influxdata

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

AN INFLUXDATA TECHNICAL PAPER

External Contributors

**Vlasta Hajek** Senior Software Engineer, Bonitoo

**Tomas Klapka** DevOps Engineer, Bonitoo

**Ivan Kudibal** Engineering Manager, Bonitoo

January 2018 (Revision 3)

# Overview

In this technical paper, we'll compare the performance and features of InfluxDB and MongoDB for common time series workloads, specifically looking at the rates of data ingestion, on-disk data compression, and query performance. This data should prove valuable to developers and architects evaluating the suitability of these technologies for their use case. Specifically, the time series data management use cases involving building DevOps Monitoring (Infrastructure Monitoring, Application Monitoring, Cloud Monitoring), IoT Monitoring, and Real-Time Analytics applications.

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

# About InfluxDB

### InfluxDB Version Tested: v1.4.2

InfluxDB is an open-source Time Series Database written in Go. At its core is a custom-built storage engine called the Time-Structured Merge (TSM) Tree, which is optimized for time series data. Controlled by a custom SQL-like query language named InfluxQL, InfluxDB provides out-of-the-box support for mathematical and statistical functions across time ranges and is perfect for custom monitoring and metrics collection, real-time analytics, plus IoT and sensor data workloads.

### 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:

# About MongoDB

## MongoDB Version Tested: v3.6.2

MongoDB is an open-source, document-oriented database, colloquially known as a NoSQL database, written in C and C++. Though it's not generally considered a true *Time Series Database* per se, its creators often promote its use for time series workloads. It offers modeling primitives in the form of timestamps and bucketing, which give users the ability to store and query time series data.

Please note that this paper does not look at the suitability of InfluxDB for workloads other than those that are time series-based. InfluxDB is not designed to satisfy document storage use cases and therefore those will not be explored in this paper. For these use cases, we recommend sticking with MongoDB or similar NoSQL databases.

- 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 | MongoDB |
|---|---|---|
| **Description** | Database designed for time series, events and metrics data management | Scalable, document-oriented NoSQL database |
| **Website** | https://influxdata.com/ | https://www.mongodb.com/ |
| **GitHub** | https://github.com/influxdata/influxdb | https://github.com/mongodb/mongo |
| **Documentation** | https://docs.influxdata.com/influxdb/latest/ | https://docs.mongodb.com/manual/ |
| **Initial Release** | 2013 | 2009 |
| **Latest Release** | v1.4.2, November 2017 | v3.6.2, January 2018 |
| **License** | Open Source, MIT | Open Source, AGPL |
| **Language** | Go | C/C++ |
| **Operating Systems** | Linux, OS X | Linux, OS X, Windows |
| **Data Access APIs** | HTTP Line Protocol, JSON, UDP | JSON, BSON |
| **Schema** | Schema-free | Schema-free |

# Summary

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 MongoDB in write throughput, on-disk compression, and query performance.***

# 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 1,000 servers over a 24-hour period, which represents a relatively modest deployment.

