MongoDB Atlas Best Practices

April 2017

Table of Contents

Introduction

MongoDB Atlas provides all of the features of MongoDB, without the operational heavy lifting required for any new application. MongoDB Atlas is available on-demand through a pay-as-you-go model and billed on an hourly basis, letting you focus on what you do best.

It's easy to get started – use a simple GUI to select the instance size, region, and features you need. MongoDB Atlas provides:

- Security features to protect access to your data
- Built in replication for always-on availability, tolerating complete data center failure
- Backups and point in time recovery to protect against data corruption
- Fine-grained monitoring to let you know when to scale. Additional instances can be provisioned with the push of a button
- Automated patching and one-click upgrades for new major versions of the database, enabling you to take advantage of the latest and greatest MongoDB features
- A choice of cloud providers, regions, and billing options

MongoDB Atlas is versatile. It's great for everything from a quick Proof of Concept, to test/QA environments, to complete production clusters. If you decide you want to bring operations back under your control, it is easy to move your databases onto your own infrastructure and manage them using MongoDB Ops Manager or MongoDB Cloud Manager. The user experience across MongoDB Atlas, Cloud Manager, and Ops Manager is consistent, ensuring that disruption is minimal if you decide to migrate to your own infrastructure.

MongoDB Atlas is automated, it's easy, and it's from the creators of MongoDB. Learn more and take it for a spin.

While MongoDB Atlas radically simplifies the operation of MongoDB there are still some decisions to take to ensure the best performance and reliability for your application. This paper provides guidance on best practices for deploying, managing, and optimizing the performance of your database with MongoDB Atlas.

This guide outlines considerations for achieving performance at scale with MongoDB Atlas across a number of key dimensions, including instance size

selection, application patterns, schema design and indexing, and disk I/O. While this guide is broad in scope, it is not exhaustive. Following the recommendations in this guide will provide a solid foundation for ensuring optimal application performance.

For the most detailed information on specific topics, please see the on-line documentation at mongodb.com. Many links are provided throughout this white paper to help guide users to the appropriate resources.

Preparing for a MongoDB Deployment

Schema Design

Developers and data architects should work together to develop the right data model, and they should invest time in this exercise early in the project. The requirements of the application should drive the data model, updates, and queries of your MongoDB system. Given MongoDB's dynamic schema, developers and data architects can continue to iterate on the data model throughout the development and deployment processes to optimize performance and storage efficiency, as well as support the addition of new application features. All of this can be done without expensive schema migrations.

Document Model

MongoDB stores data as documents in a binary representation called BSON. The BSON encoding extends the popular JSON representation to include additional types such as `int`, `long`, and `date`. BSON documents contain one or more fields, and each field contains a value of a specific data type, including arrays, sub-documents and binary data. It may be helpful to think of documents as roughly equivalent to rows in a relational database, and fields as roughly equivalent to columns. However, MongoDB documents tend to have all related data for a given record or object in a single document, whereas in a relational database that data is usually normalized across rows in many tables. For example, data that belongs to a parent-child relationship in two RDBMS tables can frequently be collapsed (embedded) into a single document in MongoDB. For operational applications, the document model makes JOINs redundant in many cases.

Where possible, store all data for a record in a single document. MongoDB provides ACID compliance at the document level. When data for a record is stored in a single document the entire record can be retrieved in a single seek operation, which is very efficient. In some cases it may not be practical to store all data in a single document, or it may negatively impact other operations. Make the trade-offs that are best for your application.

Rather than storing a large array of items in an indexed field, storing groups of values across multiple fields results in more efficient updates.

Collections

Collections are groupings of documents. Typically all documents in a collection have similar or related purposes for an application. It may be helpful to think of collections as being analogous to tables in a relational database.

Dynamic Schema & Document Validation

MongoDB documents can vary in structure. For example, documents that describe users might all contain the user id and the last date they logged into the system, but only some of these documents might contain the user's shipping address, and perhaps some of those contain multiple shipping addresses. MongoDB does not require that all documents conform to the same structure. Furthermore, there is no need to declare the structure of documents to the system – documents are self-describing.

DBAs and developers have the option to define Document Validation rules for a collection – enabling them to enforce checks on selected parts of a document's structure, data types, data ranges, and the presence of mandatory fields. As a result, DBAs can apply data governance standards, while developers maintain the benefits of a flexible document model. These are covered in the blog post Document Validation: Adding Just the Right Amount of Control Over Your Documents.

MongoDB Compass aids the identification of useful validation rules, and then these rules can created within the GUI. Refer to *Visualizing your Schema and Adding*

Validation Rules: MongoDB Compass below for more details.

Indexes

MongoDB uses B-tree indexes to optimize queries. Indexes are defined on a collection's document fields. MongoDB includes support for many indexes, including compound, geospatial, TTL, text search, sparse, partial, unique, and others. For more information see the section on indexing below.

Transactions

Atomicity of updates may influence the schema for your application. MongoDB guarantees ACID compliant updates to data at the document level. It is not possible to update multiple documents in a single atomic operation, however as with JOINs, the ability to embed related data into MongoDB documents eliminates this requirement in many cases. For use cases that do require multiple documents to be updated atomically, it is possible to implement Two Phase Commit logic in the application.

For more information on schema design, please see Data Modeling Considerations for MongoDB in the MongoDB Documentation.

Visualizing your Schema and Adding Validation Rules: MongoDB Compass

The MongoDB Compass GUI allows users to understand the structure of existing data in the database and perform ad hoc queries against it – all with zero knowledge of MongoDB's query language. Typical users could include architects building a new MongoDB project or a DBA who has inherited a database from an engineering team, and who must now maintain it in production. You need to understand what kind of data is present, define what indexes might be appropriate, and identify if Document Validation rules should be added to enforce a consistent document structure.

Without MongoDB Compass, users wishing to understand the shape of their data would have to connect to the MongoDB shell and write queries to reverse engineer the document structure, field names, and data types. Similarly, anyone wanting to run custom queries on the data would need to understand MongoDB's query language.

