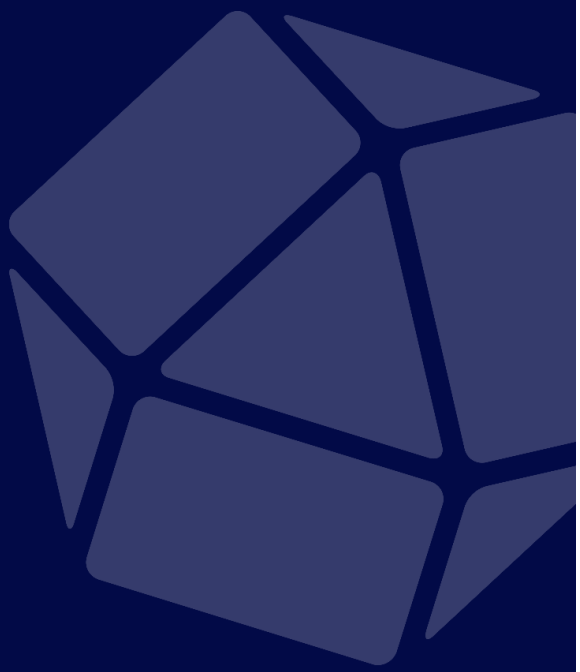
influxdata®

AN INFLUXDATA CASE STUDY

Benchmarking InfluxDB vs. Cassandra for Time Series Data, Metrics & Management

Vlasta Hajek
Senior Software Engineer, Bonitoo
Ales Pour
Engineer, Bonitoo
Ivan Kudibal
Engineering Manager, Bonitoo



OCTOBER 2022

Introduction

In this technical paper, we'll compare the performance and features of InfluxDB and Cassandra for common time series workloads, specifically looking at the rates of data ingestion, on-disk data compression, and query performance. We'll also look at a feature comparison and the resulting time required to build a complete time series solution with each tool.

Our goal with this benchmarking test was to create a consistent, up-to-date comparison that reflects the latest developments in both InfluxDB and Cassandra. Periodically, we'll re-run these benchmarks and update this document with our findings. All of the code for these benchmarks is available on GitHub. Feel free to open up issues or pull requests on that repository or if you have any questions, comments, or suggestions.

This comparison should prove valuable to developers and architects evaluating the suitability of these technologies for their use case, especially those building DevOps Monitoring (Infrastructure Monitoring, Application Monitoring, Cloud Monitoring), IoT Monitoring, and Real-Time Analytics applications.

Why time series?

Time series data has historically been associated with applications in finance. However, as developers and businesses move to instrument more in their servers, applications, network and the physical world, time series is becoming the de facto standard for how to think about storing, retrieving, and mining this data for real-time and historical insight. To learn more about why you should insist on using a purpose-built, time series backend versus attempting to retrofit a document, full-text, or RDBMS to satisfy your use case, check out the Why Time Series Matters for Metrics, Real-Time and IoT/Sensor Data technical paper.

What is time series data?

Time series data is nothing more than a sequence of values, typically consisting of successive measurements made from the same source over a time interval. Put another way, if you were to plot your values on a graph, one of your axes would always be time. For example, time series data may be produced by sensors like weather stations or RFIDs, IT infrastructure components like apps, servers, and network switches or by stock trading systems.

Time series databases are optimized for the collection, storage, retrieval, and processing of time series data; nothing more, nothing less. Compare this to document databases optimized for storing JSON documents, search databases optimized for full-text searches or traditional relational databases optimized for the tabular storage of related data in rows and columns.

Baron Schwartz has outlined some of the typical characteristics of a purpose-built time series database. These include:

- 90+% of the database's workload is a high volume of high-frequency writes.
- Writes are typically appends to existing measurements over time.
- These writes are typically done in a sequential order, for example: every second or every minute.
- If a time series database gets constrained for resources, it is typically because it is I/O bound.
- Updates to correct or modify individual values already written are rare.
- Deleting data is almost always done across large time ranges (days, months or years), rarely if ever to a specific point.
- Queries issued to the database are typically sequential per-series, in some form of sort order with perhaps a time-based operator or function applied.
- Issuing queries that perform concurrent reads or reads of multiple series are common.

