Université libre de Bruxelles

Advanced Databases

Winter Semester 2017-2018

Time Series Databases and InfluxDB

Authors:
Syeda Noor Zehra Naqvi
(000455274)
Sofia Yfantidou
(000456361)

Supervisor:
Dr. Esteban Zimányi

December 17, 2017

Contents

1 TIME SERIES & TIME SERIES DBs

1.1 Time Series

1.1.1 Definition

"Time Series is an ordered sequence of values of a variable (e.g.temperature) at equally spaced time intervals (e.g. hourly)." Thus it is a sequence of discrete-time data. For instance timestamped data, such as log files and IoT devices' measurements can be considered time series. The measurements that constitute a time series are ordered on a timeline, which reveals information about underlying patterns. Ordering matters, because there is a dependency between time and measurements and changing the order could change the meaning of the data [4]. Example time series would be the hourly measurements of temperature at a specific weather station, daily measurements of the closing price of a specific stock, etc.

1.1.2 Uses

Time series are used in various context, the most common of them being[1]:

- **Time Series Analysis:** Time Series Analysis is utilized in order to explore how a given variable changes over time. For instance Census Analysis, namely public opinion analysis on a specific matter over time, e.g. presidential candidates in U.S. elections, can be considered a Time Series Analysis task.

- **Regression Analysis:** Regression Analysis can be utilized to examine how the changes associated with a specific variable can cause shifts in other variables over the same time period. For instance, Stock Market Analysis, namely how one stock's prices over time affect other stocks' prices or unemployment over the same time period, can be considered a Regression Analysis task.

- **Time Series Forecasting:** Time Series Forecasting uses information regarding historical values and associated patterns to predict future activity. For example, Economic Forecasting, Weather Forecasting, Earthquake Forecasting (seismic time series), Sales Forecasting, etc.

[1]Time Series - Investopedia. 2017. Retrieved from `http://www.investopedia.com/terms/t/timeseries.asp`.

1.2 Time Series Databases

1.2.1 Definition

A Time Series Database (TSDB) is a database type which is optimized for time series or time-stamped data. It is built specifically for handling metrics, events or measurements that are time-stamped. A TSDB is optimized for measuring change over time. A TSDB allows its users to create, enumerate, update, destroy and organize various time series in a more efficient manner. The key difference with time series data from regular data is that mostly you ask questions about it over time. Nowadays, the majority of the companies are generating a insanely large stream of metrics and events (time series data) and hence the need of a TSDBs is unavoidable.

1.2.2 Properties

The main properties distinguishing time series data from the regular data workloads are summarization, data life cycle management, and large range scans of many records. The overview of some of the required properties of a TSDB is as follows:

- **Data Location:** If related data is not located together in the physical storage, the data queries can be really slow and even result in timeouts because non-sequential I/O operations are still very slow as compared to the sequential I/O even when using SSD. A TSDB co-locates chucks of data within the same time range on the same physical part of the database cluster and hence enables quick access for faster, more efficient analysis.

- **Fast, easy range queries:** As a TSDB keeps the co-related data together it ensures that the range queries are fast. In many cases regular databases produce an index out of memory error because of the sheer volume of time series data and subsequently affect the performance of read and write operations. In addition, it should be taken into consideration that the query language used should make it easier for users to write such queries.

- **High write performance:** A lot of databases are not able to serve requests predictably and quickly during peak loads. TSDBs should ensure high availability and high performance for both read and write

operations during peak loads because they are usually designed to stay available even under the most demanding conditions. Time series data is usually being recorded every second or even less than that, so write operations need to be fast.

- **Data compression:** As time-series data is mostly recorded per second or even with less granularity, they usually need a better data compression technique. And as the data grows older granularity becomes less important, so TSDBs should provide functionality to perform roll-ups in such scenarios for data compaction.

- **Scalability:** Time-series data increases very quickly. For example a connected car will send 25 GB of data to the cloud every hour[2]. And regular databases are not designed to handle this scalability. On the other hand time series databases are designed to take care of scale by introducing functionalities that are only possible when you treat time as your first concern. This can result in performance improvements, including: higher insertion rates, faster queries at scale, and better data compression.

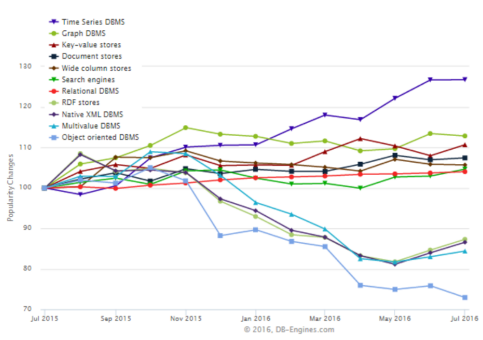
- **Usability:** TSDBs typically include functions and operations that are common to time series data analysis. For example they utilize data retention policies, continuous queries, flexible time aggregations, range queries etc. So this increases the usability by improving the user experience in case of dealing with time related analysis.

1.2.3 Popularity

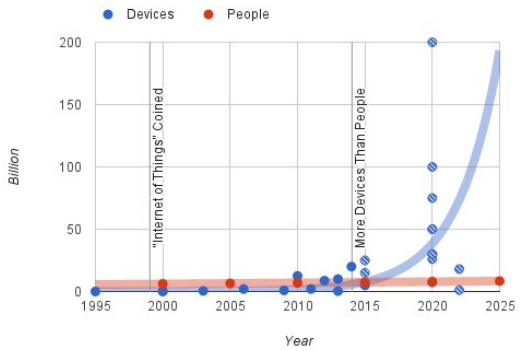
It is obvious that TSDBs are to handle time series data, but their popularity seems to have increased with the emergence of Internet of things (IoT). IoT is a network of physical devices/objects with connectivity which enables them to exchange and collect data. Such technologies are generating large amount of data which is usually time-stamped, so with the increase in popularity of IoT, TSDBs popularity increased even more, because they can be used to efficiently store sensors and devices' data in this domain. Some other common uses of TSDBs are DevOps monitoring and real time data analysis. Nowadays, many large companies like Facebook, eBay etc. are using

[2]Quartz Media. More details here: `https://qz.com/344466/connected-cars-will-send-25-gigabytes-of-data-to-the-cloud-every-hour/`.

TSDBs instead of relational databases especially for data monitoring purposes. From the graph in Figure 1a it can be seen that the popularity of TSDBs is increasing rapidly in the past couple of years. From 2015 to 2016 the popularity of TSDBs in increased by 26.7% which is twice as much as Graph database management systems which is 2nd in the list. The popularity of TSDBs continues to increase till now. In Figure 1b we visualize the increase in popularity of IoT starting from 2015 as well, to give an idea of the simultaneous grow of the two fields.



(a) DB-Engines Ranking, Popularity Trend (Source: https://www.db-engines.com).



(b) Sketching out the Internet of Things trendline, Brookings (Source: https://www.brookings.edu/).

Figure 1: Popularity Charts for TSDBs and IoT.

1.2.4 Benefits and Uses

As mentioned earlier Section 1.2.3, some common applications of TSDBs are IoT, DevOps, Data Analytics etc. But the question is that why do we prefer TSDBs over normal databases in such applications? Some additional advantages of using TSDBs are as follows:[3]

- **Massive scalability and performance:** It can be predicted from the TSDB properties described in Section 1.2.3 that an efficient and good TSDB allows an application to scale easily to support millions of

[3]Benefits of a TSDB. Retrieved from`http://basho.com/resources/time-series-databases/`.

IoT devices or time series data points in a continuous flow and perform real-time analysis.

- **Reduced downtime:** In real life scenarios there are some situations where availability is critical at all times, the architecture of a database that is built for time series data avoids any downtime for data even in the event of network partitions or hardware failures.

- **Lower costs:** Flexibility and high threshold to failure translates into fewer resources needed to manage outages. Commodity hardware used for fast and easy scaling ensures the reduction of operational and hardware costs of scaling up or down.

- **Improved business decisions:** As TSDB enables an organization to monitor and analyze data in real time, it helps it in making faster and more accurate adjustments for infrastructure changes, consumption of energy, device maintenance or other major decisions that influence the business.

Some use cases for TSDBs includes monitoring software systems like virtual machines, different services or applications, monitoring physical systems for example some equipment or machines, connected devices, the environment, home management systems, human bodies etc. Another use of TSDBs is in financial trading systems for classic securities or in crypto currencies (bit coins etc). TSDBs can also be used as eventing applications for tracking user/customer interaction data and in business intelligence tools for tracking key metrics and the general health of the business.

1.2.5 Top Time Series Databases

Top few TSDBs and their ranking can be seen in the Figure 2 according to DB-Engines Ranking of Time Series DBMS. DB-Engines [1] is an independent website which ranks databases based on search engine popularity, social media mentions, number of job offers, and technical discussion frequency. Influx DB is ranked number one in this list as of October 2017. Figure 2 also shows the historical changes of these databases and it can be seen that InfluxDB was also the top TSDB in October 2016 and it maintained it's ranking. In the following Section 2 we will discuss InfluxDB in detail. The second on the list is RRDtool which is an open source, industry standard

data logging and graphing tool for time series data. It's supported programming languages are C, C#, Java, JavaScript, Lua, Perl, PHP, Python, Ruby. Graphite is ranked third in the list which is also an open source data logging and graphing tool which is implemented in Python and supports JavaScript and Python programming languages. Both RRDtool and Graphite deals with numeric values.

