A Comparative Analysis of Influx and Timescale

Panagiotis Stefanis
Department of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el19096@mail.ntua.gr

Georgios Sotiropoulos
Department of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el19848@mail.ntua.gr

Michail Tsilimigkounakis
Department of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el19001@mail.ntua.gr

***Abstract*—In the era characterized by the prevalence of sensors and the Internet of Things, timestamped data has become ubiquitous across diverse large-scale information systems, spanning from financial indices to physical measurements. The imperative for facile and efficient storage and retrieval of time-series data has reached unprecedented levels, prompting numerous renowned data engineering and database enterprises to proffer specialized solutions tailored for timestamped data. This study undertakes a comparative analysis of two prominent database systems within this domain, evaluating their performance, compression techniques, and multi-node scalability. Specifically, the selected database systems under scrutiny are *Influx* and *Timescale*, widely acknowledged as preeminent products in terms of performance, maturity, and seamless integration. This paper aims to assist decision-makers, developers, and data engineers in making informed choices based on the specific requirements of their time-sensitive applications.**

I. Introduction

We have conducted a comprehensive exploration of the prominent database systems listed above, employing a robust benchmark suite renowned as *TSBS*, denoting the Time Series Benchmark Suite [1]. This suite encapsulates an assortment of *Go* programs and Shell scripts engineered to automate the benchmarking process for a specific category of databases. It ensures methodological coherence and comparability across a diverse array of scenarios. Utilizing the functionalities inherent in the *TSBS* tools, we generated three distinct datasets, distinguished by their varying magnitudes of size, labeled as small, medium, and large datasets. Subsequently, we ingested the generated data into both databases and assessed the relative performance of each database, discerning key metrics such as rows/sec and metrics/sec, which stand as universally acknowledged pivotal performance indicators for data ingestion.

Afterwards, we took on a study of how each database stores data on disk. This analytical endeavor involved the systematic identification of on-disk database files, precise measurement of file sizes for each database, and a analysis of diverse compression techniques, especially pertinent when dealing with time-series data. Having prepared the databases and measured insert performance, we proceeded to formulate an array of query scenarios, spanning the testing spectrum from one query to multiple protracted queries.

With the queries systematically generated and organized within well-structured directories, a plan was devised for querying the databases and consolidating results for each scenario. To uphold the highest standards of accuracy, a steadfast protocol was adhered to, involving the systematic clearance of the cache at the initiation of every new query or query-set execution. Subsequently, queries were executed under diverse scenarios and infrastructures, encompassing tests on both single and multi-node architectures. The latter aspect, focusing on tests conducted with increased data-node count, furnished valuable insights into the scalability and responsiveness of the databases.

To systematically analyze and visually represent the results gathered from multiple benchmarks, we have composed *Python* scripts designed to facilitate the visualization of the generated benchmark data. However, it is essential to note that these scripts are intended for execution on a system equipped with a Graphical User Interface (GUI). This is particularly important as the utilization of the *matplotlib*, a widely adopted *Python* plotting library, is contingent upon a GUI environment for graph generation. In the conclusive section of this document, we critically study the outcomes derived from the *TSBS* benchmark, shedding light on the distinctive advantages that one database holds over the other.

For a detailed step-by-step guide, including the specific commands used in the installation processes, please refer to our GitHub Repository [2]. There, you will find a comprehensive procedure outlined with commands, providing an in-depth resource for replicating and understanding each phase of the setup for both *Timescale* and *Influx*. Feel free to explore the repository for additional insights, documentation, and any updates related to the setup procedures. This resource is designed to facilitate a smooth and transparent replication of the processes described, enhancing your understanding and ensuring accurate implementation.

II. Single Node Setup

A. System Setup

To execute our benchmarking procedure in the single-node mode of *Timescale* (an extension of *Postgres*) and *Influx* database systems, we intend to deploy a virtual machine on the *Okeanos* Infrastructure as a Service (IaaS) cloud provider.

As a virtual machine, we have instantiated an *Ubuntu 22.04.03 LTS* instance, utilizing the 5.15.0-91-generic Linux kernel. The virtual environment is configured with an *Intel(R) Xeon(R) CPU E5-2650 v3* operating with 4 cores at 2.3GHz, 4GB of RAM, and a 30GB virtual boot drive. The virtual machine is automatically assigned a public IPv6 address, and a domain is provided to facilitate seamless SSH access. Furthermore, a public IPv4 address has been specifically assigned to enable compatibility with services requiring IPv4 connectivity, such as GitHub, which lacks native support for IPv6.

B. Influx Installation

To commence, we will install *Influx* version 1.8.10 [3]. This entails adding *Influx's* repository to the apt sources list and proceeding with the installation through the apt package manager. Following the installation, we will initiate the *Influx* service to ensure that the Database Management System (DBMS) operates in the background on our benchmarking machine.

C. Timescale Installation

The second timeseries database, *Timescale*, operates as an extension built upon *PostgresSQL*. To facilitate the installation of *Timescale*, it is imperative to first install *Postgres* [4]. We will utilize version 14.10 of *Postgres* for this purpose. As a preliminary step, we need to install specific third-party packages. Following this, we can execute a provided shell script by *Postgres* to seamlessly handle the installation process.

Upon the successful installation of *Postgres* on our virtual machine, the subsequent step involves the installation and activation of *Timescale*. For compatibility reasons, we have opted to use version 2.13.0 [4], as it was the latest version available at the time of installation. In a manner similar to the procedure employed for *Influx*, the inclusion of *Timescale*'s repository in the apt sources list is required, followed by the initiation of the installation process through the apt package manager.

Upon completion of the installation, we are required to execute a tuning script provided by *Timescale*. Finally, we utilize *Postgres*'s Command Line Interface (CLI) environment to register the *Timescale* extension with the Database Management System (DBMS).

D. TSBS Installation

The benchmarking system designated for our analysis is referred to as *TSBS* (Timeseries Database Benchmarking System) and is under the maintenance of *Timescale*. While *TSBS* is capable of benchmarking numerous timeseries databases, our focus will be on evaluating two specific databases, namely *Timescale* and *Influx*. Developed in the *Go* programming language, *TSBS* necessitates compilation on our benchmarking machine. Therefore, our initial step involves the installation of the *Go* version 1.18.1, a process facilitated through the apt package manager.

Following the successful installation of *Go*, we proceed to clone the official *TSBS* GitHub repository [1] and compile it on our system. This involves utilizing *Go's* built-in installation system to acquire the *TSBS* repository and subsequently employing the make utility for compilation. Once *TSBS* is successfully built, it becomes imperative to inform the shell about the paths of *TSBS*'s binary executables. To achieve this, we export the directory containing *TSBS*'s binaries to the PATH environment variable. The scripts housed in this directory will be utilized throughout various stages of this analysis, including data generation, query generation, data insertion, and query execution.

