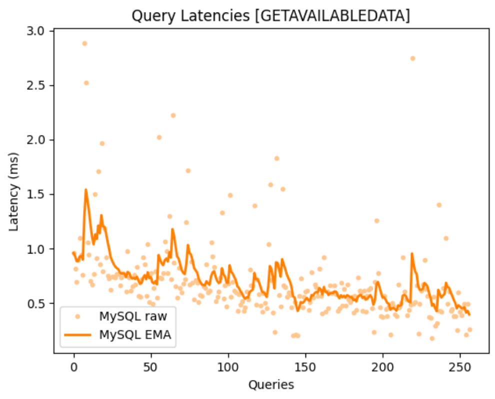


Figure 44: Latency graphed for all requests of query 3

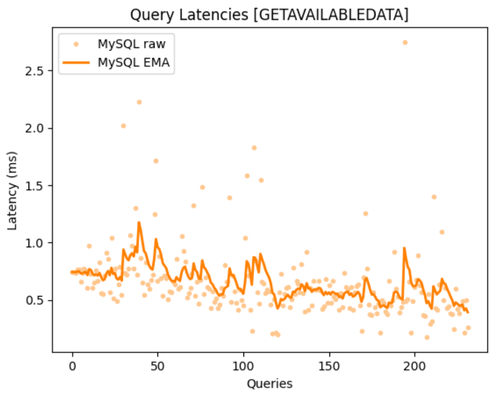


Figure 47: Latency graphed for requests of query 3 excluding the first 25 requests

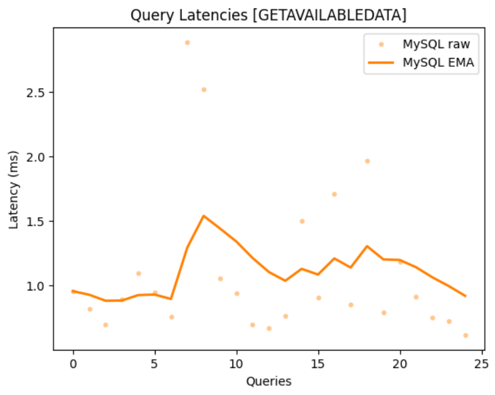


Figure 45: Latency graphed for the first 25 requests of query 3

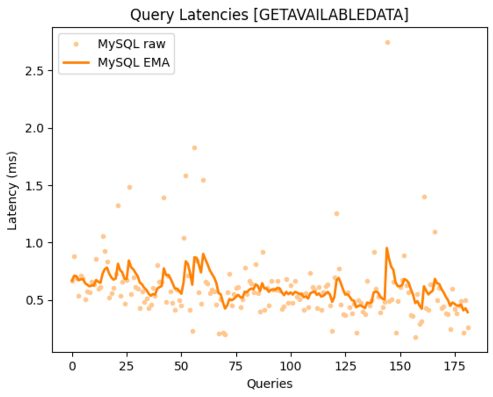


Figure 48: Latency graphed for requests of query 3 excluding the first 75 requests