Rank			DBMS	Database Model	Score		
Oct 2017	Sep 2017	Oct 2016			Oct 2017	Sep 2017	Oct 2016
1.	1.	1.	InfluxDB	Time Series DBMS	8.70	+0.22	+3.38
2.	2.	2.	RRDtool	Time Series DBMS	3.12	+0.06	+0.64
3.	3.	3.	Graphite	Time Series DBMS	2.76	+0.21	+0.86
4.	4.	4.	OpenTSDB	Time Series DBMS	1.86	-0.03	+0.39
5.	5.	5.	Kdb+	Multi-model	1.83	+0.06	+0.62
6.	6.	6.	Druid	Time Series DBMS	1.00	+0.02	+0.40
7.	7.	7.	Prometheus	Time Series DBMS	0.74	+0.07	+0.46
8.	8.	8.	KairosDB	Time Series DBMS	0.48	-0.02	+0.21
9.	9.	↑ 10.	eXtremeDB	Multi-model	0.32	-0.02	+0.13
10.	10.	↓ 9.	Riak TS	Time Series DBMS	0.24	-0.02	+0.02
11.	11.	↑ 17.	Axibase	Time Series DBMS	0.18	-0.07	+0.18
12.	12.	↑ 19.	Hawkular Metrics	Time Series DBMS	0.15	+0.02	+0.15
13.	13.	↑ 18.	Blueflood	Time Series DBMS	0.14	+0.03	+0.14
14.	↑ 16.	↑ 15.	Machbase	Time Series DBMS	0.08	+0.03	+0.06
15.	↓ 14.	↓ 13.	Warp 10	Time Series DBMS	0.06	-0.05	+0.01
16.	↓ 15.	16.	TempoIQ	Time Series DBMS	0.04	-0.02	+0.03
17.	17.	↓ 12.	Heroic	Time Series DBMS	0.00	-0.01	-0.05
17.	↑ 18.	↓ 14.	Newts	Time Series DBMS	0.00	±0.00	-0.03
17.	↑ 18.		SiriDB	Time Series DBMS	0.00	±0.00	
17.	↑ 18.	↑ 19.	SiteWhere	Time Series DBMS	0.00	±0.00	±0.00
17.	↑ 18.	↓ 11.	Yanza	Time Series DBMS	0.00	±0.00	-0.06

Figure 2: Top 21 Time Series Databases and their historical changes as of October 2017 (Source: https://db-engines.com/en/ranking/time+series+dbms).

2 INFLUXDB

2.1 General Information & Architecture

InfluxDB is an **open-source** schemaless time series database with optional closed-sourced components developed by InfluxData. It is written in Go programming language and it is optimized to handle time series data as defined in Section 1.1.1. It porvides an SQL-like query language. The open-source

version, namely the *TICK Stack* (See Image 3), provides a full time series database platform with various services including the InfluxDB core and can run on cloud and on premises on a single node. The closed-source versions, namely *InfluxEnterprise (IE)* and *InfluxCloud (IC)*, offer extra functionalities, such as high availability, scalability, backup and restore, and run either on premises (IE) or on cloud (IC). More details on the suitability of each version for specific use cases will be given in Section 2.3.

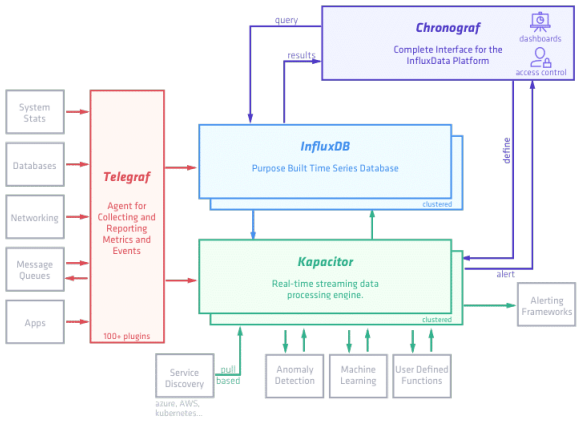


Figure 3: The open-source, Time Series Database Platform, TICK stack (Source: https://www.influxdata.com/time-series-platform/).

2.1.1 Key Concepts

Before we dive deeper into InfluxDB some key concepts should be defined. In order to illustrate these concepts in an easy way a simple real-life-like example will be utilized (See Table 1). This example is based on a similar one presented in InfluxDB's documentation [3]. It shows the number of minor and adult passengers picked up by taxi drivers Doe and Jones at locations 1 and 2 between 18/08/2015 at 00:00 and 18/08/2015 at 06:30.

First of all, the most important concept in InfluxDB is **time**. The time column is included in every InfluxDB database and stores discrete timestamps, which are associated with specific data. The next two columns, minors and adults, are **fields** (another term for attributes). Each field consists of a **field key** e.g. minors and a **field value** e.g. 1. Field values are the

name=passengers				
time	**minors**	**adults**	**location**	**driver**
2015-08-18T00:00:00Z	1	2	1	doe
2015-08-18T00:00:00Z	2	2	1	jones
2015-08-18T00:06:00Z	1	1	1	doe
2015-08-18T00:06:00Z	0	1	1	jones
2015-08-18T05:54:00Z	0	2	2	doe
2015-08-18T06:30:00Z	2	2	2	doe
2015-08-18T06:06:00Z	3	1	2	jones
2015-08-18T06:30:00Z	0	4	2	jones

Table 1: Sample time series dataset.

actual data and are always associated with a timestamp. Each field key-field value pair is a **field set**. Our dataset has 8 field sets:

- minors=1, adults=2
- minors=2, adults=2
- minors=1, adults=1
- minors=0, adults=1
- minors=0, adults=2
- minors=2, adults=2
- minors=3, adults=1
- minors=0, adults=4

The location and driver columns are called **tags**. Again. each tag consists of a **tag key** and a **tag value**. Each tag key-tag value pair constitutes a **tag set**. The following 8 tag sets are included in the dataset:

- location=1, driver=doe
- location=1, driver=jones
- location=1, driver=doe
- location=1, driver=jones
- location=2, driver=doe
- location=2, driver=doe
- location=2, driver=jones
- location=2, driver=jones

The difference between tags and fields is that tags are indexed, which means than queries on tags are faster compared to queries on simple fields, which are not indexed. Note that the primary key consists of the timestamp

Series Number	**Retention Policy**	**Measurement**	**Tag set**
Series 1	autogen	passengers	location=1, driver=doe
Series 2	autogen	passengers	location=1, driver=jones
Series 3	autogen	passengers	location=2, driver=doe
Series 4	autogen	passengers	location=2, driver=jones

Table 2: The series that constitute our dataset.

and the tags. Another important concept is **measurement** (think of it as an SQL table). The measurement technically explains our fields' content. In other words the minors and adults columns in our table contain the number of passengers (See name in the 1^{st} row of Table 1) and not their years, height, etc. A single measurement can belong to different retention policies. A **retention policy** describes how long InfluxDB stores data (DURATION) and how many copies of those data are stored in the cluster. The default retention policy with infinite duration and no replication is called **autogen**, which has an infinite duration and a replication factor on 1. Maybe the most important concept in InfluxDB is **series**, which is a collection of data with common retention policy, measurement and tag set. A **point** is is the field set in the same series with a specific timestamp e.g.*time=2015-08-18T00:00:00Z, minors=1, adults=2, location=1, driver=doe*. Our dataset consists of 4 series as shown in Table 2.

2.1.2 Sharding

Sharding is the horizontal partitioning of data in a database. Each partition is called shard. InfluxDB stores data in shard groups, which are organized by retention policy and store data with timestamps that fall within a specific time interval. The length of the aforementioned time interval depends on the duration of the retention policy (RP). The default shard group durations are 1 hour for RP less than 2 days, 1 day for RP between 2 days and 6 months and 7 days for RP greater than 6 months. The duration of the shard group is important for efficient drop operations, where data is dropped per shard, not per data point. For instance if a RP has a duration of 10 hours, it makes no sense to divide the data in 5-hour intervals. However, short shard group durations for large RP can harm compression and speed. Recommendation for appropriate sharding and schema design can be found here.

2.1.3 Storage Engine

InfluxDB currently uses its in-house built data structure, the Time Structured Merge Tree (TSM Tree). More details on this storage format will be given shortly. However, InfluxDB has utilized various storage formats over different versions. Initially it used *LevelDB* (a database based on Log Structured Merge Trees (LSM)), which optimizes write throughput and offers built-in compression. However, LevelDB does not provide hot backup functionality, which means you need to close the database to safely copy it. For this reason InfluxDB utilized LevelDB variants, such as RocksDB and HyperLevelDB, which also use LSM Trees. There is an inherent problem with LSM Trees though, deletion is an expensive operation and a time series DB requires deletions on a large scale due to automatic data retention (See Section 2.1.1). That's why InfluxDB switched to an alternative data structure, the mmap B+Tree. It used BoltDB as the underlying storage engine, which may perform slightly worse in write operations, but offers increased stability and reliability. However, they realized that when the database became larger, writes would start spiking Input Output Operations per Second (IOPS). Subsequently, the InfluxDB team decided to build their own storage format, the TSM Tree. The TSM Tree is similar to a LSM Tree in a sense that it uses write ahead log, index files that are read only, and it occasionally performs compactions to combine index files. However, it does not suffer by the deletion problem and offers better compression rates (45x improvement in disk space usage) compared to a B+ Tree[4].

2.2 Customers & Use Cases

InfluxDB has more than 70000 active installs and is preferred by customers for **DevOps Monitoring** (Infrastructure Monitoring, Application Monitoring, Cloud Monitoring), **IoT Monitoring**, and **Real-Time Analytics**. Its customers include, but are not limited to, AXA, Cisco, eBay, IBM and more. Customers' success stories per category can be found below.

[4]The New InfluxDB Storage Engine: Time Structured Merge Tree — InfluxData. 2017. Retrieved from `https://www.influxdata.com/blog/new-storage-engine-time-structured-merge-tree/`.

2.2.1 DevOps Monitoring: The IBM Case

"The IBM® Trusteer® products help detect and prevent the full range of attack vectors responsible for the majority of online, mobile and cross-channel fraud." In order to provide full protection against online fraud at all times, the Trusteer platform needs to maintain high availability. For this purpose its team uses DevOps monitoring techniques powered by InfluxDB, Telegraf[5] (another product of the InfluxData ecosystem) and Grafana[6] (open-source software for time series analytics). They use Telgraf for collecting data, InfluxDB for storing them and Grafana for analysis and visualization. They collect data on infrastructure and application performance, in order to monitor their cloud system, which contains hundreds of virtual servers.

2.2.2 IoT Monitoring: The Spiio Case

Spiio enables monitoring vertical living green walls and high value green plant installations by providing sensors and the related software to horticulturalists. Green walls are becoming more and more popular especially in large cities, bringing nature closer to city life. However, maintaining multiple green walls in a city can be a problem for horticulture professionals; that's why Spiio uses sensors to understand plant performance from data, and thus cut maintenance cost drastically, ensuring full digital control of millions of plants.