III. Data Generation

In the data generation process, we employed the `tsbs_generate_data` script provided by *TSBS*. Our initial decision aimed at creating three distinct datasets, intending to assess how the two databases would handle variations in dataset sizes. These datasets consist of a small dataset sized at 1GB, a medium dataset sized at 5GB, and a large dataset measuring 15GB.

Concerning the specification of dataset sizes, we determined the appropriate timestamps for the start and end of our data within the `tsbs_generate_data` tool. This selection allowed us to generate the three distinct datasets (small - 1GB, medium - 5GB, large - 15GB).

For the selected research use case, we focused on the *Internet of Things (IoT)*. Rigorous consistency was maintained in the application of non-random results across all data and generated queries. This approach ensures the repeatability of *IoT* data generation, facilitating adjustments in both the database size and dataset dimensions.

This use case is meant to simulate the data load in an *IoT* environment. This use case simulates data streaming from a set of trucks belonging to a fictional trucking company, simulating diagnostic data and metrics from each truck. Environmental factors are also introduced, such as out-of-order data and batch ingestion (for trucks that are offline for a period of time). It also tracks truck metadata and uses this to tie metrics and diagnostics together as part of the query set.

For those seeking additional technical insights into the data generation process, comprehensive details are available in our script located within our repository [2].

IV. Query Generation

For the query generation process, we elected to employ all query types provided by *TSBS* within the context of the *IoT* use case. Subsequently, a decision was made to assess, for each dataset, the execution of a singular query for each type. Additionally, we conducted evaluations involving the execution of ten consecutive queries of the same type. This intentional approach was aimed to scrutinize the indexing and caching performance of the respective databases, while also obtaining a more precise measurement for each query type by calculating the mean value across the executions of ten consecutive queries. The inclusion of ten consecutive queries was considered essential due to the inherent randomness in query execution, which, in a single execution, could introduce inaccuracies into our experiment.

To facilitate these procedures, we adapted and employed *TSBS*'s bash script `generate_queries.sh` to generate queries for the three databases established for both *Influx* and *Timescale*. The script, now renamed as `generate_all_queries.sh`, resides in the repository [2]. Its primary function is to parameterize and execute the Go binary `tsbs_generate_queries`.

V. Single Node Insert Performance

To assess the insertion performance of each database, we employed bash scripts, supplied by *TSBS* (Time Series Benchmark Suite). Specifically `load_timescaledb.sh` and `load_influx.sh` [2] are used to parameterize and execute the `tsbs_load_timescaledb` and `tsbs_load_influx` [1] *Go* binaries.

For `load_timescaledb`, the initial modification involves adjusting the `DATA_FILE_NAME` environment variable to specify the desired file for insertion. Subsequently, relevant PostgreSQL authentication environment variables are adjusted: `DATA_BASE_{USER_NAME| NAME| HOST| PORT| PASSWORD}`

Before executing the script, the `BULK_DATA_DIR` variable in `load_common.sh` [2] must be updated to point to the directory containing the generated data. In our case: `BULK_DATA_DIR = TimeseriesDB_Benchmarks/data_generate/iot_data`

The required modifications for running the `load_influx.sh` script mirror those for `load_timescaledb` with the only exception being, in the case of *Influx*, the password variable is not required.

We then analyzed the generated output files and generated performance graphs, utilizing a *Python* script to compare the ingestion rates in rows per second and metrics per second for each database at different sizes. The results are visually depicted in Fig. 1.

In the context of performance insertion metrics:

- Rows/s (Rows per Second): This metric indicates the rate at which individual rows or data points are being ingested, processed, or queried in the database.
- Metrics/s (Metrics per Second): This metric refers to the rate at which sets of values or measurements (metrics) are being processed. It's a higher-level metric that accounts for the fact that each row may contain multiple values or dimensions.

In terms of rows rate, *Timescale* demonstrates a peak performance for medium dataset sizes, reaching 57.99K rows/sec. In comparison, *Influx* exhibits a noticeable reduction at 45.66K rows/sec for the same dataset, reflecting a substantial decrease of 21%. In scenarios involving smaller datasets, the disparity between the two databases diminishes, with *Timescale* achieving a rate of 51.12K rows/sec and *Influx* registering 49.19K rows/sec. In the context of larger datasets, the performance differential lies between the two aforementioned scenarios, with a discernible difference of approximately 17%.

Regarding the metric rate, *Influx* consistently surpasses *Timescale*, exhibiting a substantial increment across all datasets. In particular, *Influx* attains its zenith at 388.87K metrics/sec for the small dataset, displaying a marginal reduction as the scale of the database expands by an order of magnitude. Conversely, *Timescale*, while manifesting comparatively smaller peaks, demonstrates commendable proficiency in maintaining a consistent or enhanced ingestion rate as the database size increases, in stark contrast to the deceleration observed in *Influx*.

The distinct nature of *Influx* and *Timescale* is evident from the depicted graph. *Influx* is specifically engineered for the continuous ingestion of metrics, exhibiting prowess in managing high-frequency, real-time data *Influx* within the domain of time-series data. Its storage engine and query language are optimized for this continuous ingestion pattern, making it well-suited for scenarios where new data points are arriving frequently.

In contrast, *Timescale* (as a *PostgreSQL* extension) follows a row-based approach to data storage and ingestion. Although *Timescale* is finely tuned for time-series data and incorporates extensions aimed at augmenting performance within such workloads, it remains steadfast in its adherence to the relational model. This adherence underscores the methodical insertion of data into the database structure as individual rows, aligning with conventional relational database practices.

It is imperative to acknowledge that, as elucidated in the forthcoming section concerning disk sizes, our methodology involves adhering to the default compression configurations implemented by *Timescale* and *Influx*. Notably, *Influx* employs a data compression mechanism, while *Timescale* does not. This intrinsic discrepancy in compression methodologies likely contributes to the fact that *Timescale* can sustain a more consistent or augmented insertion rate. In contrast, *Influx* exhibits a reduction in performance as the database size

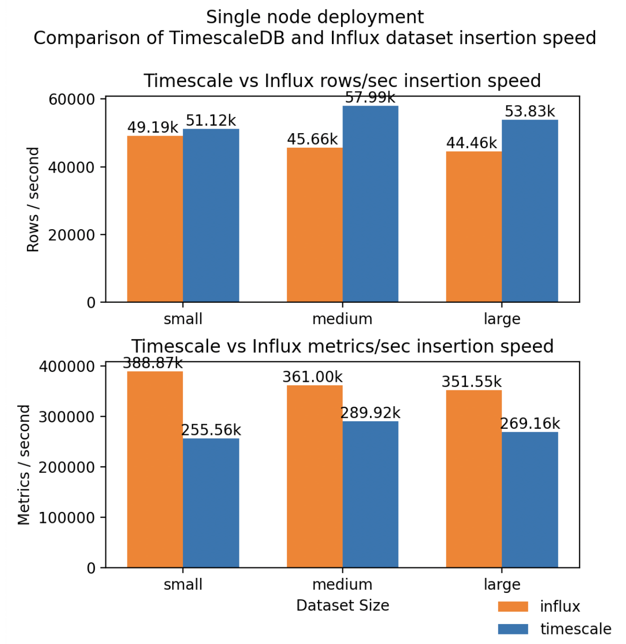


Fig. 1: Insertion performance on single node deployment.

expands, potentially indicative of an increased cost associated with compression in the context of larger datasets, as opposed to their smaller counterparts.

