influxdata

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

AN INFLUXDATA TECHNICAL PAPER

External Contributors

Vlasta Hajek Senior Software Engineer, Bonitoo

Ales Pour Engineer, Bonitoo

Ivan Kudibal Engineering Manager, Bonitoo

December 2018 (Revision 5)

Overview

In this this technical paper, we'll compare the performance and features of InfluxDB and Elasticsearch 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 the 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 Elasticsearch. 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.

About InfluxDB

InfluxDB Version Tested: v1.7.2

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 collecting, storing, retrieving and processing 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 Elasticsearch

Elasticsearch Version Tested: v6.5.0

Elasticsearch is an open source search server written in Java and built on top of Apache Lucene. It provides a distributed, full-text search engine suitable for enterprise workloads. While not a *Time Series Database* per se, Elasticsearch employs Lucene's column indexes, which are used to efficiently aggregate numeric values. Combined with query-time aggregations and the ability to index on timestamp fields (which is also important for storing and retrieving log data), Elasticsearch provides the primitives for storing and querying 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 full-text search or log management use cases and therefore will not be explored in this paper. For these use cases, we recommend sticking with Elasticsearch or similar full-text search engines.

- 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	Elasticsearch
Description	Database designed for time series, events and metrics data management	Full-text search engine based on the Apache Lucene project
Website	https://influxdata.com/	https://www.elastic.co/
GitHub	https://github.com/influxdata/influxdb	https://github.com/elastic/elasticsearch
Documentation	https://docs.influxdata.com/influxdb/latest/	https://www.elastic.co/guide/index.html
Initial Release	2013	2010
Latest Release	v1.7.2, November 2018	v6.5.0, November 2018
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, binary protocol (Java)
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 ElasticSearch in write throughput, on-disk compression, and query performance.***

Since Elasticsearch is a full-text search server and not intended for time series data out of the box, some configuration changes are recommended by Elastic for storing these types of metrics. In our testing, we found that these changes:

- Didn't have an impact on write or query performance
- Did make a difference in storage requirements

We'll cover this impact in more detail in a later section.

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.

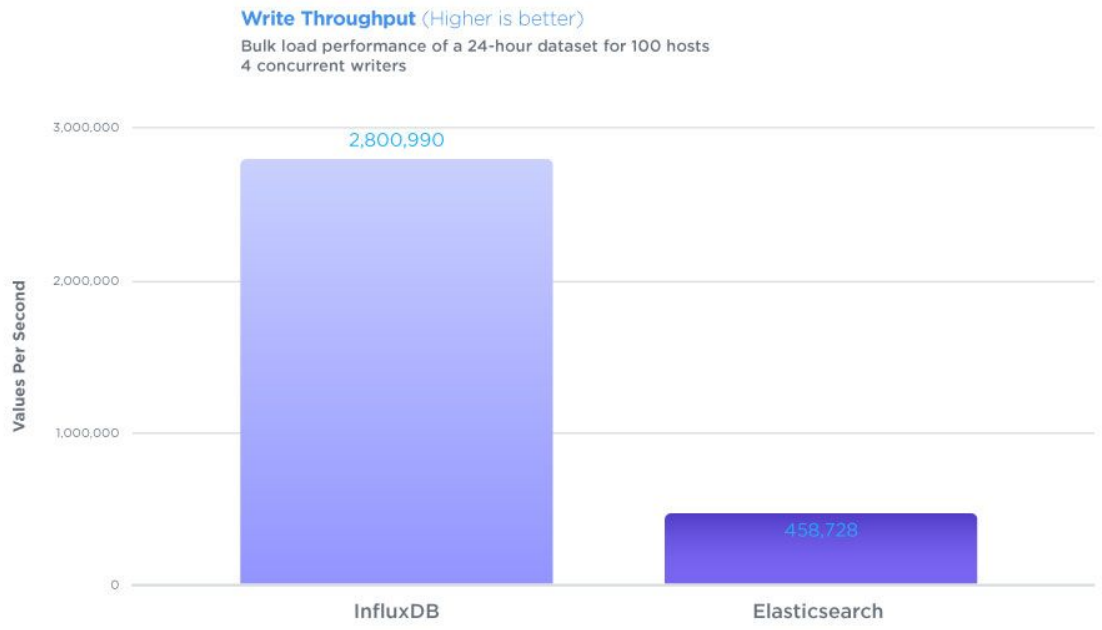
Overview of the parameters for the sample dataset

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'll 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. We found that the average throughput of Elasticsearch was **458,728 values per second** (using the aggregation template, more details below) . The same dataset loaded into InfluxDB at a rate of **2,800,990 values per second**, which corresponds to approximately **6.1x 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).