Spiio tried adopting various solutions before InfluxDB, namely IoT platforms, such as AWS Greengrass and Azure IOT, multi-purpose databases or search engines, such as MySQL, Cassandra, Elasticsearch, and even time-series databases, such as OpenTSDB. However, none of these solutions was as holistic as InfluxData. Currently, Spiio uses InfluxDB for data storage, Kapacitor[7] for real-time, streaming data analytics and Chronograf[8] for data visualization. Both Kapacitor and Chronograf are part of the InfluxData ecosystem. InfluxData enables Spiio's clients to access and share never-before possible insights on optimizing green wall maintenance by tracking

[5]Telegraf from InfluxData — Agent for Collecting & Reporting Metrics & Data. 2017. Retrieved from `https://www.influxdata.com/time-series-platform/telegraf/`.

[6]Grafana. 2017. Retrieved from `https://grafana.com/`.

[7]Kapacitor from InfluxData — Real-time streaming data processing engine. 2017. Retrieved from `https://www.influxdata.com/time-series-platform/kapacitor/`.

[8]Chronograf from InfluxData — Complete Interface for the InfluxData Platform. 2017. Retrieved from `https://www.influxdata.com/time-series-platform/chronograf/`.

the impact of factors that influence plant performance.

2.2.3 Real-Time Analytics: The eBay Case

EBay Inc. is a global e-commerce leader. Various teams inside eBay utilize InfluxDB and InfluxData ecosystem in general for DevOps monitoring and real-time analytics. Here, we will focus on real-time analytics inside eBay's Experimentation team. This team is responsible for eBay's experimentation platform, which runs more than 1500 experiments[9] and enables eBay's business users to gain insight on important analytics and answer crucial business questions. However, experiments can experience anomalies, such as traffic corruption, and here is where InfluxDB comes in. Anomalies are detected daily utilizing Anomaly/Traffic Prediction algorithms and stored into InfluxDB and are visualized in Grafana. Having this analyzed data represented in time series format is key and allows them to present it in their Grafana dashboard. This combination enables the Experimentation platform to be scalable and self-sufficient, namely by creating new dashboards and datasets automatically.

2.3 Pros & Cons

2.3.1 Pros

1. ***Tailored-made for Time Series data:*** InfluxDB is designed to handle time series data more efficiently. It is designed to have impressive write and read throughputs as will be discussed in Sections 2.5 and 3.4.

2. ***Solutions for Every Need:*** InfluxData provides an **Open-Source** core that includes InfluxDB, Kapacitor, Telegraf and Chronograf (the TICK Stack) free of charge. It also provides a **SaaS solution**, namely the InfluxCloud, that offers high availability, scalability and advanced backup and restore functionalities for users with **bigger needs and limited infrastructure**. InfluxCloud deploys servers in the U.S.A., Canada and Europe. There is also an **in-house solution** of InfluxCloud, namely InfluxEnterprise, for customers who want to utilize their

[9]Monitoring Anomalies in the Experimentation Platform. 2017. Retrieved from `http://www.ebaytechblog.com/2016/10/06/monitoring-anomalies-in-the-experimentation-platform/`.

own infrastructure or cloud services. In other words, InfluxDB offers various solutions to match every potential business need.

3. ***Holistic Solution:*** InfluxDB is designed to work perfectly along with the rest of the InfluxData ecosystem, namely Kapacitor, Telegraf and Chronograf. **In this sense it is so much more than a simple database**. It is part of a holistic solution that offers accumulation, analysis and visualization, all in one package.

4. ***Various Input Plugins:*** InfluxDB does not limit itself to one or two input methods, like other TSDBs, but it offers various input plugins free of charge. Apart from the **HTTP API**, it offers a **UDP plugin**, **Graphite plugin**, which allows input in the Graphite line protocol format, **CollectD plugin**, which allows input in collectd native format, **OpenTSDB plugin**, which allows **Telnet** and **HTTP** OpenTSDB protocol. This means that InfluxDB can act as a drop-in replacement for an OpenTSDB system.

5. ***Grafana Support:*** Grafana is the go-to software for time series analytics, with well over 100,000 active installations. Grafana has introduced a plugin for InfluxDB as a data source for their analytics dashboards.

6. ***Extensive Programming Languages Support:*** InfluxDB offers support for various programming languages, including, but not limited to: .Net, Java, Perl, PHP, Python, R, Ruby, Scala and more.

7. ***SQL-like Query Language:*** InfluxDB comes with an SQL-like query language, InfluxQL, which means it does not have a steep learning curve and is easier to write and understand by non-tech people as well compared for instance to OpenTSDB which does not provide such query language. This is extremely important when it comes to the world of businesses, where data plays a major role and is handled by people with different backgrounds.

8. ***Continuous Query Support:*** "Continuous Queries (CQ) are InfluxQL queries that run automatically and periodically on realtime data and store query results in a specified measurement."[10] CQs enable

[10]InfluxData — Documentation — Continuous Queries. 2017. Retrieved from `https://docs.influxdata.com/influxdb/v1.2/query_language/continuous_queries/`.

downsampling (roll-up) of commonly-queried, high granularity data to a lower granularity. Queries on data with lower granularity require fewer resources and are faster than queries with higher granularity.

9. ***Easy Installation:*** InfluxDB compiles into a single binary file with no dependencies, which makes it extremely easy to install and have it up and running.

10. ***Auto-Expiration:*** Time series data may become less relevant or even useless depending on the application as time goes by. InfluxDB with the use of Retention Policies as discussed in Section 2.1.1 enables automatic expiration of stale data.

11. ***Unlimited fields:*** The new storage engine of InfluxDB, TSM Tree, as discussed in Section 2.1.3, is columnar format, which means that the number of fields does not affect querying performance in a negative way. As a result the number of fields in measurement (See Section 2.1.1) do not have any limitations as well.

12. ***Built-in Web Administrator Interface:*** Like SQL, InfluxDB provides a built-in online interface for users who are not comfortable with command line interfaces and would prefer a more intuitive solution.

13. ***Extensive Documentation:*** InfluxData provides an extensive documentation guide for InfluxDB from installation to complex queries and schema optimization.

2.3.2 Cons

1. ***Scalability as a Close-Source Feature:*** When InfluxDB announced that clustering would not be included in the open-source version, it received an outcry from the community. Currently, InfluxDB high availability and scalability features are close-source. However, InfluxData provides an open-source replication solution for high availability, while many users use sharding (See Section 2.1.2) as a work-around for the missing clustering functionality in the open-source version. This for instance may make InfluxDB look unattractive for start-ups with limited budget. However, InfluxData provides various plans starting from $149

a month (or $249 a month for the cloud version), a cost not prohibiting even for smaller companies that want to invest in a good scalable solution.

2. ***Community Issues:*** Given than InfluxDB is a relatively new product, its community is again relatively small compared to solutions like Cassandra. This means that a simple Google search might not return a solution for every possible problem. A However, given than InfluxDB's popularity is rising (See Section 2.4), its community is expected to grow bigger as well.

2.3.3 When not to use InfluxDB

1. InfluxDB does not allow joins, so either design your schema such that joins are not needed. If this is not the possibility and joins can not be avoided then using influxDB is not a good idea.

2. As influxDB mostly works with frequent data, you can only group time by 1 week at maximum. If there is a requirement to group by more than a week e.g by a month, it can not be done using influxDB.

3. If clustering is required but there is no budget to buy the premium version, InfluxDB is not the best option.

4. InfluxDB is not CRUD, so if a lot of updates and deletions are required for some use case, influxDB is not recommended.

2.4 Popularity

InfluxDB is currently the most popular TSDB according to DB-Engines Rankings as seen in Figure 4 with a 3.38% increase in popularity since October 2016. The rest of the TSDBs have an increase lower that 1% or even a decrease in popularity. The ranking is based on various factors including: number of related returned results in search engines, amount of interest in the system, amount of discussion in technical forums e.g. Stack Overflow, number of job offers, number of profiles in professional networks and social network presence.

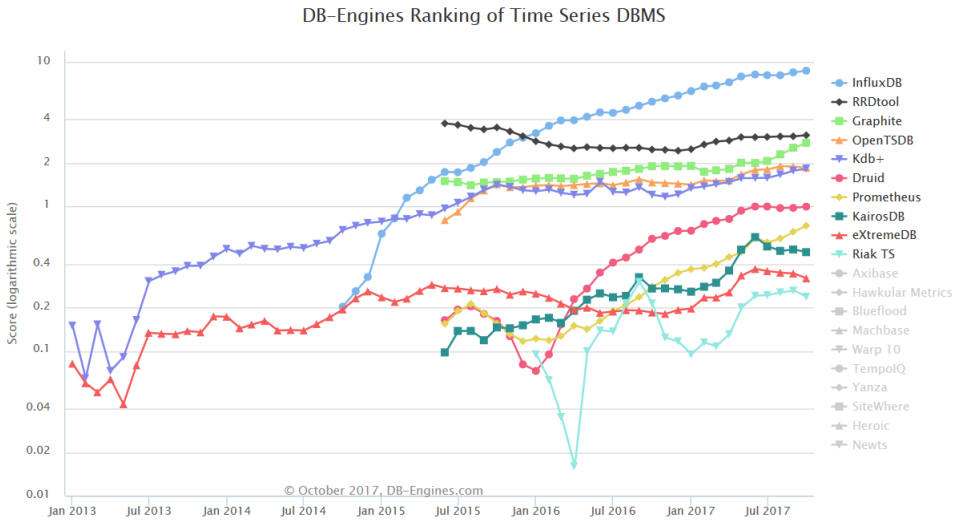


Figure 4: Ranking of the top-10 TSDBs (Source: https://db-engines.com/).

2.5 Comparisons

Figure 5 compare some characteristics of most used technologies/databases for dealing with time series data. The detailed comparison of the metrics will be done in the later Section 3.4. It can be seen that InfluxDB is released after the other competitive technologies and still among the top list. The SQL-like query language helps it make easier to use and adapt by people who are use to working with relational databases like MySQL.