VI. Single Node Size on Disk

Upon the completion of data insertion into each respective database and subsequent collection of results, our inquiry advanced to a comparative assessment of the database files stored on disk, with particular attention given to the compression strategies employed by the two databases. This becomes imperative when managing an array of rapidly accumulating daily timestamped measurements.

The storage engine that powered *Influx* 1.x and 2.x was called the *Influx* Time-Structured Merge Tree (TSM) that resembles a LSM Tree (a data structure with performance characteristics that make it suitable for providing indexed access to files with high insert volume). TSM uses run length encoding for compression. This approach is very efficient for metrics use cases where data timestamps occurred at regular intervals. *Influx* was able to store the starting time and time interval, and then calculate each time stamp at query time, based on only row count. Additionally, the TSM engine could use run length encoding on the actual field data. So, in cases where data did not change frequently and the timestamps were regular, *Influx* could compress data very efficiently. However, use cases with irregular timestamp intervals, or where the data changed with nearly every reading, reduced the effectiveness of compression. Additional insights into the intelligent and efficient methodology employed by *Influx's* storage engine to process incoming data can be found [5], [6]. A visual demonstration of our assertions is evident in the graph presented in Fig. 2, highlighting *Influx's* adeptness in achieving surprising compact database disk sizes. Notably, in datasets of 1 GB, 5 GB, and 15 GB sizes, *Influx* demonstrates an impressive storage efficiency, realizing respective sizes of 0.15 GB, 0.56 GB, and 1.5 GB.

In contrast *Timescale*, by default, opts for a strategy of not applying compression to stored data. This choice aligns with the philosophy of prioritizing performance and minimizing computational overhead during data storage operations. By abstaining from compression as the default setting, *Timescale* perhaps aims to streamline write operations and maintain responsiveness, especially in scenarios involving high-volume time-series data. The rationale behind this decision is rooted in the acknowledgment that compression introduces additional processing overhead during data writes and could potentially impact the speed of data ingestion. By favoring a non-compressed storage approach by default, *Timescale* provides users with the flexibility to fine-tune compression settings based on their specific use cases, ensuring optimal performance tailored to individual requirements. The decision by *Timescale* to abstain from data compression is depicted in the ensuing graphical representation, wherein the sizes of databases for each dataset markedly exceed those of *Influx* and, notably, surpass the sizes of the datasets themselves. Concretely, the *Timescale* database allocated 1.14 GB for a small dataset of 1 GB, 5.69 GB for a medium dataset of 5 GB, and 17.16 GB for a large dataset of 15 GB. This outcome underscores the impact of *Timescale's* uncompressed default setting on the resulting database sizes, revealing a notable divergence from the more compact storage achieved by *Influx*.

While it may appear somewhat inequitable to *Timescale* in the context of disk size comparison, we have chosen to proceed with the entire evaluation using the default settings of each database – wherein *Influx* employs compression, while *Timescale* refrains from doing so. We posit that, despite an initial impression of an unconventional comparison, each database is intricately optimized for its default settings. The perceived disadvantage concerning disk sizes is anticipated to be offset by potential gains in processing speed operations. This strategic decision aligns with the premise that default configurations are tailored to achieve optimal performance within the intended operational parameters of each respective database system.

In order to measure disk size we use ”du” command with root privileges which is a standard Linux/Unix command that allows a user to gain disk usage information. *Postgres* which is the underlying database for *Timescale* extension uses the directory `var/lib/postgresql/14/main/base` to store database files inside directories with numeric names (e.g. 34250). This numeric value is called OID and is a data type which *PostgreSQL* uses as a unique identifier (primary key) for database objects. The OID data type is implemented as an unsigned 32 bit integer.

Influx uses the directory `var/lib/influxdb/data` and stores database files inside directories with the `DATA_BASE_NAME` as name.

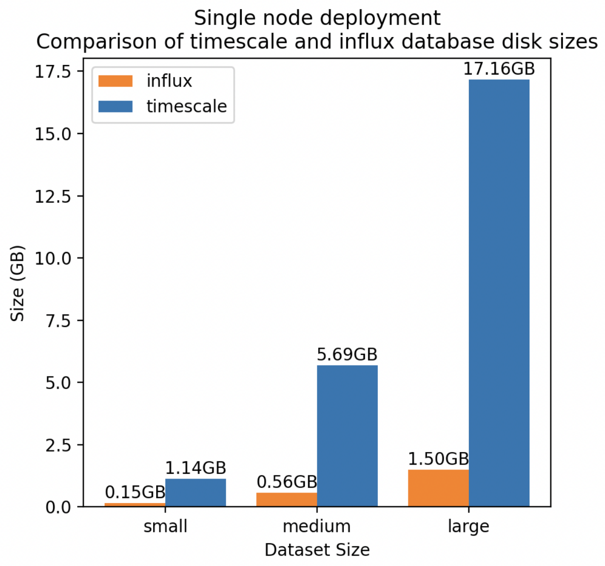


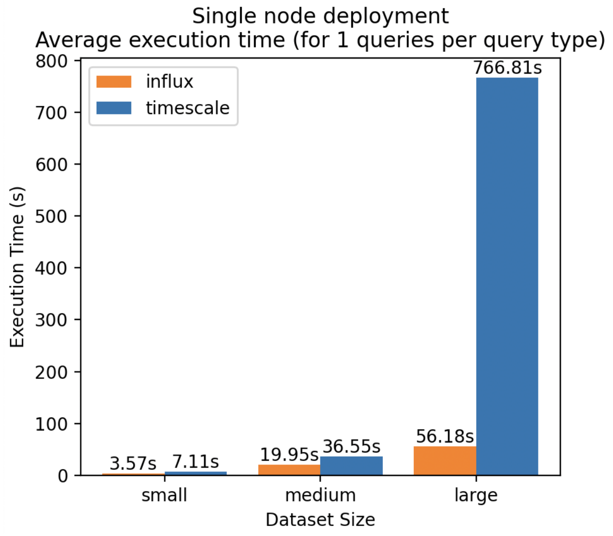
Fig. 2: Disk size comparison on single node deployment.