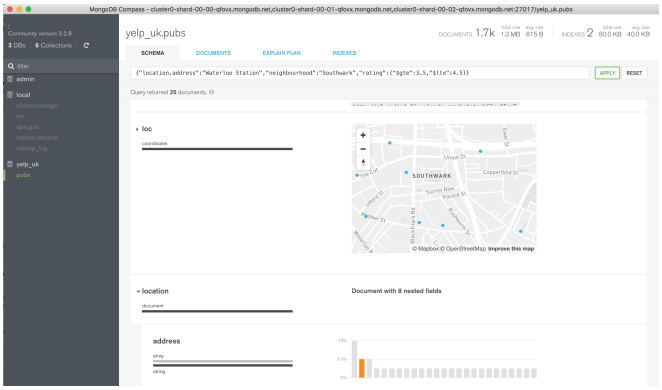

Figure 1: View schema & interactively build and execute database queries with MongoDB Compass

MongoDB Compass provides users with a graphical view of their MongoDB schema by sampling a subset of documents from a collection. By using sampling, MongoDB Compass minimizes database overhead and can present results to the user almost instantly.

Document validation allows DBAs to enforce data governance by applying checks on document structure, data types, data ranges, and the presence of mandatory fields. Validation rules can now be managed from the Compass GUI. Rules can be created and modified directly using a simple point and click interface, and any documents violating the rules can be clearly presented. DBAs can then use Compass's CRUD support to fix data quality issues in individual documents.

MongoDB Compass can be used for free during development and it is also available for production use with MongoDB Atlas Professional, MongoDB Professional, or MongoDB Enterprise Advanced subscriptions.

Application Access Patterns

The schema design has a huge influence on performance, how the application accesses the data can also have a major impact.

Searching on indexed attributes is typically the single most important pattern as it avoids collection scans. Taking it a step further, using **covered queries** avoids the need to access the collection data altogether. Covered queries

return results from the indexes directly without accessing documents and are therefore very efficient. For a query to be covered, all the fields included in the query must be present in an index, and all the fields returned by the query must also be present in that index. To determine whether a query is a covered query, use the `explain()` method. If the `explain()` output displays `true` for the `indexOnly` field, the query is covered by an index, and MongoDB queries only that index to match the query and return the results.

Rather than retrieving the entire document in your application, updating fields, then saving the document back to the database, instead issue the update to specific fields. This has the advantage of less network usage and reduced database overhead.

Document Size

The maximum BSON document size in MongoDB is 16 MB. Users should avoid certain application patterns that would allow documents to grow unbounded. For example, in an e-commerce application it would be difficult to estimate how many reviews each product might receive from customers. Furthermore, it is typically the case that only a subset of reviews is displayed to a user, such as the most popular or the most recent reviews. Rather than modeling the product and customer reviews as a single document it would be better to model each review or groups of reviews as a separate document with a reference to the product document; while also storing the key reviews in the product document for fast access.

In practice most documents are a few kilobytes or less. Consider documents more like rows in a table than the tables themselves. Rather than maintaining lists of records in a single document, instead make each record a document. For large media items, such as video or images, consider using GridFS, a convention implemented by all the drivers that automatically stores the binary data across many smaller documents.

Field names are repeated across documents and consume space – RAM in particular. By using smaller field names your data will consume less space, which allows for a larger number of documents to fit in RAM

GridFS

For files larger than 16 MB, MongoDB provides a convention called GridFS, which is implemented by all MongoDB drivers. GridFS automatically divides large data into 256 KB pieces called chunks and maintains the metadata for all chunks. GridFS allows for retrieval of individual chunks as well as entire documents. For example, an application could quickly jump to a specific timestamp in a video. GridFS is frequently used to store large binary files such as images and videos directly in MongoDB, without offloading them to a separate filesystem.

Data Lifecycle Management

MongoDB provides features to facilitate the management of data lifecycles, including Time to Live indexes, and capped collections.

Time to Live (TTL)

If documents in a collection should only persist for a pre-defined period of time, the TTL feature can be used to automatically delete documents of a certain age rather than scheduling a process to check the age of all documents and run a series of deletes. For example, if user sessions should only exist for one hour, the TTL can be set to 3600 seconds for a date field called `lastActivity` that exists in documents used to track user sessions and their last interaction with the system. A background thread will automatically check all these documents and delete those that have been idle for more than 3600 seconds. Another example use case for TTL is a price quote that should automatically expire after a period of time.

Capped Collections

In some cases a rolling window of data should be maintained in the system based on data size. Capped collections are fixed-size collections that support high-throughput inserts and reads based on insertion order. A capped collection behaves like a circular buffer: data is inserted into the collection, that insertion order is preserved, and when the total size reaches the threshold of the capped collection, the oldest documents are deleted to make room for the newest documents. For example, store

log information from a high-volume system in a capped collection to quickly retrieve the most recent log entries.

Dropping a Collection

It is very efficient to drop a collection in MongoDB. If your data lifecycle management requires periodically deleting large volumes of documents, it may be best to model those documents as a single collection. Dropping a collection is much more efficient than removing all documents or a large subset of a collection, just as dropping a table is more efficient than deleting all the rows in a table in a relational database.

Disk space is automatically reclaimed after a collection is dropped.

Indexing

Like most database management systems, indexes are a crucial mechanism for optimizing MongoDB query performance. While indexes will improve the performance of some operations by one or more orders of magnitude, they incur overhead to updates, disk space, and memory usage. Users should always create indexes to support queries, but should not maintain indexes that queries do not use. This is particularly important for deployments that support insert-heavy (or writes which modify indexed values) workloads.

To understand the effectiveness of the existing indexes being used, an `$indexStats` aggregation stage can be used to determine how frequently each index is used. This information can also be accessed through MongoDB Compass.

Query Optimization

Queries are automatically optimized by MongoDB to make evaluation of the query as efficient as possible. Evaluation normally includes the selection of data based on predicates, and the sorting of data based on the sort criteria provided. The query optimizer selects the best indexes to use by periodically running alternate query plans and selecting the index with the best performance for each query type. The results of this empirical test are stored as a cached query plan and periodically updated.