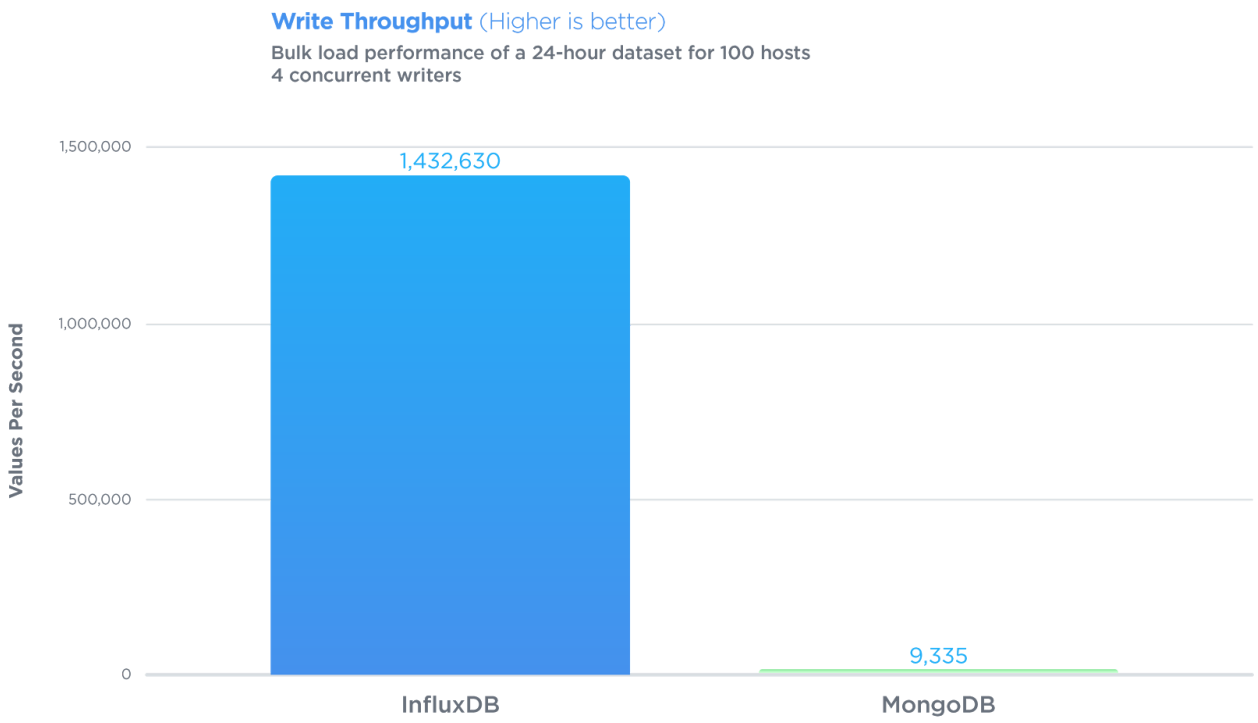
### Overview of the parameters for the sample dataset

| | |
|---|---|
| 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 detail, 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 4 worker threads (to be able to compare to the other databases tests). We found that the average throughput of MongoDB was **9,335 values per second**. The same dataset loaded into InfluxDB at a rate of **1,432,630 values per second**, which corresponds to approximately **153.47x faster ingestion** by InfluxDB (Please note: the concurrency for this test was 4 with 100 hosts reporting).

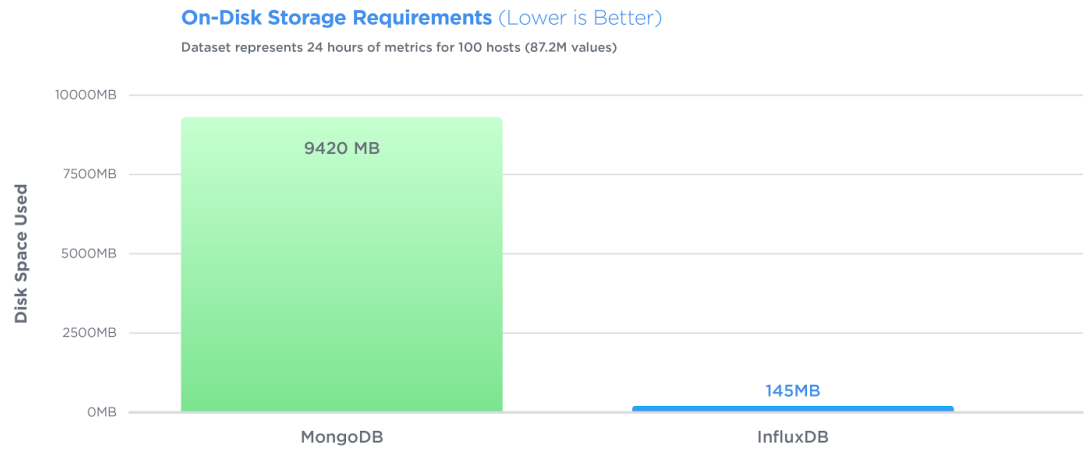



CONCLUSION:

> ***InfluxDB outperformed MongoDB by 153.47x when it came to data ingestion.***

# 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 **9.2 GB** of storage for MongoDB. The same dataset required only **145 MB** for InfluxDB, corresponding to **64.97x better compression** by InfluxDB. This results in approximately 1.74 bytes per value for InfluxDB and 204 bytes per value for MongoDB.



