Time Series Data and MongoDB: Best Practices Guide

February 2019

Table of Contents

Introduction

Time series data is increasingly at the heart of modern applications - think IoT, stock trading, clickstreams, social media, and more. With the move from batch to real time systems, the efficient capture and analysis of time series data can enable organizations to better detect and respond to events ahead of their competitors, or to improve operational efficiency to reduce cost and risk. Working with time series data is often different from regular application data, and there are best practices you should observe. In this whitepaper you will learn the following:

- What is time series data, and some of the challenges associated with this type of data
- How to build a list of discovery questions that help gather technical requirements needed for the successful delivery of a time series application using MongoDB
- How different time series schema designs can affect database performance
- How to query, analyze and present time series data managed by MongoDB

What is time series data?

While not all data is time series in nature, a growing percentage of it can be classified as time series – fueled by technologies that allow us to exploit streams of data in real time rather than in batches. In every industry and in every company there exists the need to query, analyze and report on time series data. Consider a stock day trader constantly looking at feeds of stock prices over time and running algorithms to analyze trends to identify opportunities. They are looking at data over a time interval, e.g. hourly or daily ranges. A connected car company might obtain telemetry such as engine performance and energy consumption to improve component design, and monitor wear rates so they can schedule vehicle servicing before problems occur. They are also looking at data over a time interval.

Why is time series data challenging?

Time series data can include data that is captured at constant time intervals – like a device measurement per second – or at irregular time intervals like those generated from alerts and auditing event use cases. time series data is also often tagged with attributes like the device type and location of the event, and each device may provide a variable amount of additional metadata. Data model flexibility to meet diverse and rapidly changing data ingestion and storage requirements make it difficult for traditional relational (tabular) database systems with a rigid schema to effectively handle time series data. Also, there is the issue of scalability. With a high frequency of readings generated by multiple sensors or events, time series applications can generate vast streams of data that need to be ingested and analyzed. So platforms that allow data to be scaled-out and distributed across many nodes are much more suited to this type of use-case than scale-up, monolithic tabular databases.

Time series data can come from different sources, with each generating different attributes that need to be stored and analyzed. Each stage of the data lifecycle places different demands on a data platform – from ingestion through to consumption and archival.

- During data ingestion, the database is primarily performing write intensive operations, comprising mainly inserts with occasional updates. Consumers of data may want to be alerted in real time when an anomaly is detected in the data stream during ingestion, such as a value exceeding a pre-defined threshold.
- As more data is ingested consumers may want to query it for specific insights, and to uncover trends. At this stage of the data lifecycle, the workload is read, rather than write heavy, but the database will still need to maintain high write rates as data is concurrently ingested and then queried.
- Consumers may want to query historical data and perform predictive analytics leveraging machine learning algorithms to anticipate future behavior or identify trends. This will impose additional read load on the database.
- In the end, depending on the application's requirements, the data captured may have a shelf life and needs to be archived or deleted after a certain period of time.

As you can see working with time series data is not just simply storing the data, but requires a wide range of data platform capabilities including handling simultaneous read and write demands, advanced querying capabilities, and archival to name a few.

Who is using MongoDB for time series data?

MongoDB provides all the capabilities needed to meet the demands of a highly performing time series applications. One company that took advantage of MongoDB's time series capabilities is Quantitative Investment Manager Man AHL.

Man AHL's Arctic application leverages MongoDB to store high frequency financial services market data (about 250M ticks per second). The hedge fund manager's quantitative researchers ("quants") use Arctic and MongoDB to research, construct and deploy new trading models in order to understand how markets behave. With MongoDB, Man AHL realized a 40x cost saving when compared to an existing proprietary database. In addition to cost savings, they were able to increase processing performance by 25x over the previous solution. Man AHL open sourced their Arctic project on GitHub.

Bosch Group is a multinational engineering conglomerate with nearly 300,000 employees and is the world's largest automotive components manufacturer. IoT is a strategic initiative at Bosch, and so the company selected MongoDB as the data platform layer in its IoT suite. The suite powers IoT applications both within the Bosch group and in many of its customers in industrial internet applications, such as automotive, manufacturing, smart city, precision agriculture, and more. If you want to learn more about the key challenges presented by managing diverse, rapidly changing and high volume time

series data sets generated by IoT platforms, download the Bosch and MongoDB whitepaper.

SIEMENS Siemens is is a global company focusing on the areas of electrification, automation and digitalization. Siemens developed "Monet" a platform backed by MongoDB that provides advanced energy management services. Monet uses MongoDB for real time raw data storage, querying and analytics.

Focusing on application requirements

When working with time series data it is imperative that you invest enough time to understand how data is going to be created, queried and expired. With this information you can optimize your schema design and deployment architecture to best meet the application's requirements.

You should not agree to performance metrics or SLAs without capturing the application's requirements.

As you begin your time series project with MongoDB you should get answers to the following questions:

Write workload:

- What will the ingestion rate be? How many inserts and updates per second?

As the rate of inserts increase, your design may benefit from horizontal scaling via MongoDB auto-sharding, allowing you to partition and scale your data across many nodes

- How many simultaneous client connections will there be?

While a single MongoDB node can handle many simultaneous connections from tens of thousands of IoT devices, you need to consider scaling those out with sharding to meet the expected client load.

- Do you need to store all raw data points or can data be pre-aggregated? If pre-aggregated, what summary level of granularity or interval is acceptable to store? Per minute? Every 15 minutes?

MongoDB can store all your raw data if you application requirements justify this. However, keep in mind that reducing the data size via pre-aggregation will yield lower dataset and index storage and an increase in query performance. We demonstrate this later in the guide.

- What is the size of data stored for each event?

MongoDB has an individual document size limit of 16 MB. If your application requires storing larger data within a single document, such as binary files you may want to leverage MongoDB GridFS. Ideally when storing high volume time series data it is a best practice to keep the document size small – around 1 disk block size.

Read workload:

- How many read queries per second?

A higher read query load may benefit from additional indexes or horizontal scaling via MongoDB auto-sharding.

As with write volumes, reads can be scaled with auto-sharding. You can also distribute read load across secondary replicas in your replica set.

- Will clients be geographically dispersed or located in the same region?

You can reduce network read latency by deploying read-only secondary replicas that are geographically closer to the consumers of the data.

- What are the common data access patterns you need to support? For example, will you retrieve data by a single value such as time, or do you need more complex queries where you look for data by a combination of attributes, such as event class, by region, by time.

Query performance is optimal when proper indexes are created. Knowing how data is queried and defining the proper indexes is critical to database performance. Also being able to modify indexing strategies in real time, without disruption to the system is an important attribute of a time series platform

- What analytical libraries or tools will your consumers use?

