Schema Design for Time Series Data in MongoDB (http://blog.mongodb.org/post/65517193370/schema-design-for-time-series-data-in-mongodb)

OCT 30 • POSTED 1 YEAR AGO

This is a post by Sandeep Parikh (https://twitter.com/crcsmnky), Solutions Architect at MongoDB and Kelly Stirman (https://twitter.com/kstirman), Director of Products at MongoDB.

Data as Ticker Tape

New York is famous for a lot of things, including ticker tape parades.

For decades the most popular way to track the price of stocks on Wall Street was through ticker tape, the earliest digital communication medium. Stocks and their values were transmitted via telegraph to a small device called a “ticker” that printed onto a thin roll of paper called “ a of the ticker lives on in scrolling ele most news networks, sometimes tw

Today there a t observations ordered over time. For example:

- Financial markets generate prices (we still call them “stock ticks”).
- Sensors measure temperature, barometric pressure, humidity and other environmental variables.
- Industrial fleets such as ships, aircraft and trucks produce location, velocity, and operational metrics.
- Status updates on social networks.
- Calls, SMS messages and other signals from mobile devices.
- Systems themselves write information to logs.

This data tends to be immutable, large in volume, ordered by time, and is primarily aggregated for access. It represents a history of what happened, and there are a number of use cases that involve analyzing this history to better predict what may happen in the future or to establish operational thresholds for the system.

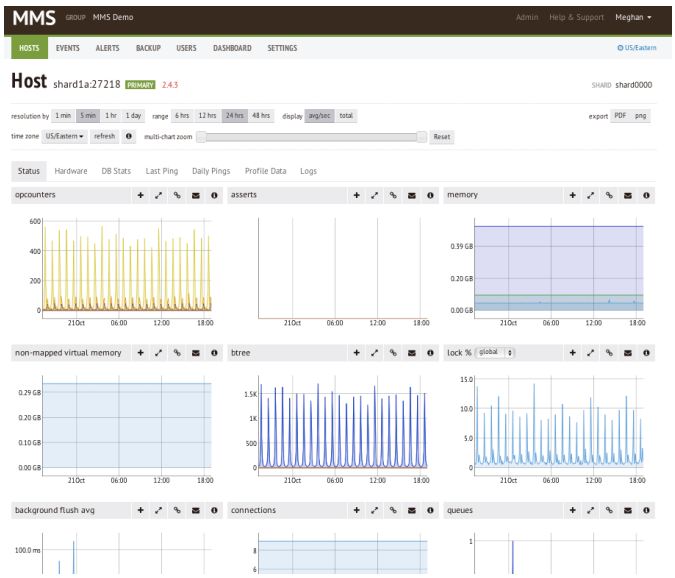
Time Series Data and MongoDB

Time series data is a great fit for MongoDB. There are many examples of organizations using MongoDB to store and analyze time series data. Here are just a few:

- Silver Spring Networks, the leading provider of smart grid infrastructure, analyzes utility meter data in MongoDB.
- EnerNOC (http://www.mongodb.com/presentations/mongodb-boston-2012/mongodb-mission-critical-enterprise) analyzes billions of energy data points per month to help utilities and private companies optimize their systems, ensure availability and reduce costs.
- Square maintains a MongoDB-based open source tool called Cube (http://square.github.io/cube/) for collecting timestamped events and deriving metrics.
- Server Density (http://www.mongodb.com/customers/server-density) uses MongoDB to collect server monitoring statistics.

- Appboy (http://blog.objectrocket.com/2013/11/12/appboy-scales-horizontally-with-objectrocket-mongodb/), the leading platform for mobile relationship management, uses MongoDB to track and analyze billions of data points on user behavior.
- Skyline Innovations (http://www.mongodb.com/presentations/mongodc-2011/time-series-data-storage-mongodb), a solar energy company, stores and organizes meteorological data from commercial scale solar projects in MongoDB.
- One of the world's largest industrial equipment manufacturers stores sensor data from fleet vehicles to optimize fleet performance and minimize downtime.

In this post, we will take a closer look at how to model time series data in MongoDB by exploring the schema of a tool that has become very popular in the community: MongoDB Management Service (MMS) (https://mms.mongodb.com/?pk_campaign=mongodb-org-blog-time-series-10-13). MMS helps users manage their MongoDB systems by providing monitoring, visualization and alerts on over 100 database metrics. Today the system monitors over 25k MongoDB servers across thousands of deployments. Every minute thousands of local MMS agents collect system metrics and ship the data back to MMS. The system processes over 5B events per day, and over 75,000 writes per second, all on less than 10 physical servers for the MongoDB tier.