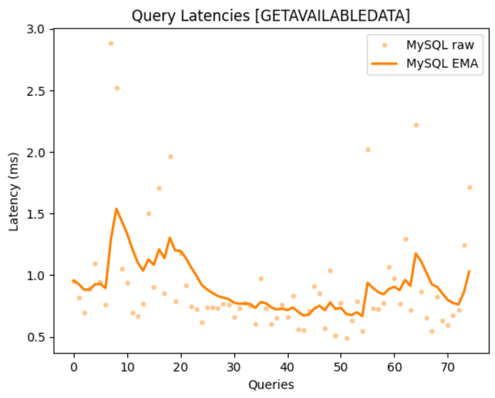


Figure 46: Latency graphed for the first 75 requests of query 3

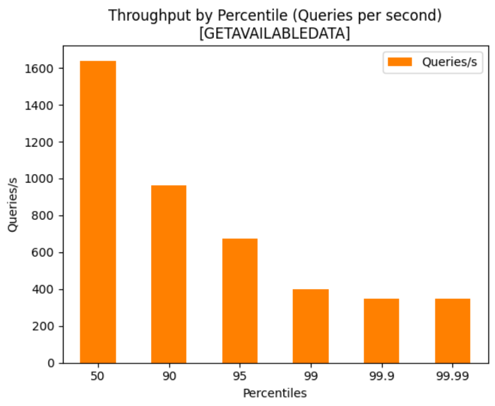


Figure 49: Throughput (queries per second) by percentile

C.8 MySQL, Query 4: GetFloorZoneData

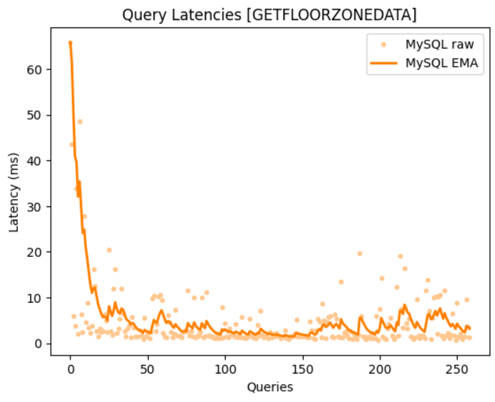


Figure 50: Latency graphed for all requests of query 4

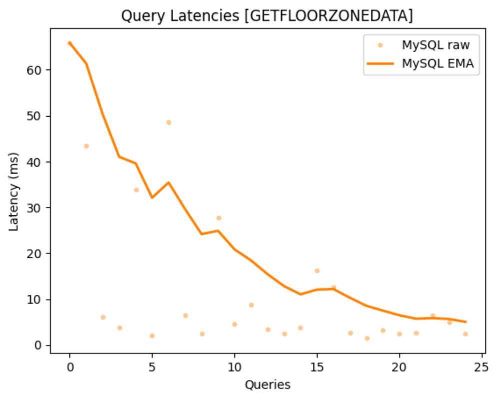


Figure 51: Latency graphed for the first 25 requests of query 4

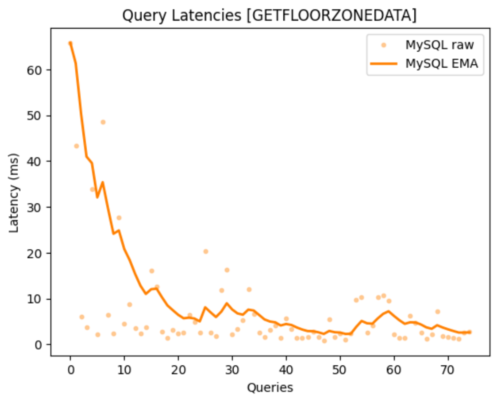


Figure 52: Latency graphed for the first 75 requests of query 4

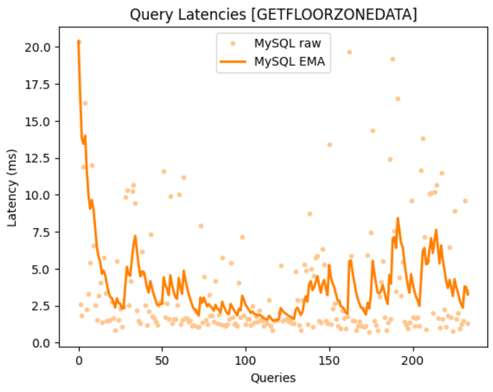


Figure 53: Latency graphed for requests of query 4 excluding the first 25 requests

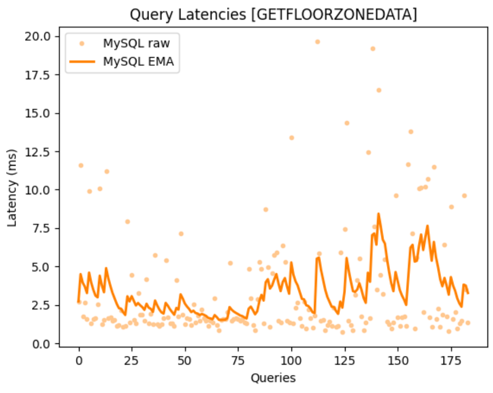


Figure 54: Latency graphed for requests of query 4 excluding the first 75 requests