MongoDB provides an `explain` plan capability that shows information about how a query will be, or was, resolved, including:

- The number of documents returned
- The number of documents read
- Which indexes were used
- Whether the query was covered, meaning no documents needed to be read to return results
- Whether an in-memory sort was performed, which indicates an index would be beneficial
- The number of index entries scanned
- How long the query took to resolve in milliseconds (when using the `executionStats` mode)
- Which alternative query plans were rejected (when using the `allPlansExecution` mode)

MongoDB provides the ability to view how a query will be evaluated in the system, including which indexes are used and whether the query is covered. This capability is similar to the Explain Plan and similar features in relational databases. You should test every query in your application using `explain()`.

The explain plan will show 0 milliseconds if the query was resolved in less than 1 ms, which is typical in well-tuned systems. When the explain plan is called, prior cached query plans are abandoned, and the process of testing multiple indexes is repeated to ensure the best possible plan is used. The query plan can be calculated and returned without first having to run the query. This enables DBAs to review which plan will be used to execute the query, without having to wait for the query to run to completion. The feedback from `explain()` will help you understand whether your query is performing optimally.

Profiling

MongoDB provides a profiling capability called Database Profiler, which logs fine-grained information about database operations. The profiler can be enabled to log information for all events or only those events whose duration exceeds a configurable threshold (whose default is 100 ms). Profiling data is stored in a capped collection

where it can easily be searched for relevant events. It may be easier to query this collection than parsing the log files.

Both the MongoDB Atlas and the MongoDB Compass GUI visualize additional real-time server statistics:

- Read and write activity
- A comprehensive overview of all operations, including counts of updates, inserts, page faults, index misses, and many other important measures of the system health

Primary and Secondary Indexes

A unique index on the `_id` attribute is created for all documents. MongoDB will automatically create the `_id` field and assign a unique value if the value is not be specified when the document is inserted. All user-defined indexes are secondary indexes. MongoDB includes support for many types of secondary indexes that can be declared on any field(s) in the document, including fields within arrays and sub-documents. Index options include:

- Compound indexes
- Geospatial indexes
- Text search indexes
- Unique indexes
- Array indexes
- TTL indexes
- Sparse indexes
- Partial Indexes
- Hash indexes
- Collated indexes for different languages

You can learn more about each of these indexes from the MongoDB Architecture Guide

Index Creation Options

Indexes and data are updated synchronously in MongoDB, thus ensuring queries on indexes never return stale or deleted data. The appropriate indexes should be determined as part of the schema design process. By default creating an index is a blocking operation in MongoDB. Because the creation of indexes can be time and resource intensive, MongoDB provides an option for creating new indexes as a background operation on both the primary and secondary members of a replica set. When the background option is enabled, the total time to create an index will be greater than if the index was created in the foreground, but it will still be possible to query the database while creating indexes.

Index Limitations

As with any database, indexes consume disk space and memory, so should only be used as necessary. Indexes can impact update performance. An update must first locate the data to change, so an index will help in this regard, but index maintenance itself has overhead and this work will reduce update performance.

There are several index limitations that should be observed when deploying MongoDB:

- A collection cannot have more than 64 indexes
- Index entries cannot exceed 1024 bytes
- The name of an index must not exceed 125 characters (including its namespace)
- In-memory sorting of data without an index is limited to 32MB. This operation is very CPU intensive, and in-memory sorts indicate an index should be created to optimize these queries

Common Mistakes Regarding Indexes

The following tips may help to avoid some common mistakes regarding indexes:

- **Use a compound index rather than index intersection:** For best performance when querying via multiple predicates, compound indexes will generally be a better option
- **Compound indexes**: Compound indexes are defined and ordered by field. So, if a compound index is defined for `last name`, `first name` and `city`, queries that specify `last name` or `last name` and `first name` will be able to use this index, but queries that try to search based on `city` will not be able to benefit from this index. Remove indexes that are prefixes of other indexes

- **Low selectivity indexes**: An index should radically reduce the set of possible documents to select from. For example, an index on a field that indicates gender is not as beneficial as an index on zip code, or even better, phone number
- **Regular expressions**: Indexes are ordered by value, hence leading wildcards are inefficient and may result in full index scans. Trailing wildcards can be efficient if there are sufficient case-sensitive leading characters in the expression
- **Negation**: Inequality queries can be inefficient with respect to indexes. Like most database systems, MongoDB does not index the absence of values and negation conditions may require scanning all documents. If negation is the only condition and it is not selective (for example, querying an orders table, where 99% of the orders are complete, to identify those that have not been fulfilled), all records will need to be scanned.
- **Eliminate unnecessary indexes**: Indexes are resource-intensive: even with they consume RAM, and as fields are updated their associated indexes must be maintained, incurring additional disk I/O overhead. To understand the effectiveness of the existing indexes being used, an `$indexStats` aggregation stage can be used to determine how frequently each index is used. If there are indexes that are not used then removing them will reduce storage and speed up writes. Index usage can also be viewed through the MongoDB Compass GUI.
- **Partial indexes**: If only a subset of documents need to be included in a given index then the index can be made partial by specifying a filter expression. e.g., if an index on the `userID` field is only needed for querying open orders then it can be made conditional on the order status being set to *in progress*. In this way, partial indexes improve query performance while minimizing overheads.

Working Sets

MongoDB makes extensive use of RAM to speed up database operations. In MongoDB, all data is read and manipulated through in-memory representations of the data. Reading data from memory is measured in nanoseconds and reading data from disk is measured in milliseconds, thus reading from memory is orders of magnitude faster than reading from disk.

The set of data and indexes that are accessed during normal operations is called the working set. It is best practice that the working set fits in RAM. It may be the case the working set represents a fraction of the entire database, such as in applications where data related to recent events or popular products is accessed most commonly.

When MongoDB attempts to access data that has not been loaded in RAM, it must be read from disk. If there is free memory then the operating system can locate the data on disk and load it into memory directly. However, if there is no free memory, MongoDB must write some other data from memory to disk, and then read the requested data into memory. This process can be time consuming and significantly slower than accessing data that is already resident in memory.