If your data consumers are using tools like Hadoop or Spark, MongoDB has a MongoDB Spark Connector that

integrates with these technologies. MongoDB also has drivers for Python, R, Matlab and other platforms used for analytics and data science.

- Does your organization use BI visualization tools to create reports or analyze the data?

MongoDB integrates with most of the major BI reporting tools including Tableau, QlikView, Microstrategy, TIBCO and others via MongoDB BI Connector. MongoDB also has a native BI reporting tool called MongoDB Charts which provides the fastest way to visualize your data in MongoDB without needing any third-party products.

Data retention and archival:

- What is the data retention policy? Can data be deleted or archived? If so, at what age?
- If archived, for how long and how accessible should the archive be? Does archive data need to be live or can it be restored from a backup?

There are various strategies to remove and archive data in MongoDB. Some of these strategies include using TTL indexes, Queryable Backups, zoned sharding (allowing you to create a tiered storage pattern), or simply creating an architecture where you just drop the collection of data when no longer needed.

Security:

- What users and roles need to be defined, and what is the least privileged permission needed for each of these entities?
- What are the encryption requirements? Do you need to support both inflight (network) and at-rest (storage) encryption of time series data?
- Do all activities against the data need to be captured in an audit log?
- Does the application need to conform with GDPR, HIPAA, PCI or any other regulatory framework?

The regulatory framework may require enabling encryption, auditing, and other security measures. MongoDB supports the security configurations necessary for these compliances through data encryption at rest and in-flight, auditing, and granular role-based access control controls.

While not an exhaustive list of all possible things to consider, it will help get you thinking about the application requirements and their impact on the design of the MongoDB schema and database configuration. In the next section we will explore a variety of ways to architect schemas for different time based scenarios. You can see first hand how these schemas have an effect on the application's performance and scale. In the end of the analysis you may find that the best schema design for your application may be leveraging a combination of schema designs. By following the recommendations we lay out below, you will have a good starting point to develop the optimal schema design for your app, and appropriately size your environment.

Designing a time series schema

Let's start by saying that there is no one canonical schema design that fits all application scenarios. There will always be trade-offs to consider regardless of the schema you develop. Ideally you want the optimal balance of memory and disk utilization to yield the best read and write performance that satisfy your application requirements, and that enables you to support both data ingest, and then analysis of time series data streams.

We will explore various schema design configurations. First, storing one document per data sample, and then bucketing the data using one document per time series time range and one document per fixed size. Storing more than one data sample per document is known as bucketing. This will be implemented at the application level and requires nothing to be configured specifically in the database. With MongoDB's flexible data model you can optimally bucket your data to yield the best performance and granularity for your application's requirements.

This flexibility also allows your data model to adapt to new requirements over time – such as capturing data from new hardware sensors that were not part of the original application design. These new sensors may provide different metadata and properties than the sensors you used when developing the initial application. WIth all this flexibility you may think that MongoDB databases are the wild west where anything goes and you can quickly end up with a database full of disorganized data. MongoDB

provides as much control as you need via schema validation that allows you full control to enforce things like the presence of mandatory fields and range of acceptable values to name a few. This tunability enables you to preserve the benefits of a flexible data model during development and test, while then enforcing data governance rules when the code goes into production.

To help illustrate how schema design and bucketing affects performance consider the scenario where we want to store and analyze historical stock price data. Our sample stock price generator application creates sample data every second for a given number of stocks that it tracks. One second is the smallest time interval of data collected for each stock ticker in this example. If you would like to generate sample data in your own environment, the StockGen tool is available on GitHub. It is important to note that although the sample data in this Guide uses stock ticks as an example, you can apply these same design concepts to any time series scenario like temperature and humidity readings from IoT sensors.

The StockGen tool used to generate sample data will generate the same data and store it in two different collections: **StockDocPerSecond** and **StockDocPerMinute** that each contain the following schemas:

Scenario 1:One document per data point

```
{
    "_id" : ObjectId("5b4690e047f49a04be523cbd"),
    "p" : 56.56,
    "symbol" : "MDB",
    "d" : ISODate("2018-06-30T00:00:01Z")
},
{
    "_id" : ObjectId("5b4690e047f49a04be523cbe"),
    "p" : 56.58,
    "symbol" : "MDB",
    "d" : ISODate("2018-06-30T00:00:02Z")
}
,...
```

Figure 1: Sample documents - one document per second granularity

Scenario 2: Time-based bucketing one document per minute

```
{
        "_id" : ObjectId("5b5279d1e303d394db6ea0f8"),
        "p" : {
        "0" : 56.56,
        "1" : 56.56,
        "2" : 56.58,
        …
        "59" : 57.02
    },
        "symbol" : "MDB",
        "d" : ISODate("2018-06-30T00:00:00Z")
    },
    {
        "_id" : ObjectId("5b5279d1e303d394db6ea134"),
        "p" : {
            "0" : 69.47,
            "1" : 69.47,
            "2" : 68.46,
            ...
            "59" : 69.45
            },
        "symbol" : "TSLA",
        "d" : ISODate("2018-06-30T00:01:00Z")
    },...
```

Figure 2: Sample documents representing one minute granularity

Note that the field, “p” contains a subdocument with the values for each second of the minute.

Schema design comparisons

Let’s compare and contrast the database metrics of storage size and memory impact based off of 4 weeks of data generated by the StockGen tool. Measuring these metrics is useful when assessing database performance. Note that the storage size and memory impact from a specific schema design does not take terabytes of data to be observable, our data sets in this example are just megabytes in size.

Effects on Data Storage

In our application the smallest level of time granularity is a second. Storing one document per second as described in Scenario 1 is the most comfortable modeling concept for those coming from a relational database background. That is because we are using one document per data point, which is similar to a row per data point in a tabular schema. This design will produce the largest number of documents and collection size per unit of time as seen in Figures 3 and 4.