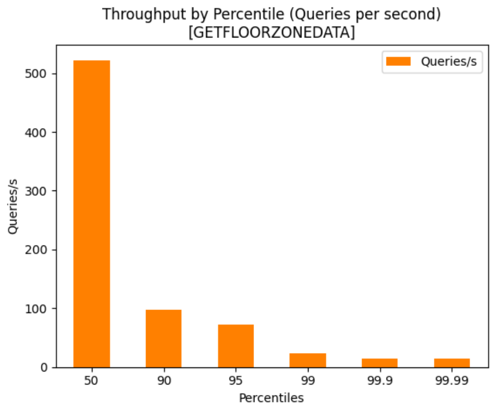


Figure 55: Throughput (queries per second) by percentile

C.9 Cassandra vs MySQL, Query 1: GetSensorData

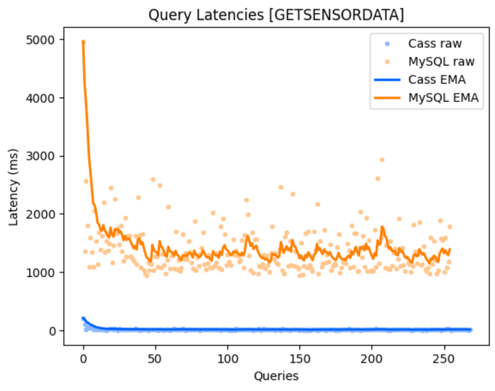


Figure 56: Latency graphed for all requests of query 1

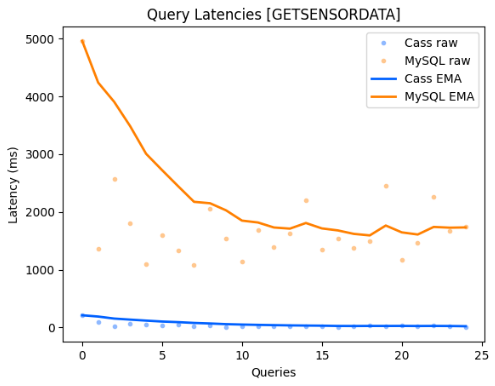


Figure 57: Latency graphed for the first 25 requests of query 1

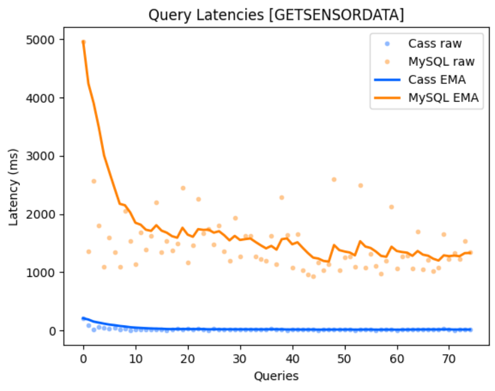


Figure 58: Latency graphed for the first 75 requests of query 1

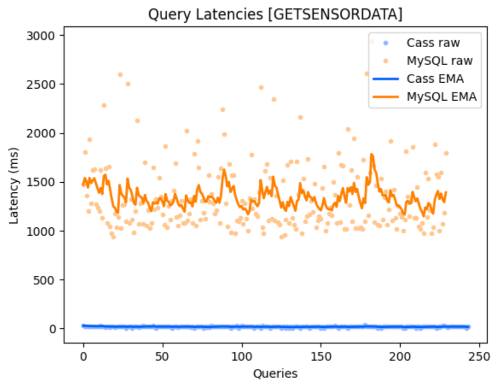


Figure 59: Latency graphed for requests of query 1 excluding the first 25 requests