Some operations may inadvertently purge a large percentage of the working set from memory, which adversely affects performance. For example, a query that scans all documents in the database, where the database is larger than available RAM on the server, will cause documents to be read into memory and may lead to portions of the working set being written out to disk. Other examples include various maintenance operations such as compacting or repairing a database and rebuilding indexes.

If your database working set size exceeds the available RAM of your system, consider provisioning an instance with larger RAM capacity (scaling up) or sharding the database across additional instances (scaling out). Scaling is an automated, on-line operation which is launched by selecting the new configuration after clicking the *CONFIGURE* button in MongoDB Atlas (Figure 2). For a discussion on this topic, refer to the section on Sharding Best Practices later in the guide. It is easier to implement sharding before the system's resources are consumed, so capacity planning is an important element in successful project delivery.

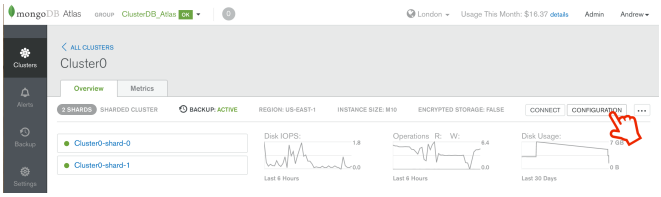

Figure 2: Reconfigure MongoDB Atlas Cluster

Data Migration

Users should assess how best to model their data for their applications rather than simply importing the flat file exports of their legacy systems. In a traditional relational database environment, data tends to be moved between systems using delimited flat files such as CSV. While it is possible to ingest data into MongoDB from CSV files, this may in fact only be the first step in a data migration process. It is typically the case that MongoDB's document data model provides advantages and alternatives that do not exist in a relational data model.

There are many options to migrate data from flat files into rich JSON documents, including `mongoimport`, custom scripts, ETL tools and from within an application itself which can read from the existing RDBMS and then write a JSON version of the document back to MongoDB.

For importing data from a pre-existing MongoDB system, MongoDB Atlas includes a live migration service built into the GUI. This functionality works with any MongoDB replica set running MongoDB 3.0 or higher.

Other tools such as `mongodump` and `mongorestore`, or MongoDB Atlas backups are also useful for moving data between different MongoDB systems. The use of `mongodump` and `mongorestore` to migrate an application and its data to MongoDB Atlas is described in the post – Migrating Data to MongoDB Atlas.

MongoDB Atlas Instance Selection

The following recommendations are only intended to provide high-level guidance for hardware for a MongoDB deployment. The specific configuration of your hardware will be dependent on your data, queries, performance SLA, and availability requirements.

Memory

As with most databases, MongoDB performs best when the working set (indexes and most frequently accessed data) fits in RAM. Sufficient RAM is the most important factor for instance selection; other optimizations may not significantly improve the performance of the system if there is insufficient RAM. When selecting which MongoDB Atlas instance size to use, opt for one that has sufficient RAM to hold the full working data set (or the appropriate subset if sharding).

If your working set exceeds the available RAM, consider using a larger instance type or adding additional shards to your system.

Storage

Using faster storage can increase database performance and latency consistency. Each node must be configured with sufficient storage for the full data set, or for the subset to be stored in a single shard. The storage speed and size can be set when picking the MongoDB Atlas instance during cluster creation or reconfiguration.

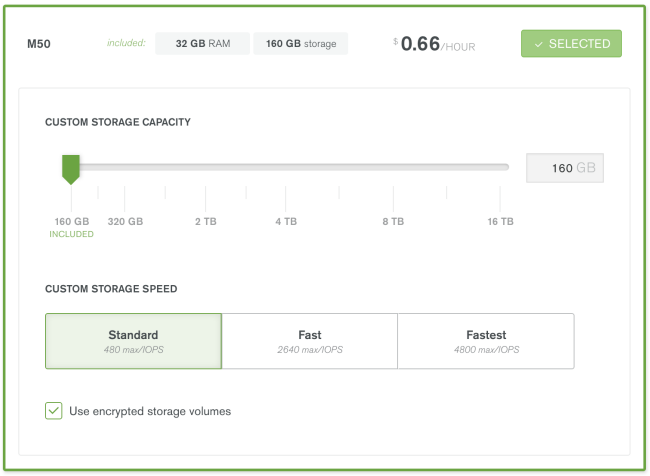

Figure 3: Select instance size and storage size & speed

Data volumes can optionally be encrypted which increases security at the expense of reduced performance.

CPU

MongoDB Atlas instances are multi-threaded and can take advantage of many CPU cores. Specifically, the total

number of active threads (i.e., concurrent operations) relative to the number of CPUs can impact performance:

- Throughput increases as the number of concurrent active operations increases up to and beyond the number of CPUs
- Throughput eventually decreases as the number of concurrent active operations exceeds the number of CPUs by some threshold amount

The threshold amount depends on your application. You can determine the optimum number of concurrent active operations for your application by experimenting and measuring throughput.

The larger MongoDB Atlas instances include more virtual CPUs and so should be considered for heavily concurrent workloads.

Scaling a MongoDB Atlas Cluster

Horizontal Scaling with Sharding

MongoDB Atlas provides horizontal scale-out for databases using a technique called sharding, which is transparent to applications. MongoDB distributes data across multiple Replica Sets called shards. With automatic balancing, MongoDB ensures data is equally distributed across shards as data volumes grow or the size of the cluster increases or decreases. Sharding allows MongoDB deployments to scale beyond the limitations of a single server, such as bottlenecks in RAM or disk I/O, without adding complexity to the application.

MongoDB Atlas supports multiple sharding policies, enabling administrators to accommodate diverse query patterns:

- **Range-based sharding:** Documents are partitioned across shards according to the shard key value. Documents with shard key values close to one another are likely to be co-located on the same shard. This approach is well suited for applications that need to optimize range-based queries.
- **Hash-based sharding:** Documents are uniformly distributed according to an MD5 hash of the shard key value. Documents with shard key values close to one another are unlikely to be co-located on the same shard. This approach guarantees a uniform distribution of writes across shards – provided that the shard key has high cardinality – making it optimal for write-intensive workloads.