VII. Single Node Average Query Performance

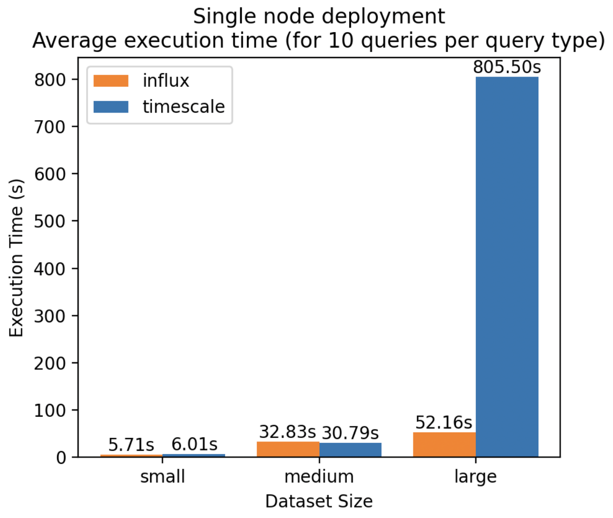
To quantify query performance across each database, we systematically gather data and construct bar graphs encompassing two distinct scenarios. The initial scenario involves the execution of a singular query per query type, with subsequent computation of the average execution time. The second scenario entails the execution of ten consecutive queries per query type. The ensuing graphs visually present the mean execution time for all query types across the three distinct datasets (small, medium, large) under both scenarios. A concise analysis detailing the comparative efficiency of databases in handling aggregates, joins, and simple projections will be expounded upon shortly towards the conclusion of this section analyzing briefly the performance for every query type.

Initially, discernible differences emerge in the execution patterns when comparing single and ten queries per type. The primary divergence is characterized by a slight convergence concerning small and medium datasets between the two databases, specifically *Influx* and *Timescale*, in the context of ten consecutive queries' execution. However, the disparity persists notably in large datasets, showcasing substantial levels and, in fact, exceeding that observed in a single query execution.

Both graphical representations depict that for small and medium datasets, distinctions between the two databases are less pronounced. Nevertheless, concerning large datasets, the disparity intensifies significantly, with *Influx* surpassing *Timescale* in performance. This observation contrasts with findings from publicly available benchmarks. Existing researches suggests that *Influx* excels in smaller databases, with its performance markedly deteriorating as data sizes increase, where *Timescale* outperforms *Influx* [7]. However, it is worth noting that the limitations of our hardware resources, encompassing CPU, memory, and disk storage, constrain our ability to scale up data and conduct experiments as the dataset size really expands. Despite our limitations we should confine our analysis to the observations discerned from our graphical representations.



(a) Average execution time (1 query) on single node deployment.



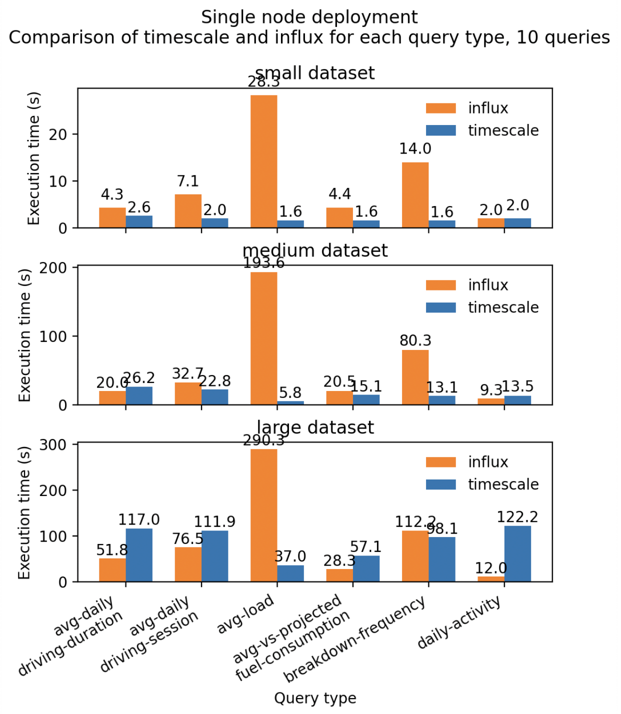
(b) Average execution time (10 query) on single node deployment.

Fig. 3: Average query execution for both scenarios.

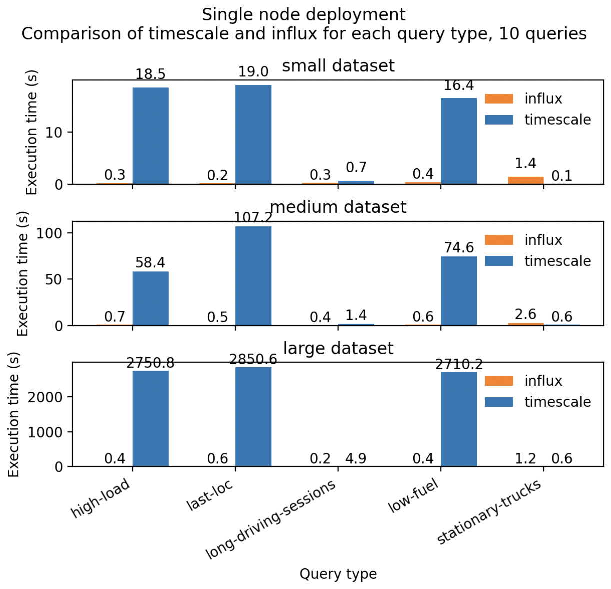
Therefore, our graphs reveal that *Influx*, particularly with our large database, consistently outperforms *Timescale*, primarily attributable to three specific query types whose execution duration are notably prolonged in *Timescale*. Further elaboration on the execution of these individual query types and briefly analysis for the execution of each query type will be provided in an upcoming section.

VIII. Single Node Query Performance per Query Type

The graphical representations depicted in Fig 4. clearly indicate a discernible trend wherein the execution time for



(a) Average execution time per query type on single node deployment (Queries 1-6).



(b) Average execution time per query type on single node deployment (Queries 7-11).

Fig. 4: Average query execution per query type (single node).

each query type increases with the expansion of the dataset. Notably, certain queries exhibit performance scaling characterized by both sub-linear and super-linear trends. Remarkably, such scaling dynamics may vary for a given query when transitioning from a small to a medium dataset and from a medium to a large dataset. It is noteworthy that specific queries deviate from conventional scaling patterns, manifesting an exceptional surge in execution time as the dataset progresses from medium to large. Particularly, high-load, last-loc, and low-fuel queries demonstrate a notable deterioration in performance, with execution times tens of times worse. Conversely, certain queries manage to maintain consistent, swift execution times and, in some instances, even exhibit improved performance despite the augmented dataset size.