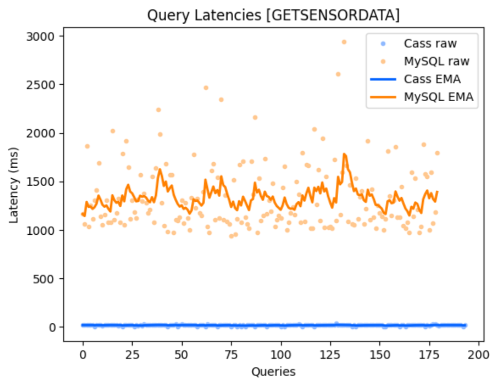


Figure 60: Latency graphed for requests of query 1 excluding the first 75 requests

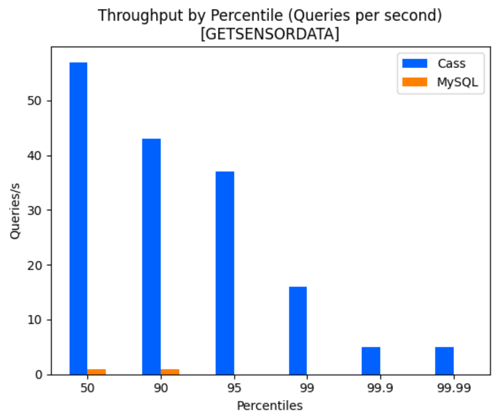


Figure 61: Throughput (queries per second) by percentile

C.10 Cassandra vs MySQL, Query 2: GetPeakData

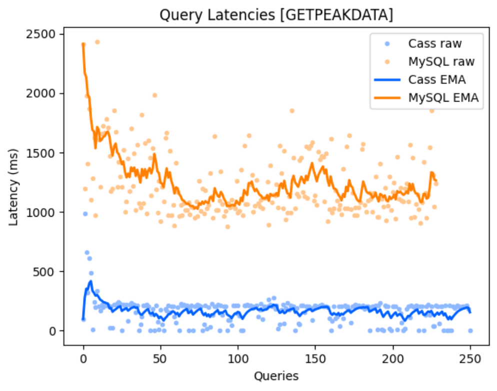


Figure 62: Latency graphed for all requests of query 2

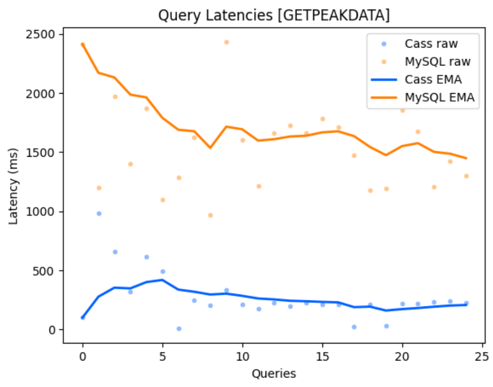


Figure 63: Latency graphed for the first 25 requests of query 2

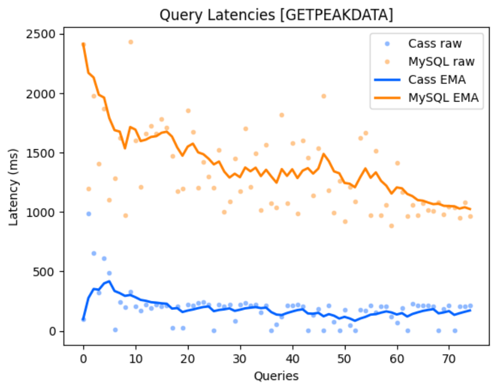


Figure 64: Latency graphed for the first 75 requests of query 2

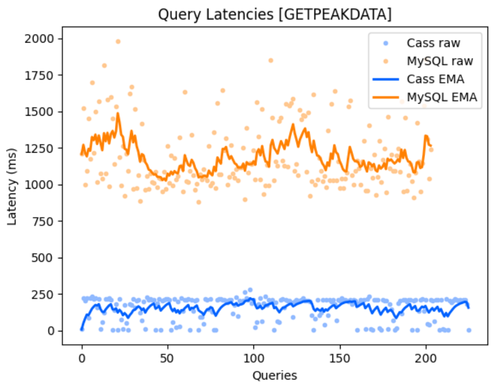


Figure 65: Latency graphed for requests of query 2 excluding the first 25 requests