3 HANDS-ON WORK

3.1 Dataset Presentation

The timestamped dataset used is provided by the NYC Taxi and Limousine Commission (TLC) and was collected by technology providers authorized under the Taxicab & Livery Passenger Enhancement Programs (TPEP/LPEP)[11]. It includes records of all the yellow cab rides from 2009 to 2017. For benchmarking purposes we utilized records from January 2016 to May 2016 (more

[11]NYC Taxi and Limousine Commission - Trip Record Data. 2017. Retrieved from `http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml`.

Name	InfluxDB	MySQL	Cassandra	ElasticSearch	OpenTSDB
Description	DBMS for storing time series, events and metrics	Widely used open source Relational DBMS	Wide-column store based on ideas of Big Table and DynamoDB	A modern search and analytics engine	Scalable Time Series DBMS based on HBase
Primary Database Model	Time Series DBMS	Relational DBMS	Wide column store	Search engine	Time Series DBMS
Initial Release	2013	1995	2008	2010	2011
Current Release	v1.3, July 2017	5.7.20, October 2017	3.11.1, October 2017	5.6.3, October 2017	2.3.0, December 2016
Implementation Language	Go	C and C++	JAVA	JAVA	JAVA
SQL	SQL-like query language	Yes	SQL-like DML and DDL statements (CQL)	No	No
In Memory Capabilities	Yes	Yes	Yes	No	No
Triggers	No	Yes	Yes	Yes	No
DATA Access APIs	HTTP Line Protocol, JSON, UDP	ADO.NET, JDBC, ODBC	JSON, CQL	JSON, Binary protocol (JAVA)	HTTP, Telnet
Schema-free	Yes	No	Yes	Yes	Yes

Figure 5: System Properties Comparison (Source: https://db-engines.com/).

than 30,000,000 records) and we removed specific attributes that were not relevant to the Time Series concepts. Finally, each record includes the following information:

- **VendorID:** A code indicating the TPEP provider that provided the record.
  1= Creative Mobile Technologies, LLC; 2= VeriFone Inc.
- **tpep_pickup_datetime:** The date and time when the meter was engaged.
- **Passenger_count:** The number of passengers in the vehicle.
- **Trip_distance:** The elapsed trip distance in miles reported by the taximeter.
- **RateCodeID:** The final rate code in effect at the end of the trip.
  1= Standard rate, 2=JFK, 3=Newark, 4=Nassau or Westchester, 5=Negotiated fare, 6=Group ride
- **Payment_type:** A numeric code signifying how the passenger paid for the trip.
  1= Credit card, 2= Cash, 3= No charge, 4= Dispute, 5= Unknown, 6= Voided trip
- **Fare_amount:** The time-and-distance fare calculated by the meter.
- **Extra:** Miscellaneous extras and surcharges. Currently, this only includes the $0.50 and $1 rush hour and overnight charges.
- **MTA_tax:** $0.50 MTA tax that is automatically triggered based on the metered rate in use.
- **Improvement_surcharge:** $0.30 improvement surcharge assessed trips at the flag drop.
- **Tip_amount:** Tip amount – This field is automatically populated for credit card tips. Cash tips are not included.
- **Tolls_amount:** Total amount of all tolls paid in trip.
- **Total_amount:** The total amount charged to passengers. Does not include cash tips.

3.2 InfluxDB Tutorial

3.2.1 Database Setup

Setting up InfluxDB on Windows is a relatively easy process. After downloading the official Windows Binaries, we immediately ran the server (influxd.exe) and then the CLI (influx.exe). InfluxDB is an out-of-the-box platform. We also downloaded Chronograf, an open-source monitoring and visualization UI for InfluxDB. Locally it can be found on port 8888 of localhost by default after running the chronograf.exe file. Chronograf enables better visualization of the results, especially the SELECT statements, as it creates graphs, tables and CSV files for query results. However, due to these extra features it is slower than a typical UI, such as the visualization tool of SQL Server Management Studio.

3.2.2 Schema Design

There are certain tips for designing the InfluxDB Schema[12]:

- Your field's unit of measurement should be reflected by the measurement. For instance if your measurement is *number_of_passengers*, the fields in this measurement cannot include the fare of the ride. As limiting as it sounds, it ensures efficient schema design for your database.
- Tags are indexed and fields are not, so store data in tags if they are commonly-queried meta data or used in $GROUP\ BY$ clauses. Store data in fields if they are used with aggregation functions.
- Avoid having too many series. In other words, avoid tags that are too varied like IDs.
- Avoid putting more than one piece of information in one tag. For instance convert $location = north - NY$ to two tags, $location = NY$ and $region = north$.
- Adjust your Shard duration to your Retention Policy (See Section 2.1.2). For longer Retention Policies, increasing the shard group duration can improve compression and write speed. A recommendation is to adjust

[12]InfluxData — Documentation — Schema Design. 2017. Retrieved from `https://docs.influxdata.com/influxdb/v1.3/concepts/schema_and_data_layout/`.

your Shard duration so that is is two times your longest typical query's time range.

- Keep in mind that InfluxDB is not designed to support joins, thus your schema should support all potential queries without the need for a join.

Taking into consideration the above mentioned tips, we designed our schema as follows:

- **Measurement:** fare (includes all the fare related fields measured in dollars), **Tags:** VendorID, RatecodeID, payment_type, **Fields:** extra, fare_amount, mta_tax, tip_amount, tolls_amount, improvement_surcharge, total_amount
- **Measurement:** passengers (includes the passenger related fields measured in number of passengers), **Tags:** VendorID, RatecodeID, payment_type, **Fields:** passenger_count
- **Measurement:** distance (includes the trip distance related fields measured in miles), **Tags:** VendorID, RatecodeID, payment_type, **Fields:** trip_distance

For benchmarking purposes we utilized only one Retention Policy, the default autogen, as well as the default shard duration. Moreover, given the specified measurements, tags and RP we ended up with 171 series (Series 1: distance, RatecodeID=1, VendorID=1, payment_type=1 and Series 2: distance, RatecodeID=1, VendorID=1, payment_type=2 etc.).

3.2.3 Data Import

There are various ways to import data to InfluxDB, including the CLI, client libraries and plugins, as well as the built-in HTTP API. It should be noted that Time Series data is usually imported into the database in real-time e.g. sensor data, log files data, so InfluxDB is designed to best handle these real-life scenarios. Thus file support is limited to files following the line protocol syntax[13]. CSV files should first be converted to line protocol format and then get imported into the database.

[13]InfluxData — Documentation — Line Protocol Tutorial. 2017. Retrieved from `https://docs.influxdata.com/influxdb/v1.3/write_protocols/line_protocol_tutorial/`.

To this end we tried various open-source converters (csv2influx,csv2influxdb, csv-to-influxdb, etc.), that convert CSV files to an InfluxDB acceptable format. Most of them suffer from limited functionality and unresolved bugs. Finally, we utilized csv-to-influxdb, which allows specifying timestamp-column, tag-columns, measurement and batch size at conversion time. Batch size is important since InfluxDB allows up to 5000 rows to be imported in one batch. Note that if the timestamp column name is different than *time*, then InfluxDB automatically creates a timestamp column called *name*, containing the import time. Since import time is not needed in our case, we renamed our pick-up time column to *time*.

However, for using the CSV file with the aforementioned converter, we needed to modify it accordingly. First and foremost, we needed to split each CSV file into 3 files containing only the columns needed for the specified measurement (fare, distance, passengers as seen in Section 3.2.2). For each subfile We needed to remove useless columns, format dates so that the follow a specific format (2016-01-15 00:00:00), format floating point numbers so they all follow the same format (13.45), and change the delimiter to comma. Handling files with millions of rows in Microsoft Excel is not possible (limit of 1,048,576 rows), so we wrote our own Java code that performs this formatting.

A conversion and import command using csv-to-influxdb in command line looks like this:

```
.\csv-to-influxdb_windows_amd64.exe -d cabs -m fare
-t VendorID,RatecodeID,payment_type -ts time
yellow_tripdata_fare_2016-05.csv
```

The command above is of a specific format, *csv-to-influxdb [options] csv-filepath*, where the *[options]* used refer to:

- –server, -s Server address (default http://localhost:8086)
- –database, -d Database name (default test)
- –measurement, -m Measurement name (default data)
- –tag-columns, -t Comma-separated list of columns to use as tags
- –timestamp-column, -ts Header name of the column to use as the timestamp (default timestamp)
- –batch-size, -b Batch insert size (default 5000)

3.2.4 Basic Queries

InfluxQL is an SQL-like query language provided by InfluxDB, which provides statements for **data and schema exploration**, **database management**, **continuous queries**, **mathematical and aggregation function and authentication and authorization**.

For **Data Exploration**, InfluxQL supports basic *SELECT* statement, as well as clauses, such as *WHERE, GROUP BY, INTO, ORDER BY, LIMIT* and *OFFSET*. Moreover, InfluxQL provides subqueries functionality as an alternative to SQL's *HAVING* clause. Examples of such statements will be given in Section 3.3.3. Moreover, it specifically supports *SLIMIT* and *SOFFSET* for limiting and offsetting point the number of series returned respectively.

For instance, the query below uses an InfluxQL function and a time interval in the *GROUP BY* clause to calculate the average total fare for each 1week interval in the query's time range. *SLIMIT 1* requests a single series associated with the fare measurement.

```
SELECT MEAN(total_amount)
FROM fare
WHERE time >= '2016-01-01 00:00:00' AND
time <= '2016-01-31 00:00:00'
GROUP BY *,time(1w)
SLIMIT 1
```

InfluxQL also provides a *TIMEZONE* clause (*tz()*) which returns the UTC offset for the specified timezone. For instance appending *tz('America / Chicago')* in the previous query would return a time column that would like like *2016-01-01T19:00:00**-05:00***.