Comparison at a glance

	InfluxDB	Cassandra
Description	Database designed for time series, events and metrics data management	Distributed, non-relational database
Website	https://influxdata.com/	http://cassandra.apache.org/
GitHub	https://github.com/influxdata/influxdb	https://github.com/apache/cassandra
Initial Release	2013	2008
Latest Release	v1.8.10, October 2021	v4.0.5, July 2022
License	Open Source, MIT	Open Source, Apache
Language	Go	Java

Operating Systems	Linux, OS X	Linux, OS X, Windows
Data Access APIs	HTTP Line Protocol, JSON, UDP	JSON, CQL
Schema	Schema-free	User-defined schema

Overview

Before we dig into the benchmarks themselves, it's important to point out that doing a head-to-head comparison of InfluxDB and Cassandra for time series workloads is impossible without first writing some application code to make up for Cassandra's lacking functionality. Effectively, you'd need to rewrite some portion of InfluxDB in your own application.

In building a representative benchmark suite, we identified the most commonly evaluated characteristics for working with time series data. As we'll describe in additional detail below, we looked at performance across three vectors:

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

CONCLUSION:

> ***InfluxDB outperformed Cassandra in write throughput, on-disk compression, and query performance.***

It's important to emphasize that achieving optimal query performance with Cassandra requires additional application-level processing. This could have a huge impact in production deployments where each query places additional load on the application servers.

The dataset

For this benchmark, we focused on a dataset that models a common DevOps monitoring and metrics use case, where a fleet of servers are periodically reporting system and application metrics at a regular time interval. We sampled 100 values across 9 subsystems (CPU, memory, disk, disk I/O, kernel, network, Redis, PostgreSQL, and Nginx) every 10 seconds. For the key comparisons, we looked at a dataset that represents 100 servers over a 24-hour period, which represents a relatively modest deployment.

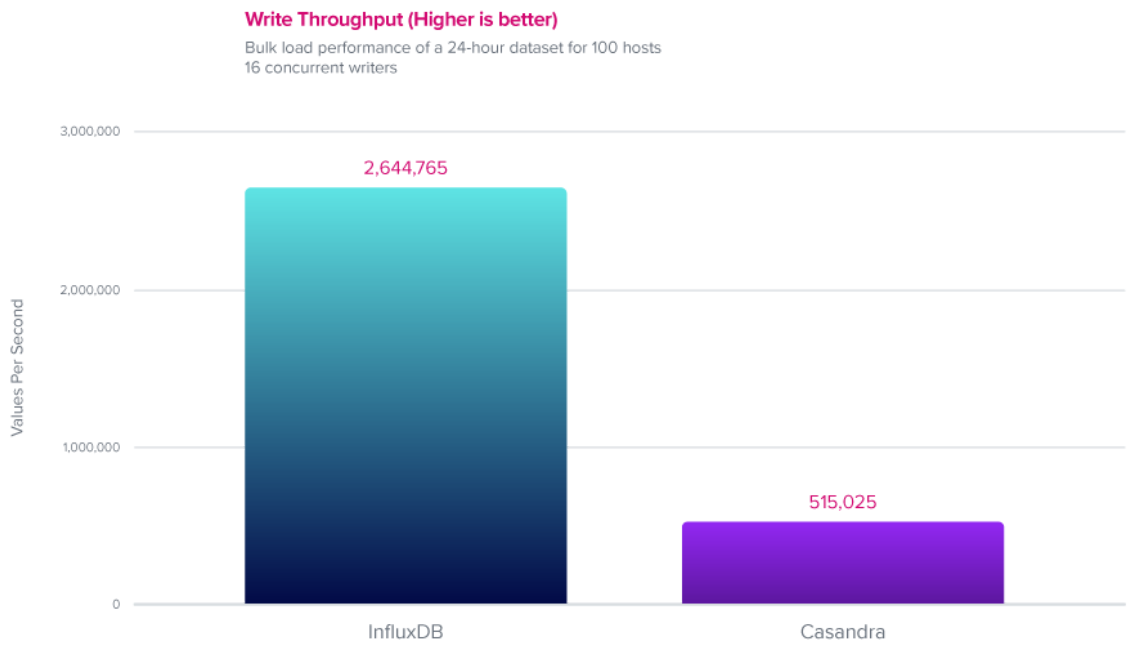
Overview of the parameters for the sample dataset