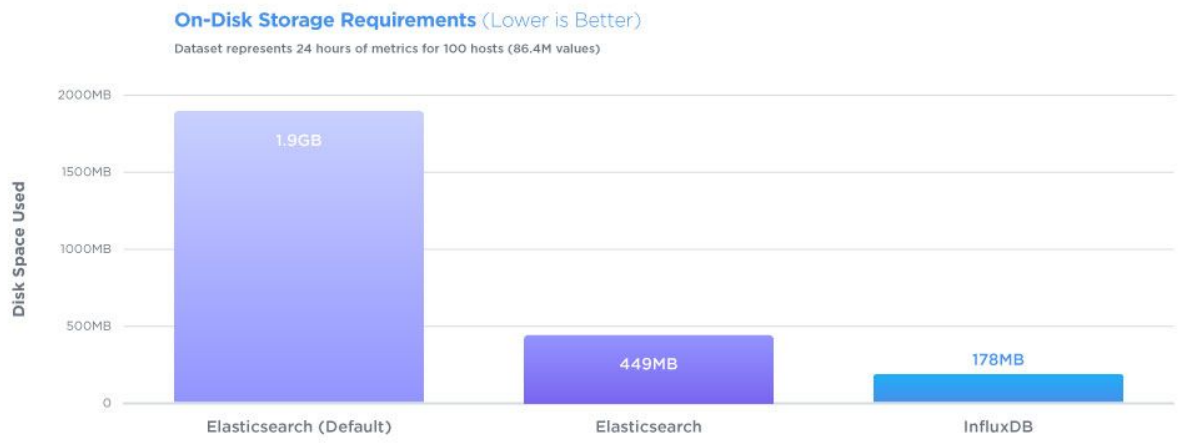
CONCLUSION:

> ***InfluxDB outperformed Elasticsearch by 6.1x when examining data ingestion.***

On-Disk Storage Requirements

As mentioned above, we chose to utilize Elasticsearch in the recommended configuration for time series data. However, we also wanted to give some insight into how the storage requirements compared against the default Elasticsearch configuration as well.

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 **449 MB** for Elasticsearch with the aggregate schema and **1.9 GB** for Elasticsearch with the default schema. The same dataset required only **178 MB** for InfluxDB, corresponding to **2.5x and 10x better compression** by InfluxDB, respectively. This results in approximately 2,15 bytes per value for InfluxDB and 5.39 bytes per value for Elasticsearch (23.4 for the default schema).



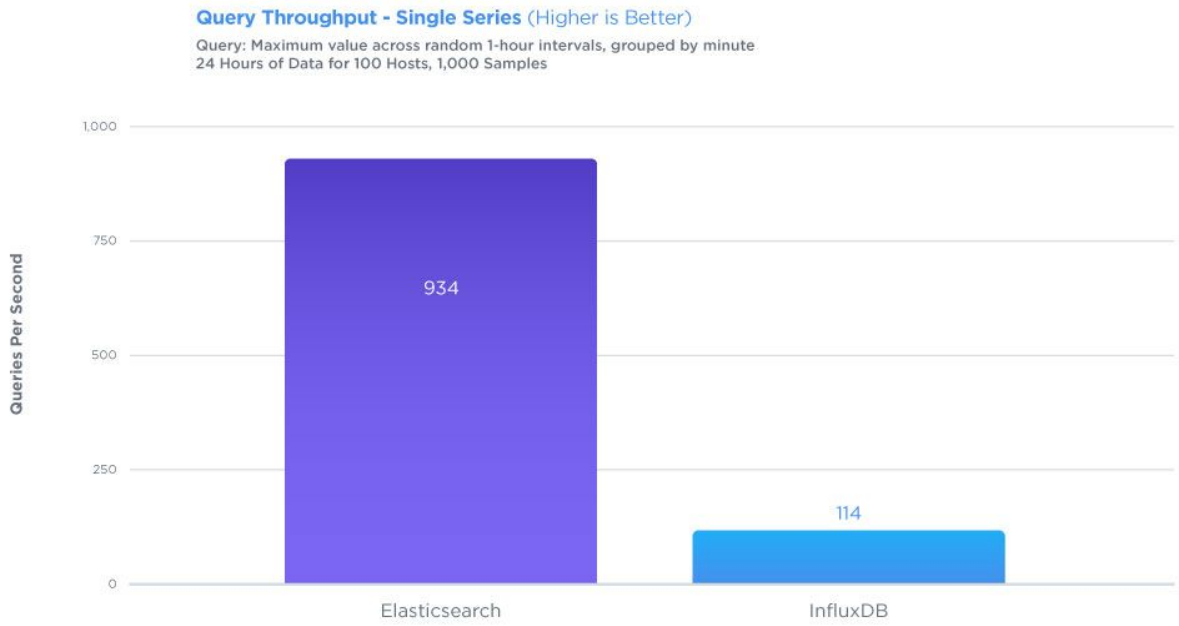
Largely, the additional storage requirement for Elasticsearch with the default configuration comes from the persistence of the `_source` data, which is a byproduct of full-text search features such as highlighting, where the original source document is required. However, even with that data discarded, the Lucene-based DocValues storage format provided by Elasticsearch gives sub-optimal compression when compared to InfluxDB for time series workloads.