(https://mms.mongodb.com/?pk_campaign=mongodb-org-blog-time-series-10-13)

Schema Design and Evolution

How do you store time series data in a database? In relational databases the answer is somewhat straightforward; you store each event as a row within a table. Let's say you were monitoring the amount of system memory used per second. In that example you would have a table and rows that looked like the following:

timestamp	memory_used
2013-10-10T23:06:37.000Z	1000000
2013-10-10T23:06:38.000Z	2000000

If we map that storage approach to MongoDB, we would end up with one document per event:

```
{
  timestamp: ISODate("2013-10-10T23:06:37.000Z"),
  type: ”memory_used”,
  value: 1000000
},
{
  timestamp: ISODate("2013-10-10T23:06:38.000Z"),
  type: ”memory_used”,
  value: 15000000
}
```

While this approach is valid in MongoDB, it doesn't take advantage of the expressive nature of the document model. Let's take a closer look at how we can refine the model to provide better performance for reads and to improve storage efficiency.

The Document-Oriented Design

A better schema approach looks like the following, which is not the same as MMS but it will help to understand the key concepts. Let's call it the document-oriented design:

```
{
  timestamp_minute: ISODate("2013-10-10T23:06:00.000Z"),
  type: “memory_used”,
  values: {
    0: 999999,
    …
    37: 1000000,
    38: 1500000,
    …
    59: 2000000
  }
}
```

We store multiple readings in a single document: one document per minute. To further improve the efficiency of the schema, we can isolate repeating data structures. In the ```timestamp_minute``` field we capture the minute that identifies the document, and for each memory reading we store a new value in the ```values``` sub-document. Because we are storing one value per second, we can simply represent each second as fields 0 - 59.

More Updates than Inserts

In any system there may be tradeoffs regarding the efficiency of different operations, such as inserts and updates. For example, in some systems updates are implemented as copies of the original record written out to a new location, which requires updating of indexes as well. One of MongoDB’s core capabilities is the in-place update mechanism: field-level updates are managed in place as long as the size of the document does not grow significantly. By avoiding rewriting the entire document and index entries unnecessarily, far less disk I/O is performed. Because field-level updates are efficient, we can design for this advantage in our application: with the document-oriented design there are many more updates (one per second) than inserts (one per minute).

For example, if you wanted to maintain a count in your application, MongoDB provides a handy operator that increments or decrements a field. Instead of reading a value into your application, incrementing, then writing the value back to the database, you can simply increase the field using $inc:

```
```{ $inc: { pageviews: 1 } }```
```

This approach has a number of advantages: first, the increment operation is atomic - multiple threads can safely increment a field concurrently using $inc. Furthermore, this approach is more efficient for disk operations, requires less data to be sent over the network and requires fewer round trips by omitting the need for any reads. Those are three big wins that result in a more simple, more efficient and more scalable system. The same advantages apply to the use of the $set operator.

The document-oriented design has several benefits for writing and reading. As previously stated, writes can be much faster as field-level updates because instead of writing a full document we’re sending a much smaller delta update that can be modeled like so:

```
db.metrics.update(
  {
    timestamp_minute: ISODate("2013-10-10T23:06:00.000Z"),
    type: ”memory_used”
  },
  {$set: {“values.59”: 2000000 } }
)
```

With the document-oriented design reads are also much faster. If you needed an hour’s worth of measurements using the first approach you would need to read 3600 documents, whereas with this approach you would only need to read 60 documents. Reading fewer documents has the benefit of fewer disk seeks, and with any system fewer disk seeks usually results is significantly better performance.

A natural extension to this approach would be to have documents that span an entire hour, while still keeping the data resolution per second:

```
{
  timestamp_hour: ISODate("2013-10-10T23:00:00.000Z"),
  type: “memory_used”,
  values: {
    0: 999999,
    1: 1000000,
    …,
    3598: 1500000,
    3599: 2000000
  }
}
```
```

One benefit to this approach is that we can now access an hour's worth of data using a single read. However, there is one significant downside: to update the last second of any given hour MongoDB would have to walk the entire length of the "values" object, taking 3600 steps to reach the end. We can further refine the model a bit to make this operation more efficient:

```
{
  timestamp_hour: ISODate("2013-10-10T23:00:00.000Z"),
  type: "memory_used",
  values: {
    0: { 0: 999999, 1: 999999, …, 59: 1000000 },
    1: { 0: 2000000, 1: 2000000, …, 59: 1000000 },
    …,
    58: { 0: 1600000, 1: 1200000, …, 59: 1100000 },
    59: { 0: 1300000, 1: 1400000, …, 59: 1500000 }
  }
}
```

```
db.metrics.update(
  {
    timestamp_hour: ISODate("2013-10-10T23:00:00.000Z"),
    type: "memory_used"
  },
  {$set: {"values.59.59": 2000000 } }
)
```