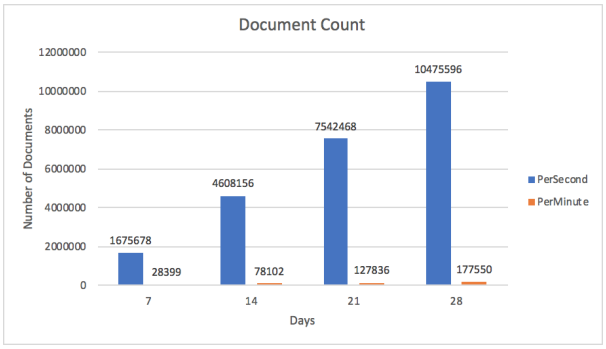

Figure 3: Document count per day comparing per second vs per minute schema design

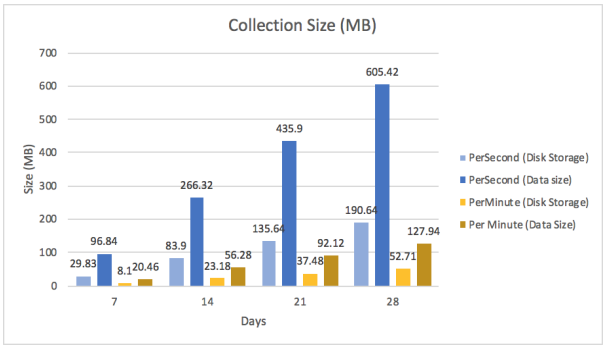

Figure 4: Comparison between data size and storage size for each scenario

Figure 4 shows two sizes per collection. The first value in the series is the size of the collection that is stored on disk while the second value is the size of the data in the database. These numbers are different because MongoDB's WiredTiger storage engine supports compression of data at rest. Logically the PerSecond collection is 605MB, but on disk it is reduced to around 190 MB of storage space.

Effects on memory utilization

A large number of documents will not only increase data storage consumption but increase index size as well. An index was created on each collection and covered the symbol and date fields. Unlike some key-value databases that position themselves as time series databases, MongoDB provides secondary indexes giving you flexible access to your data and allowing you to optimize query performance of your application.

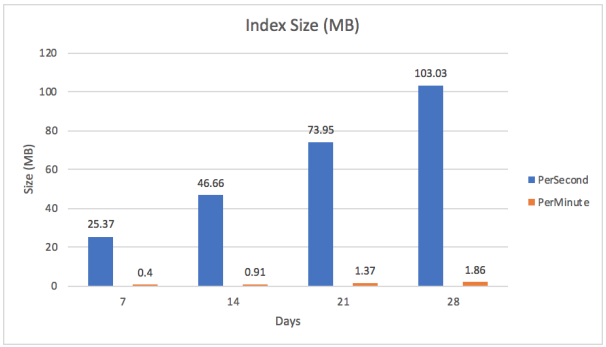

Figure 5: Index Size (MB) comparison between PerSecond and PerMinute

The size of the index defined in each of the two collections are seen in Figure 5. Optimal performance of MongoDB happens when indexes and most recently used documents fit into the memory allocated by the WiredTiger cache (we call this the "working set"). In our example we generated data for just 5 stocks over the course of 4 weeks. Given this small test case our data already generated an index that is 103MB in size for the PerSecond scenario. Keep in mind that there are some optimizations such a index prefix compression that help reduce the memory footprint of an index. However, even with these kind of optimizations, proper schema design is important to prevent runaway index sizes. Given the trajectory of growth, any changes to the application requirements, like tracking more than just 5 stocks or more than 4 weeks of prices in our sample scenario, will put much more pressure on memory and eventually require indexes to page out to disk. When this happens your performance will be degraded. To mitigate this situation, consider scaling horizontally.

Horizontally scaling

As your data grows in size you may end up scaling horizontally when you reach the physical limits (i.e. of the

RAM, CPU, or I/O) of the server hosting the primary mongod in your MongoDB replica set.

By horizontally scaling via MongoDB Sharding performance can be improved since the indexes and data will be spread over multiple MongoDB nodes. Database operations are no longer directed at a specific primary node. Rather they are mediated by an intermediate service called a query router (mongos) which sends the operation to the specific nodes that contain the data that satisfy the query. Note that this is completely transparent to the application – MongoDB handles all of the routing for you.

Scenario 3: Size based bucketing

The key learnings when comparing the previous schema design scenarios is that bucketing data has significant advantages. Time-based bucketing as described in scenario 2 collects an entire minutes worth of data into a single document. However, in time-based applications such as IoT, sensor data may be generated at irregular intervals and some sensors may provide more data than others. In these scenarios, time-based bucketing may not be the optimal approach to schema design. An alternative strategy is size based bucketing. With size based bucketing we design our schema around one document per a predetermined number of emitted sensor events, or for the entire day, whichever threshold is reached first.

To see size based bucketing in action consider the scenario where you are storing sensor data and limiting the bucket size to 200 events per document, or a single day (whichever comes first). Note: The 200 limit is an arbitrary number used in this example and can be changed as needed, without application changes or schema migrations.

```
{
  "_id" : ObjectId("5c66a3e29a368016fbdb48bb"),
  "day" : ISODate("2019-02-15T00:00:00Z"),
  "deviceid" : 2,
  "sensorid" : 2003,
  "first" : ISODate("2019-02-15T05:34:00Z"),
  "last" : ISODate("2019-02-15T05:37:19Z"),
  "nsamples" : 200,
  "samples" : [
    {
      "val" : 51.76,
      "time" : ISODate("2019-02-15T05:34:00Z")
    },
    {
      "val" : 51.74,
      "time" : ISODate("2019-02-15T05:34:01Z")
    },

    … Repeated array of values, removed from brevity

}
```

Figure 6: Size based bucketing for sparse data

An example size based bucket is shown in figure 6. In this design trying to limit inserts per document to an arbitrary number or a specific time period may seem difficult, however, it is easy to do using an upsert, as shown in the following code example:

```
from datetime import date, timedelta, datetime as dt
...
sampletime = dt.now()
sample = {val:59,time:sampletime}
day = dt.today().replace(hour=0, minute=0, \
second=0, microsecond=0)

db.iot.updateOne({deviceid:1234,sensorid:3, \
nsamples:{$lt:200},day:day},
             {$push:{samples:sample},
              $min:{first:sample.time},
              $max:{last:sample.time},
              $inc:{nsamples:1}},{upsert:true} )
```

Figure 7: Sample code to add to the size based bucket

As new sensor data is ingested it is simply appended to the document until the number of samples hit 200, then a new document is created because of our upsert:true clause.