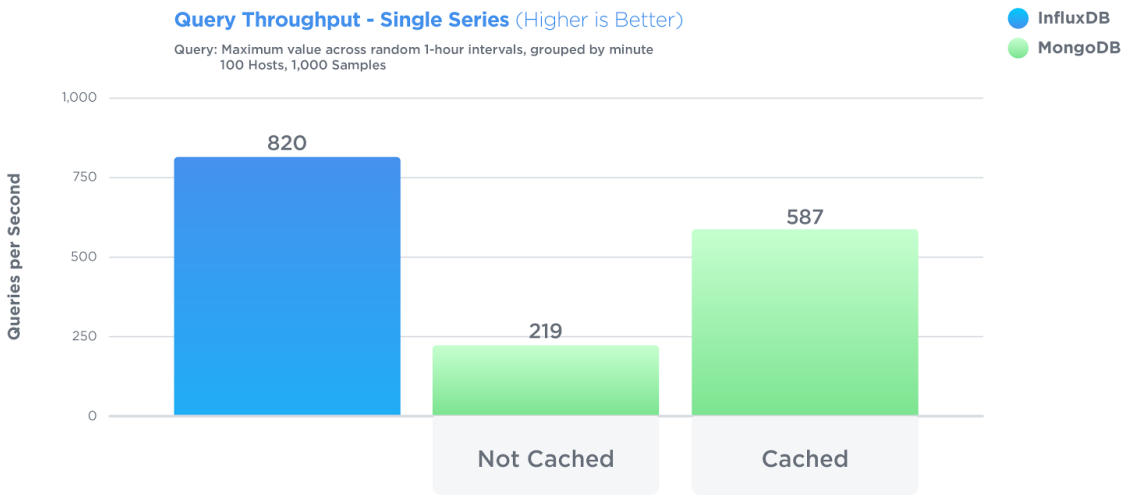

Largely, the additional storage requirement for MongoDB comes from its design as a document-oriented database. Without the time series-optimized compression for values and timestamps that InfluxDB has with its TSM storage engine, MongoDB ends up requiring considerably more space on disk to store the same dataset.

CONCLUSION:

> ***InfluxDB outperformed MongoDB by delivering 64.97x 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 reduce variability, the query times were averaged over 1,000 runs. With concurrent 4 worker threads, we found that the mean query response time for MongoDB was **4.56ms (219.19 queries/sec)**. The same query took an average of **1.22ms (820 queries/sec)** on InfluxDB, demonstrating approximately **3.73x** higher performance. As concurrency increased, the performance difference remained relatively consistent.

Average query response time of the second run -- cached results -- took 1.70ms (**587 queries per sec**).

CONCLUSION:

> ***InfluxDB outperformed MongoDB by delivering up to 3.73x faster query performance.***

# Testing Hardware

All of the tests performed were conducted on two dedicated machines running Ubuntu 16.04 LTS, on AWS infrastructure node image c4.4xlarge: Intel Xeon E5-2666 v3 2.9GHz, 16 vCPU, 30GB RAM, 1x EBS Provisioned 6000 IOPS SSD 120GB. For each test, one machine served as the data load and query client, and the other ran only the database server. Both machines were connected within a 1.29 Gbits/sec network.

# User Experience Comparison

## Overview

MongoDB is a general-purpose document store. MongoDB is intended to store "schema-less" data, in which each object may have a different structure. In practice, MongoDB is typically used to store large, variable-sized payloads represented as JSON or BSON objects.

