influxdata®

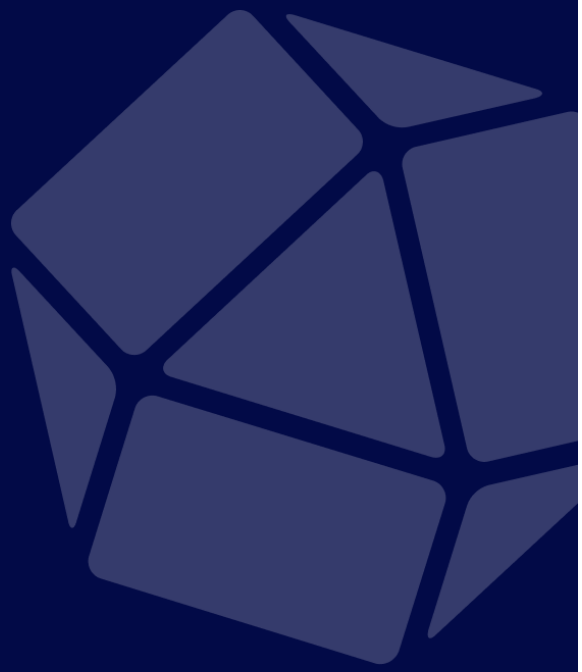
AN INFLUXDATA CASE STUDY

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

Vlasta Hajek
Senior Software Engineer, Bonitoo

Ales Pour
Engineer, Bonitoo

Ivan Kudibal
Engineering Manager, Bonitoo



OCTOBER 2022 REVISION 4

Introduction

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. 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 MongoDB. 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.

Some of the typical characteristics of a purpose-built time series database 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	MongoDB
Description	Database designed for time series, events and metrics data management	Scalable, document-oriented NoSQL database; **Dedicated storage engine for time series data**
Website	https://influxdata.com/	https://www.mongodb.com/
GitHub	https://github.com/influxdata/influxdb	https://github.com/mongodb/mongo
Initial Release	2013	2009
Latest Release	v1.8.10, October 2021	v5.0.6, January 2022

License	Open Source, MIT	Open Source, SSPL
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

Overview

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 100 servers over a 24-hour period, which represents a relatively modest deployment.

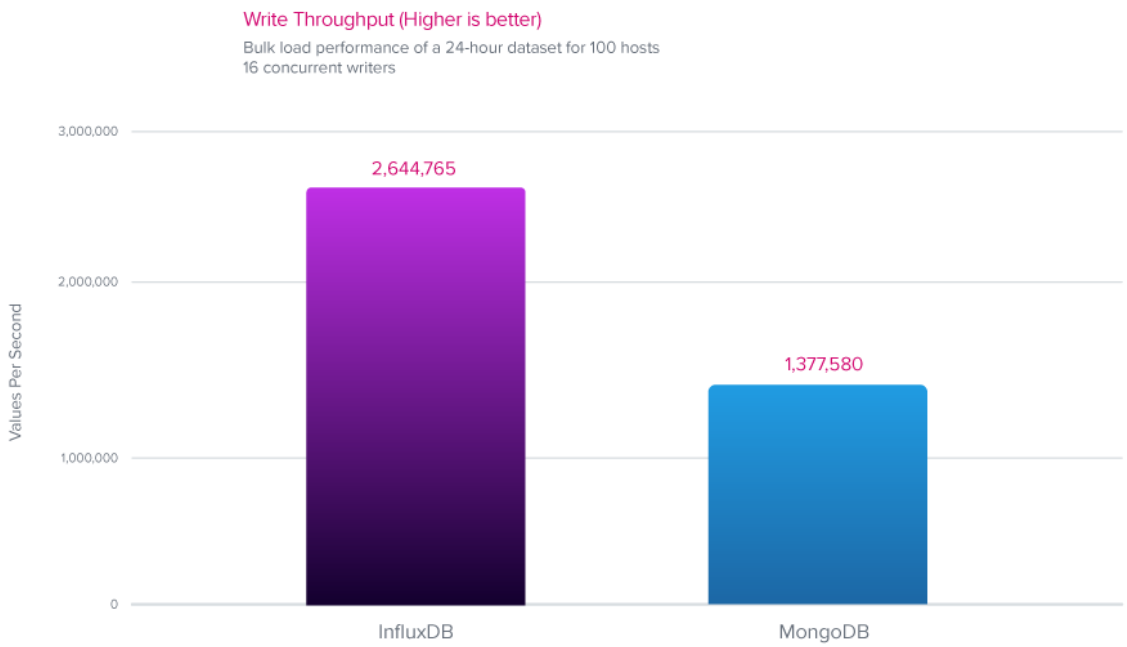
Overview of the parameters for the sample dataset

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 database tests). We found that the average throughput of MongoDB was **1,377,580 values per second**. The same dataset loaded into InfluxDB at a rate of **2,644,765 values per second**, which corresponds to approximately **1.9x faster ingestion** by InfluxDB (Please note: the concurrency for this test was 16 with 100 hosts reporting).