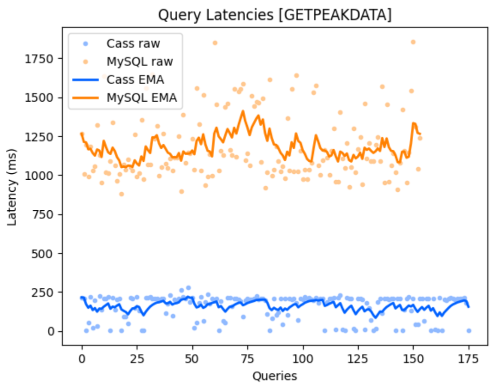


Figure 66: Latency graphed for requests of query 2 excluding the first 75 requests

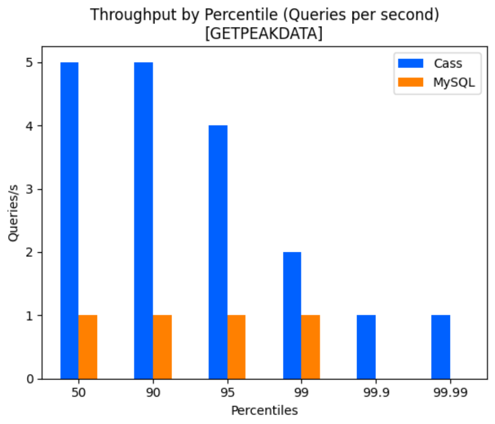


Figure 67: Throughput (queries per second) by percentile

C.11 Cassandra vs MySQL, Query 3: GetAvailableData

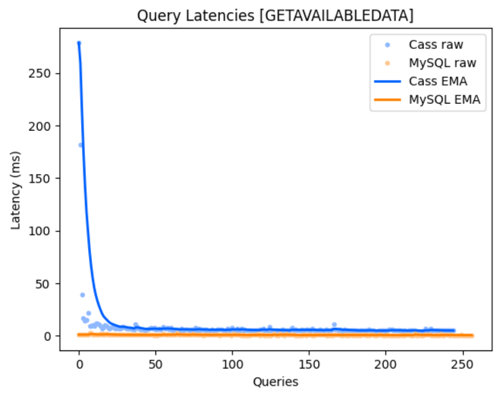


Figure 68: Latency graphed for all requests of query 3

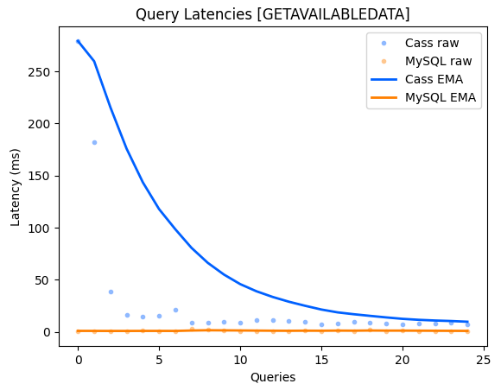


Figure 69: Latency graphed for the first 25 requests of query 3

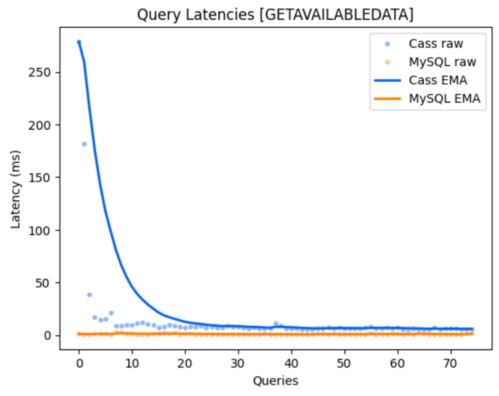


Figure 70: Latency graphed for the first 75 requests of query 3

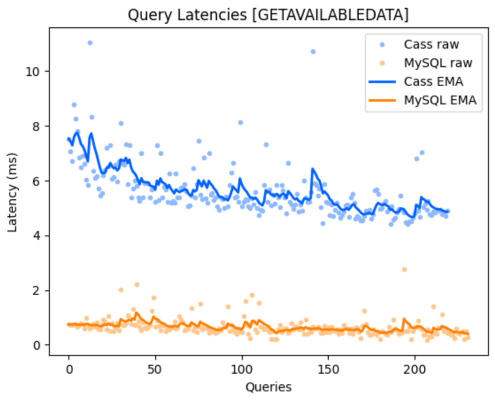


Figure 71: Latency graphed for requests of query 3 excluding the first 25 requests

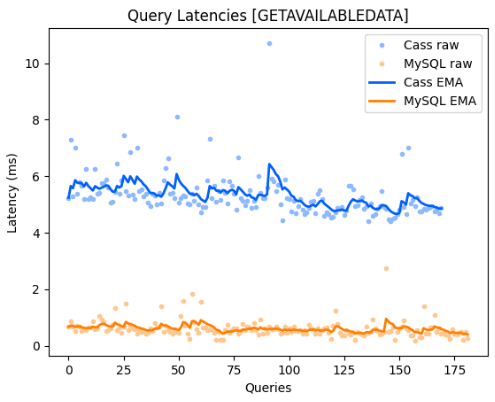


Figure 72: Latency graphed for requests of query 3 excluding the first 75 requests