MongoDB supports secondary indexes. For example, an index in this scenario could be on {deviceid:1,sensorid:1,day:1,nsamples:1}. When we are updating data based on a day it is an exact match in our index. We could also add an index for the first and last fields. This will help when we specify a date, or a date range query. For example, if we wanted to query for all the samples that occured in the 06:55 minute we could use an

aggregation pipeline query with three stages. The first stage will take the entire collection and return just those documents which have a timestamp between our minute.

```
$match: {
 $or:[
   {"first":
    {$gte:
      ISODate('2019-02-15T06:55:00.000+00:00')
      }},
  { "last":
    {$lte:
      ISODate('2019-02-15T06:56:00.000+00:00')
      }}
   ]
}
```

Now that we have only documents that contain data within the desired range we will unwind the array to create documents for each time value.

```
$unwind: {
  path: "$samples"
}
```

Finally we can filter out the documents to return just those documents that are within our minute.

```
$match: {
  "samples.time": {
  $gte:
    ISODate('2019-02-15T06:55:00.000+00:00'),
  $lt:
    ISODate('2019-02-15T06:56:00.000+00:00')}
   }
```

The complete query is as follows:

```
db.collection.aggregate (
[ {$match: {
 $or:[
   {"first": {
     $gte:
       ISODate('2019-02-15T06:55:00.000+00:00')}},
  { "last": {
     $lte:
       ISODate('2019-02-15T06:56:00.000+00:00')}}
   ]
}
}, {$unwind: {
  path: "$samples"
}}, {$match: {
  "samples.time": {
   $gte:
     ISODate('2019-02-15T06:55:00.000+00:00'),
   $lt:
    ISODate('2019-02-15T06:56:00.000+00:00')}
   }}]
)
```

More information on querying data will be covered later in this paper.

Bucketing data in a fixed size will yield very similar database storage and index improvements as seen when bucketing per time in scenario 2. It is one of the most efficient ways to store sparse IoT data in MongoDB.

What to do with old data

Should we store all data in perpetuity? Is data older than a certain time useful to your organization? How accessible should older data be? Can it be simply restored from a backup when you need it, or does it need to be online and accessible to users in real time as an active archive for historical analysis? As we discussed earlier, these are some of the questions that should be asked prior to going live.

Establishing data retention requirements is by far the most important topic to address in any time series application. Afterall, if the business only cares about 1 month of data and you are only ingesting a small amount of transactions a day with no desire to retain or query anything beyond that, you may not have to worry about schema at all. Of course, they will never come back a month after production release and change their mind. As you know that never happens. In case of some unforeseen change of business requirement, rest assured that there are multiple approaches to handling aged data in MongoDB and depending on your specific requirements some may be more applicable than others. The key learning is to choose the one that best fits your requirements.

Pre-aggregation

Does your application really need a single data point for every event generated years ago? In most cases the resource cost of keeping this granularity of data around outweighs the benefit of being able to query down to this level at any time. In most cases data can be pre-aggregated and stored for fast querying. In our stock example, we may want to only store the closing price for each day as a value. In most architectures, pre-aggregated values are stored in a separate collection since typically queries for historical data are different than real-time

queries. Usually with historical data, queries are looking for trends over time versus individual real-time events. By storing this data in different collections you can increase performance by creating more efficient indexes as opposed to creating more indexes on top of real-time data.

Offline archival strategies

When data is archived, what is the SLA associated with retrieval of the data? Is restoring a backup of the data acceptable or does the data need to be online and ready to be queried at any given time? Answers to these questions will help drive your archive design. If you do not need real-time access to archival data you may want to consider backing up the data and removing it from the live database. Production databases can be backed up using MongoDB Ops Manager or if using the MongoDB Atlas service you can use a fully managed backup solution.

Removing documents using remove statement

Once data is copied to an archival repository via a database backup or an ETL process, data can be removed from a MongoDB collection via the remove statement as follows:

```
db.StockDocPerSecond.remove ( { "d" :
    { $lt: ISODate( "2018-03-01" ) } } )
```

In this example all documents that have a date before March 1st, 2018 defined on the “d” field will be removed from the StockDocPerSecond collection.

You may need to set up an automation script to run every so often to clean out these records. Alternatively you can avoid creating automation scripts in this scenario by defining a time to live (TTL) index.

Removing documents using a TTL Index

A TTL index is similar to a regular index except you define a time interval to automatically remove documents from a collection. In the case of our example, we could create a TTL index that automatically deletes data that is older than 1 week.

```
db.StockDocPerSecond.createIndex( { "d": 1 },
    { expireAfterSeconds: 604800 } )
```

Although TTL indexes are convenient, keep in mind that the TTL check happens every minute or so and the interval can not be configured. If you need more control so that deletions won’t happen during specific times of the day you may want to schedule a batch job that performs the deletion in lieu of using a TTL index.

Removing documents by dropping the collection

It is important to note that using the remove command or TTL indexes will cause high disk I/O. On a database that may be under high load already this may not be desirable. The most efficient and fastest way to remove records from the live database is to drop the collection. If you can design your application such that each collection represents a block of time so that when you need to archive or remove data all you need to do is drop the collection. This may require some smarts within your application code to know which collections should be queried, however, the cost benefit may outweigh this change. When you issue a remove, MongoDB also has to remove data from all affected indexes as well and this could take a while depending on the size of data and indexes. Dropping a collection is the fastest way to remove data and indexes.

Online archival strategies

If archival data still needs to be accessed in real-time consider how frequently these queries occur and if storing only pre-aggregated results can be sufficient.

Sharding archival data

One strategy for archiving data and keeping the data accessible real-time is by using zoned sharding to partition the data. Sharding not only helps with horizontally scaling the data out across multiple nodes, but you can tag shard ranges so partitions of data are pinned to specific shards. A cost savings measure could be to have the archival data live on shards running lower cost disks and periodically adjusting the time ranges defined in the shards themselves. These ranges would cause the MongoDB balancer to automatically migrate the data between these

storage layers, providing you with tiered, multi-temperature storage. Review our tutorial for creating tiered storage patterns with zoned sharding for more information.

Accessing archived data via queryable backups

If your archive data is not accessed that frequently and the query performance does not need to meet any strict latency SLAs consider backing the data up and using the Queryable Backups feature of MongoDB Atlas or MongoDB Ops Manager. Queryable Backups allow you to connect to your backup and issue read-only commands to the backup itself, without having to first restore the backup.

Querying data from the data lake

MongoDB is an inexpensive solution not only for long term archival but for your data lake as well. Companies who have made investments in technologies like Apache Spark can leverage the MongoDB Spark Connector. This connector materializes MongoDB data as DataFrames and Datasets for use with Spark and machine learning, graph, streaming, and SQL APIs.

Key Takeaways on Schema design

Once an application is live in production and is multi-terabytes in size, any major change can be very expensive from a resource standpoint. Consider the scenario where you have 6 TB of IoT sensor data and accumulating new data at a rate of 50,000 inserts per second. Performance of reads is starting to become an issue and you realize that you have not properly scaled out the database. Unless you are willing to take application downtime, a change of schema in this configuration – i.e. moving from raw data storage to bucketed storage – may require building out shims, temporary staging areas and all sorts of transient solutions to move the application to the new schema. The moral of the story is to plan for growth and properly design the best time series schema that fits your application's SLAs and requirements.