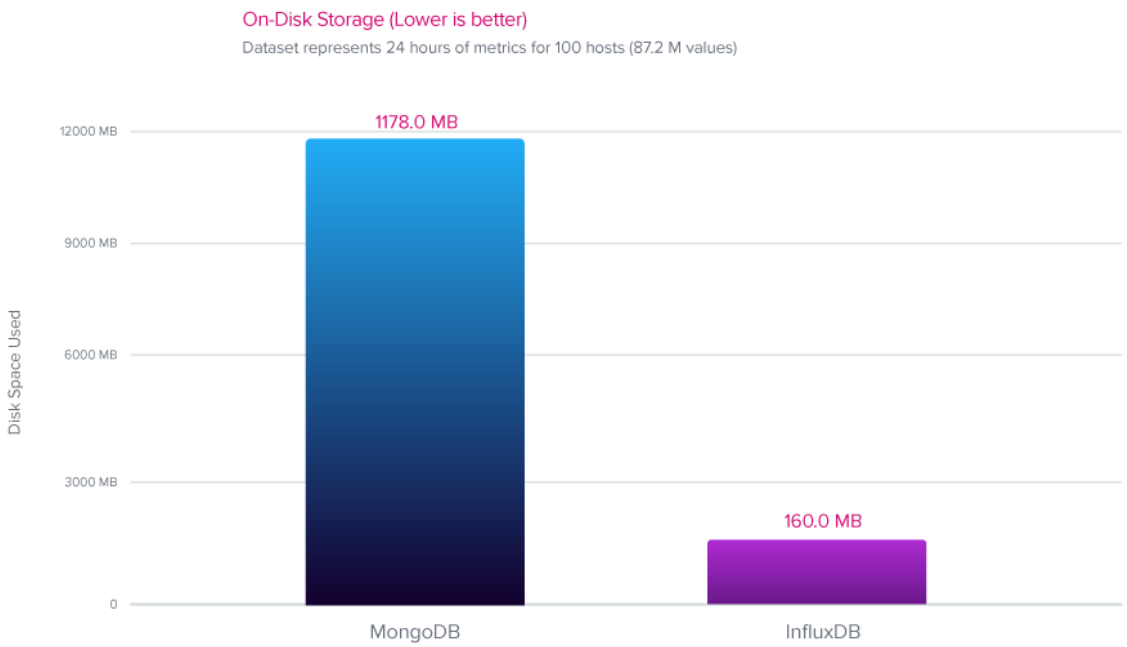
This write throughput stays relatively consistent across larger datasets (i.e. 48 hours, 72 hours, 96 hours).

CONCLUSION:

> ***InfluxDB outperformed MongoDB by 1.9x 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 **1178.0 MB for MongoDB**. The same dataset required **160.0 MB for InfluxDB**, corresponding to **7.3 better compression by InfluxDB**. This results in approximately 2.15 bytes per value for InfluxDB and 14 bytes per value for MongoDB.

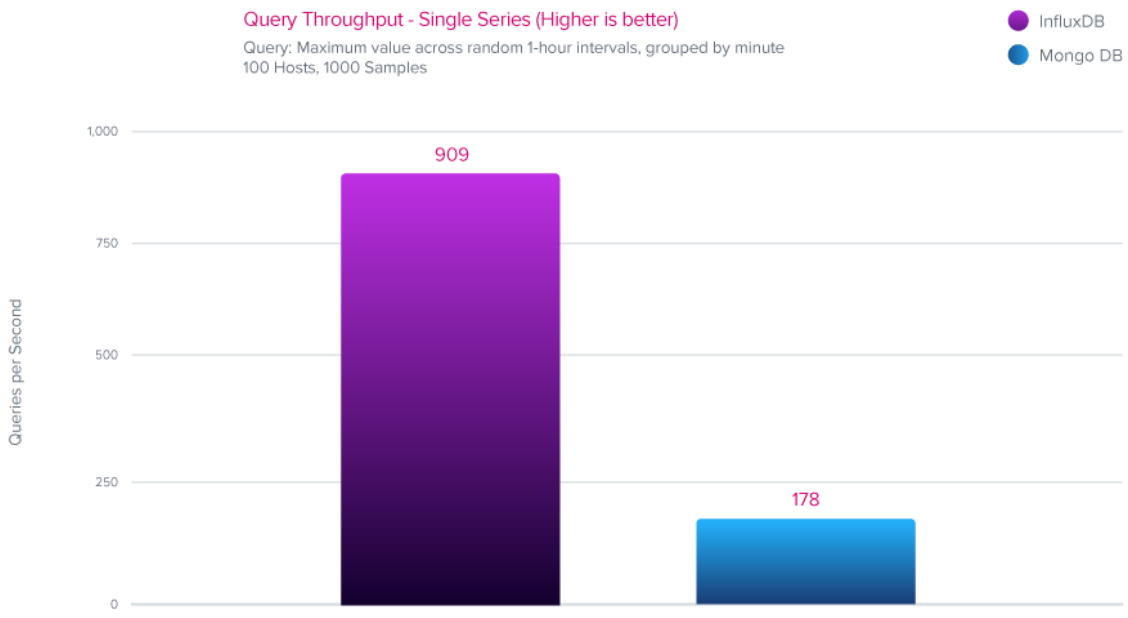


CONCLUSION:

> ***InfluxDB outperformed Mongo by delivering 7.3x 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 a single worker thread, we found that the mean query response time for **MongoDB was 5.6ms (178 queries/sec)**. The same query took an average of **1.1ms (909 queries/sec) for InfluxDB**, demonstrating approximately **5x higher performance**. As concurrency increased, the performance difference remained relatively consistent.