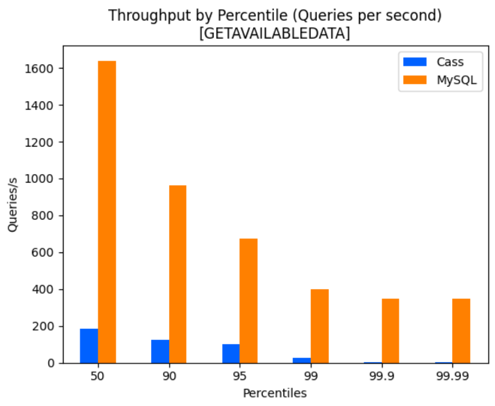


Figure 73: Throughput (queries per second) by percentile

C.12 Cassandra vs MySQL, Query 4: GetFloorZoneData

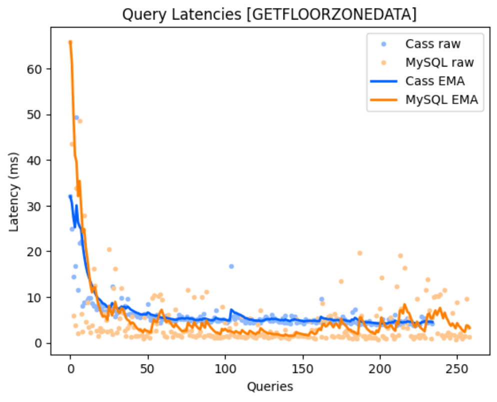


Figure 74: Latency graphed for all requests of query 4

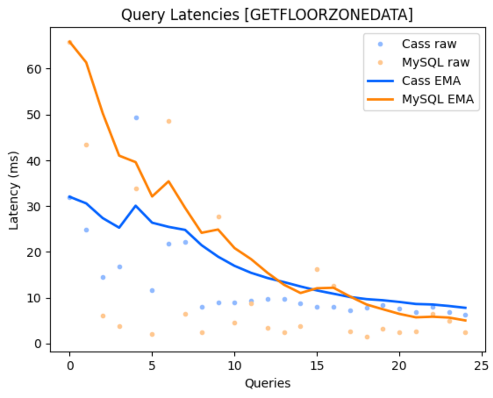


Figure 75: Latency graphed for the first 25 requests of query 4

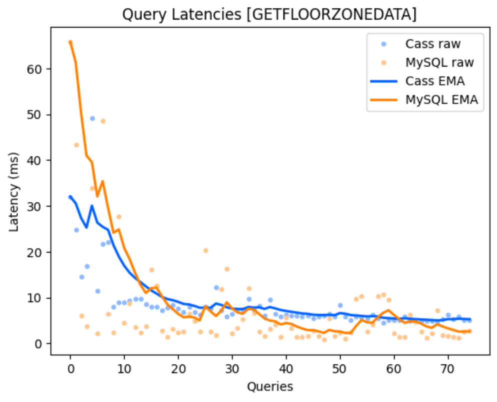


Figure 76: Latency graphed for the first 75 requests of query 4

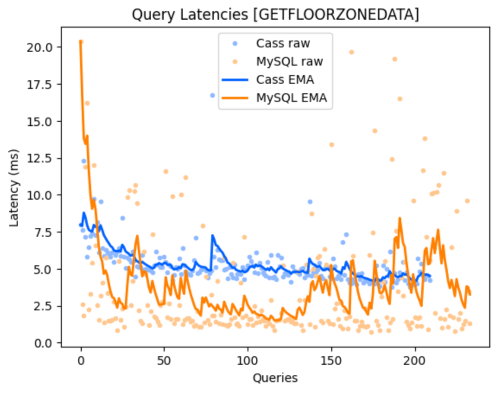


