A MongoDB White Paper

MongoDB Operations Best Practices

MongoDB 3.6
November 2017

mongoDB

Table of Contents

Introduction

MongoDB is a high-performance, scalable, distributed database designed for a broad array of modern applications. It is used by organizations of all sizes to power mission-critical, operational applications where low latency, high throughput, and continuous availability are critical requirements of the system.

While some aspects of MongoDB are different from traditional relational databases, the concepts of the system, its operations, policies, and procedures will be familiar to staff who have deployed and operated other database systems. Organizations have found that DBAs and operations teams are able to preserve existing investments by integrating MongoDB into their production environments, without needing to customize established operational processes or tools.

This paper provides guidance on best practices for deploying and managing MongoDB. It assumes familiarity with the architecture of MongoDB and an understanding of concepts related to the deployment of enterprise software.

This guide is aimed at users managing everything themselves. A dedicated guide is provided for users of the MongoDB database as a service – MongoDB Atlas Best Practices.

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.

Roles and Responsibilities

As with any database, applications deployed on MongoDB require careful planning and the coordination of a number of roles in an organization's IT teams to ensure successful deployment and operation. Organizations tend to find that most of the same individuals and their respective roles for traditional database applications are appropriate for a MongoDB environment: Database Administrators, System Administrators, Application Developers, Network Administrators, Business Analysts, and Data Architects.

In smaller organizations it is common for IT staff to fulfill multiple roles, whereas in larger companies it is more common for each role to be assumed by an individual or team dedicated to those tasks. For example, in a large investment bank there may be a very strong delineation between the functional responsibilities of a DBA and those of a system administrator.

Database Administrator (DBA)

As with other database systems, many factors should be considered in designing a MongoDB system for a desired performance and availability SLA. The DBA should be involved early in the project regarding discussions of the data model, the types of queries that will be issued to the system, the query volume, the availability and recovery goals, and the desired performance characteristics.

System Administrator (Sysadmin)

Sysadmins typically perform a set of activities similar to those required in managing other applications, including upgrading software and hardware, managing storage, system monitoring, and data migration. MongoDB users have reported that their sysadmins have had no trouble learning to deploy, manage, and monitor MongoDB because no special skills are required.

Application Developer

The application developer works with other members of the project team to ensure the requirements regarding functionality, deployment, security, and availability are clearly understood. The application itself is written in a language such as Java, C#, PHP, or Ruby. Data will be stored, updated, and queried in MongoDB, and language-specific drivers are used to communicate between MongoDB and the application. The application developer works with the data architect to define and evolve the data model and to define the query patterns that should be optimized. The application developer works with the database administrator, sysadmin, and network administrator to define the deployment and availability requirements of the application.

Network Administrator

A MongoDB deployment typically involves multiple servers distributed within and across multiple data centers. Network resources are a critical component of a MongoDB system. While MongoDB does not require any custom configurations or resources as compared to other database systems, the network administrator should be consulted to ensure the appropriate policies, procedures, configurations, capacity, and security settings are implemented for the project.

Data Architect

While modeling data for MongoDB is typically simpler than modeling data for a relational database, there tend to be multiple options for a data model, and each has trade-offs regarding performance, resource utilization, ease of use, and other areas. The data architect can carefully weigh these options with the development team to make informed decisions regarding the design of the schema. Typically the data architect performs tasks that are more proactive in nature, whereas the database administrator may perform tasks that are more reactive, once the application has been deployed.

Business Analyst

Business Analysts need to draw insights from the data held in MongoDB in order to drive the direction of the business. In particular, they need to be able to view the data in multiple ways, often combining data from multiple collections, and from multiple databases. While the overall schema is more likely to be driven by the operational requirements, the Business Analyst has a vested interest in making sure that appropriate indexes are in-place for their real-time analytics requirements as well as methods to work with the in their preferred analytics and visualization platforms.

Preparing for a MongoDB Deployment

MongoDB Pluggable Storage Engines

MongoDB exposes a storage engine API, enabling the integration of pluggable storage engines that extend MongoDB with new capabilities, and enable optimal use of specific hardware architectures. MongoDB ships with multiple supported storage engines:

- The default **WiredTiger storage engine**. For most applications, WiredTiger's granular concurrency control and native compression will provide the best all-around performance and storage efficiency for the broadest range of applications.
- The **Encrypted storage engine**, protecting highly sensitive data, without the performance or management overhead of separate files system encryption. The Encrypted storage is based upon WiredTiger and so throughout this document, statements regarding WiredTiger also apply to the Encrypted storage engine. This engine is part of MongoDB Enterprise Advanced.
- The **In-Memory storage engine**, delivering predictable latency coupled with real-time analytics for the most demanding, applications. This engine is part of MongoDB Enterprise Advanced.
- The **MMAPv1 storage engine**, an improved version of the storage engine used in pre-3.x MongoDB releases. MMAPv1 was the default storage engine in MongoDB 3.0 and earlier. MMAPv1 should only be used for legacy applications.

MongoDB uniquely allows users to mix and match multiple storage engines within a single MongoDB cluster. This flexibility provides a more simple and reliable approach to meeting diverse application needs for data. Traditionally, multiple database technologies would need to be managed to meet these needs, with complex, custom integration code to move data between the technologies, and to ensure consistent, secure access. While each storage engine is optimized for different workloads, users still leverage the same MongoDB query language, data model, scaling, security, and operational tooling independent of the engine they use. As a result most of best practices in this guide apply to all of the supported storage engines. Any differences in recommendations between the storage engines are noted.

Mixing storage engines within a single MongoDB replica set also makes it easy to evaluate and migrate between each of them. Upgrades to the WiredTiger storage engine are non-disruptive for existing replica set deployments; applications will be 100% compatible, and migrations can be performed with zero downtime through a rolling upgrade of the MongoDB replica set. WiredTiger is the default storage engine for new MongoDB deployments from MongoDB 3.2; if another engine is preferred then start the `mongod` using the `--storageEngine` option. If a 3.2+ `mongod` process is started and one or more databases already exist, then it will use whichever storage engine those databases were created with.

Review the documentation for a checklist and full instructions on the migration process.

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.

