Benchmarking TimescaleDB vs. MongoDB for Time-Series Data

A comparison of two databases in terms of scale, data model, insert performance, and query performance

TIMESCALE

Overview

Time-series data is popping up in many places: DevOps and monitoring, industrial manufacturing, financial trading and risk management, sensor data, ad tech, application eventing, smart home, autonomous vehicles, and more. Since the data piles up very quickly, the biggest challenge with storing time-series is scale. While we've done research to prove that SQL is definitely scalable, a lot of people's' first thoughts go to NoSQL (i.e., non-relational databases) when they want to scale.

Enter MongoDB. MongoDB is among the best known NoSQL solutions, emerging at the end of the last decade to become the face of NoSQL and the foundation of a multibillion-dollar company. Initially used as a simple document store for web apps to prototype quickly and scale easily, its proponents now boast its usefulness across a variety of domains, including time-series.

However, is it really the right solution for time-series data? We decided to evaluate for ourselves to answer that question, even trying multiple methods to make sure we were fair to MongoDB. We found that users get 20% higher insert performance, up to 1400x faster queries, and simpler queries when using TimescaleDB vs. MongoDB for time-series data. While MongoDB's JSON-like document store may make it a jack-of-all-trades type of database, and perhaps a master of some (e.g., web applications), time-series is not one of them.

Methodology

We evaluated two methods of using MongoDB 3.6 as a time series database: (1) a naive, document-per-event method and (2) a method recommended by MongoDB users (and MongoDB itself) that aggregates events into hourly documents. The first method has fast writes and is extremely simple to implement, but offers poor query performance and disk space usage.

The second method achieves good query performance on simple queries, but has worse write performance, higher implementation complexity, and fails to deliver good query performance on more complex time-series queries when compared to TimescaleDB.

More specifically, compared to MongoDB, TimescaleDB exhibits:

- Comparable (method 1) to 20% better (method 2) write performance.
- Faster queries, often ranging from 5x to even 1,400x improvements.
- Far simpler implementation, especially when compared to method 2.

MongoDB methods

Before diving into write and read performance numbers, let's take a moment to examine in more detail the two methods we evaluated for storing time-series data in MongoDB.

Method 1: Document per event (aka "Mongo-naive")

As mentioned, we tested two methods for storing time series data in MongoDB. The first, which we'll refer to as "Mongo-naive," is straightforward: each time-series reading is stored as a document. So for our given use case (see setup below) of monitoring CPU metrics, the JSON document looks like this:

```
{
   "measurement" : "cpu",
   "timestamp_ns" : NumberLong("1451606420000000000"),
   "fields" : {
       "usage_user" : 98.15664166391042,
       "usage_guest_nice" : 85.53066998716453,
       …
   },
   "tags" : {
       "hostname" : "host_2019",
       "datacenter" : "us-east-1b",
       …
   }
}
```

Conceptually and in implementation, this method is very simple, so it seems like a tempting route to go: batch all measurements that occur at the same time into one document along with their associated tags, and store them as one document. Indeed, this approach yields very good write performance (and in our tests, noticeably higher than others have reported), and it is easy to implement. However, we'll see later that, even with indexes, the query performance with this method leaves a lot to be desired. Further, this method eats up disk space, using up nearly 50% more than method 2.

Method 2: Aggregate readings hourly per device (aka "Mongo-recommended")

This second method is one that MongoDB itself (and other blogs) recommends—and therefore we call "Mongo-recommended"—when it comes to time series. The idea is to create a document for each device, for each hour, which contains a 60 by 60 matrix representing every second of every minute in that hour. This document is then updated each time a new reading comes in, rather than doing a new document insert:

```
{
 "measurement" : "cpu",
 "doc_id" : "host_15_20160101_00",  // for quick update lookup
 "key_id" : "20160101_00",  // YYYYMMDD_hh
 "tags" : {
    "hostname" : "host_6",
    "datacenter" : "ap-southeast-1b",
    ....
     },
 "events" : [
   [
     {
       "timestamp_ns" : NumberLong("1451606420000000000"),
       "usage_user" : 98.15664166391042,
       "usage_guest_nice" : 85.53066998716453,
       ...
     },
     ... // (59 elements elided)
   ],
   ... // (59 elements elided)
 ]
}
```