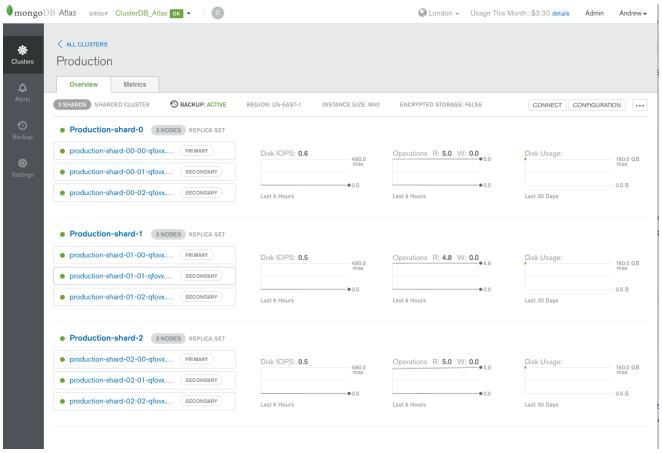

Figure 4: Horizontal Scaling with MongoDB Atlas

Users should consider deploying a sharded cluster in the following situations:

- **RAM Limitation:** The size of the system's active working set plus indexes is expected to exceed the capacity of the maximum amount of RAM in the system
- **Disk I/O Limitation:** The system will have a large amount of write activity, and the operating system will not be able to write data fast enough to meet demand, or I/O bandwidth will limit how fast the writes can be flushed to disk
- **Storage Limitation:** The data set will grow to exceed the storage capacity of a single node in the system

Applications that meet these criteria, or that are likely to do so in the future, should be designed for sharding in advance rather than waiting until they have consumed available capacity. Applications that will eventually benefit from sharding should consider which collections they will want to shard and the corresponding shard keys when designing their data models. If a system has already reached or exceeded its capacity, it will be challenging to deploy sharding without impacting the application's performance.

Between 1 and 12 shards can be self-configured in MongoDB Atlas GUI; customers interested in more than 12 shards should contact MongoDB.

Sharding Best Practices

Users who choose to shard should consider the following best practices.

Select a good shard key: When selecting fields to use as a shard key, there are at least three key criteria to consider:

1. Cardinality: Data partitioning is managed in 64 MB chunks by default. Low cardinality (e.g., a user's home country) will tend to group documents together on a small number of shards, which in turn will require frequent rebalancing of the chunks and a single country is likely to exceed the 64 MB chunk size. Instead, a shard key should exhibit high cardinality.
2. Insert Scaling: Writes should be evenly distributed across all shards based on the shard key. If the shard key is monotonically increasing, for example, all inserts will go to the same shard even if they exhibit high cardinality, thereby creating an insert hotspot. Instead, the key should be evenly distributed.
3. Query Isolation: Queries should be targeted to a specific shard to maximize scalability. If queries cannot be isolated to a specific shard, all shards will be queried in a pattern called scatter/gather, which is less efficient than querying a single shard.
4. Ensure uniform distribution of shard keys: When shard keys are not uniformly distributed for reads and writes, operations may be limited by the capacity of a single shard. When shard keys are uniformly distributed, no single shard will limit the capacity of the system.

For more on selecting a shard key, see Considerations for Selecting Shard Keys.

Avoid scatter-gather queries: In sharded systems, queries that cannot be routed to a single shard must be broadcast to multiple shards for evaluation. Because these queries involve multiple shards for each request they do not scale well as more shards are added.

Use hash-based sharding when appropriate: For applications that issue range-based queries, range-based sharding is beneficial because operations can be routed to the fewest shards necessary, usually a single shard. However, range-based sharding requires a good understanding of your data and queries, which in some cases may not be practical. Hash-based sharding ensures a uniform distribution of reads and writes, but it does not provide efficient range-based operations.

Apply best practices for bulk inserts: Pre-split data into multiple chunks so that no balancing is required during the insert process. For more information see Create Chunks in a Sharded Cluster in the MongoDB Documentation.

Add capacity before it is needed: Cluster maintenance is lower risk and more simple to manage if capacity is added before the system is over utilized.

Continuous Availability & Data Consistency

Data Redundancy

MongoDB maintains multiple copies of data, called replica sets, using native replication. Replica failover is fully automated in MongoDB, so it is not necessary to manually intervene to recover nodes in the event of a failure.

A replica set consists of multiple replica nodes. At any given time, one member acts as the primary replica and the other members act as secondary replicas. If the primary member fails for any reason (e.g., a failure of the host system), one of the secondary members is automatically elected to primary and begins to accept all writes; this is typically completed in 2 seconds or less and reads can optionally continue on the secondaries.

Sophisticated algorithms control the election process, ensuring only the most suitable secondary member is promoted to primary, and reducing the risk of unnecessary failovers (also known as "false positives"). The election algorithm processes a range of parameters including analysis of histories to identify those replica set members that have applied the most recent updates from the primary and heartbeat and connectivity status.

A larger number of replica nodes provides increased protection against database downtime in case of multiple

machine failures. A MongoDB Atlas replica set can be configured with 3, 5, or 7 replicas.

More information on replica sets can be found on the Replication MongoDB documentation page.

Write Guarantees

MongoDB allows administrators to specify the level of persistence guarantee when issuing writes to the database, which is called the write concern. The following options can be selected in the application code:

- **Write Acknowledged:** This is the default write concern. The `mongod` will confirm the execution of the write operation, allowing the client to catch network, duplicate key, Document Validation, and other exceptions
- **Replica Acknowledged:** It is also possible to wait for acknowledgment of writes to other replica set members. MongoDB supports writing to a specific number of replicas. This mode also ensures that the write is written to the journal on the secondaries. Because replicas can be deployed across racks within data centers and across multiple data centers, ensuring writes propagate to additional replicas can provide extremely robust durability
- **Majority:** This write concern waits for the write to be applied to a majority of replica set members, and that the write is recorded in the journal on these replicas – including on the primary

Read Preferences

Updates are typically replicated to secondaries quickly, depending on network latency. However, reads on the secondaries will not normally be consistent with reads on the primary. Note that the secondaries are not idle as they must process all writes replicated from the primary. To increase read capacity in your operational system consider sharding. Secondary reads can be useful for analytics and ETL applications as this approach will isolate traffic from operational workloads. You may choose to read from secondaries if your application can tolerate eventual consistency.