Upon comparing the various query types, it becomes evident that in the case of queries avg-load and breakdown-frequency, *Influx* demonstrates a notable decrease in speed, with the disparity expanding as the dataset increases , except for a discernible convergence between the two databases when progressing from medium to large dataset in the breakdown-frequency query type scenario. Both queries involve sub-queries and group-by operations with multiple columns for grouping. The substantial performance gap observed may potentially be attributed to the selection of columns for index creation by these two timeseries databases.

Conversely, queries labeled as high-load, last-loc, and low-fuel exhibit a pronounced performance discrepancy, favoring *Influx*. In contrast to *Influx's* slower queries (specifically avg-load and breakdown-frequency), this distinction grows exponentially more substantial with the expansion of the dataset, resulting in execution times thousands of times worse than those observed with *Influx*. These three queries share a common characteristic in that they abstain from applying temporal filters to the timeseries data, thus doing a traversal of the entire dataset while conducting group-by operations and elementary mathematical calculations. This exhaustive dataset traversal imposes a considerable cost on the database, particularly when executing group-by operations based on a specific column. While it is not universally accurate to assert that the degradation is consistently exponential, it is customary for the performance impact to become more discernible with larger datasets. The reason as to why *Influx* has the ability to sustain optimal performance under these circumstances remains unknown.

Additionally, detailed information about each query type, including its corresponding SQL code and a concise explanation, is available in our repository within the `queries_generate` folder [2].

IX. Multi Node Setup

A. System Setup

To evaluate the scalability of each database, we established a cluster comprising three nodes (virtual machines provided by *Okeanos*). For each virtual machine, we instantiated an instance of *Ubuntu 22.04.03 LTS*, running the 5.15.0-91-generic

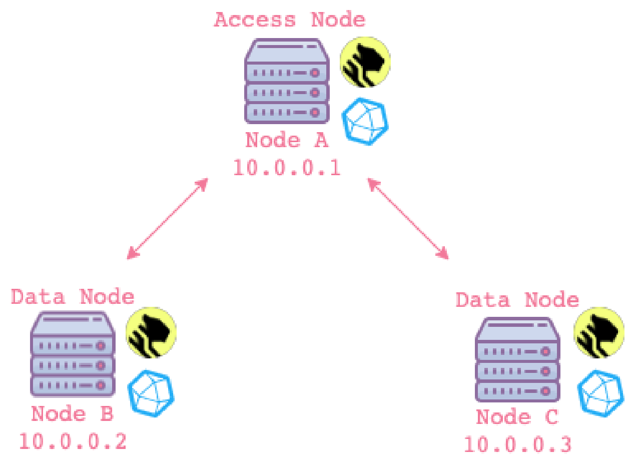


Fig. 5: Multi Node Setup.

Linux kernel. The virtual environment is equipped with an *Intel(R) Xeon(R) CPU E5-2650 v3* operating at 2.3GHz, 4GB of RAM, and a 30GB virtual boot drive. These virtual machines are connected to a virtual private network (VPN) configured through *Okeanos*. In order to establish internet connectivity for all three machines, *Network Address Translation (NAT)* was implemented on Node A, which is equipped with the public IPv4 address in this particular instance. This configuration was essential to enable license key verification for the *Influx* Enterprise data version. However, an elaboration on this aspect will be provided later in this section, specifically within the part of the multi-node *Influx* installation. *Timescale* and *Influx* will be installed on each node, and we will employ the same datasets used in the single-node experiment (small - 1GB, medium - 5GB, and large - 15GB).

B. *Influx Installation*

To implement *Influx* in cluster mode, we had to transition to the Enterprise Edition, the sole version supporting clustering. *Influx Enterprise* 1.11.3 is a commercial product, but for this project, we utilized the 14-day free trial. The *Influx Enterprise* Meta node installer was deployed in Node A, while the *Influx Enterprise* Data node installer was installed in Nodes B and C. Subsequently, we modified the configuration files of *Influx* on all three nodes to incorporate the license key obtained from *Influx Enterprise*. Following this, Node A (meta node) and Node B, Node C (data nodes) were added to the cluster, and subsequently inserted data into the retention policy. The same dataset and queries from the single-node experiment were used.

C. *Timescale Installation*

For the *Timescale* cluster, Node A was designated as the access node, while Nodes B and C were assigned as data nodes. We installed *Timescale* 2.13.1 on all three nodes, mirroring the setup from the single-node experiment. Subsequently, we adjusted the configuration files of *PostgreSQL* for all three nodes to establish cluster connectivity. Further modifications were made to the configuration of Node A to function as the access node and the configurations of Nodes B and C to operate as data nodes. Following these adjustments, we initiated the creation of the databases Node A, specifying Nodes B and C as the data nodes that would host the data of each database. Subsequently, we populated the hypertables with data using the same queries employed in the single-node experiment.

D. *TSBS Installation*

It is necessary to install *TSBS* on Node A to facilitate the execution of *TSBS* scripts for both loading data into the databases and executing queries. The installation procedure mirrors the one elucidated earlier in the single-node setup section.

X. Multi Node Insert Performance

To evaluate the insertion performance of each database cluster, we utilized bash scripts provided by the *Time Series Benchmark Suite (TSBS)*, following a similar approach as in the single node comparison. Notably, we modified the `--do-create-db` parameter to false in the `load_timescale.sh` script, facilitating the loading of data into pre-configured and empty *Timescale* databases with a clustered setup. Additionally we adjust the `--urls` parameter in the `load_influx.sh` bash script to include the URLs of both data nodes.

Subsequently, we scrutinized the output files generated during the process and constructed performance graphs. A *Python* script was employed to compare the ingestion rates in terms of rows per second and metrics per second for each database at various sizes. The resulting visual representations are presented in Fig. 6.

In terms of rows rate, *Influx* demonstrates a peak performance for medium dataset size, reaching 135.08K rows/sec. In comparison, *Timescale* exhibits a noticeable reduction at 96.34K rows/sec for the same dataset, reflecting a substantial decrease of 28%. In scenarios involving smaller datasets, the disparity between the two databases diminishes, with *Timescale* achieving a rate of 82.86K metrics/sec and *Influx* registering 80.63K metrics/sec. In the context of larger datasets, *Timescale* has a rate of 101.1K rows/sec while *Influx* has a rate of 124.85K rows/sec making it 19% faster for the corresponding dataset.

Regarding the metric rate, *Influx* consistently surpasses *Timescale*, exhibiting a substantial increment across all datasets. In particular, *Influx* attains its maximum at 1067.98K metrics/sec for the medium dataset, while *Timescale* peaks out at 505.48K metrics/sec for the large dataset.

At this point, it's reasonable to assert that the insertion speeds of both *Influx* and *Timescale* experienced significant enhancements, measured in both rows per second and metrics per second, with the transition to a 3-node cluster. This notable increase in performance metrics stands out when compared to the insertion speeds observed in the single-node configuration.