This chapter of the Guide analyzed three different schema designs for storing time series data from stock prices. Is the schema that won in the end for this stock price database the one that will be the best in your scenario? Maybe. Due to the nature of time series data and it's typically rapid ingestion, the answer may in fact be leveraging a combination of collections that target a read or write heavy use case. The good news is that with MongoDB's flexible schema it is easy to make changes. In fact you can run two different versions of the app writing two different schemas to the same collection. However don't wait until your query performance starts suffering to figure out an optimal design as migrating TBs of existing documents into a new schema can take time, resource, and delay future releases of your application. You should undertake real world testing before commiting on a final design. Quoting a famous proverb, "Measure twice and cut once."

Schema design tips:

- MMAPV1 storage engine is deprecated, use the default WiredTiger storage engine. Note that if you read older schema design best practices from a few years ago they were often built on the older MMAPV1 technology
- Understand what the data access requirements are from your time series application
- Schema design impacts resources. "Measure twice and cut once" with respect to schema design and indexes
- Test schema patterns with real data and a real application if possible
- Bucketing data reduces index size and thus massively reduce hardware requirements time series applications traditionally capture very large amounts of data, so only create indexes where they will be useful to the app's query patterns
- Consider more than one collection, one focused on write heavy inserts and recent data queries and another collection with bucketed data focused on historical queries on pre-aggregated data
- Use MongoDB monitoring tools like MongoDB OpsManager to monitor the size of your indexes. When the size starts to approach the amount of memory on the server hosting MongoDB, consider horizontally scaling out to spread the index and load over multiple servers

- Determine at what point data expires, and what action to take such as archival or deletion

Querying, Analyzing, and Presenting time series Data

As of now we reviewed the key questions you need to ask to understand query access patterns to your database, and explored various schema design patterns for time series data and how they affect MongoDB resources. Now let's look at how to gain insights into our data. Specifically how to query, analyze and present time series data stored in MongoDB. Knowing how clients will connect to query your database will help guide you in designing both the data model and optimal database configuration. There are multiple ways to query MongoDB. You can use native tools such as the MongoDB Shell command line and MongoDB Compass a GUI-based query tool. Programatically MongoDB data is accessed via an extensive list of MongoDB drivers. There are drivers available for practically all the major programming languages including C#, Java, NodeJS, Go, R, Python, Ruby, and many others.

MongoDB also provides third-party BI reporting tool integration through the use of the MongoDB BI Connector. Popular SQL-based reporting tools like Tableau, Microsoft PowerBI, QlikView, and TIBCO Spotfire can leverage data directly in MongoDB without the need to ETL data into another platform for querying. MongoDB Charts, currently in Beta, provides the fastest way to visualize your MongoDB data without the need for third party products or flattening your data so that it can be read by SQL-bases BI tool.

For advanced analytical operations on time series data, such as those those found in artificial intelligence and machine learning solutions, MongoDB integrates with many third party data science tools and platforms including Apache Spark. See the "Advanced Analytics with MongoDB" section later in this paper for more details.

Processing Time Series Data with the Aggregation Framework

The MongoDB Aggregation Framework allows developers to express functional pipelines that perform preparation, transformations, and analysis of data. This is accomplished through the use of stages which perform specific operations like multi-faceted aggregations, grouping, matching, joining, and sorting, to name a few. This reshaped data flows through the stages and its corresponding processing is referred to as the Aggregation Pipeline. Conceptually it is similar to the data flow through a Unix shell command line pipeline. Data gets input from the previous stage, work is performed and the stage's output serves as input to the next processing stage until the pipeline ends. Figure 8 shows how data flows through a pipeline that consists of a match and group stage.

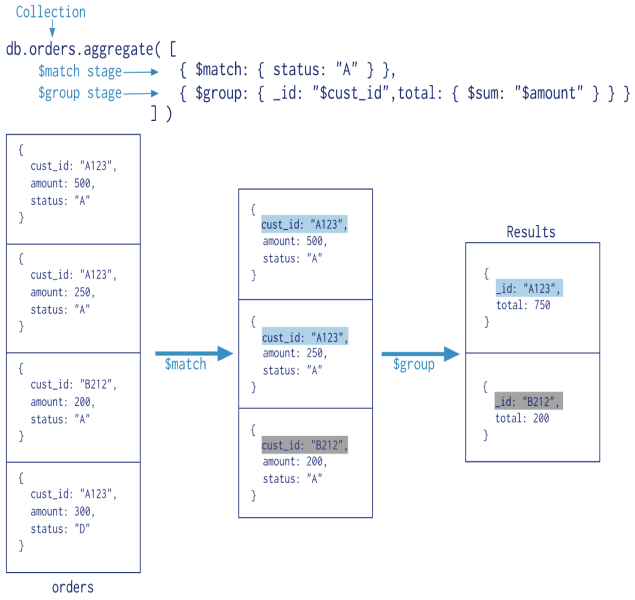

Figure 8: Sample data flows through the Aggregation Pipeline

$match is the first stage In this simple two stage pipeline. $match will take the entire orders collection as an input and provide as an output a filter with the list of documents where the field, "status" contains the "A" value. The second stage will take these filtered documents as input and perform a grouping of the data to produce the desired query result as an output. While this is a simple example keep in mind that you can build out extremely sophisticated processing pipelines leveraging 100+ operators and over

25 different stage classes allowing you to do things like transformations, redacting, sorting, grouping, matching, faceted searches, graph traversals, and joins between different collections to name a few. You can work with data in ways that are just impossible with other distributed databases, and which are highly complex to build and optimize with SQL.

With our time series data we are going to use MongoDB Compass to issue an ad-hoc query that finds the day high price for a given stock. Compass is the GUI tool that allows you to easily explore your data. A useful feature is the ability to visually construct an aggregation pipeline by assembling stages onto a canvas, inspecting the results at each stage, and then exporting the resultant pipeline as code for copying and pasting into your app.

Finding day high for a given stock

Before diving into the query itself, recall that our StockGen application, produced 1 month of stock price data for 5 stocks we wanted to track. One of the two collections created is called, "StockDocPerMinute" (PerMinute) and it contains a document that represents a minute of data for a specific stock symbol as shown in Figure 9.

```
{
    "_id" : ObjectId("5b57a8fae303d36d6df69cd3"),
    "p" : {
                "0" : 58.75,
                "1" : 58.75,
                "2" : 59.45,
...up to…
                "58" : 58.57,
                "59" : 59.01
    },
    "symbol" : "FB",
    "d" : ISODate("2018-07-14T00:00:00Z")
}
```

Figure 9: Sample StockDocPerMinute document

Consider the scenario where the application requests the day high price for a given stock ticker over time. Without the Aggregation Framework this query would have to be either accomplished by retrieving all the data back into the app and using client side code to compute the result, or by defining a map-reduce function in Javascript. Both options are not optimal from a performance or developer productivity perspective.