Reading from the primary replica is the default configuration as it guarantees consistency. If higher read throughput is required, it is recommended to take advantage of MongoDB's auto-sharding to distribute read operations across multiple primary members.

There are applications where replica sets can improve scalability of the MongoDB deployment. For example, analytics and Business Intelligence (BI) applications can execute queries against a secondary replica, thereby reducing overhead on the primary and enabling MongoDB to serve operational and analytical workloads from a single deployment.

A very useful option is `primaryPreferred`, which issues reads to a secondary replica only if the primary is unavailable. This configuration allows for the continuous availability of reads during the short failover process.

For more on the subject of configurable reads, see the MongoDB Documentation page on replica set Read Preference.

Managing MongoDB: Provisioning, Monitoring and Disaster Recovery

Created by the engineers who develop the database, MongoDB Atlas is the simplest way to run MongoDB, making it easy to deploy, monitor, backup, and scale MongoDB.

MongoDB Atlas incorporates best practices to help keep managed databases healthy and optimized. They ensure operational continuity by converting complex manual tasks into reliable, automated procedures with the click of a button:

- **Deploy.** Using your choice of instance size, number of replica set members, and number of shards
- **Scale.** Add capacity, without taking the application offline
- **Point-in-time, Scheduled Backups.** Restore complete running clusters to any point in time with just a few clicks, because disasters aren't predictable

- **Performance Alerts.** Monitor system metrics and get custom alerts

Deployments and Upgrades

All the user needs to do in order for MongoDB Atlas to automatically deploy the cluster is to select a handful of options:

- Instance size
- Storage size (optional)
- Storage speed (optional)
- Data volume encryption
- Number of replicas in the replica set
- Number of shards (optional)
- Automated backups

The database nodes will automatically be kept up date with the latest stable MongoDB and underlying operating system software versions; rolling upgrades ensure that your applications are not impacted during upgrades.

Monitoring & Capacity Planning

System performance and capacity planning are two important topics that should be addressed as part of any MongoDB deployment. Part of your planning should involve establishing baselines on data volume, system load, performance, and system capacity utilization. These baselines should reflect the workloads you expect the system to perform in production, and they should be revisited periodically as the number of users, application features, performance SLA, or other factors change.

Featuring charts and automated alerting, MongoDB Atlas tracks key database and system health metrics including disk free space, operations counters, memory and CPU utilization, replication status, open connections, queues, and node status.

Historic performance can be reviewed in order to create operational baselines and to support capacity planning. Integration with existing monitoring tools is also straightforward via the MongoDB Atlas RESTful API, making the deep insights from MongoDB Atlas part of a consolidated view across your operations.

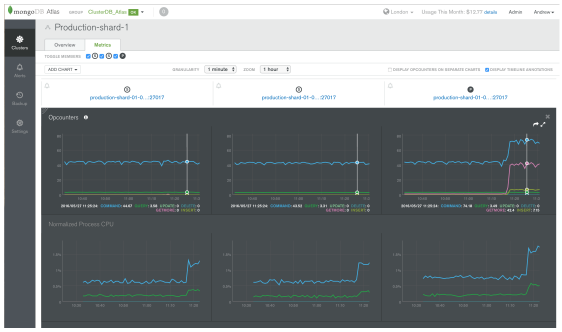

Figure 5: Database monitoring with MongoDB Atlas GUI

MongoDB Atlas allows administrators to set custom alerts when key metrics are out of range. Alerts can be configured for a range of parameters affecting individual hosts and replica sets. Alerts can be sent via email, webhooks, Flowdock, HipChat, and Slack or integrated into existing incident management systems such as PagerDuty.

When it's time to scale, just hit the CONFIGURATION button in the MongoDB Atlas GUI and choose the required instance size and number of shards – the automated, on-line scaling will then be performed.

Things to Monitor

MongoDB Atlas monitors database-specific metrics, including page faults, ops counters, queues, connections and replica set status. Alerts can be configured against each monitored metric to proactively warn administrators of potential issues before users experience a problem. The MongoDB Atlas team are also monitoring the underlying infrastructure, ensuring that it is always in a healthy state.

Application Logs And Database Logs

Application and database logs should be monitored for errors and other system information. It is important to correlate your application and database logs in order to determine whether activity in the application is ultimately responsible for other issues in the system. For example, a spike in user writes may increase the volume of writes to MongoDB, which in turn may overwhelm the underlying storage system. Without the correlation of application and database logs, it might take more time than necessary to establish that the application is responsible for the

increase in writes rather than some process running in MongoDB.

Page Faults

When a working set ceases to fit in memory, or other operations have moved working set data out of memory, the volume of page faults may spike in your MongoDB system.

Disk

Beyond memory, disk I/O is also a key performance consideration for a MongoDB system because writes are journaled and regularly flushed to disk. Under heavy write load the underlying disk subsystem may become overwhelmed, or other processes could be contending with MongoDB, or the storage speed chosen may be inadequate for the volume of writes.

CPU

A variety of issues could trigger high CPU utilization. This may be normal under most circumstances, but if high CPU utilization is observed without other issues such as disk saturation or `pagefaults`, there may be an unusual issue in the system. For example, a MapReduce job with an infinite loop, or a query that sorts and filters a large number of documents from the working set without good index coverage, might cause a spike in CPU without triggering issues in the disk system or `pagefaults`.

Connections

MongoDB drivers implement connection pooling to facilitate efficient use of resources. Each connection consumes 1MB of RAM, so be careful to monitor the total number of connections so they do not overwhelm RAM and reduce the available memory for the working set. This typically happens when client applications do not properly close their connections, or with Java in particular, that relies on garbage collection to close the connections.

Op Counters

The utilization baselines for your application will help you determine a normal count of operations. If these counts start to substantially deviate from your baselines it may be an indicator that something has changed in the application, or that a malicious attack is underway.