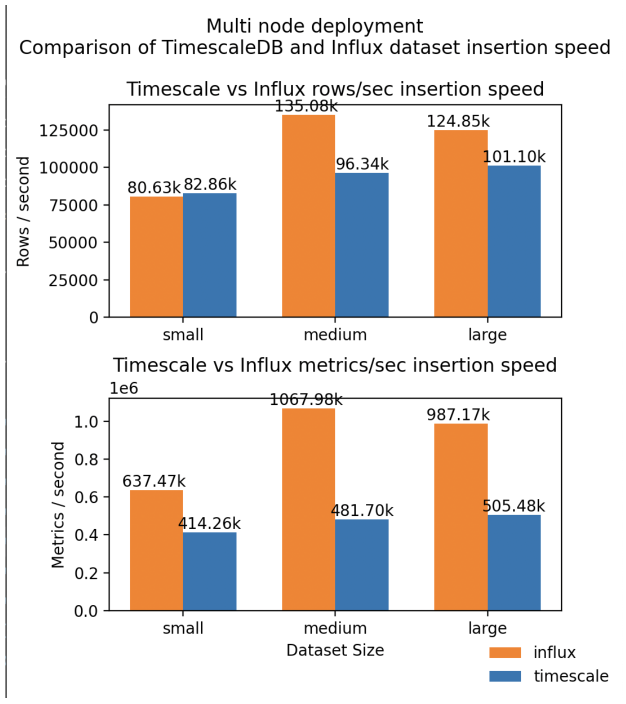


Fig. 6: Insertion performance on multi node deployment.

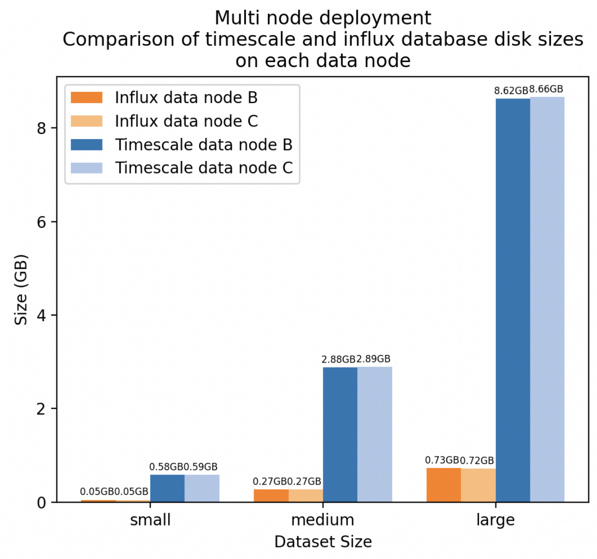


Fig. 7: Disk size comparison on multi node deployment.

This phenomenon can be explained by the fact that the insertion of a dataset across multiple nodes concurrently distributes the inherently slow input/output (IO) write operations, thus enhancing the overall operational efficiency. The associated overhead for write synchronization is comparatively lower than the expense incurred by waiting for IO write operations on a singular machine.

XI. Multi Node Size on Disk

After the successful completion of data insertion into the respective database clusters and subsequent result collection, our investigation progressed to a comparative evaluation of the stored database files on individual data nodes.

The storage mechanism driving our *Influx* cluster is the *Influx Time-Structured Merge Tree (TSM)*, elucidated in the preceding single-node comparison.

The graphical representation in Fig. 7 illustrates the distribution of database sizes across the two data nodes. Reiterating our observations, we note a noteworthy reduction in the cumulative database size in *Influx* compared to *Timescale*, attributable to *Influx's* inherent compression. Additionally, a uniform distribution of the database size across the two data nodes is evident.

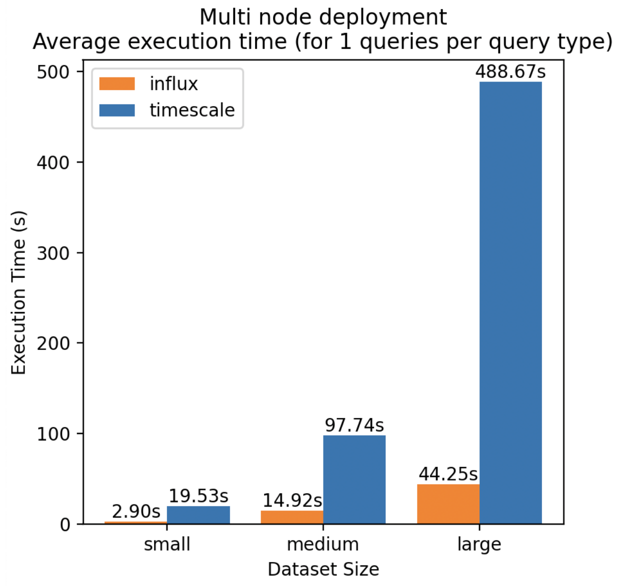
Similarly, within the *Timescale* environment, the database size exhibits an equitable distribution across the two data nodes. Nevertheless, the total database size is substantially larger, primarily attributed to *Timescale's* default absence of data compression, as underscored in our earlier single-node comparison.

A noteworthy observation is the comparison between the cumulative disk size of data nodes and the disk size occupied by a database when operating on a single machine. In the case of *Influx*, this comparison reveals a marginal advantage in favor of a multi-node setup, where the cumulative disk size is slightly less than that of a single-node configuration. Conversely, *Timescale* appears to consume more disk space for data when distributed across multiple data nodes. Despite the relatively slight disparities in cumulative sizes compared to the single-node configuration, on the order of tens of megabytes for our dataset, this constitutes a significant metric in evaluating the efficiency of data storage in multi node set up between the two databases, namely *Influx* and *Timescale*.

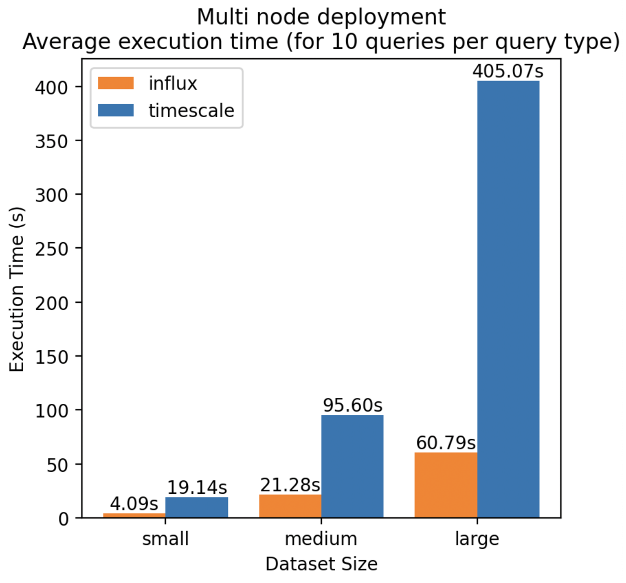
XII. Multi Node Average Query Performance

To assess query performance across individual database clusters, we systematically collect data and generate bar graphs representing two scenarios, mirroring the single-node testing approach. The first scenario involves a single query per query type, with the average execution time recorded. In the second scenario, ten consecutive queries per query type are executed. The resulting graphs visually depicts the mean execution time for all query types across three distinct datasets (small, medium, large) in both scenarios.