Notice the sample document has a subdocument which contains data for an entire minute interval. Using the Aggregation Framework we can easily process this subdocument by transforming the subdocument into an array using the $objectToArray expression, calculating the maximum value and projecting the desired result.

Using MongoDB Compass we can construct the query using the Aggregation Pipeline Builder as follows:

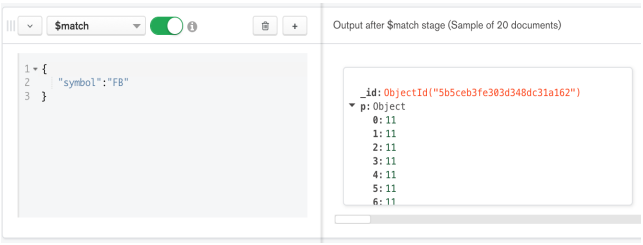

Figure 10: First stage is $match stage

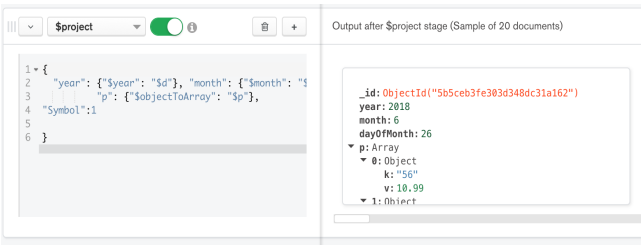

Figure 11: Second stage is $project stage

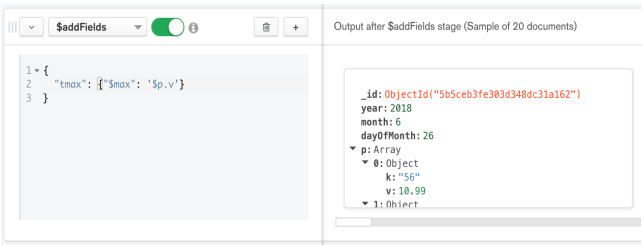

Figure 12: Third stage is $addFields stage

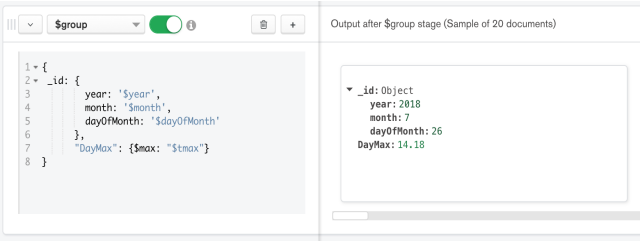

Figure 13: Fourth stage is $group stage

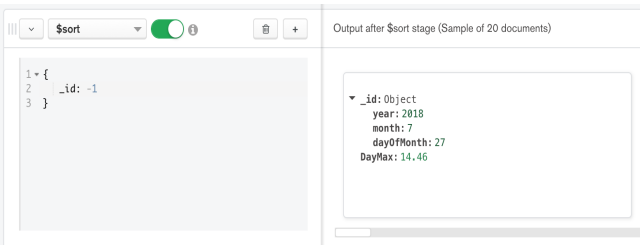

Figure 14: Fifth stage is $sort stage

We can see the output of the last stage is showing the maximum value per day for each day. Using the aggregation pipeline builder we didn't need to write a single line of code. For reference, the complete query built by MongoDB Compass in the previous figures is as follows:

```
db.getCollection('StockDocPerMinute')
.aggregate([
{ $match: { "symbol" : "FB" } },
{ "$project":
    {   "year": {"$year": "$d"},
        "month": {"$month": "$d"},
        "dayOfMonth": {"$dayOfMonth": "$d"},
        "p": {"$objectToArray": "$p"},
        "Symbol":1
    } },
{"$addFields": {"tmax": {"$max": '$p.v'}}},
{"$group": {
    _id: {
        year: '$year',
        month: '$month',
        dayOfMonth: '$dayOfMonth'
        },
    "DayMax": {$max: "$tmax"},
    }
    },
{$sort: {_id: -1}}
])
```

Leveraging Views

MongoDB read-only views provide a layer of abstraction from the underlying collection. This abstraction provides both a security and a usability benefit. From a security standpoint, this layer allows administrators to define separate access controls to the view and no access to the underlying collection. The end result is a user that has access to query the view but no access to the underlying collection. This is important as it mitigates a scenario where your user account gets compromised and the attacker now has access to all the information in the collection as opposed to only just some of the data in the collection. Locking down access control this way is referred to in security best practices as the principle of least privileges. If you want to learn more about access control with Views

read the blog post, "Providing Least Privilege Access to MongoDB data".

Another benefit of Views is that they hide complex schemas from the user. This can be beneficial if users are allowed to directly connect and query the data in a collection. For example, given our current schema design, if a user wanted to query the StockDocPerMinute collection to return a list of stock prices for a given stock symbol they could issue the following aggregation query:

```
db.StockDocPerMinute.aggregate([
{ $match: { "symbol":"FB" } },
{"$project":
    {"p":
        {"$objectToArray": "$p"},
        "d":1,
        "symbol":1}},
{"$unwind": "$p"},
{"$project":
    {"_id": 0,
    "symbol":1,
    "Timestamp": {
        "$dateFromString":
            {"dateString":
                {"$concat":
                    [{"$dateToString":
                        {"format": "%Y-%m-%dT%H:%M:",
                            "date": "$d"}},
                        {"$concat": ["$p.k", "Z"]
                        }]}}},
                        "price":"$p.v"}}
])
```

A few samples of the resulting query are as follows:

```
{
    "symbol": "FB",
    "Timestamp":ISODate("2018-06-26T00:00:56.000Z"),
    "price": 10.99
},
{
    "symbol": "FB",
    "Timestamp":ISODate("2018-06-26T00:00:54.000Z"),
    "price": 10.99
},..
```

We can make a better user experience by wrapping this query in a view as follows:

```
db.createView("ViewStock","StockDocPerMinute",
[
{"$project": {"p": {"$objectToArray": "$p"},
    "d":1, "symbol":1}},
{"$unwind": "$p"},
{"$project": {"_id": 0,
    "symbol":1,
    "Timestamp": {
    "$dateFromString": {"dateString":
        {"$concat": [{"$dateToString":
            {"format": "%Y-%m-%dT%H:%M:",
            "date": "$d"}},
            {"$concat": ["$p.k", "Z"]}]}}},
            "price":"$p.v"}}
])
```

Now, all the user has to do to query for the first price entry for the "FB" stock is this statement:

```
db.ViewStock.find({"symbol":"FB"})
    .sort({d:-1}).limit(1)
```

Note that as MongoDB read-only views are materialized at run-time, the latest results are always available with each query.