Number of servers	100
Values measured per server	100
Measurement interval	10s
Dataset duration(s)	24h
Total values in dataset	87,264,000

This is only a subset of the entire benchmark suite, but it's a representative example. At the end of this paper we will discuss other variables and their impacts on performance. If you're interested in additional details, you can read more about the testing methodology on GitHub.

Write performance

To test write performance, we concurrently batch loaded the 24-hour dataset with 16 worker threads (to be able to compare to the other databases' tests). We found that the average throughput of Cassandra was **515,025 values per second**. The same dataset loaded into InfluxDB at a rate of **2,644,765 values per second**, which corresponds to approximately **5x faster ingestion** by InfluxDB. (Remember: the concurrency for this test was 16, with 100 hosts reporting.)



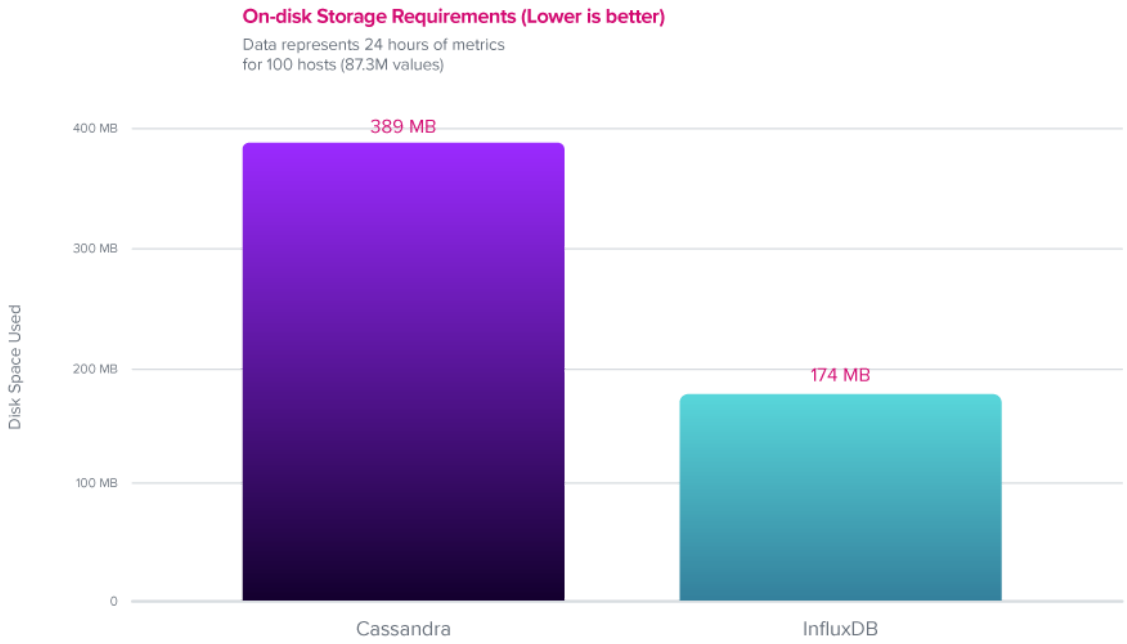
This write throughput stays relatively consistent across larger datasets (i.e. 48 hours, 72 hours, 96 hours).

CONCLUSION:

> ***InfluxDB outperformed Cassandra by 5x when evaluating data ingestion performance.***

On-disk storage requirements

For the same 24-hour dataset outlined above, we looked at the amount of disk space used after writing all values and allowing each database's native compaction process to finish. We found that the dataset required **389 MB for Cassandra**. The same dataset required **174 MB for InfluxDB**, corresponding to **2.4x better compression by InfluxDB**. This results in approximately 2.15 bytes per value for InfluxDB and 4.6 bytes per value for Cassandra.