The topic of schema design is significant, and a full discussion is beyond the scope of this guide. A number of resources are available on-line, including conference presentations from MongoDB Solutions Architects and users, as well as no-cost, web-based training provided by MongoDB University. MongoDB Global Consulting Services offers assistance in schema design as part of the Development Rapid Start service.. The key schema design concepts to keep in mind are as follows.

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`, `decimal`, 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 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.

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.

Document validation allows DBAs to enforce data governance by applying checks on document 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 3.6 introduces the option to add Schema Validation based on syntax derived from the proposed IETF JSON Schema standard. When specifying the JSON Schema, the DBA can specify whether documents can include additional fields, not specified in the schema.

As an example, you can add a JSON Schema to enforce these rules:

- Each document must contain a field named *lineItems*
- The document may optionally contain other fields
- *lineItems* must be an array where each element:
    - Must contain a *title* (string), *price* (number no smaller than 0)
    - May optionally contain a boolean named *purchased*
    - Must contain no further fields

```
db.createCollection( "orders",
  {validator: {$jsonSchema:

    {
      properties: {
        lineItems:

        {type: "array",
        items:{
          properties: {
            title: {type: "string"},
            price: {type: "number",
                    minimum: 0.0},
            purchased: {type: "boolean"}
          },
          required: ["_id", "title", "price"],
          additionalProperties: false
        }
      }
    },
    required: ["lineItems"]
  }}
})
```

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.

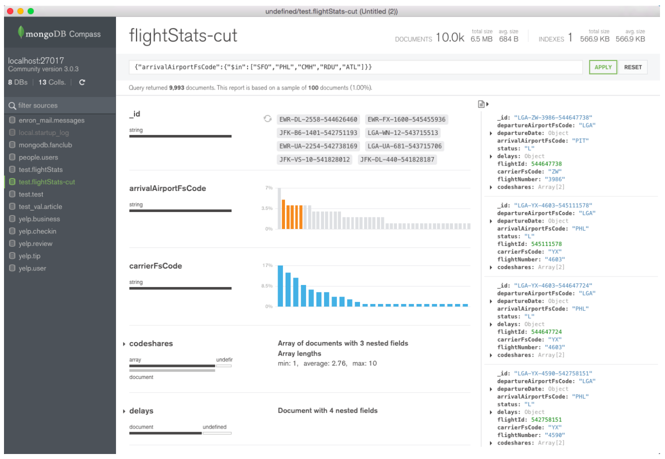
Transactions

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.



```
{"arrivalAirportFsCode":{"$in":["SFO","PHL","CMH","RDU","ATL"]}}
```

Query returned **9,993** documents. This report is based on a sample of **100** documents (1.00%).

```
EWR-DL-2558-544626460   EWR-FX-1600-545455936
JFK-B6-1401-542751193   LGA-WN-12-543715513
EWR-UA-2254-542738169   LGA-UA-681-543715706
JFK-VS-10-541828012     JFK-DL-440-541828187
```

Array of documents with 3 nested fields
Array lengths
min: 1, average: 2.76, max: 10

Document with 4 nested fields

```
  _id: "LGA-ZW-3986-544647738"
  departureAirportFsCode: "LGA"
▸ departureDate: Object
  arrivalAirportFsCode: "PIT"
  status: "L"
▸ delays: Object
  flightId: 544647738
  carrierFsCode: "ZW"
  flightNumber: "3986"
▸ codeshares: Array[2]

  _id: "LGA-YX-4603-545111578"
  departureAirportFsCode: "LGA"
▸ departureDate: Object
  arrivalAirportFsCode: "PHL"
  status: "L"
  flightId: 545111578
  carrierFsCode: "YX"
  flightNumber: "4603"
▸ codeshares: Array[2]

  _id: "LGA-YX-4603-544647724"
  departureAirportFsCode: "LGA"
▸ departureDate: Object
  arrivalAirportFsCode: "PHL"
  status: "L"
▸ delays: Object
  flightId: 544647724
  carrierFsCode: "YX"
  flightNumber: "4603"
▸ codeshares: Array[2]

  _id: "LGA-YX-4590-542758151"
  departureAirportFsCode: "LGA"
▸ departureDate: Object
  arrivalAirportFsCode: "PHL"
  status: "L"
▸ delays: Object
  flightId: 542758151
  carrierFsCode: "YX"
  flightNumber: "4590"
▸ codeshares: Array[2]
```

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

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.

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.

From MongoDB 3.6, developers can view documents as conventional tables, in addition to JSON documents.

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.

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.

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 in MongoDB.

Data Lifecycle Management

MongoDB provides features to facilitate the management of data lifecycles, including Time to Live indexes, and capped collections. In addition, by using MongoDB Zones, administrators can build highly efficient tiered storage models to support the data lifecycle. By assigning shards to Zones, administrators can balance query latency with storage density and cost by assigning data sets based on a value such as a timestamp to specific storage devices:

- Recent, frequently accessed data can be assigned to high performance SSDs with Snappy compression enabled.
- Older, less frequently accessed data is tagged to lower-throughput hard disk drives where it is compressed with zlib to attain maximum storage density with a lower cost-per-bit.
- As data ages, MongoDB automatically migrates it between storage tiers, without administrators having to build tools or ETL processes to manage data movement.

You can learn more about sharding using Zones later in this guide.

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.

When using WiredTiger, disk space is automatically reclaimed after a collection is dropped.

Indexing

Like most database management systems, indexes are a crucial mechanism for optimizing system performance in MongoDB. 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.

For operational simplicity, the MongoDB Ops Manager and Cloud Manager platforms can identify missing indexes and then automate the process of rolling them out – while avoiding any application impact.

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. MongoDB Compass visualizes index coverage, enabling you to determine which specific fields are indexed, their type, size, and how often they are used.

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)

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.

MongoDB Compass provides the ability to visualize explain plans, presenting key information on how a query performed – for example the number of documents returned, execution time, index usage, and more. Each stage of the execution pipeline is represented as a node in a tree, making it simple to view explain plans from queries distributed across multiple nodes.