In the tests involving ten consecutive queries per query type, *Influx* cluster consistently demonstrates significantly better query time execution on average. The performance gap between the two databases widens notably for the large dataset. Specifically, the *Influx* cluster surpasses the *Timescale* cluster by 4.67 times for the small dataset, 4.5 times for the medium dataset, and 6.6 times for the large dataset. However, it's important to note that this average may not accurately represent overall performance, as it is influenced largely by three specific query types (high-load, last-loc, and low-fuel) where *Timescale* exhibits significantly higher execution times. Hence, a comprehensive analysis evaluating the effectiveness of databases based on query types will be presented towards



(a) Average execution time (1 query) on multi node deployment.



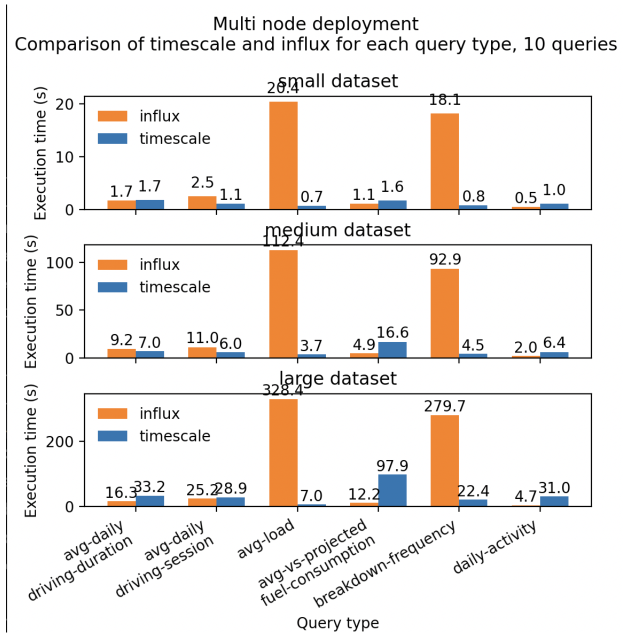
(b) Average execution time (10 query) on multi node deployment.

Fig. 8: Average query execution for both scenarios.

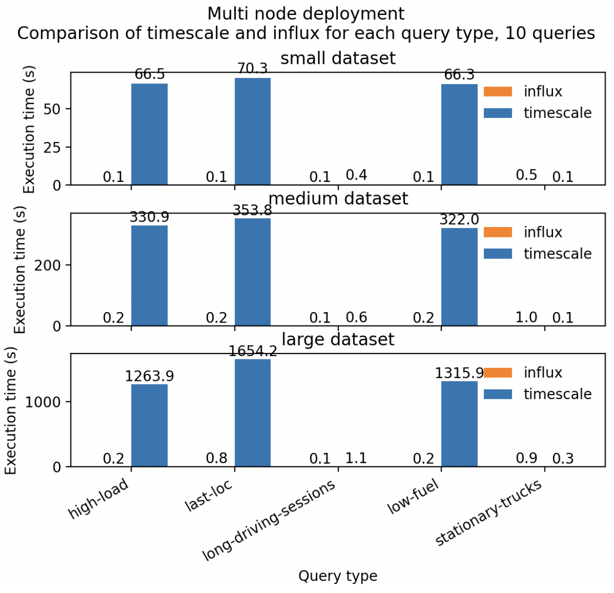
the end of this section, providing a concise summary of the performance associated with each query type. Similarly to the single node testing, graphs show that for large datasets, *Influx* outperforms *Timescale*, diverging from public benchmarks. Previous studies suggest *Influx* excels in smaller databases, but hardware limitations may impede scaling up data for experiments. However, our analysis should prioritize insights from our experiments.

XIII. Multi Node Query Performance per Query Type

The visual representations generated from the datasets explicate that, similarly to the single-node deployment scenario, a conspicuous trend is observed in which the speed of query execution gradually decelerates as the datasets expand in size. This pattern remains consistent in the more extensive datasets, mirroring the characteristics observed in the single-node deployment. The aforementioned observations persist as well in the extremes of the dataset spectrum within this specific topology.



(a) Average execution time per query type on multi node deployment (Queries 1-6).



(b) Average execution time per query type on multi node deployment (Queries 7-11).

Fig. 9: Average query execution per query type (multi node).

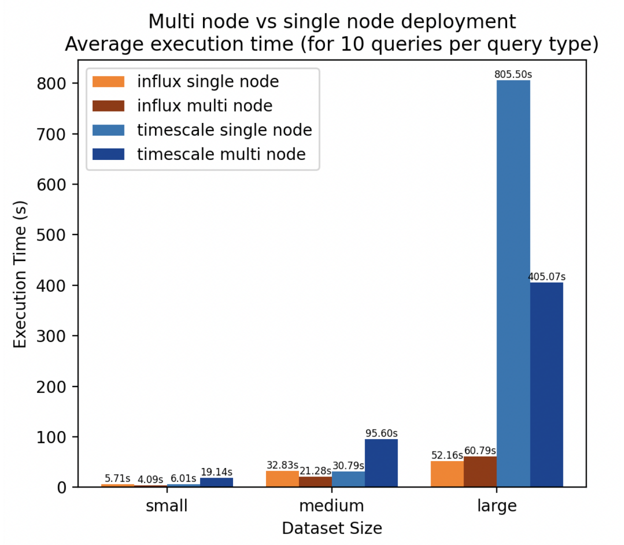


Fig. 10: Single Node vs Multi Node Avg Query Time.