CONCLUSION:

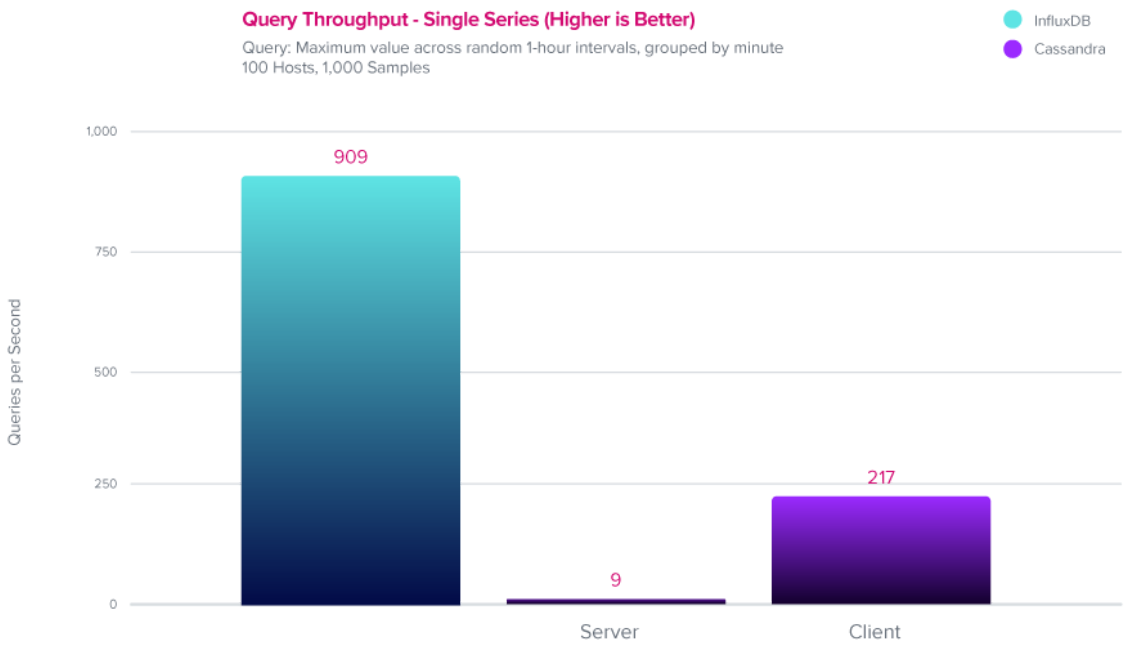
> ***InfluxDB outperformed Cassandra by delivering 2.4x better on-disk compression.***

Query performance

To test query performance, we chose a query that aggregates data for a single server 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. Querying an individual time series is common for many IoT use cases as well.

To add some detail to our earlier statements on query performance, we looked at two different scenarios for Cassandra — one where the data aggregation processing happened on the client, and another where Cassandra was asked to return results for a set of queries, which forced all of the data aggregation processing to happen on the server side.

As you can see from the graph below, Cassandra was only able to deliver comparable performance with InfluxDB in the scenario where we moved data aggregation to our client-side application logic. However, Cassandra **performed up to 100x slower** when it was responsible for handling all of the data aggregation on the server.



To reduce experimental variability, the query times were averaged over 1000 runs. With 1 worker threads we found that the worst-case mean query response time with server aggregation for **Cassandra was 106ms (9 queries/sec)**. The same query took an average of **1.1ms (909 queries/sec) for InfluxDB**, and **4.6ms (217 queries/sec)** when doing client-side, application-level data aggregation for Cassandra, which required custom development as mentioned above.

Since achieving this optimal performance from Cassandra requires users to employ the application-level processing we've discussed (and required transmitting more raw data over the network), we wanted to look at the impact of queries that require analyzing more data.