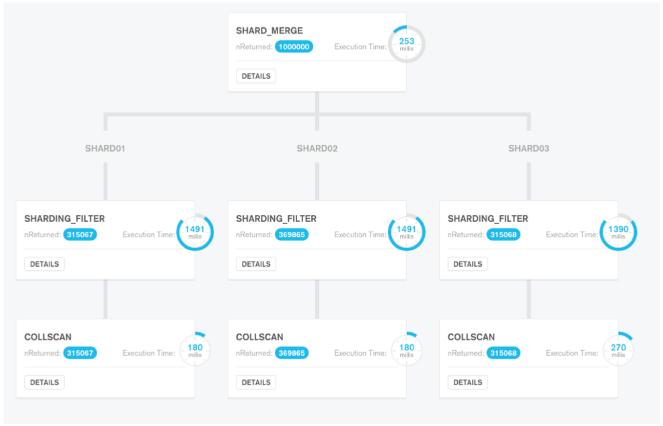


Figure 2: MongoDB Compass visual query plan for performance optimization across distributed clusters

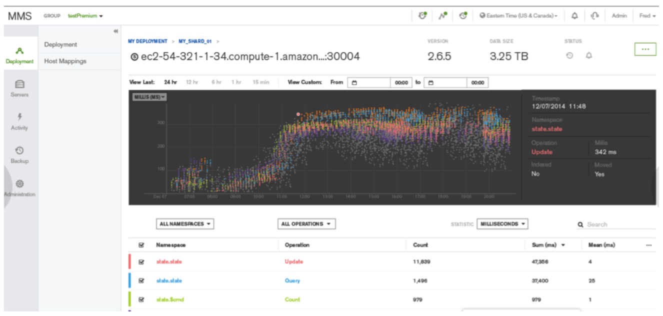
If the application will always use indexes, MongoDB can be configured through the `notablescan` setting to throw an error if a query is issued that requires scanning the entire collection.

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.

MongoDB Ops Manager and Cloud Manager (discussed later in the guide) can be used to visualize output from the profiler when identifying slow queries. The Visual Query Profiler provides a quick and convenient way for operations teams and DBAs to analyze specific queries or query families. The Visual Query Profiler (as shown in Figure 3) displays how query and write latency varies over time – making it simple to identify slower queries with common access patterns and characteristics, as well as identify any latency spikes. A single click in the Ops Manager UI activates the profiler, which then consolidates and displays metrics from every node in a single screen.

The Visual Query Profiler will analyze the data – recommending additional indexes and optionally add them through an automated, rolling index build.



ec2-54-321-1-34.compute-1.amazon...:30004 | 2.6.5 | 3.25 TB

Timestamp: 12/07/2014 11:48
Namespace: state.state
Operation: Update
Millis: 342 ms
Indexed: No
Moved: Yes

Namespace	Operation	Count	Sum (ms)	Mean (ms)
state.state	Update	11,839	47,356	4
state.state	Query	1,496	37,400	25
state.$cmd	Count	979	979	1

Figure 3: Visual Query Profiling in MongoDB Ops Manager

Performance Advisor (avaialble from Ops Manager 3.6) continuously highlights slow-running queries and provides intelligent index recommendations to improve performance. Using Ops Manager automation, the administrator can then roll out the recommended indexes automatically, without incurring any application downtime.

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.

A common practice is to build the indexes in the foreground, first on the secondaries and then on the demoted primary. Ops Manager and Cloud Manager automate this process.

In addition, multiple indexes can be built concurrently in the background. Refer to the Build Index on Replica Sets documentation to learn more about considerations for index creation and on-going maintenance.

Managing Indexes with the MongoDB WiredTiger Storage Engine

All storage engines fully support MongoDB's rich indexing functionality. When using the WiredTiger storage engine, there are some additional optimizations that you can take advantage of:

- By default, WiredTiger uses prefix compression to reduce index footprint on both persistent storage and in RAM. This enables administrators to dedicate more of the working set to manage frequently accessed documents. Compression ratios of around 50% are typical, but users are encouraged to evaluate the actual ratio they can expect by testing their own workloads.
- Administrators can place indexes on their own separate storage volume, allowing for faster disk paging and lower contention.

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. The WiredTiger storage engine manages data through its internal cache but it also benefits from pages held in the filesystem cache.

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.

Page faults occur when MongoDB attempts to access data that has not been loaded in RAM. If there is free memory then the operating system can locate the page on disk and load it into memory directly. However, if there is no free memory, the operating system must write a page that is in memory to disk, and then read the requested page into memory when it is required by the application. This process can be time consuming and will be 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 increasing RAM capacity or adding sharding the database across additional servers. For a discussion on this topic, refer to the section on Sharding Best Practices. It is far easier to implement sharding before the system's resources are consumed, so capacity planning is an important element in successful project delivery.

When the default WiredTiger storage engine is used, see the documentation for configuring the WiredTiger internal cache size.

MongoDB Setup and Configuration

Setup

MongoDB provides repositories for .deb and .rpm packages for consistent setup, upgrade, system integration, and configuration. This software uses the same binaries as the tarball packages provided from the MongoDB Downloads Page. The MongoDB Windows package is available via the downloadable binary installed via its MSI. Binaries for OS X are also provided in a tarball[1].

Database Configuration

User should store configuration options in `mongod`'s configuration file. This allows sysadmins to implement consistent configurations across entire clusters. The configuration files support all options provided as command line options for `mongod`. Popular tools such as Chef and Puppet can be used to provision MongoDB instances. The provisioning of complex topologies comprising replica sets and sharded clusters can be automated by the the Ops Manager and Cloud Manager platforms, which are discussed later in this guide.

Upgrades

Users should upgrade software as often as possible so that they can take advantage of the latest features as well

1. OS X is intended as a development rather than a production environment

as any stability updates or bug fixes. Upgrades should be tested in non-production environments to validate correct application behavior.

Customers can deploy rolling upgrades without incurring any downtime, as each member of a replica set can be upgraded individually without impacting database availability. It is possible for each member of a replica set to run under different versions of MongoDB, and with different storage engines. As a precaution, the MongoDB release notes should be consulted to determine if there is a particular order of upgrade steps that needs to be followed, and whether there are any incompatibilities between two specific versions. Upgrades can be automated with Ops Manager and Cloud Manager.