MMS Implementation

In MMS users have flexibility to view monitoring data at varying levels of granularity. These controls appear at the top of the monitoring page:

These controls inform the schema design for MMS, and how the data needs to be displayed. In MMS, different resolutions have corresponding range requirements - for example, if you specify that you want to analyze monitoring data at the granularity of "1 hr" instead of "1 min" then the ranges also become less granular, changing from hours to days, weeks and months:

To satisfy this approach in a scalable manner and keep data retention easy to manage, MMS organizes monitoring data to be very efficient for reads by maintaining copies at varying degrees of granularity. The document model allows for efficient use of space, so the tradeoff is very reasonable, even for a system as large as MMS. As data ages out, collections that are associated with ranges of time are simply dropped, which is a very efficient operation. Collections are created to represent future ranges of time, and these will eventually be dropped as well. This cycle maintains a rolling window of history associated with the functionality provided by MMS.

In addition, to support the "avg/sec" display option the system also tracks the number of samples collected and the sum of all readings for each metric similar to the following example:

```
{
  timestamp_minute: ISODate("2013-10-10T23:06:00.000Z"),
  num_samples: 58,
  total_samples: 108000000,
  type: "memory_used",
  values: {
    0: 999999,
    …
    37: 1000000,
    38: 1500000,
    …
    59: 1800000
  }
}
```

The fields "num_samples" and "total_samples" are updated as new readings are applied to the document:

```
db.metrics.update(
  {
    timestamp_minute: ISODate("2013-10-10T23:06:00.000Z"),
    type: “memory_used”
  },
  {
    {$set: {“values.59”: 2000000 }},
    {$inc: {num_samples: 1, total_samples: 2000000 }}
  }
)
```

Computing the average/sec is straightforward and requires no counting or processing, just a single read to retrieve the data and a simple application-level operation to compute the average. Note that with this model we assume a consistent cadence of measurements - one per second - that we can simply aggregate at the top of the document to report a rolled-up average for the whole minute. Other models are possible that would support inconsistent measurements and flexible averages over different time frames.

Another optimization used in MMS is preallocating all documents for the upcoming time period; MMS never causes an existing document to grow or be moved on disk. A background task within the MMS application performs inserts of empty “shell” documents including the subdocument schema but with all zeroes for the upcoming time periods before they are recorded. With this approach fields are always incremented or set without ever growing the document in size, which eliminates the possibility of moving the document and the associated overhead. This is a major performance win and another example of ensuring in-place updates within the document-oriented design.

Conclusion

MongoDB offers many advantages for storing and analyzing time series data, whether it’s stock ticks, tweets or MongoDB metrics. If you are using MongoDB for time series data analysis, we want to hear about your use case. Please continue the conversation by commenting on this post with your story.

More Information

- Time Series Data Part 1: Sensor Management (recorded presentation) (https://www.mongodb.com/presentations/mongodb-time-series-data-part-1-setting-stage-sensor-management)
- Time Series Data Part 2: Using the Aggregation Framework and Hadoop (recorded presentation) (https://www.mongodb.com/presentations/mongodb-time-series-data-part-2-analyzing-time-series-data-using-aggregation-framework)
- Time Series Data Part 3: Sharding (recorded presentation) (https://www.mongodb.com/presentations/mongodb-time-series-data-part-3-sharding)
- MongoDB for operational intelligence (http://docs.mongodb.org/ecosystem/use-cases/#operational-intelligence)
- MongoDB Management Service (MMS) (https://mms.mongodb.com/?pk_campaign=mongodb-org-blog-time-series-10-13)
- Webinar Series starting July 2014: MongoDB for Time Series Data (http://www.mongodb.com/lp/webinar-series/time-series-july-2014)

This post was updated in December 2014 to include additional resources and updated links.