CONCLUSION:

> ***InfluxDB outperformed Cassandra by delivering up to 100x faster query performance.***

Testing hardware

All of the tests performed were conducted on two virtual machines in AWS infrastructure, running Ubuntu 16.04 LTS. We used instance type r4.4xlarge (Intel Xeon E5-2686 v4 2.3GHz, 16 vCPU, 122 GB RAM, 1x EBS Provisioned 6000 IOPS SSD 250GB) for a database server and c4.xlarge instance type (Intel Xeon E5-2666v3 2.9GHz, 4 vCPU, 7.5GB RAM) for a client host with the data load and query clients.

User experience comparison

The user experiences of InfluxDB and Cassandra differ in six key ways: upfront engineering work, write complexity, read complexity, query flexibility, maintenance burden, and mental models. Cassandra was designed as a building block to create custom distributed data systems, while InfluxDB is designed specifically with time series data as a first-class citizen.

Upfront engineering work

Cassandra requires upfront engineering effort to be useful. Using Cassandra required us to be familiar with Cassandra column families, rows, wide rows, CQL, storage options, partition keys, and secondary indexes. These are general Cassandra concepts and are not particular to the time series use case.

Cassandra also requires domain-specific decision-making. For our time series use case, we made decisions about the data storage (e.g. using TimeWindowCompactionStrategy and LZ4Compressor), what time intervals to use for bucketing our data (e.g. one wide row per day), whether to use time-based rollups, and how to best use server-side aggregations.

Write complexity

We had to choose — ahead of time — a granularity of time by which to shard our time series data. This is because each 'wide row' in Cassandra has a maximum effective width, both in terms of performance tradeoffs and due to hard data size limits. We examined our benchmark data generator, calculated how many points it would create per series per time interval, and decided to create a separate wide row for each 24-hour period for each series. During this engineering process, we had to ensure that our Cassandra clients and servers used consistent time bucket policies. Successfully implementing this required a great deal of care and upfront assumption-making.

Cassandra forced us to make these decisions upfront, before writing a single data point. It required experimentation and tuning, and could have made for a brittle system. We thought of other ways to store data, but they required our ingestion software to be even more complicated. (For example, we considered using adaptively-sized time intervals, but that would require writing a separate runtime index that maintains mappings from series names to their time intervals.)

Writing to Cassandra requires building complicated 'smart clients' in which bugs cannot be tolerated, and initial data modeling assumptions are very difficult to change.

In contrast, InfluxDB is a fully-featured time series database and thus requires no engineering work (besides learning basic InfluxQL) to ingest data correctly and performantly.

Read complexity

Cassandra requires custom software to query time series data. Our Cassandra client implementation has three components, each of which required experimentation and careful thought to build correctly.

The first component in our query pipeline is the query planner. Cassandra does not directly support time series queries, so we had to build a system to translate time series query templates into actual Cassandra CQL requests. (An example high-level query we developed is the following: aggregate the maximum CPU usage of a set of eight servers over the course of an hour, in one-minute intervals.)

Building the query planner required lots of upfront design and continued experimentation to make it both correct and fast. Because our benchmarking setup uses pre-generated data, we were able to verify the correctness of our client programs. Unfortunately, in production scenarios, it is difficult to check that a query planner returned correct results. We believe that this tension could be a significant source of bugs in Cassandra deployments.

Our custom Cassandra query planner supports two execution strategies: 1) with server aggregations, and 2) without server aggregations. It turns out that using server aggregation functions requires more round-trip requests (but lower overall bandwidth), and skipping server aggregation requires fewer round-trip requests (but higher bandwidth).

The second component in our Cassandra query pipeline is the query executor. The query executor is responsible for taking the output of the query planner and dispatching requests to a Cassandra cluster. It also manages receiving the results. Our query executor handles multiple high-level queries in parallel, and, within each, dispatches raw CQL requests serially. The query executor uses the `gocql` client library, configured for Cassandra's binary protocol version 4 to achieve the best performance. Our client uses low-memory iterators to read the CQL results from Cassandra. A source of complexity arises out of the fact that the query executor is necessarily customized for each query plan: the query dispatch and client-side aggregation logic is not amenable to code abstractions.