CONCLUSION:

> ***InfluxDB outperformed Elasticsearch by delivering 2.5x 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 1 worker thread, we found that the mean query response time for Elasticsearch was **8.73ms** (114 queries/sec). The same query took an average of **1.07ms** (934 queries/sec) on InfluxDB, demonstrating approximately **8.2x** faster query performance than Elasticsearch.

CONCLUSION:

> ***InfluxDB responded 8.2x faster to query requests.***

Testing Hardware

All of the tests performed were conducted on two virtual machines in AWS, running Ubuntu 16.04 LTS. We used the 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 Elasticsearch differ in two key ways: syntax and convenience, and mental models. Elastic was designed for full-text search while InfluxDB was designed with time series as a first-class citizen. This section of the paper is largely subjective so your mileage may vary.

Syntax and Convenience

Elasticsearch's query language is JSON. This can be both good and bad: while it's immediately readable for most developers, hand-writing queries in JSON might feel awkward. For example, remembering to skip final commas when writing JSON arrays could be frustrating.

Additionally, the Elasticsearch HTTP API allows many syntactically-valid JSON requests regardless of the intended semantics. This means that if a mistake is made in an index template declaration (by incorrectly nesting an aggregation clause), the server would readily accept the input. For example, in Elasticsearch 5.6.3 and later, the `minimum_should_match` parameter is no longer recognized in certain contexts. However, Elasticsearch would still silently allow it to be included in a query.

InfluxDB's query language, InfluxQL, provides a relatively concise way to work with time series. For example, compare these two logically-equivalent queries:

InfluxDB

```
SELECT mean(usage_user) from cpu where time >= '2018-01-12T04:29:14-08:00' and
time < '2018-01-13T04:29:14-08:00' group by time(1h)
```

Elasticsearch

```
{
  "size" : 0,
  "aggs": {
    "result": {
      "filter": {
        "range": {
          "timestamp": {
            "gte": "2018-01-12T04:29:14-08:00",
            "lt": "2018-01-13T04:29:14-08:00"
          }
        }
      },
      "aggs": {
        "result2": {
          "date_histogram": {
            "field": "timestamp",
            "interval": "1h",
            "format": "yyyy-MM-dd-HH"
          },
          "aggs": {
            "avg_of_field": {
              "avg": {
                 "field": "usage_user"
              }
            }
          }
        }
      }
    }
  }
}
```

Queries in Elasticsearch are more verbose, even for relatively simple tasks.

Another difference between the two databases is type inference. Both databases have fields that are strongly-typed, and that type is inferred from the first value they see for that field.

In Elasticsearch, for example, if a user creates a document with field `foo` set to `bar`, it will correctly infer that field `foo` is a variable-length string field. If another document is then inserted with field `foo` set, the database will reject any value that is not a string.

In Elasticsearch, this type inference can cause unexpected errors. If a document is created with field `bar` set to 1, Elasticsearch can't know what kind of number it is—is it an integer, float, bignum, or some other type? Elasticsearch assumes that numbers without decimal points are integers by default. This can be especially problematic when a value changes from an ambiguous whole number, such as 0, to a nearby floating point value, such as 0.1. In this case, the solution is to always print the decimal point, but it requires more user intervention to avoid this confusion.

In contrast, InfluxDB requires values to conform to a small set of types, each with their own syntax:

Boolean: `true, false`
Integer: `0i, 123i`
Float: `0, 0.0, 123.0`
String: `"foo"`

Because integers are suffixed with an `i`, there is no ambiguity when dealing with numerical values, and no type inference problems. All other values are stored natively as 64-bit floating point numbers.

Mental Models

As noted already, Elasticsearch is a full-text search server which also happens to have a datastore that can be used for time series data. On the other hand, InfluxDB is purpose-built to support time series data.

Elasticsearch's flexibility comes at a price: any particular use case needs to be modeled to correctly utilize the primitives Elasticsearch provides.

For example, while evaluating the differences between Elasticsearch's default indexing template and the recommended configuration for time series data, it was necessary to know all about the details of how Elasticsearch and Lucene store data on disk. The result was a set of design decisions that took into account how Elasticsearch works, the shape of the data, and the expected queries. This end-to-end thinking is needed when configuring any generalized datastore: using it optimally requires knowing how the internal mechanisms work and requires a much steeper learning curve.

InfluxDB requires fewer decisions from the user because it is purpose-built for the time series use case. It makes it easier to think directly in terms of the data, with the concepts of “measurements”, “tags”, and “fields”.

Summary

In the course of this benchmarking paper we looked at the performance of InfluxDB and Elasticsearch 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 Elasticsearch across all three tests by a significant margin. Specifically:

- InfluxDB outperformed Elasticsearch by 6.1x when it came to data ingestion
- InfluxDB outperformed Elasticsearch by delivering 2.5x better compression
- InfluxDB outperformed Elasticsearch by up to 8.2x when measuring query performance

It’s also important to note that configuring Elasticsearch wasn’t trivial—it requires up-front decisions about indexing, heap sizing, and how to work with the JVM. 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

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

Benchmarking Resources

- Benchmarking Code, Methodology and Documentation on GitHub

Have Questions or Need Help?

- Community
- Technical Support
- Virtual Training