Figure 77: Latency graphed for requests of query 4 excluding the first 25 requests

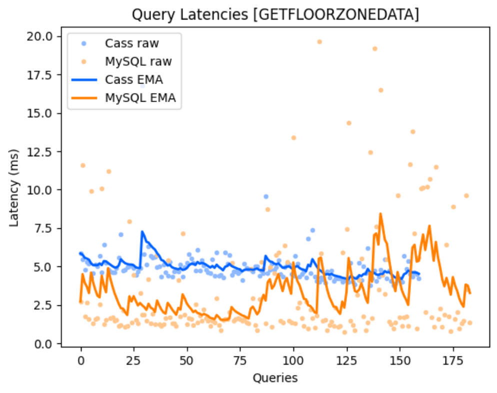


Figure 78: Latency graphed for requests of query 4 excluding the first 75 requests

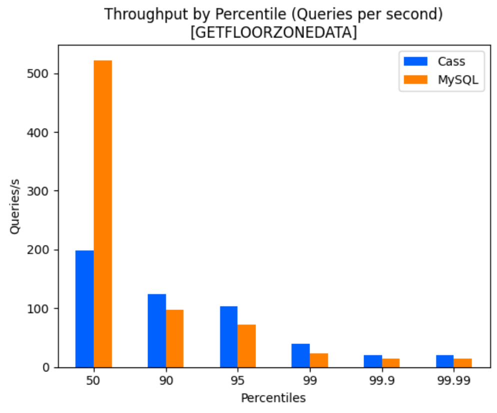
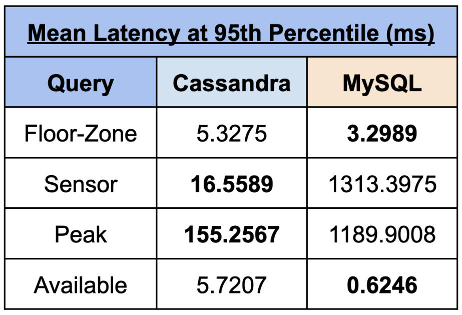


Figure 79: Throughput (queries per second) by percentile

C.13 Average Comparison with All Queries



Mean Latency at 95th Percentile (ms)		
Query	**Cassandra**	**MySQL**
Floor-Zone	5.3275	**3.2989**
Sensor	**16.5589**	1313.3975
Peak	**155.2567**	1189.9008
Available	5.7207	**0.6246**