The third and final component in the query pipeline is the client-side query result aggregator. Because each high-level query generates many Cassandra requests, our query pipeline must efficiently aggregate those results to form the result for the high-level query. In particular, the result aggregator must perform both time window bucketing and final aggregations over each bucket. (An example would be merging the data for eight different measurements together, with the `max` function, and putting the results in the correct buckets.)

We took care to implement client-side aggregation functions that are both commutative and associative. We also built the aggregators to be constant-space, so that the client would not exhaust RAM when collating results from Cassandra. Generally, implementing the Cassandra query engine required careful thought, and a great deal of engineering time and risk.

It is worth noting that Cassandra 4.1 is expected to support time aggregation directly in CQL which will greatly simplify time-series queries and the query performance may improve.

In comparison, InfluxDB is straightforward. The InfluxQL query language enables users to write human-friendly queries, in the spirit of SQL, and InfluxDB servers will do everything needed to compute the results.

Query flexibility

One aspect of the Cassandra philosophy is that users are expected to write custom support code for each type of query they wish to run. This is made explicit in the official Cassandra documentation:

> *"Step 1: Determine What Queries to Support: Try to determine **exactly** what queries you need to support."*
>
> – Basic Rules of Cassandra Data Modeling

When writing the Cassandra time series benchmarker, we found this advice to be entirely correct. Cassandra required us to write custom code for each type of query we considered executing. Although this is an expected property of working with Cassandra, it is a severe tradeoff to make: each new query type, and the data used to fulfill that query, must be prepared ahead of time.

In general, Cassandra is a good choice when the problem at hand is extremely well-defined and involves huge amounts of data. In contrast, InfluxDB permits users to run ad-hoc queries without any upfront engineering time.

Maintenance burden

Another aspect of Cassandra systems is that the database is not responsible for maintaining data integrity when values are denormalized.

When a Cassandra developer wants to do time-based rollups (i.e. precomputing the aggregate value of a series over fixed time intervals), then she will have to write application code to ensure that it is correct. This can cause a large maintenance burden, as developers are forced to:

1. Ensure all 'smart clients' are correct when writing denormalized data
2. Guarantee the correctness of 'smart client' behavior during a production code update and rollout
3. Guarantee the operational stability of these auxiliary processes in production

Thankfully, we were able to skip many of these issues because our code was not used in a production environment.

InfluxDB, by contrast, requires no support systems to be fully functional.

Mental models

Cassandra and InfluxDB are fundamentally different pieces of software. Cassandra is one of many building blocks used in systems that require highly specialized behavior involving large amounts of data. InfluxDB is purpose-built exactly to solve the problems of storing and querying large volumes of time series data. This means that, while Cassandra can be made to support a huge number of use cases, the labor, time, and risk involved may be high.

In conclusion, using Cassandra for time series data would require implementing additional functionality that InfluxDB gives you right out-of-the-box.

Summary

In the course of this benchmarking paper, we looked at the performance of InfluxDB and Cassandra performance across three vectors:

- Data ingest performance - measured in values per second
- On-disk storage requirements - measured in bytes
- Mean query response time - measured in milliseconds

The benchmarking tests and resulting data demonstrated that InfluxDB outperformed Cassandra across all three tests by a significant margin. Specifically:

- InfluxDB outperformed Cassandra by 5x for data ingestion

- InfluxDB outperformed Cassandra by delivering 2.4x better compression
- InfluxDB outperformed Cassandra by up to 100x when measuring query performance

We discussed this in depth above, but it's important to remember that in order to get optimal performance with time series data from Cassandra, upfront design decision making and custom application code are required. InfluxDB, on the other hand, is ready to use for time series workloads out of the box with no additional configuration.

InfluxDB documentation, downloads & guides

Get InfluxDB

Try InfluxDB Cloud for Free

Get documentation

Additional tech papers

Join the InfluxDB community