You can also use the aggregation framework with views. Here is the query for all "FB" stock ticker data for a specific day.

```
db.ViewStock.aggregate({ $match:
    { "symbol":"FB",
        "Timestamp" :
        { $gte: ISODate("2018-06-26"),
            $lt: ISODate("2018-06-27") }}},
    { $sort: { "Timestamp": -1 }} )
```

Querying time series data with third-party BI Reporting tools

Users may want to leverage existing investments in third-party Business Intelligence Reporting and Analytics tools. To enable these SQL-speaking tools to query data in MongoDB, you can use intermediary service called the MongoDB BI Connector. Using the BI Connector, you can start to visualize and analyze your time series data, as it is streamed into the MongoDB platform. Your reports can be updated as new events are generated, giving users real time insight into the business.

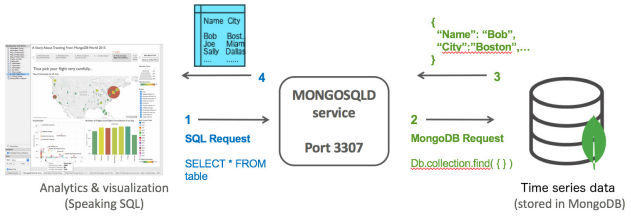

Figure 15: Query MongoDB data with your favorite SQL-based reporting tools using the BI Connector

The BI Connector service presents a port that resembles a MySQL Server to the client application and accepts client connections issuing SQL queries. The BI Connector service then translates these queries into the MongoDB Query Language (MQL) and submits the query to the MongoDB database. Results are returned from MongoDB and flattened into a tabular structure and sent back to the SQL speaking client. This flow is seen in detail in Figure 15.

To illustrate the MongoDB BI Connector in action, let's consume our time series data with Tableau Desktop and the MongoDB BI Connector. Tableau Desktop has a connection option for MongoDB. Using that option and connecting to the port specified in the BI Connector we see that Tableau enumerates the list of tables from our MongoDB database in Figure 16.

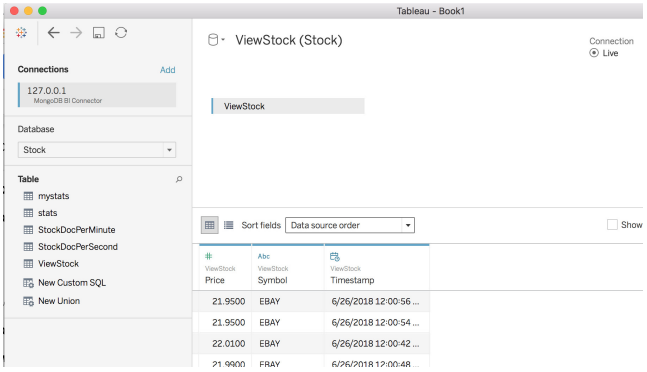

Figure 16: Data Source view in Tableau showing information from MongoDB

These tables are really our collections in MongoDB. Continuing with the Worksheet view in Tableau we can continue and build out a report showing price over time using the View we created earlier in this document.

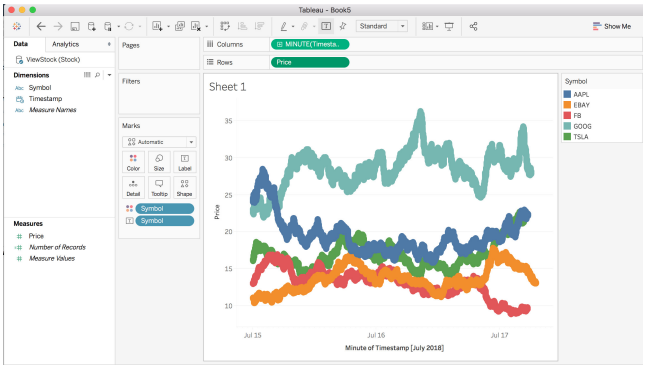

Figure 17: Sample Tableau worksheet (price over time)

MongoDB Charts

The fastest way to visualize data in MongoDB is with MongoDB Charts. Currently available in Beta it provides users with a web console where they can build and run reports directly from data stored in MongoDB. With Charts there is no special service that needs to run in order to query MongoDB. There is also no need to move the data out or transform it into a different format to be queried. Data can be queried directly as rich documents stored MongoDB. As with other read-only connections, you can connect Charts to secondary replica nodes, therefore isolating analytics and reporting queries from the rest of the cluster serving the operational time series apps. To see

how MongoDB Charts can represent data from the StockGen tool, check out the price over time line graph as shown in Figure 18.

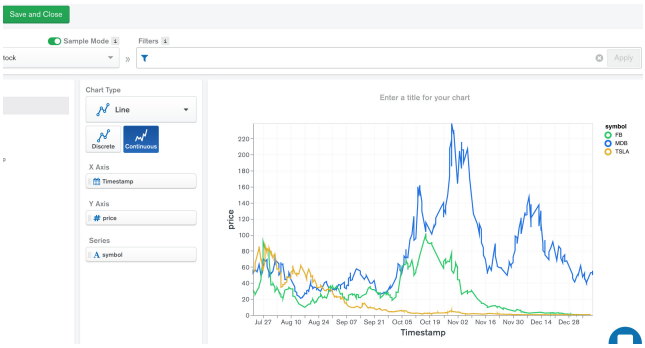

Figure 18: MongoDB Charts showing price over time

At the time of this writing MongoDB Charts is in Beta, therefore details and screenshots may vary from with the final release.