Like what you see? Get MongoDB updates straight to your inbox (http://www.mongodb.com/newsletter)

TAGS:

MONGODB (HTTP://BLOG.MONGODB.ORG/TAGGED/MONGODB) | SCHEMA DESIGN (HTTP://BLOG.MONGODB.ORG/TAGGED/SCHEMA-DESIGN) | TIME SERIES (HTTP://BLOG.MONGODB.ORG/TAGGED/TIME-

(http://twitter.com/home?status=Schema Design for Time Series Data (https://www.facebook.com/sharer.php?u=http%3A%2F%2Fblog.mongodb.org%2Fpost%2F65600957178%2Fschema-design-for-time-series-data-in-mongodb) (https://plus.google.com/share?url=http%3A%2F%2Fblog.mongodb.org%2Fpost%2F65600957178%2Fschema-design-for-time-series-data-in-mongodb)

instead of nested array? Maybe you can help me with a more efficient way of my approaches:

```
timestamp_day = '2014-02-07T00:00:00';
items = {'17': {'15': 0.0, '45': 0.0, '0': 0.0, '30': 0.0}, '16': {'15': 0.0, '45': 0.0,
```

Python:

```
data = []
for each in meter_data:
    timestamp_day = each.get('timestamp')
    for (hour, data) in each.get('data').items():
        timestamp = timestamp_day.replace(hour=int(hour))
        for minute, value in data.items():
            timestamp = timestamp_day.replace(minute=int(minute))
            data.append([timestamp, value])
```

see more

5 • Share ›

dmerriman Mod ➚ wiesson • a year ago

in some cases $unwind and $project in an aggregate command might yield what you want

another approach would be to use the built-in map reduce to unfold the data in a javascript function, kind of like the python above but could run server side and emit several date/value pairs. you would then not really reduce just use an identity function for that

1 • Share ›

Alain Scialoja ➚ dmerriman • 4 months ago

In case the timestamps have to be unfolded and the cadence of data is not regular, would it be more performant to use map reduce or to store minutes and seconds as values instead of keys?

1 • Share ›

wiesson ➚ dmerriman • a year ago

Thanks! Do you/MongoDB provide some examples? Another important feature is that I could sum the values per hour, day, week, 2 weeks, month ...

I'm currently testing a lot with NodeJS and Python to build an easy setup to push data to the mongodb and if it is needed transform it into useful charts and reports. I'm looking really forward to replace our MySQL system.

• Share ›

Luc • 5 months ago

Thanks, this is a really great article.

In my use case, I use mongodb for time-series as well but the cadence of measurements are not consistent:
- for some sensors I get data every couple of seconds
- for some other sensors I get data with a non regular step between one measurement and the next one (could be 5 sec, 1 minute, ...)

Is there an approach like the one presented here that could be used for this use case ?

2 • Share ›

Glavin Wiechert • 8 months ago

I am developing an open-source project called Cobra that I am using to experiment with MongoDB and Time Series data. I have implemented a server in Node.js and a graphing web-app using Cubism to see data streaming in real-time.
Check it out if you're interested in learning more about MongoDB and Time-Series data like I am: https://github.com/Glavin001/C...

1 • Share ›

Pieter Willem Jordaan • 5 months ago

How should collection names be used to allow for dropping expired data? At what granularity level do you split on collection level.

• Share ›

dmerriman Mod ➚ Pieter Willem Jordaan • 5 months ago

whatever is convenient for the use case. you won't be able to have a query span

collections, so dividing up too finely might impact your ability to query. and a thousand collections could be a slight dba hassle (but would work).

for example if i need 90 days of history, i might have a collection per month, and thus have up to 4 collections at a given point of time; when i start a new month that would become a fifth collection, i can at that time drop the oldest of the 4.

see also capped collections, which auto-expire data, but also have limitations and thus aren't good for the example in the blog post (at least as it is formulated)

Pieter Willem Jordaan ➚ dmerriman • 5 months ago

Thanks for the answer Dwight.
In you experience with some of the use cases you've had to deal with, are TTL collections feasible for, say for instance, this blog post's example?

dmerriman Mod ➚ Pieter Willem Jordaan • 5 months ago

yes, depending on performance requirements. dropping a collection is cheap. even if fairly fast, dropping individual documents, hundreds of millions of them, will take a bit longer / consume more resources in the ejection.

we might ask what about fragmentation given so many deletes, but for mmap storage engine, with powerof2sizes on (defaults on in latest versions), i think it is ok -- and not ok if off. that said, your inserts and such won't be as contiguous physically on disk as they might have been with the drop collection approach. which may or may not matter depending on what you are doing and type of storage. that's for mmap storage engine; wildtiger storage engine will have a different performance profile

Pieter Willem Jordaan ➚ dmerriman • 5 months ago

Once again a prompt and concise answer. Thanks for your recommendations. I will have to do some production simulations in order to determine what best fits my needs.