Figure 80: Average comparison for Cassandra and MySQL latencies on IoT data over all queries at the 95th percentile — lower is better

C.14 Resource Usgae

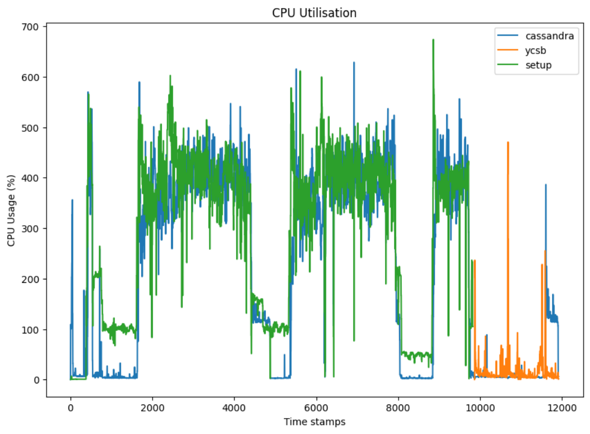


Figure 81: Docker CPU usage for Cassandra benchmark

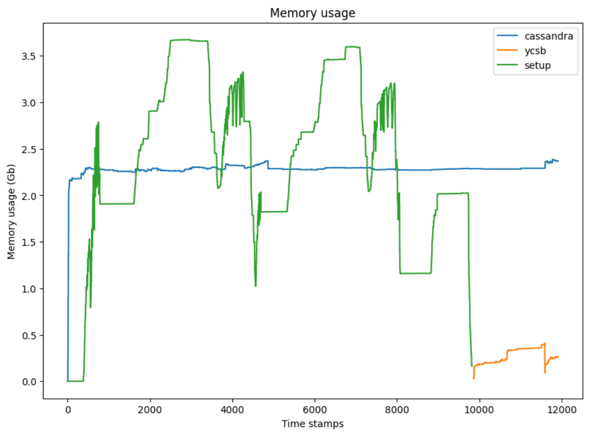


Figure 82: Docker memory usage for Cassandra benchmark

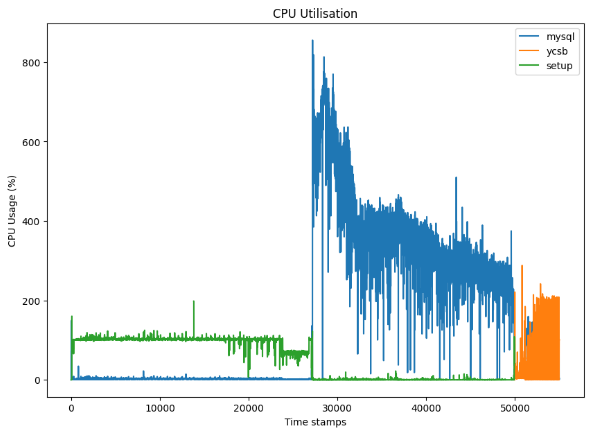


Figure 83: Docker CPU usage for MySQL benchmark

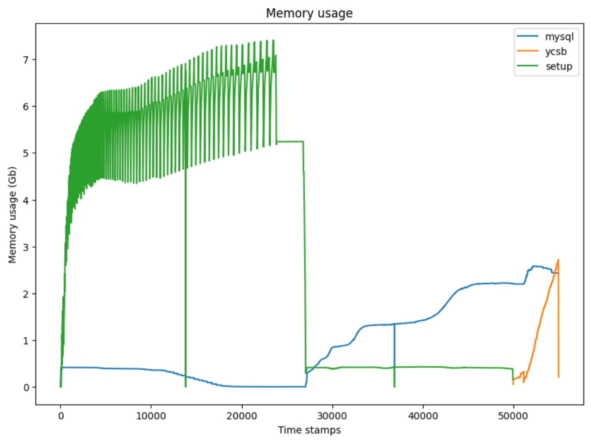


Figure 84: Docker memory usage for MySQL benchmark