The most important **Schema Exploration** statements and their results can be seen in Figure 6. Regarding **Database Management** statements like *DROP/CREATE database, DROP series, DROP measurement, DROP shard* and *DROP/ALTER/CREATE retention policy* are provided and are self explanatory. However, when authorization is enabled these commands are only available to admin users. Moreover the database can be backed up using the command *.\influxd backup -database [database name] -retention [retention policy] -since [start date for backup] [destination folder]* and restored using first the command *influxd restore -metadir [path to InfluxDB metadata folder] [backup path]* for restoring the metadata and the command *influxd restore -database [database name] -datadir [path to InfluxDB data*

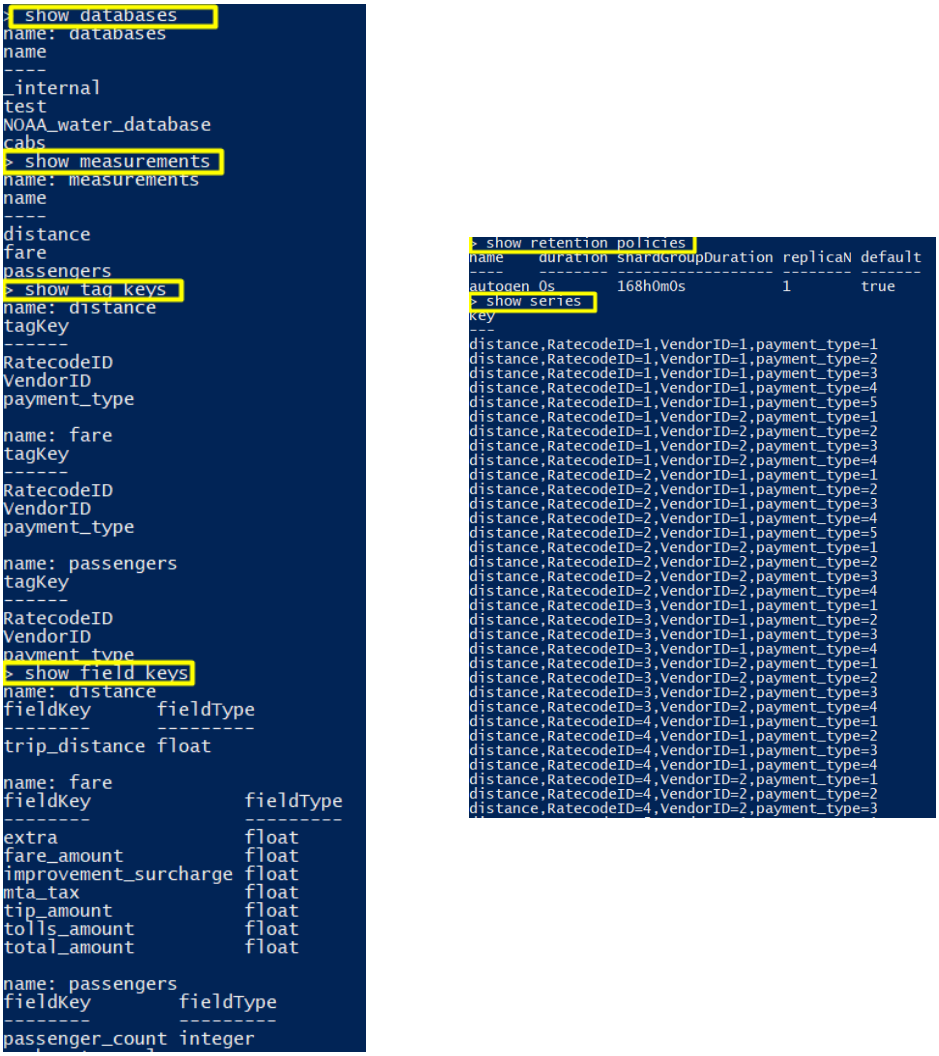


Figure 6: Basic Schema Exploration statements.

folder] [backup path] for actually restoring the database.

InfluxDB provides support for **Continues Queries**, namely queries that run automatically and periodically on real-time data and store query results in a specified measurement. This is extremely important when it comes to time series and real time analysis for the reasons below[14]:

[14]InfluxData — Documentation — Continuous Queries. 2017. Retrieved from `https://docs.influxdata.com/influxdb/v1.3/query_language/continuous_queries`.

- **Downsample Data.** Use CQs to automatically downsample high precision data to a lower precision and remove the high precision data from the database if not needed.
- **Pre-calculate Expensive Queries.**
- **Substitute *HAVING* clauses and nested functions.**

An example of such query can be seen below.

```
1 CREATE CONTINUOUS QUERY "cq_average_fare" ON "cabs"
2 RESAMPLE EVERY 30m
3 BEGIN
4   SELECT mean("total_fare")
5   INTO "average_fare"
6   FROM "fare"
7   GROUP BY time(1h)
8 END
```

cq_average_fare calculates the per hour average of total fare from the *fare* measurement and stores the results in the *average_fare* measurement in the *cabs* database.

cq_average_fare executes at 30-minute intervals (*EVERY* clause). Every 30 minutes, *cq_average_fare* runs a single query that covers the time range for the current time bucket, that is, the one-hour time bucket that intersects with current time (*now()*).

Finally InfluxDB provides various **Functions and Operators**, including, but not limited to, **Aggregations** e.g. MEAN, INTEGRAL, MODE, STANDARD DEVIATION, etc., **Selectors** e.g. PERCENTILE, SAMPLE (uses reservoir sampling), TOP, BOTTOM (Note that TOP and BOTTOM in InfluxDB is different than in SQL Server. They return the N greatest or lowest values for a specified field and not just the N latest or oldest ones.) **Transformations** e.g. HISTOGRAM, MOVING AVERAGE, DERIVATIVE and **Predictors** e.g. HOLT WINTERS method based on seasonality.

Since Holt Winters is quite unique in InfluxDB, it is worth mentioning the structure of queries containing the method (the clauses in square brackets are optional). There are two methods, HOLT_WINTERS which returns only the predicted values, and HOLT_WINTERS_WITH-FIT, which returns both the actual and predicted values. They both take as arguments the field that should be predicted, the number N of instances to be predicted and the

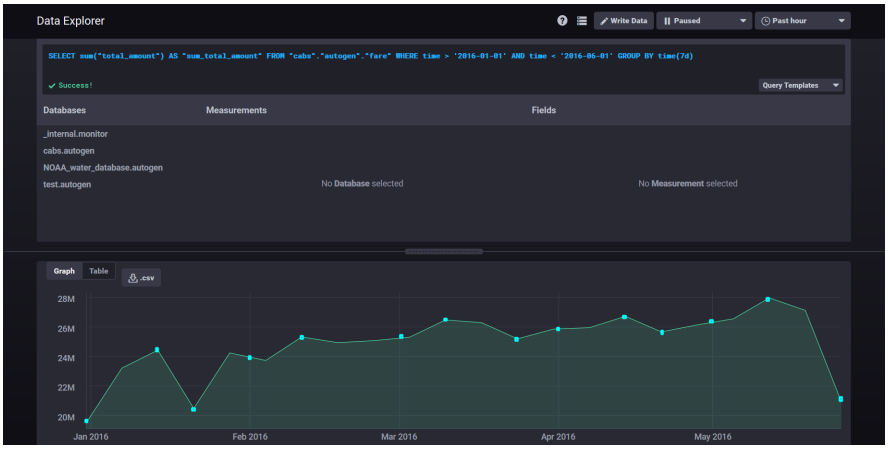


Figure 7: Identifying seasonality patterns for the sum of total fares in Chronograf. The dots represent the approximate seasonal data points. Each month contains 4 dots with similar pattern.

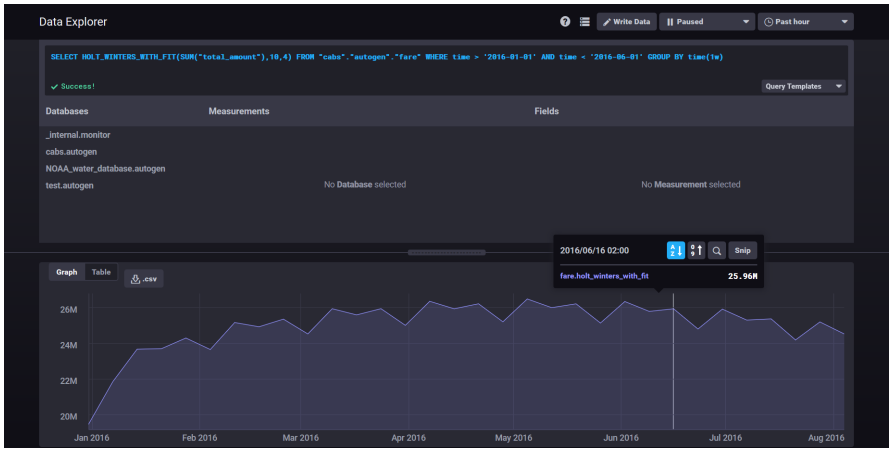


Figure 8: Executing the Holt Winters function in Chronograf.

seasonality pattern S. The method must be used along with a GROUP BY time() clause.

The N predicted values occur at the same interval as the GROUP BY time() interval. If your GROUP BY time() interval is 1w and N is 10 you'll receive ten predicted values that are each 1 week apart. S is the seasonal pattern parameter and delimits the length of a seasonal pattern according to the GROUP BY time() interval. If your GROUP BY time() interval is 1w and S is 4, then the seasonal pattern occurs every 4 weeks (1 month), that is, every four data points. If seasonality is not needed S can be set to 0 or 1.

```
SELECT HOLT_WINTERS[_WITH-FIT](<function>(<field_key>),<N>,<S>)
[INTO_clause]
FROM_clause
[WHERE_clause]
GROUP_BY_clause
[ORDER_BY_clause]
[LIMIT_clause]
[OFFSET_clause]
[SLIMIT_clause]
[SOFFSET_clause]
```

An example of a Holt Winters query can be seen below. The query predicts $N = 10$ weekly future values for the sum of total_fares after 1^{st} of June. Note that the predicted values will be in weekly time intervals because of the group by query. The seasonality pattern is set to 4 because we noticed that there is a slight pattern on the sum of total fares on a monthly basis (See Figure 7). The results can be seen in Figure 8.

```
SELECT HOLT_WINTERS_WITH_FIT(SUM(total_amount),10,4)
FROM fare
WHERE time > '2016-01-01' AND time < '2016-06-01'
GROUP BY time(1w)
```

3.3 Benchmarking SQL Server vs InfluxDB

3.3.1 Query Properties

To get an overview on possible query types, we distinguish five aspects of query properties [2]:

- Query Interval

- Aggregation
- Object Identity
- Dimension
- Condition Type

However, some of these properties are not relevant to InfluxDB. Firstly, Dimension is not relevant since we are dealing solely with Time Dimension. InfluxDB is not a spatial Database. Also, Condition Type is not relevant, as InfluxDB supports only single object operations. According to InfluxDB documentation[15] "Currently, there is no way to perform cross-measurement math or grouping. All data must be under a single measurement to query it together. InfluxDB is not a relational database and mapping data across measurements is not currently a recommended schema".