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.

The `mongoimport` and `mongoexport` tools are provided with MongoDB for simple loading or exporting of data in JSON or CSV format. These tools may be useful in moving data between systems as an initial step. Other tools such as `mongodump` and `mongorestore`, or Ops Manager and Cloud Manager backups are useful for moving data between different MongoDB systems.

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.

Hardware

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, availability requirements, and the capabilities of the underlying hardware infrastructure. MongoDB has extensive experience helping customers to select hardware and tune their configurations and we frequently work with customers to plan for and optimize their MongoDB systems. The Health Check, Operations Rapid Start, and Production Readiness consulting packages can be especially valuable in helping select the appropriate hardware for your project.

MongoDB was specifically designed with commodity hardware in mind and has few hardware requirements or limitations. Generally speaking, MongoDB will take advantage of more RAM and faster CPU clock speeds.

Memory

MongoDB makes extensive use of RAM to increase performance. Ideally, the working set fits in RAM. As a general rule of thumb, the more RAM, the better. As workloads begin to access data that is not in RAM, the performance of MongoDB will degrade, as it will for any database. The default WiredTiger storage engine gives more control of memory by allowing users to configure how much RAM to allocate to the WiredTiger internal cache – defaulting to 60% of RAM minus 1 GB. WiredTiger also exploits the operating system's filesystem cache which will grow to utilize the remaining memory available.

Storage

MongoDB does not require shared storage (e.g., storage area networks). MongoDB can use local attached storage as well as solid state drives (SSDs). Most disk access patterns in MongoDB do not have sequential properties, and as a result, customers may experience substantial performance gains by using SSDs. Good results and strong price to performance have been observed with SATA SSD and with PCIe. Commodity SATA spinning drives are comparable to higher cost spinning drives due to the non-sequential access patterns of MongoDB: rather than spending more on expensive spinning drives, that budget may be more effectively invested on increasing RAM or using SSDs. Another benefit of using SSDs is the performance benefit of flash over hard disk drives if the working set no longer fits in memory.

While data files benefit from SSDs, MongoDB's journal files are good candidates for fast, conventional disks due to their high sequential write profile. See the section on journaling later in this guide for more information.

Most MongoDB deployments should use RAID-10. RAID-5 and RAID-6 have limitations and may not provide sufficient performance. RAID-0 provides good read and write performance, but insufficient fault tolerance. MongoDB's replica sets allow deployments to provide stronger availability for data, and should be considered with RAID and other factors to meet the desired availability SLA.

If using Amazon EC2 then select the required IOPS rate using the Provisioned-IOPS option when configuring storage to provide consistent storage performance.

As with networking, use paravirtualized drivers for your storage when running on VMs.

Compression

MongoDB natively supports compression when using the default WiredTiger storage engine. Compression reduces storage footprint by as much as 80%, and enables higher storage I/O scalability as fewer bits are read from disk. As with any compression algorithm, administrators trade storage efficiency for CPU overhead, and so it is important to test the impacts of compression in your own environment.

MongoDB offers administrators a range of compression options for documents, indexes, and the journal. The default Snappy compression algorithm provides a good balance between high document and journal compression ratio (typically around 70%, dependent on the data) with low CPU overhead, while the optional zlib library will achieve higher compression, but incur additional CPU cycles as data is written to and read from disk. Indexes in WiredTiger uses prefix compression, which serves to reduce the in-memory footprint of index storage, freeing up more of the working set for frequently accessed documents. Administrators can modify the default compression settings for all collections. Compression can also be specified on a per-collection basis during collection creation.

CPU

MongoDB will deliver better performance on faster CPUs. The MongoDB WiredTiger storage engine is better able to saturate multi-core processor resources than the MMAPv1 storage engine. The Encrypted Storage engine adds an average of 15% overhead compared to WiredTiger due to a portion of available CPU being used for encryption/decryption – the actual impact will be dependent on your data set and workload.

Process Per Host

For best performance, users should run one `mongod` process per host. With appropriate sizing and resource allocation using virtualization or container technologies, multiple MongoDB processes can run on a single server without contending for resources. Using the WiredTiger storage engine, administrators will need to calculate the appropriate cache size for each instance by evaluating what portion of total RAM each of them should use, and splitting the default `cache_size` between each.

For availability, multiple members of the same replica set should never be co-located on the same physical hardware or share any single point of failure such as a power supply. When running in the cloud, make use of your virtualization provider's ability to deploy across availability zones to ensure that members from each replica set are geographically dispersed and do not share the same power, hypervisor or network.

Virtualization and IaaS

Customers can deploy MongoDB on bare metal servers, in virtualized environments and in the cloud. Performance will typically be best and most consistent using bare metal, though many MongoDB users leverage infrastructure-as-a-service (IaaS) platforms such Amazon Web Services' Elastic Compute Cloud (AWS EC2), Google Compute Engine, Microsoft Azure, Rackspace and others.

Sizing for mongos and Config Server Processes

For sharded systems, additional processes must be deployed alongside the `mongod` data storing processes: `mongos` query routers and config servers. Shards are physical partitions of data spread across multiple servers. For more on sharding, please see the section on horizontal scaling with shards. Queries are routed to the appropriate shards using a query router process called `mongos`. The metadata used by `mongos` to determine where to route a query is maintained by the config servers. Both `mongos` and config server processes are lightweight, but each has somewhat different sizing requirements.

Within a shard, MongoDB further partitions documents into chunks. MongoDB maintains metadata about the relationship of chunks to shards in the config database. Three or more config servers are maintained in sharded deployments to ensure availability of the metadata at all times. Shard metadata access is infrequent: each `mongos` maintains a cache of this data, which is periodically updated by background processes when chunks are split or migrated to other shards, typically during balancing operations as the cluster expands and contracts. The hardware for a config server should therefore be focused on availability: redundant power supplies, redundant network interfaces, redundant RAID controllers, and redundant storage should be used. Config servers can be deployed as a replica set with up to 50 members.