Both because of MongoDB's generality, and because of its design as a schema-less datastore, MongoDB does not take advantage of the highly-structured nature of time series data. In particular, time series data is composed of tags (key/value string pairs) and sequences of time-stamped numbers (which are the values being measured). As a result, MongoDB must be specifically configured to work with time series data.

In comparison, InfluxDB is a special-purpose Time Series Database. Thus, it automatically takes advantage of the structure of time series data.

## Mental Models

MongoDB is not out-of-the-box ready for time series usage. It requires users to be conversant in its particular lexicon. The vocabulary of a MongoDB system is focused on "collections", "documents", "arrays", and "indexes". None of these map directly to time series data. In all cases, engineering tradeoffs must be made to achieve that mapping.

In contrast, InfluxDB directly uses time series concepts like "measurements", "tags", and "values". This is because InfluxDB is specifically designed to make it easy to store and analyze time series data.

As such, InfluxDB is much simpler to think about when working in the domain of time series data. The cognitive burden on an InfluxDB user is much lower than when using MongoDB.

## System Configuration

Upon startup, the MongoDB process advises the server administrator to disable transparent hugepages. We configured our machines to match this recommendation.

## Inapplicable Advice from MongoDB Documentation

There are many official blog posts and documentation articles that attempt to recommend the best ways to use MongoDB for time series data. However, once the full time series storage problem is considered, those recommendations no longer apply.

For example, we commonly saw advice to use a multi-level data storage strategy, in which each MongoDB document stores an array of point data. The stated goal of this design is to choose a trade-off between the number of separate documents in a collection, the raw data size, and the index size. Specifically, each document would represent a predetermined span of time (typically 60 seconds). Within that document, the array would be pre-allocated to represent at most "one point per second".

There are numerous problems with this approach: 1) It does not permit variable-rate data, which is common in realistic workloads. 2) It does not permit storing different tags with each value, which is critical for performing ad-hoc querying. 3) It requires application-specific tuning, which is brittle and stressful to maintain.

## Schema Design

Designing a time series application with MongoDB requires significant up-front engineering investment. The decisions made in the planning stage of a MongoDB-based application will have long-lasting impacts on what can be done with the data. For example, some faster configurations store tag data in separate collections, which can make it awkward or impossible to perform ad-hoc querying. The choices to be made include: How many MongoDB collections to use? How will numeric values be stored (all as float, or some as integers and some as floats)? How much to rely on MongoDB's generic compression, versus use an application-specific optimization? To save disk space, should tags be normalized, which introduces coordination problems and is not an idiomatic MongoDB design? Is the write-throughput benefit of using multiple collections worth the complexity? If using many collections, how to create them without race conditions?

The schema that we developed is derived from first principles to most correctly and performantly map MongoDB to the time series use case, with particular attention paid to supporting ad-hoc querying. We rely on the copy-on-write and Snappy compression features of MongoDB's new WiredTiger storage engine. In MongoDB, our time series data uses one document per value. Each document contains the name of the measurement (e.g. "cpu" or "redis"), the name of the field (e.g. "usage_user" or "mem_fragmentation_ratio"), the set of tags as an array of objects (e.g. `{key: "hostname", val: "host_42"}`), the timestamp in nanoseconds, and the numeric value being stored. All points are stored in one collection. This design is maximally flexible at query time, allowing ad-hoc analysis similar to that of InfluxDB, but has the tradeoff of

causing a huge amount of disk space to be used. Also, in absolute terms, write throughput is low.

After experimentation, we chose to use a single compound index, which has the following form: `[measurement, tags, field, timestamp_ns]`.

Notably, although MongoDB supports so-called "covered queries" to reduce disk seeks, they do not apply in the time series case because tag data requires a multikey index.

We chose one configuration in the space of possible MongoDB configurations. Much thought was put into this design, and we strove to make MongoDB perform the best way we knew how.

In contrast, InfluxDB required zero schema design.

## Compression and Disk Usage

MongoDB uses a configurable compressor for raw collection data. In particular, the WiredTiger engine supports Snappy, which we used. Indexes are compressed with prefix compression. We checked our index statistics to verify that compression was being used. Unfortunately, these generic compression methods were not enough to mitigate the disk space needed by MongoDB. In our benchmarks, MongoDB required over an order of magnitude more storage space than InfluxDB.