XIV. Single Vs Multi Node

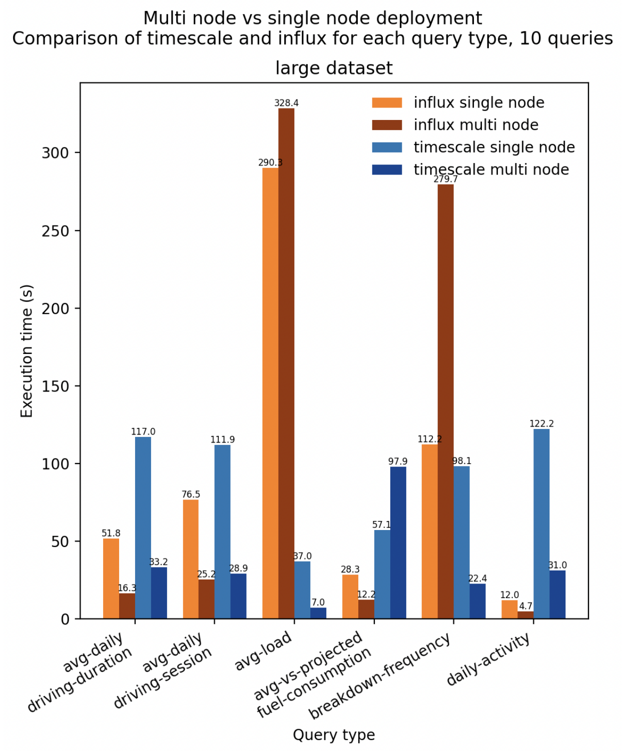
A. Single Vs Multi node average execution time

In the graphical representation depicting the average query time, the transition from single to multi-node deployment in Influx reveals an enhancement in average performance for small and medium datasets, albeit a deterioration in performance for the large dataset. The observed performance improvement can be attributed to the parallel execution across the two nodes. Conversely, the decline in performance for the large dataset may be due to queries where the synchronization overhead across nodes outweighs the benefits of parallelization, a phenomenon exacerbated with larger datasets.

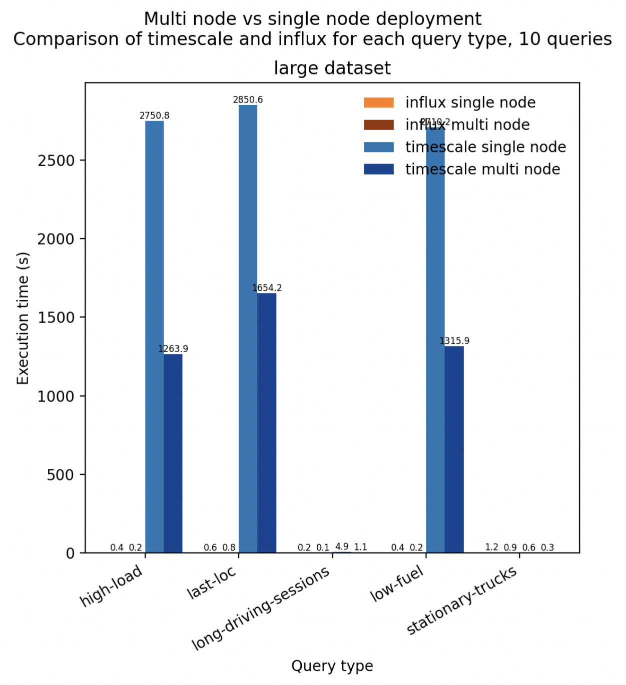
Conversely, Timescale exhibits, on average, an inverse behavior, with performance degradation observed for small and medium datasets and a substantial increase (doubling) for the large dataset. This pattern could potentially be attributed to a distinct approach in query execution and parallelization adopted by Timescale, altering the dynamic between parallelization and synchronization overhead to favor larger datasets. Notably, as further elaborated upon later, certain instances of notably poor Timescale performance exhibit a significantly greater scaling factor when executed on multiple nodes.

B. Single Vs Multi node execution time per query

In the presented graphs (Fig. 11), the disparity per query type is evident for both *Influx* and *Timescale*, as discussed previously. Of greater significance is the observation of how these differences evolve with the transition from a single to a multi-node deployment. In the case of *Influx*, there is an overall performance improvement across most queries, with some experiencing performance gains of up to threefold. Conversely, the two queries that exhibited suboptimal performance in the single-node setup demonstrated an even more pronounced decline in the multi-node configuration. This, coupled with



(a) Single Node vs Multi Node Query Time per Query Type (Query 1-6).



(b) Single Node vs Multi Node Query Time per Query Type (Query 7-11).

Fig. 11: Single Node Vs Multi Node.

their already elevated execution times among *Influx* queries, contributed to the performance degradation observed in the large dataset within the average query time execution metric.

Contrastingly, *Timescale* exhibited a remarkable and unexpected improvement in performance across all queries, with only one exception. Execution times were either i nthe worst case halved or even reached peaks of 5.2 times faster (in the case of avg-load). Particularly surprising was the twofold performance gain observed in queries that had previously performed inadequately on *Timescale* (high-load, last-loc, and low-fuel). These findings support the proposition that the prolonged execution times for these queries result from the necessity for *Timescale* to traverse the entire dataset. When this operation is distributed across two data nodes (assuming processing occurs at the data nodes rather than the access node, a plausible scenario to avoid extensive data movement across the network), an anticipated acceleration in performance is expected. Consequently, this elucidates the rationale behind the discernibly lower average query time observed on *Timescale* for the large dataset.

XV. Summary

In this paper, our objective is to conduct a comparative analysis of *Influx* and *Timescale*, focusing on insertion speeds, disk size, and query speeds in both single node and multi-node deployments using IoT data. Our findings reveal that *Influx* and *Timescale* exhibit comparable insertion speeds, while *Influx* outperforms in average query speeds. However, it is important to note that this discrepancy is partially attributed to three specific queries that exhibit significantly longer execution times in *Timescale*. This observation diverges from conclusions drawn in publicly available benchmarks, where existing research suggests that *Influx* excels in smaller databases, experiencing performance degradation as data sizes increase, with *Timescale* outperforming *Influx* in such scenarios. It is crucial to acknowledge the limitations imposed by our hardware resources, encompassing CPU, memory, and disk storage, constrain our capacity to scale up data and conduct experiments as the dataset size expands significantly.

It is imperative to underscore that *Timescale* additionally incorporates support for columnar compression, a feature that holds the potential to diminish the database size by as much as 90%. Moreover, this compression mechanism has the ancillary effect of expediting queries, particularly those of an analytical nature executed over substantial datasets, utilizing the inherent advantages of a columnar data structure [8].

For an exhaustive, step-by-step guide, inclusive of specific commands utilized in the installation processes, we direct readers to consult our GitHub Repository [2]. We encourage exploration of the repository for supplementary insights, documentation, and updates pertaining to the setup procedures. This resource is crafted to facilitate a seamless and transparent replication of the outlined processes, thereby enhancing comprehension and ensuring accurate implementation.

References

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

[2] https://github.com/EEMplekei/TimeseriesDB_Benchmarks

[3] https://docs.influxdata.com/influxdb/v1/introduction/install/

[4] https://docs.timescale.com/self-hosted/latest/install/installation-linux

[5] https://influxdata.com/blog/improved-data-ingest-compression-influxdb-3-0/

[6] https://docs.influxdata.com/influxdb/v1/concepts/storage_engine/

[7] https://medium.com/timescale/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877

[8] https://www.timescale.com/learn/reasons-to-choose-timescale-as-your-influxdb-alternative