Typically multiple `mongos` instances are used in a sharded MongoDB system. It is not uncommon for MongoDB users to deploy a `mongos` instance on each of their application servers. The optimal number of `mongos` servers will be determined by the specific workload of the application: in some cases `mongos` simply routes queries to the appropriate shard, and in other cases `mongos` must route them to multiple shards and merge the result sets. To estimate the memory requirements for each `mongos`, consider the following:

- The total size of the shard metadata that is cached by `mongos`
- 1MB for each application connection

The `mongos` process uses limited RAM and will benefit more from fast CPUs and networks.

Operating System and File System

Configurations for Linux

Only 64-bit versions of operating systems are supported for use with MongoDB.

Version 2.6.36 of the Linux kernel or later should be used for MongoDB in production.

Use EXT4 or XFS file systems; avoid EXT3, EXT3 is quite old and is not optimal for most database workloads. For example, MMAPv1 preallocates space for data. In EXT3 preallocation will actually write `0`s to the disk to allocate the space, which is time consuming. In EXT4 and XFS preallocation is performed as a logical operation, which is much more efficient.

With the WiredTiger storage engine, use of XFS is strongly recommended to avoid performance issues that have been observed when using EXT4 with WiredTiger.

For MongoDB on Linux use the following recommended configurations:

- Turn off `atime` for the storage volume with the database files.
- Do not use Huge Pages virtual memory pages, MongoDB performs better with normal virtual memory pages.
- Disable NUMA in your BIOS or invoke `mongod` with NUMA disabled.
- Ensure that `readahead` settings for the block devices that store the database files are relatively small as most access is non-sequential. For example, setting `readahead` to 32 (16 KB) is a good starting point.
- Synchronize time between your hosts – for example, using NTP. This is especially important in sharded MongoDB clusters. This also applies to VM guests running MongoDB processes.

Linux provides controls to limit the number of resources and open files on a per-process and per-user basis. The default settings may be insufficient for MongoDB. Generally MongoDB should be the only process on a system, VM, or container to ensure there is no contention with other processes.

While each deployment has unique requirements, the following configurations are a good starting point for `mongod` and `mongos` instances. Use `ulimit` to apply these settings:

- `-f` (file size): unlimited
- `-t` (CPU time): unlimited
- `-v` (virtual memory): unlimited
- `-n` (open files): above 20,000
- `-m` (memory size): unlimited
- `-u` (processes/threads): above 20,000

For more on using `ulimit` to set the resource limits for MongoDB, see the MongoDB Documentation page on Linux ulimit Settings.

Networking

Always run MongoDB in a trusted environment with network rules that prevent access from all unknown entities. There are a finite number of predefined processes that communicate with a MongoDB system: application servers, monitoring processes, and other MongoDB processes running in a replica set or sharded cluster.

From the MongoDB 2.6 release onwards, the binaries from the official MongoDB RPM and DEB packages bind to `localhost` by default. With MongoDB 3.6, this default behavior is extended to all MongoDB packages across all platforms. As a result, all networked connections to the database will be denied unless explicitly configured by an administrator. Review the documentation. If your system has more than one network interface, bind MongoDB processes to the private or internal network interface.

Detailed information on default port numbers for MongoDB, configuring firewalls for MongoDB, VPN, and other topics is available in the MongoDB Security Tutorials. Review the Security section later in this guide for more information on best practices on securing your deployment.

MongoDB 3.6 introduces IP whitelisting, allowing administrators to configure MongoDB to only accept external connections from approved IP addresses or CIDR ranges that have been explicitly added to the whitelist.

When running on virtual machines, use paravirtualized drivers to implement an optimized network and storage interfaces that passes instructions between the virtual machine and the hypervisor with minimal overhead.

Network Compression

As a distributed database, MongoDB relies on efficient network transport during query routing and inter-node replication. MongoDB 3.4 introduced a new option to compress the wire protocol used for intra-cluster communications. MongoDB 3.6 extends wire protocol compression to network traffic between the client and the database. Based on the snappy compression algorithm, network traffic can be compressed by up to 70%, providing major performance benefits in bandwidth-constrained environments, and reducing networking costs.

Compression is off by default, but can be enabled by setting `networkMessageCompressors` to `snappy`.

Compressing and decompressing network traffic requires CPU resources – typically low single digit percentage overhead. Compression is ideal for those environments where performance is bottlenecked by bandwidth, and sufficient CPU capacity is available.

Production-Proven Recommendations

The latest recommendations on specific configurations for operating systems, file systems, storage devices, and other system-related topics are maintained in the MongoDB Production Notes.

Continuous Availability

Under normal operating conditions, a MongoDB system will perform according to the performance and functional goals of the system. However, from time to time certain inevitable failures or unintended actions can affect a system in adverse ways. Hard drives, network cards, power supplies, and other hardware components will fail. These risks can be mitigated with redundant hardware components. Similarly, a MongoDB system provides configurable redundancy throughout its software components as well as configurable data redundancy.

Journaling

MongoDB implements write-ahead journaling of operations to enable fast crash recovery and durability in the storage engine. In the case of a server crash, journal entries are recovered when the server process is restarted.

The behavior of the journal is dependent on the configured storage engine:

- The WiredTiger journal ensures that writes are persisted to disk between checkpoints. WiredTiger uses checkpoints to flush data to disk by default every 60 seconds after the prior flush or after 2GB of data has been written. Thus, by default, WiredTiger can lose more than 60 seconds of writes if running without journaling – though the risk of this loss will typically be much less if using replication to other nodes for additional durability. The WiredTiger write ahead log is not necessary to keep the data files in a consistent state in the event of an unclean shutdown, and so it is safe to run without journaling enabled, though to ensure durability the "replica safe" write concern should be used (see the Write Availability section later in the guide for more information).
- MMAPv1 journal commits to disk are issued at least as often as every 100 ms by default or 30ms if the journal is on a separate device. In addition to providing durability, the journal also prevents corruption in the case of an unclean shutdown of the system. By default, journaling is enabled for MongoDB with MMAPv1. No production deployment should run without the journal configured.

Another feature of both the WiredTiger and MMAPv1 storage engines is the ability to compress the journal on disk, thereby reducing storage space.