As a result we exclude these properties from the queries below. Below, we present queries in InfluxQL that utilize combinations of the remaining properties. **The equivalent SQL queries are excluded from this report but can be found in the delivered project folder.** In SQL a non clustered index was created for the timestamp column to imitate InfluxDB's key and improve SQL's performance. When the index was removed, point queries on timestamp were up to 10 times slower. Note that point queries utilizing time use precision in seconds in InfluxDB. Now, in our case such point queries do not make sense but will be used for the sake of benchmarking. However, there might be cases where precision in seconds is actually relevant.

CLUSTERING IS A PREMIUM FEATURE IN INFLUXDB. THAT'S WHY WE COULD NOT TEST OUR QUERIES IN A CLUSTERED ENVIRONMENT. ALL BENCHMARKING QUERIES WERE EXECUTED IN A SINGLE NODE ENVIRONMENT.

3.3.2 Hardware Specifications

The benchmarking was executed using a Dell Inspiron 7559 laptop with the following specifications:

- **OS Name:** Microsoft Windows 10 Home

[15] InfluxData — Documentation — Frequently Asked Questions. 2017. `https://docs.influxdata.com/influxdb/v1.4/troubleshooting/frequently-asked-questions`.

- **System Type:** x64-based PC
- **Processor:** Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHz, 2301 Mhz, 4 Core(s), 4 Logical Processor(s)
- **Installed Physical Memory (RAM):** 16,0 GB
- **L2 Cache Size:** 1024 KB
- **L3 Cache Size:** 6144 KB

3.3.3 Benchmarking Queries

Query 1 (Range, Aggregation, Unknown): What was the total fare calculated per hour by the meters on New Year's Day 2016?

```
SELECT SUM(fare_amount)
FROM fare
WHERE time>='2016-01-01 00:00:00' AND time<'2016-01-02 00:00:00'
GROUP BY time(1h)
```

Query 2 (Unbounded, Aggregation, Unknown): Which TPEP provider has the maximum average trip distance for all recorder rides?

```
SELECT MAX(meanDistance),VendorID
FROM (
  SELECT MEAN(trip_distance) AS meanDistance
  FROM distance
  WHERE time>='2016-01-01 00:00:00'
  GROUP BY VendorID
)
```

Query 3 (Point, Aggregation, Unknown): How many passengers had payment disputes with drivers per TPEP provider for all recorded rides?

```
SELECT COUNT(passenger_count)
FROM passengers
WHERE payment_type='4'
GROUP BY VendorID
```

Query 4 (Range, Aggregation, Known): What was the total fare calculated by the meters per day in January 2016 for Creative Mobile Technologies?

```
SELECT SUM(fare_amount)
FROM fare
WHERE time>='2016-01-01' AND time<'2016-02-01' AND VendorID='1'
GROUP BY time(1d)
```

Query 5 (Unbounded, Aggregation, Known): How many passengers did VeriFone Inc. transport in total for all recorded rides per week?

```
SELECT COUNT(passenger_count)
FROM passengers
WHERE time>='2016-01-01' AND VendorID='2'
GROUP BY time(1w)
```

Query 6 (Point, Aggregation, Known): How many passengers traveled to JFK airport with VeriFone Inc. for all recorded rides?

```
SELECT COUNT(passenger_count)
FROM passengers
WHERE AND RatecodeID='2' AND VendorID='2'
```

Query 7 (Range, No Aggregation, Unknown): Which trip type (Rate Code) gave the company the highest single total fare amount in January 2016?

```
SELECT MAX(total_amount), RatecodeID
FROM fare
WHERE time>'2016-01-01' AND time<'2016-02-01'
```

Query 8 (Unbounded, No Aggregation, Unknown): Which TPEP Provider covered the longest distance on a single ride since March 2016?

```
SELECT MAX(trip_distance), VendorID
FROM distance
WHERE time>'2016-03-01'
```

Query 9 (Point, No Aggregation, Unknown): We noticed that there is an abnormally high fare value for timestamp 2016-03-10 22:59:51. Which TPEP provider charged a client with an abnormal fare and what is this fare?

```
SELECT VendorID, total_amount FROM fare
WHERE time = '2016-03-10 22:59:51'
```

Query 10 (Range, No Aggregation, Known): Give the top-10 greatest tip amounts along with payment type for trips with John F. Kennedy International Airport as destination for spring 2016.

```
SELECT TOP(tip_amount,10), payment_type
FROM fare
WHERE RatecodeID='2'
AND time>='2016-03-01' AND time<'2016-06-01'
```

Query 11 (Unbounded, No Aggregation, Known): Give the 10 lowest non-zero tips for individual rides with disputed fares since 2016.

```
SELECT BOTTOM(tip_amount,10)
FROM fare
WHERE payment_type='4' AND tip_amount > 0
AND time>='2016-01-01'
```

Query 12 (Point, No Aggregation, Known): Give exact fare amount and payment type for a ride starting at midnight on the 1^{st} of January.

```
SELECT fare_amount, payment_type
FROM fare
WHERE time = '2016-01-01 00:00:00'
```

3.3.4 Benchmarking Query Results

1. **Write Performance:** InfluxDB is designed to digest huge loads of streaming data (IoT, devOps, etc.) with the help of Kapacitor. However, for our benchmarking we used historical data in CSV format.

Query	Interval	Aggregation	Object Identity	SQL Time	InfluxDB Time
1	Range	Yes	Unknown	786	70
2	Unbounded	Yes	Unknown	3104.8	1537
3	Point	Yes	Unknown	621	33.2
4	Range	Yes	Known	2271.8	160.13
5	Unbounded	Yes	Known	2960	672.939
6	Point	Yes	Known	651.5	33.9
7	Range	No	Unknown	3079.1	1402.475
8	Unbounded	No	Unknown	2759.7	1584.96
9	Point	No	Unknown	4	2.5
10	Range	No	Known	663	628.14
11	Unbounded	No	Known	642.9	36
12	Point	No	Known	3.2	3

Table 3: Execution time in milliseconds (ms) for different query types in SQL Server and InfluxDB.

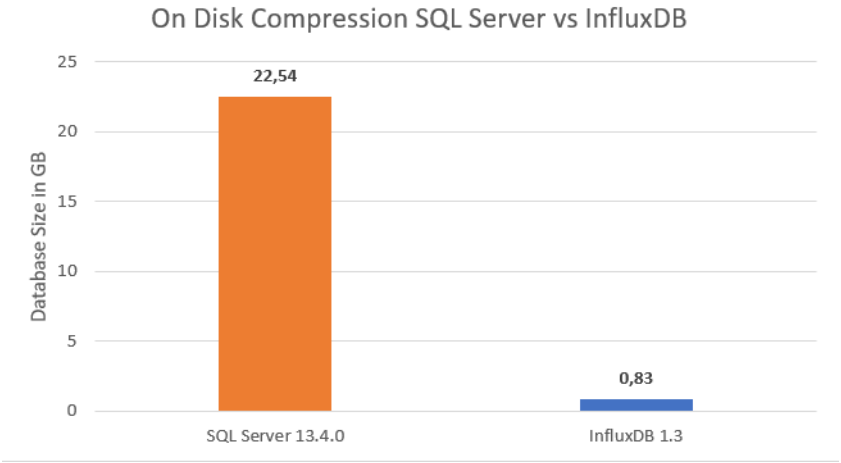


Figure 9: On-disk Compression: SQL Server vs InfluxDB.

InfluxDB does not have an official plugin that handles insertions from CSV, thus an external tool was used as mentioned in Section 3.2.3. This tool converts CSV data to the appropriate format and then inserts them to InfluxDB. However, SQL Server is designed to work with CSV data. Thus, the write throughputs cannot be comparable, as in InfluxDB we had to convert and then insert data, while in SQL Server we could import data immediately. That's why we decided to exclude write throughput from this report.

2. **On-disc storage Requirements:** For the same 30+ million rows dataset, after writing all the values, the space consumed by the data set in case of SQL Server was 22.54 GB. However, InfluxDB only required 0.83 GB. This results in approximately 692.3 bytes per record for SQL Server and 25.49 bytes per record for InfluxDB. For both databases the default configuration was used. See Figure 9 for a visualization of this comparison.

   Conclusion: InfluxDB outperformed SQL Server in on-disk performance by 27x using default configuration.

3. **Query Performance:** Query performance was tested by using the 12 benchmarking queries mentioned in Section 3.3.3 which cover 3 query properties (Query Interval, Aggregation and Object Identity) on a single node. Each query was executed 11 times and its average execution time was calculated excluding the first execution. It was found that the average of all query responses for SQL Server was 1462.25 ms and for InfluxDB was 513.68 ms (See Figure 11). The individual average execution times per query can be seen in Table 3 and Figure 10. We notice that for queries 9 and 12 InfluxDB has similar performance to SQL Server. These are point queries on the timestamp and the SQL database has an index built on the timestamp. If this index is removed its performance decreases 10x. Furthermore, query 10 for which InfluxDB and SQL Server perform similarly uses function TOP of InfluxDB. We noticed that this function is slower than the respective BOTTOM one. This behavior based on common sense is not rational and could be the result of a bug. Other than these 3 queries, for the remaining 9 queries InfluxDB outperforms SQL Server.

   Conclusion: InfluxDB is up to 20x faster than SQL Server in query performance with an average of 8x faster. Note that since InfluxDB's

query performance increases by adding extra nodes as will be seen in Section 3.4, we expect InfluxDB's query performance to improve even more in multi-node environments.