Advanced Analytics with MongoDB

The value of data to a business is derived from analyzing the data. Whether it's looking at trends such as the 5 day moving average of stock ticker data or using machine learning algorithms to uncover new statistical patterns for determining component failure via predictive analytics, MongoDB enables your business to gain the most value from your data. These advanced analytical queries can make use of MongoDB's aggregation framework as well as leverage many of the popular machine learning and artificial intelligence platforms today including solutions like Apache Spark. The MongoDB Connector for Apache Spark exposes all of Spark's libraries, including Scala, Java, Python and R. This enables you to use the Spark analytics engine for big data processing of your time series data extending analytics capabilities of MongoDB even further to perform real-time analytics and machine learning. The connector materializes MongoDB data as DataFrames and Datasets for analysis with machine learning, graph, streaming, and SQL APIs. The Spark connector takes advantage of MongoDB's aggregation pipeline and rich secondary indexes to extract, filter, and process only the range of data you need! No wasted time extracting and loading data into another database in order to query your MongoDB data using Spark!

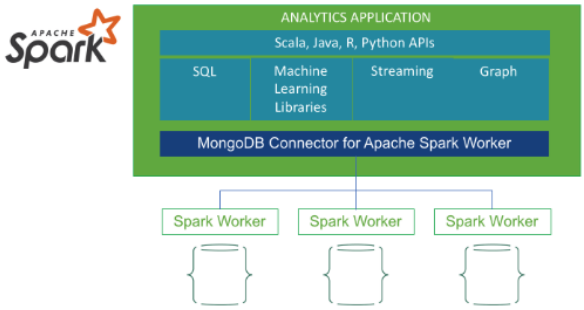

Figure 19: Spark connector for MongoDB

The MongoDB R driver provides developers and statisticians a first class experience with idiomatic, native language access to MongoDB, enterprise authentication, and full support for BSON data types. Using the extensive libraries available to R you could query MongoDB time series data and determine locally weighted regression as seen in Figure 20.

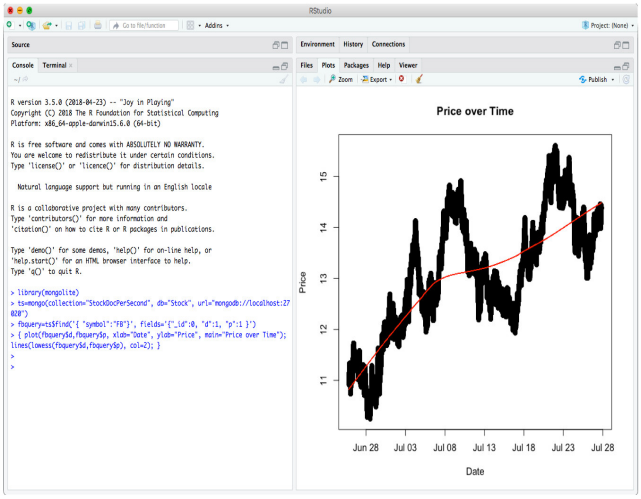

Figure 20: Scatter Plot showing price over time and smoothing of per second data

The R driver for MongoDB is available via the CRAN R Archive. Once installed you can connect to your MongoDB database and return dataframes that can be used in R calculations. The above graph was produced with the following code using R Studio:

```
library(mongolite)
ts=mongo(collection="StockDocPerSecond",
        db="Stock",
        url="mongodb://localhost:27020")
fbquery=ts$find('{ "symbol":"FB"}',
        fields='{"_id":0, "d":1, "p":1 }')
plot(fbquery$d,fbquery$p,
        xlab="Date",
        ylab="Price",
        main="Price over Time");
lines(lowess(fbquery$d,fbquery$p), col=2);
```

Summary

While not all data is time series in nature, a growing percentage of it can be classified as time series – fueled by technologies that allow us to exploit streams of data in real time rather than in batches. In every industry and in every company there exists the need to query, analyze and report on time series data. One such use case is the Internet of Things (IoT). IoT generates a plethora of time series data. Larger IoT solutions involve supporting a variety of hardware and software devices for data ingestion, supporting real-time and historical analysis, security, high availability and managing time series data at scale to name a few. MongoDB is powering mission critical IoT applications worldwide. For more information on IoT at MongoDB check out the Internet of Things website.

Like IoT and other time series applications, it is one thing to prototype a sample but handling terabytes of production data efficiently is on a different playing field. With MongoDB it is easy to prototype then horizontally scale out time series workloads. Through the use of replica sets, read-only clients can connect to replica-set secondaries to perform their queries leaving the primary to focus on writes. Write heavy workloads can scale horizontally via sharding. While an in depth analysis of MongoDB architecture is out of scope for this whitepaper you can find lots of useful information in the MongoDB Architecture whitepaper.

In this whitepaper we covered some thought provoking questions with respect to your specific application requirements. We looked at a few different time series schema designs and their impact on MongoDB performance. Finally we explored how to query time series data using the MongoDB Aggregation Framework and MongoDB Compass as well as other methods like using the BI Connector and analytical languages like R. MongoDB provides all the functionality, performance and tools needed for a successful time series application.

We Can Help

We are the MongoDB experts. Over 6,600 organizations rely on our commercial products. We offer software and services to make your life easier:

MongoDB Enterprise Advanced is the best way to run MongoDB in your data center. It's a finely-tuned package of advanced software, support, certifications, and other services designed for the way you do business.

MongoDB Atlas is a database as a service for MongoDB, letting you focus on apps instead of ops. With MongoDB Atlas, you only pay for what you use with a convenient hourly billing model. With the click of a button, you can scale up and down when you need to, with no downtime, full security, and high performance.

MongoDB Stitch is a serverless platform which accelerates application development with simple, secure access to data and services from the client – getting your apps to market faster while reducing operational costs and effort.

MongoDB Mobile (Beta) MongoDB Mobile lets you store data where you need it, from IoT, iOS, and Android mobile devices to your backend – using a single database and query language.

MongoDB Cloud Manager is a cloud-based tool that helps you manage MongoDB on your own infrastructure. With automated provisioning, fine-grained monitoring, and continuous backups, you get a full management suite that reduces operational overhead, while maintaining full control over your databases.

MongoDB Consulting packages get you to production faster, help you tune performance in production, help you scale, and free you up to focus on your next release.

MongoDB Training helps you become a MongoDB expert, from design to operating mission-critical systems at scale. Whether you're a developer, DBA, or architect, we can make you better at MongoDB.

Resources

For more information, please visit mongodb.com or contact us at sales@mongodb.com.

Case Studies (mongodb.com/customers)
Presentations (mongodb.com/presentations)
Free Online Training (university.mongodb.com)
Webinars and Events (mongodb.com/events)
Documentation (docs.mongodb.com)
MongoDB Enterprise Download (mongodb.com/download)
MongoDB Atlas database as a service for MongoDB (mongodb.com/cloud)
MongoDB Stitch backend as a service (mongodb.com/cloud/stitch)

mongoDB

US 866-237-8815 ▪ INTL +1-650-440-4474 ▪ info@mongodb.com