This method does make it possible to do some efficient filtering when it comes to queries, but comes with a more cumbersome implementation and decreased (albeit not terrible) write performance. For example, to efficiently manage writes, a client-side cache of which documents are already made needs to be kept so that a more costly "upsert" (i.e. insert if it doesn't exist, otherwise update) pattern is not needed.

Further, while queries for this method are typically more performant, designing the query in the first place requires more effort, especially when reasoning about which aggregate documents can be filtered/pruned.

Finally, this approach limits the granularity of your data. In particular, if you wanted to support millisecond precision, you would have to change the design to aggregate on a minutely basis, as the max document size in MongoDB (16MB) does not lend itself to further nesting. Beyond millisecond precision is probably infeasible.

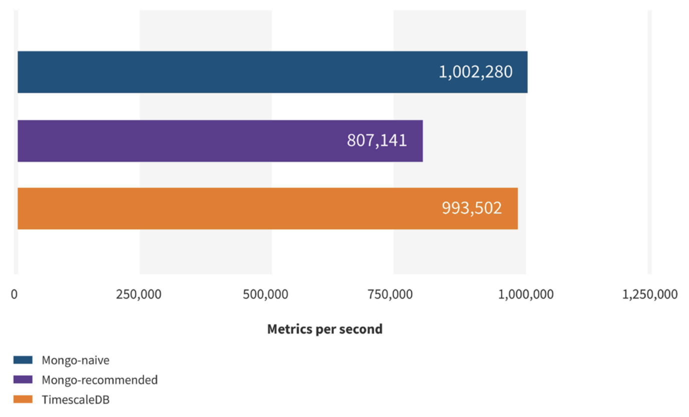
However, because the data in our evaluation was only at the granularity of seconds, and given the query performance we measured, we ultimately decided that this method is probably the best method for comparison against TimescaleDB. We include write performance numbers and some query numbers for Mongo-naive to show how we reached that conclusion.

Write performance & disk usage

Because NoSQL databases typically trade off some guarantees of relational databases, most engineers would expect MongoDB to achieve better write performance/throughput, making it an inviting choice for ingesting time-series data, which can be at a rate of thousands of readings per second.

While it's true that plain PostgreSQL does tend to lose write throughput as the dataset size grows, fortunately TimescaleDB's chunking mechanism keeps write performance high. As a result, we find that our write performance is actually comparable to MongoDB at its fastest (Mongo-naive), as shown in the figure below. These insert numbers were achieved using 8 concurrent clients inserting data into each setup.

Write Performance (MongoDB vs TimescaleDB)



The simplicity of Mongo-naive has it outperforming Mongo-recommended by over 20%, but TimescaleDB is able to achieve similar performance (nearly 1M metrics per second).

All three setups are able to achieve write performance that makes them suitable for time-series data, but Mongo-naive and TimescaleDB certainly stand a cut above. Mongo-recommended experiences extra cost in having to occasionally make new larger documents (e.g., when a new hour or device is encountered).

While Mongo-naive is indeed fast for writes, it suffers significantly when it comes to query performance (as we'll see below). Ultimately that poor performance makes it an impractical setup for time-series data overall.

Mongo-naive also uses more disk space than either Mongo-recommended or TimescaleDB for the full dataset:

Disk Usage

Mongo-naive	31GB
Mongo-recommended	22GB
TimescaleDB	25GB

Therefore, for MongoDB's most practical setup (Mongo-recommended), TimescaleDB achieves write performance that is >20% over MongoDB.