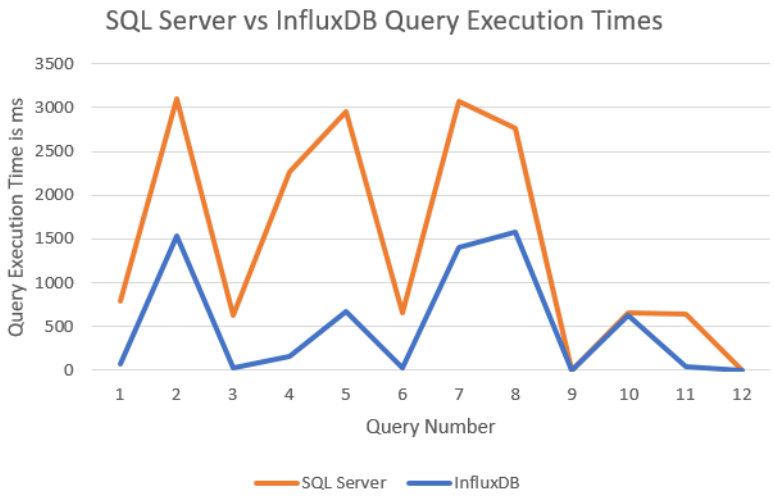


Figure 10: Average execution time per query number in ms.

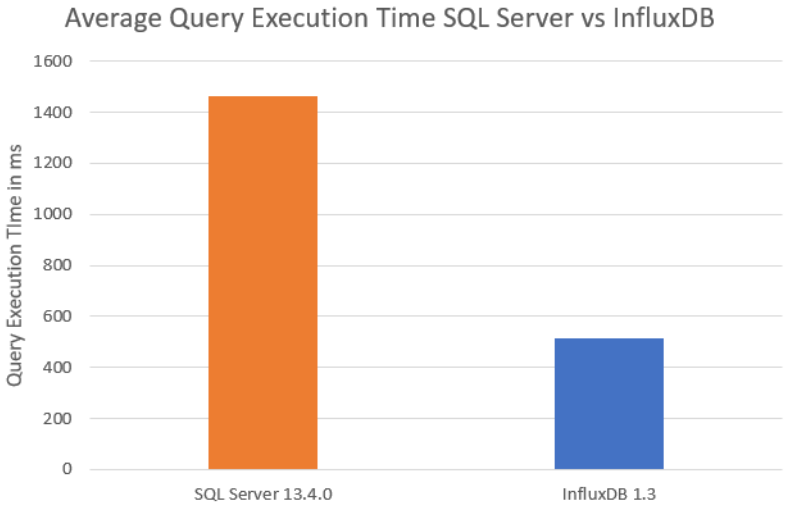


Figure 11: Average execution time for all queries in ms.

3.4 Benchmarking

This section will compare performance of 3 commonly used technologies for time-series data: Cassandra, OpenTSDB and Elasticsearch with InfluxDB. See Figure 5 to see the comparison of some basic features of these technologies. **Please note that all the content and statistics used in this section are from the official benchmarking by influxDB [3].** The following paragraphs introduce the metrics and data set against which this benchmarking is done.

Metrics: The benchmarking is done against the most commonly evaluated characteristics for working with time-series data which are:

1. Data ingest performance - measured in values per second.
2. On-disk storage requirements - measured in Gigabytes.
3. Mean query response time - measured in milliseconds.

The Data Set: The dataset used for these benchmarks, models a common DevOps monitoring and metrics use case, where a number of servers are periodically reporting system and application metrics at a regular time interval. Overview of the data can be seen in Table 4.

Parameters	**Cassandra**	**OpenTSDB**	**Elasticsearch**
Number of Servers	1000	1000	100
Values measured per Server	100	100	100
Measurement Interval	10s	10s	10s
Dataset duration(s)	24h	4h	24h, 48h, 72h, 96h
Total values in dataset	864,000,000 per day	144,000,000	864,000,000 per day

Table 4: Overview of the Parameters for the Sample Dataset.

3.4.1 InfluxDB vs. Cassandra

Let us compare the performance of InfluxDB and Cassandra with respect to the the 3 above mentioned vectors [5].

1. **Write Performance:** To test write performance, batch of 24-hour dataset with 4 worker threads was concurrently loaded (to match the number of cores on the server). The average throughput of Cassandra was found to be 90,333 values per second .The same dataset loaded into InuxDB at a rate of 476,460 values per second (See Figure 12).

Figure 12: Write Performance: InfluxDB Vs. Cassandra

Conclusion: InuxDB performed better than Cassandra by 5.3x when comparing data ingestion performance.

2. **On-disk storage Requirments:** For the same 24-hour dataset, after writing all the values the space consumed by the data set in case of Cassandra was 13.0 GB however influxDB only required 1.4 GB. This results in approximately 1.77 bytes per value for InuxDB and 16.15 bytes per value for Cassandra. See Figure 13 for detailed view.

   Conclusion: According to this benchmark, InfluxDB outperformed Cassandra by 9.3x better on-disk compression.

3. **Query Performance:** To test query performance an aggregation query was chosen that aggregates data for a single server (single time series) over a random 1-hour period of time, grouped into one-minute intervals, potentially representing a single line on a visualization - a common DevOps monitoring and metrics function. It is a very common use case for IoT. Cassandra has 2 scenarios, one where all of the query processing is handled on the client side, and another where Cassandra was

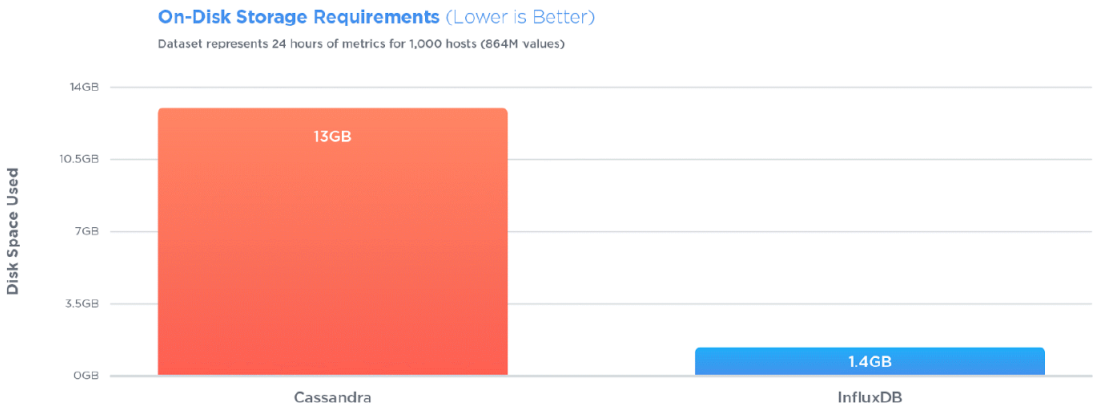


Figure 13: On-disk Compression: InfluxDB Vs. Cassandra

asked to return results for a set of queries, which forced all of the query processing to happen on the server side. Figure 14 shows that Cassandra was only able to deliver performance comparable to InfluxDB in the scenario where the application handled the query load. However, InfluxDB outperformed Cassandra by being x20 times faster than it when processing happened on the server side. And if we increase time period to 12 hours or if we increase number of time series, that is query over multiple servers, Cassandra is even slower and in such scenarios even if all the query load is handled by client side, Cassandra is way behind than InfluxDB.

Conclusion: InfluxDB is upto 168x faster than Cassandra in query performance (server side aggregations) [5].

3.4.2 InfluxDB vs. Elasticsearch

The statistics of benchmarking of InfluxDB against Elasticsearch are taken from the technical benchmarking report from official influx data benchmarking [6].

1. **Write Performance:** For testing of write performance, 24h dataset with 4 worker threads (to match the number of cores on the server) were loaded. The average throughput of Elasticsearch was found to be 115,422 values per second, however InfluxDB loaded the same data at

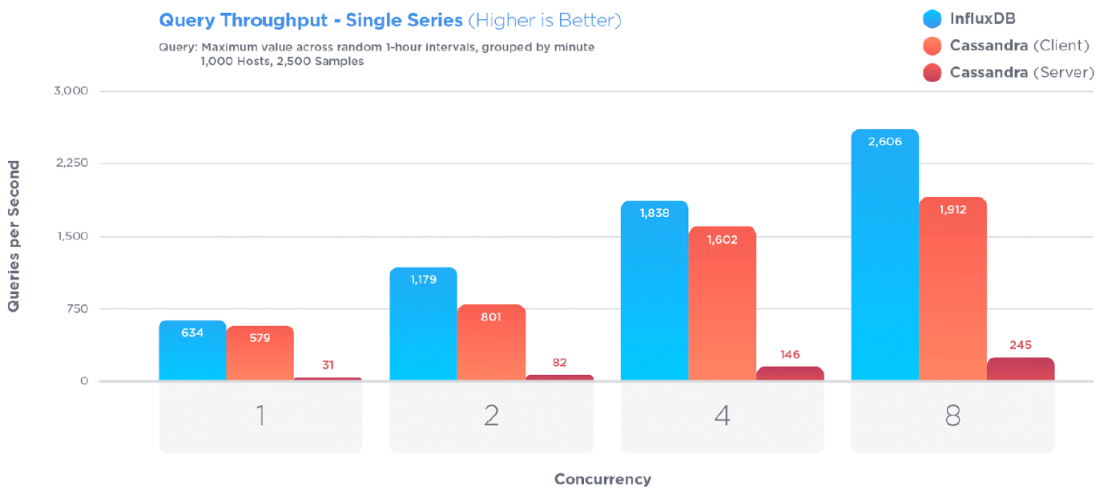


Figure 14: Query Performance: InfluxDB Vs. Cassandra

a rate of 926,389 values per second. The write throughput remained almost constant with larger data sets (48h, 72h and 96h).