In contrast, InfluxDB uses compression algorithms that are tailored for time series data. This results in less disk space usage.

## Ad-hoc Query Composition

MongoDB requires the user to construct short "aggregation" programs to analyze data. These are JSON documents that specify hybrid imperative/declarative computation over the data in a collection. For example, here is an example aggregation query for MongoDB:

```
db.point_data.aggregate(
   [
      {
           $match: {
               measurement: "cpu",
               timestamp_ns: { $gte: 1451607326000000000, $lt:
1451610926000000000},
               field: "usage_user",
               tags: {$in: [{key: "hostname", val: "host_0"}]},
           },
      },
```

```
      {
        $project: {
           _id: 0,
           time_bucket: { $subtract: ["$timestamp_ns",  { $mod:
["$timestamp_ns", 60e9] } ] },
           field: 1,
           value: 1,
           measurement: 1
        }
      },
      {
        $group : {
           _id : {time_bucket: "$time_bucket", tags: "$tags"},

           max_value: { $max: "$value" }
        }
      },
      {
        $sort : { '_id.time_bucket': 1 }
      }
   ]
)
```

In contrast, the equivalent InfluxQL query is:

```
SELECT max(usage_user) from cpu where hostname = 'host_0' and time >=
'2016-01-01T00:15:26Z' and time < '2016-01-01T01:15:26Z' group by time(1m)
```

Clearly, InfluxQL is more succinct. The reason for this is that InfluxQL is specifically designed for ad-hoc analysis of time series data.

## Go Language Support

As of this writing, the canonical MongoDB driver for the Go language is *mgo*. Although easy to use, it imposes a mandatory serialization overhead on the user. In particular, it forces all queries to be serialized to BSON at query time, instead of ahead-of-time. This prevented us from squeezing out as much performance as we would have liked from the query benchmarking software.

In contrast, InfluxDB uses a straightforward HTTP API that is amenable to ahead-of-time generation. Also, InfluxDB has a bulk protocol that allows clients to perform zero heap allocations on the write path when using a standard HTTP client library.

### User Experience Conclusion

In summary, MongoDB is a general purpose document-oriented database, and InfluxDB is a special-purpose Time Series Database. As a result, InfluxDB is orders of magnitude more convenient to use when storing and analyzing time series data.

# Summary

In the course of this benchmarking paper, we looked at the performance of InfluxDB and MongoDB 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 MongoDB in data ingestion and on-disk storage by a significant margin. Specifically:

- InfluxDB outperformed MongoDB by 153.47x when it came to data ingestion
- InfluxDB outperformed MongoDB by delivering 64.97x better compression
- InfluxDB outperformed MongoDB by up to 3.73x when measuring query performance

InfluxDB and MongoDB performed similarly on query response time as concurrency increased.

It's also important to note that configuring MongoDB to work with time series data wasn't trivial. It requires up-front decisions about how to structure your collections and data types, which can be very time consuming and will have long-lasting impacts on how you can interact with your data and what types of queries you can run. InfluxDB, on the other hand, is ready to use for time series workloads out-of-the-box with no additional configuration.

In conclusion, we highly encourage developers and architects to run these benchmarks themselves to independently verify the results on their hardware and datasets of choice. However, for those looking for a valid starting point on which technology will give better time series data ingestion, compression and query performance "out-of-the-box", InfluxDB is the clear winner across all these dimensions, especially when the datasets become larger and the system runs over a longer period of time.

# What's Next?

### InfluxDB Documentation, Downloads & Guides

- 1.4.2 Download
- 1.4 Installation Guide
- 1.4 Getting Started
- 1.4 Schema Design
- 1.4 Line Protocol Reference
- 1.4 Key Concepts

### Benchmarking Resources

- Benchmarking Code, Methodology and Documentation on GitHub

### Have Questions or Need Help?I

- Community
- Technical Support
- Virtual Training