For additional guarantees, the administrator can configure the journaled write concern, whereby MongoDB acknowledges the write operation only after committing the data to the journal. When using a write concern greater than 1 and the v1 replication protocol[2], the application will not receive an acknowledgement until the write has been journalled on the specified number of secondaries and when using a write concern of “majority” it must also be journalled on the primary.

2. Enhanced (v1) replication protocol – earlier versions are referred to as v0

Locating MongoDB's journal files and data files on separate storage arrays can help performance. The I/O patterns for the journal are very sequential in nature and are well suited for storage devices that are optimized for fast sequential writes, whereas the data files are well suited for storage devices that are optimized for random reads and writes. Simply placing the journal files on a separate storage device normally provides some performance enhancements by reducing disk contention.

Learn more about journaling from the documentation.

Data Redundancy

MongoDB maintains multiple copies of data, called replica sets, using native replication. Users should use replica sets to help prevent database downtime. 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.

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, heartbeat and connectivity status, and user-defined priorities assigned to replica set members. For example, administrators can configure all replicas located in a secondary data center to be candidates for election only if the primary data center fails. Once the new primary replica set member has been elected, remaining secondary members are automatically start replicating from the new primary. If the original primary comes back on-line, it will recognize that it is no longer the primary and will reconfigure itself to become a secondary replica set member.

The number of replica nodes in a MongoDB replica set is configurable, and a larger number of replica nodes provides increased protection against database downtime in case of multiple machine failures. While a node is down MongoDB will continue to function. The DBA or sysadmin should work to recover or replace the failed replica in order to mitigate the temporarily reduced resilience of the system.

Replica sets also provide operational flexibility by providing sysadmins with an option for performing hardware and software maintenance without taking down the entire system. Using a rolling upgrade, secondary members of the replica set can be upgraded in turn, before the administrator demotes the master to complete the upgrade. This process is fully automated when using Ops Manager or Cloud Manager – discussed later in this guide.

Consider the following factors when developing the architecture for your replica set:

- Ensure that the members of the replica set will always be able to elect a primary. A strict majority of voting cluster members must be available and in contact with each other to elect a new primary. Therefore you should run an odd number of members but if you do have an even number of members then prevent one member from voting or add an arbiter (a member of the replica set that exists solely for participating in election of the primary) to one of your application servers so that there is an odd number of voting members. There should be at least three replicas with copies of the data in a replica set.
- Best practice is to have a minimum of 3 data centers so that a majority is maintained after the loss of any single site. If only 2 sites are possible then know where the majority of members will be in the case of any network partitions and attempt to ensure that the replica set can elect a primary from the members located in that primary data center.
- Consider including a hidden member in the replica set. Hidden replica set members can never become a primary and are typically used for backups, or to run applications such as analytics and reporting that require isolation from regular operational workloads. Delayed replica set members can also be deployed that apply changes on a fixed time delay to provide recovery from unintentional operations, such as accidentally dropping a collection.

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

Multi-Data Center Replication

MongoDB replica sets allow for flexible deployment designs both within and across data centers that account for failure at the server, rack, and regional levels. In the case of a natural or human-induced disaster, the failure of a single data center can be accommodated with no downtime when MongoDB replica sets are deployed across data centers.

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 configured on a per connection, per database, per collection, or even per operation basis. The options are as follows:

- **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.
- **Journal Acknowledged:** The `mongod` will confirm the write operation only after it has flushed the operation to the journal on the primary. This confirms that the write operation can survive a `mongod` crash and ensures that the write operation is durable on disk.
- **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 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. This also

ensures that the write is recorded in the journal on these replicas – including on the primary.

- **Data Center Awareness:** Using tag sets, sophisticated policies can be created to ensure data is written to specific combinations of replicas prior to acknowledgment of success. For example, you can create a policy that requires writes to be written to at least three data centers on two continents, or two servers across two racks in a specific data center. For more information see the MongoDB Documentation on Data Center Awareness.

Read Preferences

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. Another configuration option directs reads to the replica closest to the user based on ping distance, which can significantly decrease the latency of read operations in globally distributed applications at the expense of potentially reading slightly stale data.

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.

Read Concerns

To ensure isolation and consistency, the `readConcern` can be set to `majority` to indicate that data should only be returned to the application if it has been replicated to a majority of the nodes in the replica set, and so cannot be rolled back in the event of a failure.

MongoDB 3.4 added a new readConcern level of "Linearizable". The linearizable read concern ensures that a node is still the primary member of the replica set at the time of the read, and that the data it returns will not be rolled back if another node is subsequently elected as the new primary member. Configuring this read concern level can have a significant impact on latency, therefore a maxTimeMS value should be supplied in order to timeout long running operations.

Causal Consistency

Causal consistency – guarantees that every read operation within a client session will always see the previous write operation, regardless of which replica is serving the request. By enforcing strict, causal ordering of operations within a session, causal consistency ensures every read is always logically consistent, enabling monotonic reads from a distributed system – guarantees that cannot be met by most multi-node databases. Causal consistency allows developers to maintain the benefits of strict data consistency enforced by legacy single node relational databases, while modernizing their infrastructure to take advantage of the scalability and availability benefits of modern distributed data platforms.

Primary Elections

In the event that a primary node is unavailable, reads may continue if you are using an read preference other than the default of `primary` but any writes to that replica set will fail until either:

- The primary becomes available again
- An election is held and a secondary becomes the new primary

If the primary is only unavailable for a short period (e.g., due to a transient network glitch) then the best approach would be to wait for it to come back on-line. However, if the primary is not going to come back quickly, then you want the system to quickly elect and promote a new primary member to take over. Clearly, there's a balance to be reached based on service requirements, the system configuration and the environment in which the database is running.

MongoDB 3.2 introduced an enhanced replication protocol that delivers faster service recovery in the event of a primary failure, as well as stricter durability guarantees. The enhanced replication protocol extends the Raft consensus algorithm to offer greater deployment flexibility while maintaining compatibility with replication constructs offered in earlier MongoDB releases. Specifically, the protocol maintains support for replica set arbiters, replica set member election priorities, and secondary members replicating from other secondaries to enable chained replication.