Queues

If MongoDB is unable to complete all requests in a timely fashion, requests will begin to queue up. A healthy deployment will exhibit very short queues. If metrics start to deviate from baseline performance, requests from applications will start to queue. The queue is therefore a good first place to look to determine if there are issues that will affect user experience.

Shard Balancing

One of the goals of sharding is to uniformly distribute data across multiple servers. If the utilization of server resources is not approximately equal across servers there may be an underlying issue that is problematic for the deployment. For example, a poorly selected shard key can result in uneven data distribution. In this case, most if not all of the queries will be directed to the single `mongod` that is managing the data. Furthermore, MongoDB may be attempting to redistribute the documents to achieve a more ideal balance across the servers. While redistribution will eventually result in a more desirable distribution of documents, there is substantial work associated with rebalancing the data and this activity itself may interfere with achieving the desired performance SLA.

If in the course of a deployment it is determined that a new shard key should be used, it will be necessary to reload the data with a new shard key because designation and values of the shard keys are immutable. To support the use of a new shard key, it is possible to write a script that reads each document, updates the shard key, and writes it back to the database.

Replication Lag

Replication lag is the amount of time it takes a write operation on the primary replica set member to replicate to

a secondary member. A small amount of delay is normal, but as replication lag grows, significant issues may arise.

If this is observed then replication throughput can be increased by moving to larger MongoDB Atlas instances or adding shards.

Disaster Recovery: Backup & Restore

A backup and recovery strategy is necessary to protect your mission-critical data against catastrophic failure, such as a software bug or a user accidentally dropping collections. With a backup and recovery strategy in place, administrators can restore business operations without data loss, and the organization can meet regulatory and compliance requirements. Taking regular backups offers other advantages, as well. The backups can be used to seed new environments for development, staging, or QA without impacting production systems.

MongoDB Atlas backups are maintained continuously, just a few seconds behind the operational system. If the MongoDB cluster experiences a failure, the most recent backup is only moments behind, minimizing exposure to data loss.

In additional, MongoDB Atlas includes queryable backups, which allows you to perform queries against existing snapshots to more easily restore data at the document/object level. Queryable backups allow you to accomplish the following with less time and effort:

- Restore a subset of objects/documents within the MongoDB cluster
- Identify whether data has been changed in an undesirable way by looking at previous versions alongside current data
- Identify the best point in time to restore a system by comparing data from multiple snapshots

mongodump

`mongodump` is a tool bundled with MongoDB that performs a live backup of the data in MongoDB. `mongodump` may be used to dump an entire database, collection, or result of a query. `mongodump` can produce a dump of the data that reflects a single moment in time by dumping the oplog entries created during the dump and then replaying it during `mongorestore`, a tool that imports content from BSON database dumps produced by `mongodump`.

In the vast majority of cases, MongoDB Atlas backups delivers the simplest, safest, and most efficient backup solution. `mongodump` is useful when data needs to be exported to another system, when a local backup is needed, or when just a subset of the data needs to be backed up.

Integrating MongoDB with External Monitoring Solutions

The MongoDB Atlas API provides integration with external management frameworks through programmatic access to automation features and alerts.

APM Integration

Many operations teams use Application Performance Monitoring (APM) platforms to gain global oversight of their complete IT infrastructure from a single management UI. Issues that risk affecting customer experience can be quickly identified and isolated to specific components – whether attributable to devices, hardware infrastructure, networks, APIs, application code, databases and, more.

The MongoDB drivers include an API that exposes query performance metrics to APM tools. Administrators can monitor time spent on each operation, and identify slow running queries that require further analysis and optimization.

In addition, MongoDB Atlas provides packaged integration with the New Relic platform. Key metrics from MongoDB Atlas are accessible to the APM for visualization, enabling MongoDB health to be monitored and correlated with the rest of the application estate.

As shown in Figure 6, summary metrics are presented within the APM's UI. Administrators can also run New Relic Insights for analytics against monitoring data to generate dashboards that provide real-time tracking of Key Performance Indicators (KPIs).

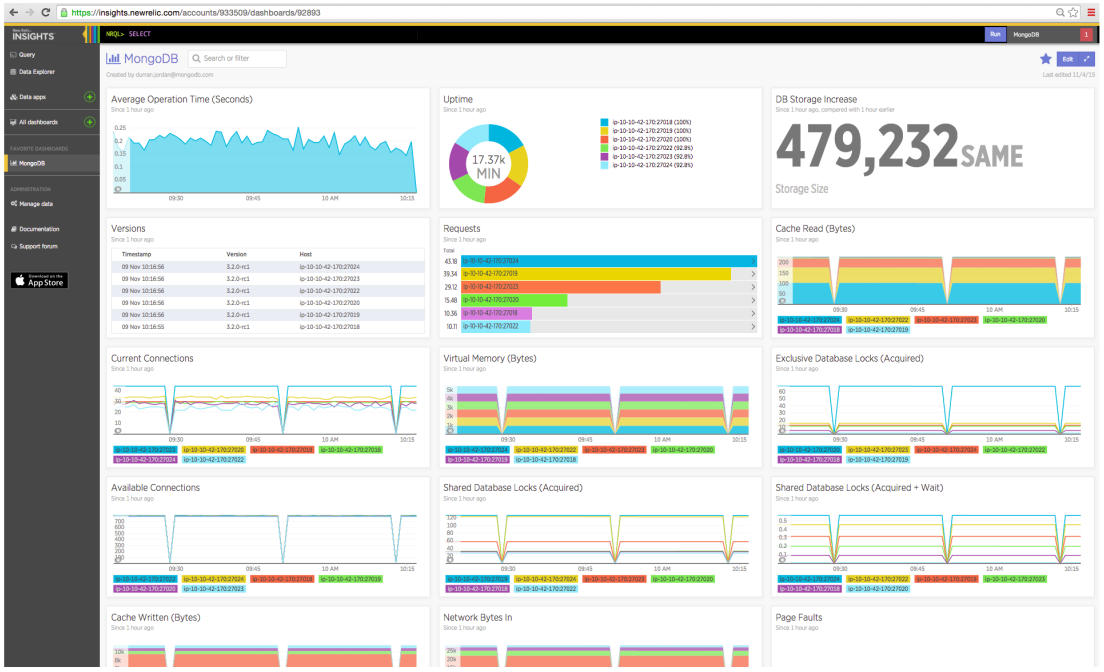

Figure 6: MongoDB integrated into a single view of application performance

Security

As with all software, MongoDB administrators must consider security and risk exposure for a MongoDB deployment. There are no magic solutions for risk mitigation, and maintaining a secure MongoDB deployment is an ongoing process.