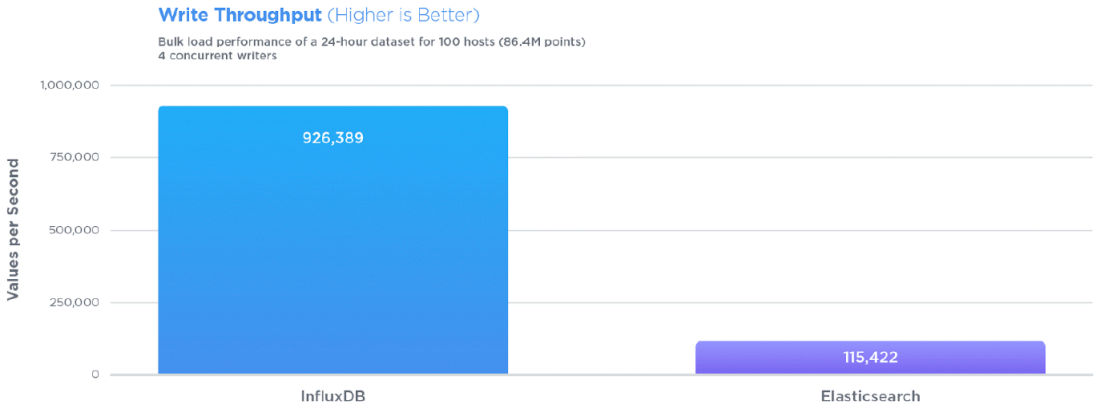


Figure 15: Write Throughput: InfluxDB Vs. Elasticsearch

Conclusion: InfluxDB was proven to be 8 times better than Elasticsearch in write performance (See Figure 15).

2. **On-disk storage Requirments:** To test on-disk compression the

benchmarking is done both against recommended configuration for time series data and default configuration of Elasticsearch. For 24 hour data-set the amount of space utilized by InfluxDB was 127 MB, however Elastic search required 2.1 GB with default settings and 502 MB with the recommended configurations. So approximately space needed for InfluxDB was 1.54 bytes per value and 6.09 bytes per value for Elasticsearch. Figure 16 shows this comparison in form of bar chart.

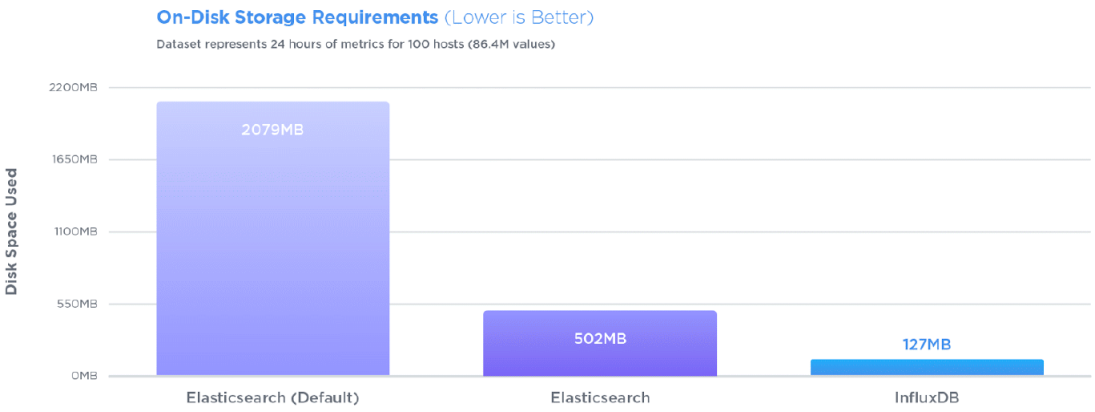


Figure 16: On-disk Compression: InfluxDB Vs. Elasticsearch

Conclusion: InfluxDB outperformed Elasticsearch in on-disk compression by 4x and 16x in recommended and default configuration respectively.

3. **Query Performance:** Query performance was tested by aggregating value on random 1-hour period of time, grouped into one-minute intervals, representing a single line on a visualization (Querying single time series is a common use case for DevOps and IoT). It was found that the mean query response time for Elasticsearch was 4.98ms (201 queries/sec) and for InfluxDB was 1.26ms (794 queries/sec) for the same query, which shows that InfluxDB was 4x faster in this case. It can be seen in Figure 17 that as size of the dataset increases performance of Elasticsearch degrades while the performance of InfluxDB remain constant.

   Conclusion: InfluxDB outperformed Elasticsearch by proving to be

Figure 17: Query Performance: InfluxDB Vs. Elasticsearch

4x to 10x better query performance depending on how large is the data-set.

3.4.3 InfluxDB vs. OpenTSDB

To compare InfluxDB with OpenTSDB, the benchmarking done in this section is taken from the technical benchmarking report from official influx data benchmarking [7].

1. **Write Performance:** For testing of write performance, 24h dataset with 4 worker threads (to match the number of cores on the server) were loaded. In addition to that, because target OpenTSDB setup required 6 servers in total (4 HBase nodes + 2 OpenTSDB daemons, and exclusive of the additional Zookeeper node) compared to just a single InfluxDB node, so write performance was looked at per server basis. Additionally, replication factor of 1 with in H-base is used to make a fair comparison with a single node of InfluxDB. The average throughput of OpenTSDB was found to be 35,648 values per second (per server). Data ingestion for InfluxDB was at a rate of 179,814 values per second (per server) for the same database. per-server throughput.

   Conclusion: InfluxDB outperformed OpenTSDB by 5.0x when eval-

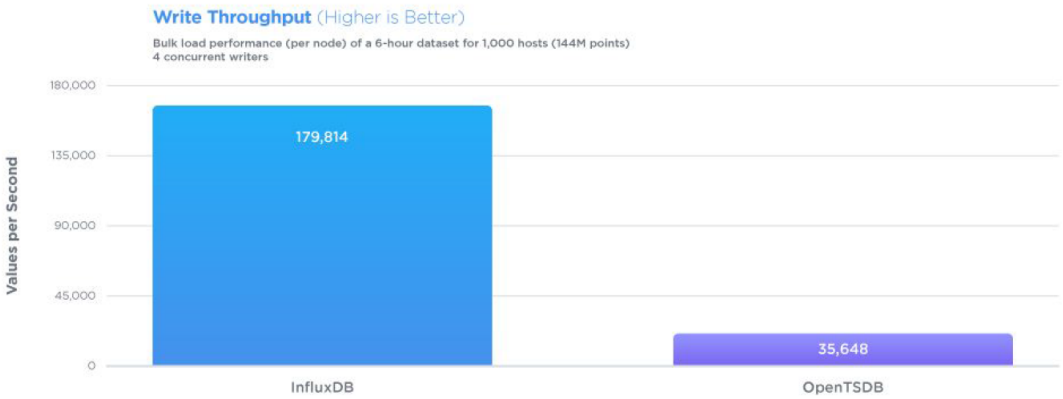


Figure 18: Write Throughput: InfluxDB Vs. OpenTSDB

uating write throughput.

2. **On-disk storage Requirments:** For the data mentioned above, the amount of disk space consumed by OpenTSDB was 5.8 GB. The same dataset required only 351MB for InfluxDB. This results in approximately 2.44 bytes per value for InfluxDB and 40.3 bytes per value for OpenTSDB.

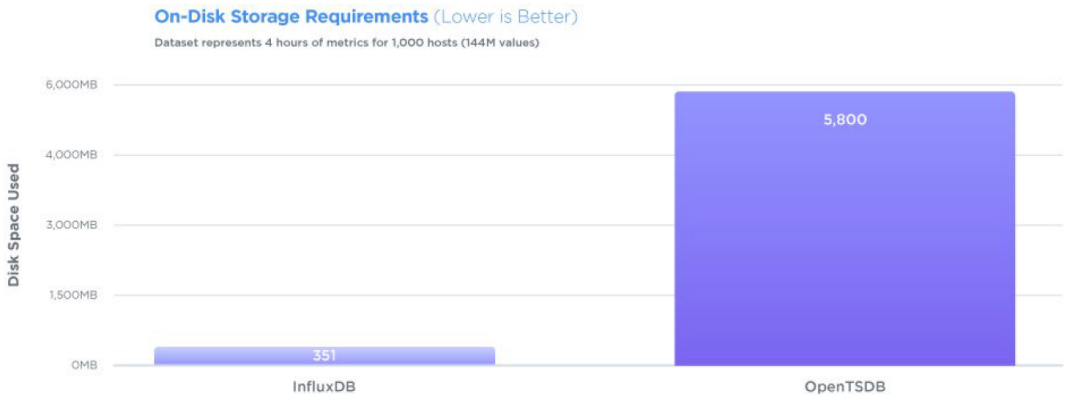


Figure 19: On-disk Compression: InfluxDB Vs. OpenTSDB

Conclusion: InfluxDB outperformed OpenTSDB by 16.5x when eval-

uating on-disk compression.

3. **Query Performance:** To test query performance, the query selected is the one that aggregates data for 8 servers over a random 1-hour period of time, grouped into one-minute intervals, potentially representing multiple lines on a visualization which is a common DevOps monitoring and metrics function. Concurrency was increased in the queries, starting from 1 worker upto 32 workers to check how each database perform in the situation of increasing work load. It can be see from Figure 20, InfluxDB performed better than OpenTSDB in all scenarios and there was only a slight variance across different concurrencies. If we see the architecture of OpenTSDB, these results are logical because in OpenTSDB each query has to reach out to H-base to retrieve data before performing query which adds to latency and hence results in slower response time.

Figure 20: Query Performance: InfluxDB Vs. OpenTSDB

Conclusion: InfluxDB was proven to be 4.0x faster than OpenTSDB in query performance.

References

[1] DB-Engines. Db-engines ranking, 2017.

[2] Düntgen, C., Behr, T., and Güting, R. H. Berlinmod: A benchmark for moving object databases. *The VLDB Journal 18*, 6 (Dec. 2009), 1335–1368.

[3] InfluxData. Influxdb version 1.3 documentation, 2017.

[4] Joshi, P., Massaron, L., and Hearty, J. *Python: Real World Machine Learning*. Packt Publishing, 2017.

[5] Persen, T., and Winslow, R. Influx db vs. cassandra for time-series data, metrics & management. Tech. rep., September 2016.

[6] Persen, T., and Winslow, R. Influx db vs. elasticsearch for time-series data, metrics & management. Tech. rep., September 2016.

[7] Persen, T., and Winslow, R. Influx db vs. opentsdb for time-series data, metrics & management. Tech. rep., November 2016.