The enhanced replication protocol reduces the failover interval by optimizing the algorithms used to detect replica set primary failures and elect a new primary. Failover time is dependent on several factors, including network latency. It is important for the system to avoid unnecessary failovers, and to provide flexibility for the needs of different deployments. Use the `electionTimeoutMillis` parameter to tune your systems for optimal failover behavior:

- Higher values result in slower failovers but decreased sensitivity to network latency and load on the primary node.
- Lower values result in faster failover, but increased sensitivity to network latency and load on the primary.

`electionTimeoutMillis` defaults to 10,000 milliseconds (10 seconds) – in most environments this will provide a robust cluster which is tolerant to normal network variability but if you have stricter failover requirements and predictable networking infrastructure then you can test with lower values.

Scaling a MongoDB System

Horizontal Scaling with Sharding

MongoDB 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 supports three types of sharding policy, 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.
- **Zones:** MongoDB Zones (described as tag-aware sharding in earlier MongoDB releases) allow precise control over where data is physically stored, accommodating a range of deployment scenarios – for example by geography, by hardware configuration, or by application. Administrators can continuously refine data placement rules by modifying shard key ranges, and MongoDB will automatically migrate the data to its new Zone. MongoDB 3.4 added new helper functions and additional options in Ops Manager and Cloud Manager to configure Zones, essential for managing large deployments.

While sharding is very powerful, it can add operational complexity to a MongoDB deployment and it has additional infrastructure requirements. As a result, users should shard as necessary and when indicated by actual operational requirements.

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.
- **Location-aware requirements:** The data set needs to be assigned to a specific data center to support low latency local reads and writes. Alternatively, to create multi-temperature storage infrastructures that separate hot and cold data onto specific volumes.

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.

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.

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

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.

Run three or more configuration servers to provide redundancy. Production deployments must use three or more config servers. Config servers should be deployed in a topology that is robust and resilient to a variety of failures.

Use replica sets. Sharding and replica sets are absolutely compatible. Replica sets should be used in all deployments, and sharding should be used when appropriate. Sharding allows a database to make use of multiple servers for data capacity and system throughput. Replica sets maintain redundant copies of the data across servers, server racks, and even data centers.

Use multiple `mongos` instances.

Apply best practices for bulk inserts. Pre-split data into multiple chunks so that no balancing is required during the insert process. Alternately, disable the balancer during bulk loads[3]. Also, use multiple `mongos` instances to load in parallel for greater throughput. For more information see Create Chunks in a Sharded Cluster in the MongoDB Documentation.

Dynamic Data Balancing

As data is loaded into MongoDB, the system may need to dynamically rebalance chunks across shards in the cluster using a process called the balancer. The balancing operations attempt to minimize the impact to the performance of the cluster by only moving one chunk of documents at a time, and by only migrating chunks when a distribution threshold is exceeded. It is possible to disable

3. From MongoDB 3.4. the balancer process supports parallel data migrations, multiple node pairs can perform balancing migrations simultaneously, significantly improving balancing throughput.

the balancer or to configure when balancing is performed to further minimize the impact on performance.

Geographic Distribution

Shards can be configured such that specific ranges of shard key values are mapped to a physical shard location. Zone sharding allows a MongoDB administrator to control the physical location of documents in a MongoDB cluster, even when the deployment spans multiple data centers in different regions.

It is possible to combine the features of replica sets, Zone sharding, read preferences, and write concern in order to provide a deployment that is geographically distributed, enabling users to read and write to their local data centers. An administrator can restrict sharded collections to a specific set of shards, effectively federating those shards for different users. For example, one can tag all USA data and assign it to shards located in the United States.

To learn more, download the MongoDB Multi-Datacenter Deployments Guide.

Managing MongoDB: Provisioning, Monitoring and Disaster Recovery

Created by the engineers who develop the database, Ops Manager is the simplest way to run MongoDB, making it easy for operations teams to deploy, monitor, backup, and scale MongoDB. Many of the capabilities of Ops Manager are also available in the MongoDB Cloud Manager service hosted in the cloud. Today, Cloud Manager supports thousands of deployments, including systems from one to hundreds of servers. Organizations who run their deployments with MongoDB Enterprise Advanced can choose between Ops Manager and Cloud Manager.

Ops Manager and Cloud Manager incorporate 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 or via an API call:

- **Deploy.** Any topology, at any scale
- **Upgrade.** In minutes, with no downtime
- **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.
- **Queryable Backups.** Avaialble in MongoDB Atlas and Ops Manager 3.6 and later, allow partial restores of selected data, and the ability to query a backup file in-place, without having to restore it.
- **Performance Alerts.** Monitor 100+ system metrics and get custom alerts before the system degrades.
- **Roll Out Indexes.** Avoid impact to the application by introducing new indexes node by node – starting with the secondaries and then the demoted primary.
- **Manage Zones.** Configure sharding Zones to mandate what data is stored where.
- **Data Explorer.** Examine the database's schema by running queries to review document structure, viewing collection metadata, and inspecting index usage statistics (Atlas and Ops Manager 3.6+).

The Operations Rapid Start service Give your operations and devops teams the skills and tools to run and manage MongoDB with confidence. This engagement offers introductory administrator training and custom consulting to help you set up and use either MongoDB Ops Manager or MongoDB Cloud Manager (available for those operations teams who do not want to maintain their own management and backup infrastructure in-house).

Deployments and Upgrades

Ops Manager coordinates critical operational tasks across the servers in a MongoDB system. It communicates with the infrastructure through agents installed on each server. The servers can reside in the public cloud or a private data center. Ops Manager reliably orchestrates the tasks that administrators have traditionally performed manually – deploying a new cluster, upgrades, creating point in time backups, rolling out new indexes, and many other operational tasks.

Ops Manager is designed to adapt to problems as they arise by continuously assessing state and making adjustments as needed. Here’s how:

- Ops Manager agents are installed on servers (where MongoDB will be deployed), either through configuration tools such as Chef or Puppet, or by an administrator.
- The administrator creates a new design goal for the system, either as a modification to an existing deployment (e.g., upgrade, oplog resize, new shard), or as a new system.
- The agents periodically check in with the Ops Manager central server and receive the new design instructions.
- Agents create and follow a plan for implementing the design. Using a sophisticated rules engine, agents continuously adjust their individual plans as conditions change. In the face of many failure scenarios – such as server failures and network partitions – agents will revise their plans to reach a safe state.
- Minutes later, the system is deployed – safely and reliably.