CONCLUSION:

> ***InfluxDB outperformed MongoDB by delivering up to 5x 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

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.

As of version 5, MongoDB has added a dedicated storage engine for working with time series data. These time series collections can be queried similarly to a normal MongoDB collection but do have several tradeoffs and limitations. For example, the types of secondary indexes that can be used with MongoDB's time series collections are restricted. The ability to update and delete data is also limited compared to conventional MongoDB collections. Features like database triggers, Atlas Search, GraphQL API, change streams, and schema validation are also not supported by time series collections.

In comparison, InfluxDB is a special-purpose time series database. Thus, it automatically takes advantage of the structure of time series data. InfluxDB also includes usability features for downsampling data to lower granularity after a set period of time while MongoDB only supports outright deleting and evicting data after a defined time period. InfluxDB also supports workloads at nanosecond timestamp precision while MongoDB is limited to millisecond precision.

System configuration

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

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 floats, or some as integers and some as floats)? How much to

rely on MongoDB's default 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? Another approach could be to store each measurement in a separate collection. Such design would then face limitations such as lack of querying/aggregation across multiple collections.

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 `ztsd` compression features of MongoDB's new WiredTiger storage engine. In MongoDB, our time series data uses one document per multiple values for a measurement with the same set of tags and timestamp, ie. it corresponds to a point in InfluxDB protocol. Each document contains the name of the measurement (e.g. "cpu" or "redis"), the set of tags as an array of objects (e.g. `{ "key": "hostname", "val": "host_42"})`, the set of fields for given measurement (e.g. `{ "usage_user": 65, "usage_system": 7, ... }`) as a subdocument, and the timestamp in nanoseconds. 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 large amount of disk space to be used. Also, in absolute terms, write throughput is low with higher cardinality.

We used a time series collection created with a time series field set to `{ timeField: "timestamp", metaField: "tags" }`. After experimentation, we chose to use a single secondary compound index, which has the following form: `[{ "tags": 1 }, { "timestamp": 1 }]`. Since time series collection can only index timeField and metaField fields, we included measurement with tags. Otherwise, we would prefer to have measurement a top-level document field and create compound index on measurement, tags and timestamp fields.

Another limitation is that text index type would be more appropriate for tags, but is not supported for time series collections as of now.

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 uses `zstd` by default for time series. Indexes are compressed with prefix compression. We checked our index statistics to verify that compression was being used. In our benchmarks, MongoDB required almost 2x more storage space than InfluxDB. Please note that although these benchmarks were run against MongoDB 5.0.6, starting with MongoDB 5.2, column-based compression was added which allows for more specialized compression algorithms to be used for each data type in a time series. However, maximizing the performance requires adding application logic to handle things like removing fields if arrays or objects are empty for certain data points.

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: {
               "tags.measurement": "cpu",
               timestamp: { $gte: ISOdate(”2016-01-01T00:15:26Z”), $lt:
ISODate(”2016-01-01 01:15:26”) },
               ”tags.hostname”: { $in: ["host_42"] },
           },
      },
      {
        $project: {
           _id: 0,
           time_bucket: { $dateTrunc: { date: ”$timestamp”, unit:
”minute” }},
           ”fields.usage_user”: 1,
        }
      },
```

```
      {
            "$unwind": "$fields",
       },
      {
        $group : {

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

In contrast, the equivalent InfluxQL query is:

```
SELECT max(usage_user) from cpu where hostname = 'host_42' 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.

And the equivalent Flux query is:

```
from(bucket: "telegraf")
  |> range(start: 2016-01-01T00:15:26Z, stop: 2016-01-01T01:15:26Z)
  |> filter(fn: (r) => r["_measurement"] == "cpu")
  |> filter(fn: (r) => r["_field"] == "usage_user")
  |> filter(fn: (r) => r["cpu"] == "cpu-total")
  |> filter(fn: (r) => r["host"] == "host_42")
  |> aggregateWindow(every: 1m, fn: max, createEmpty: false)
  |> yield(name: "max")
```

Go language support

As of this writing, the canonical MongoDB driver for the Go language is *mongo*. 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 with recently added but still somewhat limited time series support, 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 1.9x** when it came to data ingestion
- InfluxDB outperformed MongoDB by delivering **7.3x better compression**
- **InfluxDB outperformed MongoDB by up to 5x** 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.

InfluxDB documentation, downloads & guides

Get InfluxDB

Try InfluxDB Cloud for Free

Get documentation

Additional tech papers

Join the InfluxDB community

influxdata®

Try InfluxDB

Get InfluxDB

Contact us for a personalized demo influxdata.com/get-influxdb/