Query performance

Before we compared MongoDB against TimescaleDB, we first considered the query performance between the two MongoDB methods. By this point, Mongo-naive had demonstrated better write performance with a simpler implementation at the cost of additional disk usage, but we suspected that Mongo-recommended would outperform Mongo-naive for query performance. If there were a clear winner between the two methods for MongoDB on simple queries, we could save ourselves some time by not implementing our full query set against both methods.

We first compared the two MongoDB methods using 3 “single rollup” (groupby) queries (on time) and one “double rollup” query (on time and device hostname). The three single rollup queries were run in 500 different permutations (i.e., with random time ranges and hosts) and the double rollup query was run in 100 different permutations, from which we recorded the mean and standard deviation.

Here are the results:

Query Comparison (Mongo-naive & Mongo-recommended)

	Mongo-naive		Mongo-recommended		Ratio (naive / recommended)
	Mean (ms)	Std Dev (ms)	Mean (ms)	Std Dev (ms)	Mean
single-groupby(1,1)	4.55	1.10	6.98	1.22	0.65
single-groupby(1,12)	1,034.22	777.19	47.58	16.32	21.74
single-groupby(8,1)	147.98	55.60	55.37	36.10	2.67
double-groupby	388,327.01	57,359.39	156,543.87	3,300.44	2.48

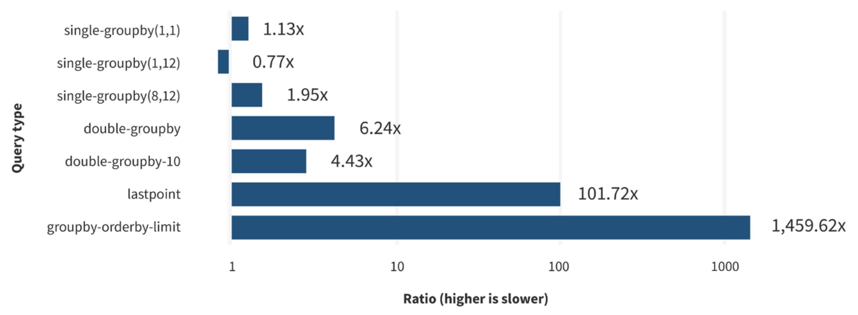
‘single-groupby(1,1)’ retrieves the MAX of one cpu metric per minute for 1 hour across 1 host.
‘single-groupby(1,12)’ retrieves the MAX of one cpu metric per minute for 12 hours across 1 host.
‘single-groupby(8,1)’ retrieves the MAX of one cpu metric per minute for 1 hour across 8 hosts.
‘double-groupby’ retrieves the AVG of one cpu metric per hour per device for 24 hours.

Mongo-recommended out performs Mongo-naive by 2x-20x depending on the query Mongo-naive does outperform Mongo-recommended on the shortest query (shaving a few milliseconds), but aside from that, it is anywhere from 2x slower up to nearly 22x slower than Mongo-recommended.

Given this significant difference in query performance compared to the more modest different in write performance (a loss of 20% for Mongo-recommended), we decided that the recommended approach from MongoDB and others was indeed probably the best setup for time-series data. Thus, the remainder of this post, we use the “Mongo-recommended” setup whenever benchmarking MongoDB.

Here is an overview of query performance:

MongoDB Query Performance slowdown vs TimescaleDB



Single rollups on time

The first set of queries that we will compare are the same first three we looked at when comparing the two methods above:

TimescaleDB v MongoDB: Single rollups on time

	MongoDB	TimescaleDB	Ratio (MongoDB / TimescaleDB)
single-groupby(1,1)	6.98ms	6.15ms	1.13x
single-groupby(1,12)	47.58ms	61.90ms	0.77x
single-groupby(8,1)	55.37ms	28.37ms	1.95x

All queries were run with 500 different parameter permutations across 8 workers.

The results (mean query latency) here are a bit mixed, with TimescaleDB performing better on one, MongoDB performing better on another, and a third being pretty much a tie. All in all, the the raw difference in milliseconds make most of these queries roughly equivalent in a practical sense.