Defense in Depth

A Defense in Depth approach is recommended for securing MongoDB deployments, and it addresses a number of different methods for managing risk and reducing risk exposure.

MongoDB Atlas features extensive capabilities to defend, detect, and control access to MongoDB, offering among the most complete security controls of any modern database:

- **User Rights Management.** Control access to sensitive data using industry standard mechanisms for authentication and authorization at the database level
- **Encryption.** Protect data in motion over the network and at rest in persistent storage

To ensure a secure system right out of the box, authentication and IP Address whitelisting are automatically enabled.

Review the security section of the MongoDB Atlas documentation to learn more about each of the security features discussed below.

IP Whitelisting

Clients are prevented from accessing the database unless their IP address (or a CIDR covering their IP address) has been added to the IP whitelist for your MongoDB Atlas group.

VPC Peering

Virtual Private Cloud (VPC) Peering allows users to create an extended, private network that connects the AWS VPC housing their application servers with the VPC containing the backend databases. VPC peering achieves this connectivity without using public IP addresses, and without needing to whitelist every client in your MongoDB Atlas group.

Authorization

MongoDB Atlas allows administrators to define permissions for a user or application, and what data it can access when querying MongoDB. MongoDB Atlas provides the ability to provision users with roles specific to a database, making it possible to realize a separation of duties between different entities accessing and managing the data.

Additionally, MongoDB's Aggregation Framework Pipeline includes a stage to implement Field-Level Redaction, providing a method to restrict the content of a returned document on a per-field level, based on user permissions. The application must pass the redaction logic to the database on each request. It therefore relies on trusted middleware running in the application to ensure the redaction pipeline stage is appended to any query that requires the redaction logic.

Encryption

MongoDB Atlas provides encryption of data in flight over the network and at rest on disk.

Support for SSL/TLS allows clients to connect to MongoDB over an encrypted channel. Clients are defined as any entity capable of connecting to MongoDB Atlas, including:

- Users and administrators
- Applications
- MongoDB tools (e.g., mongodump, mongorestore)
- Nodes that make up a MongoDB Atlas cluster, such as replica set members and query routers.

Data at rest can optionally be protected using encrypted data volumes.

Read-Only, Redacted Views

DBAs can define non-materialized views that expose only a subset of data from an underlying collection, i.e. a view that filters out specific fields. DBAs can define a view of a collection that's generated from an aggregation over another collection(s) or view.

Views are defined using the standard MongoDB Query Language and aggregation pipeline. They allow the inclusion or exclusion of fields, masking of field values, filtering, schema transformation, grouping, sorting, limiting, and joining of data using $lookup and $graphLookup to another collection.

You can learn more about MongoDB read-only views from the documentation.

Considerations for Proofs of Concept

Generic benchmarks can be misleading and misrepresentative of a technology and how well it will perform for a given application. MongoDB instead recommends that users model and benchmark their applications using data, queries, instance sizes, and other aspects of the system that are representative of their intended application. The following considerations will help you develop benchmarks that are meaningful for your application:

- **Model your benchmark on your application**: The queries, data, system configurations, and performance goals you test in a benchmark exercise should reflect the goals of your production system. Testing assumptions that do not reflect your production system is likely to produce misleading results.
- **Create chunks before loading, or use hash-based sharding**: If range queries are part of your benchmark use range-based sharding and create chunks before loading. Without pre-splitting, data may be loaded into a shard then moved to a different shard as the load progresses. By pre-splitting the data, documents will be loaded in parallel into the appropriate shards. If your benchmark does not include range queries, you can use hash-based sharding to ensure a uniform distribution of writes.
- **Prime the system for several minutes**: In a production MongoDB system the working set should fit in RAM, and all reads and writes will be executed against RAM. MongoDB must first fetch the working set into RAM, so prime the system with representative queries for several minutes before running the tests to

get an accurate sense for how MongoDB will perform in production.

- **Monitor everything to locate your bottlenecks**: It is important to understand the bottleneck for a benchmark. Depending on many factors any component of the overall system could be the limiting factor. A variety of popular tools can be used with MongoDB – many are listed in the manual.
- **Profiling**: MongoDB provides a profiling capability called Database Profiler, which logs fine-grained information about database operations. The profiler can be enabled to log information for all events or only those events whose duration exceeds a configurable threshold (whose default is 100 ms). Profiling data is stored in a capped collection where it can easily be searched for relevant events.

Conclusion

MongoDB is the next-generation database used by the world's most sophisticated organizations, from cutting-edge startups to the largest companies, to create applications never before possible at a fraction of the cost of legacy databases. MongoDB is the fastest-growing database ecosystem, with over 15 million downloads, thousands of customers, and over 1,000 technology and service partners.

MongoDB Atlas automates the operational tasks that usually burdens the user, freeing you up to focus on what you do best – delivering great applications. There remain some tasks that will keep your application running smoothly and quickly; this paper describes those best practices.

We Can Help

We are the MongoDB experts. Over 2,000 organizations rely on our commercial products, including startups and more than a third of the Fortune 100. 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 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 Professional helps you manage your deployment and keep it running smoothly. It includes support from MongoDB engineers, as well as access to MongoDB Cloud Manager.

Development Support helps you get up and running quickly. It gives you a complete package of software and services for the early stages of your project.

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)

New York • Palo Alto • Washington, D.C. • London • Dublin • Barcelona • Sydney • Tel Aviv
US 866-237-8815 • INTL +1-650-440-4474 • info@mongodb.com