Kyle • 5 months ago

This is a more general MongoDB question for this schema design. How do you update the <field> value to properly log each new data point?

From the article:

```
db.metrics.update(
{
  timestamp_hour: ISODate("2013-10-10T23:00:00.000Z"),
  type: “memory_used”
},
  {$set: {“values.59.59”: 2000000 } }
)
```

How would I iterate over each data point and update the field operator respectively. Such as updating "values.0.1" then "values.0.2"` etc.

Gabriel • 7 months ago

If you cannot get data from all preallocated dates. How do you distinguish between preallocated and actual zeros?

Nagesh • 10 months ago

Can you show me an example of merging two different time-series ? Suppose, I have two different time series, t1 and t2 that publish data at some arbitrary data. I want a third time series t3, that takes a slice (say, 1 min wide) of t1 and t2 and merges them and finally sorts them ascending.

Camilo López A. • 10 months ago

Hey, great article!. Could you please go deeper on how are you managing collections that represent a period of time? How do you name such collections? How do you decide when to change to the next collection? How do you query this model when you need to retrieve data that is under two or more collections?

• Share ›

crcsmnky ➚ Camilo López A. • 10 months ago

There are different ways to attack this part of the problem and it really comes down to maintaining some level of simplicity without forcing you to do too much work in your app code.

The way MMS Monitoring organizes collections is based on the way the UI is organized and by data retention policies. It uses a similar approach to the following:

For each granularity level (1m, 5m, etc) there are collections for each possible zoom level (1m -> 1hr, 6hr, 12hr, 24hr, 48hr, etc.).

So when a metric comes in, you would deconstruct the timestamp to figure out which collections should be written to and then issue multiple writes for that metric value. The naming convention isn't the most important part, it's the lookup function that takes in a timestamp and returns a list of collections to insert into.

For querying, we just use the user-selected display mode (1m, 5m, etc.) and the zoom level to determine which collection to read from. The purpose here is to manage the complexity on writes and simplify reads.

Bear in mind that this is just one approach. There are certainly other ways of accomplishing this.

1 • Share ›

Jenny Blunt • a year ago

After over a year looking for something to do this, you have saved my bacon :)

Do you have any resources that would explain the most efficient ways to iterate and graph such data?

• Share ›

programmer2123 • a year ago

59: 2000000,

.....

20,000: 100000,
}
}

Is this a better design than a single document for each event, because of the size of the sub-document being unlimited. I am mainly concerned about document growth causing disk fragmentation, see below comment:

Some updates to documents can increase the size of documents. These updates include pushing elements to an array (i.e. $push) and adding new fields to a document. If the document size exceeds the allocated space for that document, MongoDB will relocate the document on disk. Relocating documents takes longer than in place updates and can lead to fragmented storage. Although MongoDB automatically adds padding to document allocations to minimize the likelihood of relocation, data models should avoid document growth when possible.

see more

• Share ›

crcsmnky ➚ programmer2123 • a year ago

It's best to manage your documents to minimize constant growth because of the overhead of relocating documents, which negatively impacts the speed of in-place updates.

Generally, I would avoid having thousands of elements in a sub-document or array and instead consider a more granular "bucket" size. If you absolutely know that you'll need to have this many elements in a sub-document/array and you are worried about growth you could pre-create the structure for this many elements in advance (with all entries in the sub-doc set to 0) but this still is very large to manage. You need to balance not only the document complexity but also the bandwidth needed to send these large documents back and forth. This is why keeping documents with a "bucket" size of a minute or an hour work best.

1 • Share ›

cfarrugia • a year ago

Say i have the following document:

{
timestamp_hour: ISODate("2013-10-10T23:00:00.000Z")

type: “memory_used”,
values: {
0: 999999,
1: 1000000,
…,
3598: 1500000,
3599: 2000000
}
}

How would I go about getting the max value within the hour?

• Share ›

crcsmnky ↗ cfarrugia • a year ago

The best way to accomplish that is via the Aggregation Framework, using $max operator in the $group pipeline stage. See here for some more information: http://docs.mongodb.org/manual...

• Share ›

John L Page ↗ cfarrugia • 8 months ago

Look at the $max UPDATE operator - this means you can add the new value (use multiple dimensions in that array too Minute.second not just second) but at the top you can have maxvalue: and on each update use {$max:{maxvalue:x}} to keep track of the highest seen.

• Share ›

(http://blog.mongodb.org/) ‹ (http://blog.mongo