Rollups on time and device

While single rollups are pretty comparable across the two systems, other more complex queries are not. Turning back to the 'double-groupby' query used previously, where the rollup occurs on time andon device name, we can see where TimescaleDB shows big gains.

In the table below, we show one metric being aggregated and also 10 metrics being aggregated. TimescaleDB is between 4–6x faster than MongoDB in these cases. And now that the queries may take 10s of seconds (rather than milliseconds), those kind of differences are very noticeable.

TimescaleDB v MongoDB: Rollups on time and device

	MongoDB	TimescaleDB	Ratio
double-groupby	156,543.87ms	25,097.10ms	6.24x
double-groupby-10	190,725.64ms	43,045.91ms	4.43x

All queries were run with 100 different parameter permutations across 4 workers.

Last point per device & last N before time cutoff

Finally we look at two types of queries where TimescaleDB outperforms MongoDB by an ever wider margin. The first is a query ('lastpoint') that finds the latest reading for every device in the dataset (quite often seen in some use cases, e.g., IoT).

While our dataset has all devices reporting at consistent intervals, this query is troublesome in the general case because it could be that some devices have not reported in quite a long time, potentially causing a lot of documents (MongoDB) or rows (TimescaleDB) to be scanned.

We are able to do some clever query construction in both to get a list of distinct devices which allows both setups to stop searching when every device has a point associated with it. We utilize JOINs in both systems. However, while MongoDB does support JOINs, they are not the bread-and-butter that they are for relational databases like TimescaleDB. Therefore, results were not good for MongoDB:

TimescaleDB v MongoDB: Last point per device & last N before time cutoff

	MongoDB	TimescaleDB	Ratio
lastpoint	46,060.99ms	452.80ms	101.72x

All queries were run with 100 different parameter permutations across 8 workers.

The second complex query we analyzed ('groupby-orderby-limit') does a single rollup on time to get the MAX reading of a CPU metric on a per-minute basis for the last 5 intervals for which there are readings before a specified end time. (This is the type of query one often sees in monitoring workloads.)

This is tricky because those last 5 intervals could be the 5 minutes prior to the end time, or if there is no data for some minute periods (i.e., "gaps") they could be spread out, potentially needing a search from the beginning just to find all 5.

In fact, a full table scan was the query strategy needed for MongoDB, while TimescaleDB has intelligent planning to utilize its indexes, causing a result massively in favor of TimescaleDB:

TimescaleDB v MongoDB: groupby-orderby-limit

	MongoDB	TimescaleDB	Ratio
groupby-orderby-limit	217,336.69ms	148.90ms	1,459.62x

All queries were run with 100 different parameter permutations across 8 workers.

Query language comparison

We also compared the query language differences between TimescaleDB and MongoDB for that last "groupby-orderby-limit" query. As we can see, not only is TimescaleDB much more performant, but the query language (i.e., SQL) is also much simpler and easier to read (a crucial criteria for sustainable software development):

TimescaleDB query (SQL):

```
SELECT date_trunc('minute', time) AS minute, max(usage_user)
FROM cpu
WHERE time < '2016-01-01 19:47:52.646325 -7:00'
GROUP BY minute
ORDER BY minute DESC
LIMIT 5
```

Here's that same query expressed in MongoDB:

MongoDB query:

```
[
 {
   $match: {
     measurement: "cpu",
     key_id: {
       $in: ["20160101_00", "20160101_01", "20160101_02", "20160101_03", "20160101_04", "20160101_05", "20160101_06",
"20160101_07", "20160101_08", "20160101_09", "20160101_10", "20160101_11", "20160101_12", "20160101_13", "20160101_14",
"20160101_15", "20160101_16", "20160101_17", "20160101_18", "20160101_19", "20160101_20", "20160101_21", "20160101_22",
"20160101_23", "20160102_00", "20160102_01", "20160102_02", "20160102_03", "20160102_04", "20160102_05", "20160102_06",
"20160102_07", "20160102_08", "20160102_09", "20160102_10", "20160102_11", "20160102_12", "20160102_13", "20160102_14",
"20160102_15", "20160102_16", "20160102_17", "20160102_18", "20160102_19", "20160102_20", "20160102_21", "20160102_22",
"20160102_23", "20160103_00", "20160103_01", "20160103_02", "20160103_03", "20160103_04", "20160103_05", "20160103_06",
"20160103_07", "20160103_08", "20160103_09", "20160103_10", "20160103_11", "20160103_12", "20160103_13"]
     }
   }
 },
 {
   $project: {
     _id: 0,
     key_id: 1,
     tags: "$tags.hostname",
     events: 1
   }
 },
 {$unwind: "$events"},
 {
   $project: {
     key_id: 1,
     tags: 1,
     events: {
       $filter: {
         input: "$events",
         as: "event",
         cond: {
           $and: [
             {$gte: ["$$event.timestamp_ns", 1451606400000000000]},
             {$lt: ["$$event.timestamp_ns", 1451827606646325489]}
           ]
         }
       }
     }
   }
 },
 {$unwind:$events},
 {
   $project: {
     time_bucket: {
       $subtract: [
         "$events.timestamp_ns",
         {$mod: ["$events.timestamp_ns", 60000000000]}
       ]
     },
     field: "$events.usage_user"
   }
 },
 {
   $group: {
     _id: "$time_bucket",
     max_value: {$max: "$field"}
   }
 },
 {$sort: {_id: -1}},
 {$limit: 5}

]
```

Two things that jump out besides the verbose JSON syntax:

- First, for efficiently stopping the query, the client running the query will have to compute the subset of documents to look in, which creates the lengthy list in the first $match aggregator below.
- Second, to unpack the 60x60 matrices in each document, the $unwind/$project/$unwind pattern is needed to efficiently expand those matrices while removing empty time periods. This pattern is actually needed for almost all of the queries we looked at here, which makes all the queries verbose and potentially daunting to debug.

Summary

In conclusion, users get 20% higher insert performance, up to 1400x faster queries, and simpler queries with TimescaleDB vs. MongoDB.

Is it surprising that a time-series database like TimescaleDB out-performs a general purpose document store when it comes to time-series data? Not exactly, but there are enough blogs, talks, and other material out there showing that people are using MongoDB for time-series data that we thought we needed to do an evaluation.

Understandably, for many users, MongoDB seems to offer the benefit of ease-of-use and a quick setup time. Yet, for time-series data, setting up MongoDB to actually be performant is not simple and requires careful thought about your document design. If you just dump each reading into a new document, you're in for bad time when the data accumulates and you want to start querying it. If you're going to take the time to write all the client-side code for its recommended time-series method, you've already done a lot more work than you would need to setup and start using TimescaleDB.

MongoDB may be a great non-relational document store, but it just isn't that great for time-series data. However, for time-series data with TimescaleDB, you get all the benefits of a reliable relational database (i.e., PostgreSQL) with better performance than a popular NoSQL solution like MongoDB. Based on our analysis, TimescaleDB is the clear choice.

Next steps

In this paper, we performed a detailed comparison of TimescaleDB and MongoDB for time-series data. One of the worst mistakes a business can make is investing in a technology that will limit it in the future, let alone be the wrong fit today. That's why we encourage you to take a step back and analyze your stack before you find your database infrastructure crumbling to the ground.

If you are ready to get started with TimescaleDB, follow the instructions below. If you have any questions along the way, please contact us.

Where to go from here...

- Install TimescaleDB
- Join the Slack channel
- Reach out to hello@timescale.com and let us know how we can help

Additional resources

- Benchmark using the Time Series Benchmark Suite (TSBS)
- Follow our blog for database insights, engineering updates, tutorials, and more