Beyond deploying new databases, Ops Manager and Cloud Manager can "attach to" or import existing MongoDB deployments and take over their control.

In addition to initial deployment, Ops Manager and Cloud Manager make it possible to dynamically resize capacity by adding shards and replica set members. Other maintenance tasks such as upgrading MongoDB or resizing the oplog can be reduced from dozens or hundreds of manual steps to the click of a button, all with zero downtime.

A common DBA task is to roll out new indexes in production systems. In order to minimize the impact to the live system, the best practice is to perform a rolling index build – starting with each of the secondaries and finally applying changes to the original primary, after swapping its role with one of the secondaries. While this rolling process can be performed manually, Ops Manager and Cloud Manager can automate the process across MongoDB replica sets, reducing operational overhead and the risk of failovers caused by incorrectly sequencing management processes.

Administrators can use the Ops Manager interface directly, or invoke the Ops Manager RESTful API from existing enterprise tools, including popular monitoring and orchestration frameworks. Specific integration is provided with the leading Application Performance Management (APM) tools. Details are included later in this section of the guide.

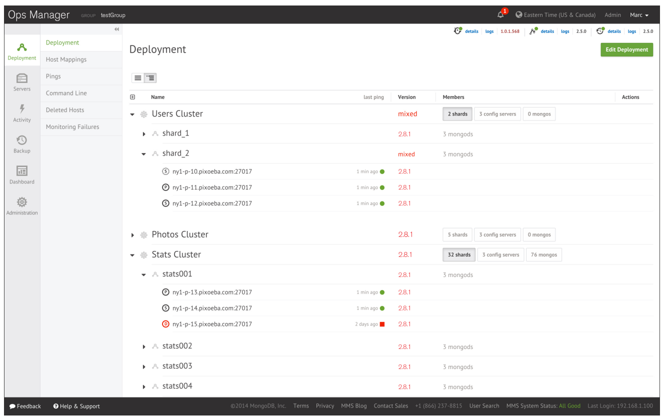
Monitoring & Capacity Planning

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.

Baselines will help you understand when the system is operating as designed, and when issues begin to emerge that may affect the quality of the user experience or other factors critical to the system. It is important to monitor your MongoDB system for unusual behavior so that actions can be taken to address issues proactively. The following represents the most popular tools for monitoring MongoDB, and also describes different aspects of the system that should be monitored.

Monitoring with Ops Manager and Cloud Manager

Featuring charts, custom dashboards, and automated alerting, Ops Manager tracks 100+ key database and systems health metrics including operations counters, memory and CPU utilization, replication status, open connections, queues, and any node status.



Deployment

Name	last ping	Version	Members	Actions
Users Cluster		mixed	2 shards / 3 config servers / 0 mongos	
shard_1		2.8.1	3 mongods	
shard_2		mixed	3 mongods	
ny1-p-10.pixoeba.com:27017	1 min ago ●	2.8.1		
ny1-p-11.pixoeba.com:27017	1 min ago ●	2.8.1		
ny1-p-12.pixoeba.com:27017	1 min ago ●	2.8.1		
Photos Cluster		2.8.1	5 shards / 3 config servers / 0 mongos	
Stats Cluster		2.8.1	32 shards / 3 config servers / 76 mongos	
stats001		2.8.1	3 mongods	
ny1-p-13.pixoeba.com:27017	1 min ago ●	2.8.1		
ny1-p-14.pixoeba.com:27017	1 min ago ●	2.8.1		
ny1-p-15.pixoeba.com:27017	2 days ago ■	2.8.1		
stats002		2.8.1	3 mongods	
stats003		2.8.1	3 mongods	
stats004		2.8.1	3 mongods	

Figure 4: Ops Manager: simple, intuitive, and powerful. Deploy and upgrade entire clusters with a single click.

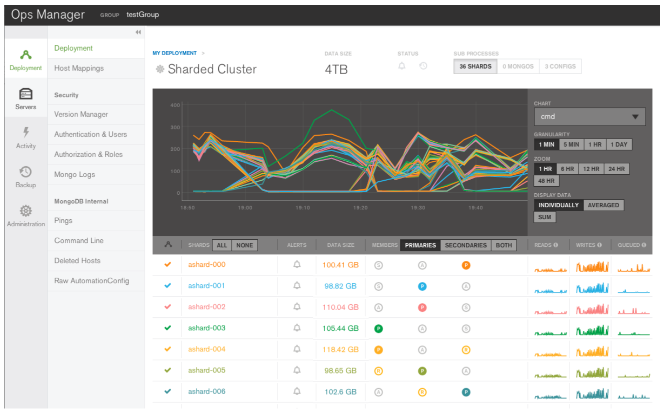
The metrics are securely reported to Ops Manager and Cloud Manager where they are processed, aggregated, alerted and visualized in a browser, letting administrators easily determine the health of MongoDB in real-time. Views can be based on explicit permissions, so project team visibility can be restricted to their own applications, while systems administrators can monitor all the MongoDB deployments in the organization.

Ops Manager allows telemetry data to be collected every 10 seconds.

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 Ops Manager RESTful API, making the deep insights from Ops Manager part of a consolidated view across your operations.

Ops Manager 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, replica sets, agents, and backup. Alerts can be sent via SMS, email, webhooks, Flowdock, HipChat, and Slack or integrated into existing incident management systems such as PagerDuty to proactively warn of potential issues, before they escalate to costly outages.

If using Cloud Manager, access to monitoring data can also be shared with MongoDB support engineers, providing fast issue resolution by eliminating the need to ship logs between different teams.



Sharded Cluster — DATA SIZE 4TB — SUB PROCESSES 36 SHARDS, 0 MONGOS, 3 CONFIGS

	SHARDS	ALERTS	DATA SIZE	MEMBERS			READS	WRITES	QUEUED
✓	ashard-000		100.41 GB	S	A	P			
✓	ashard-001		98.82 GB	S	P	A			
✓	ashard-002		110.04 GB	A	P	S			
✓	ashard-003		105.44 GB	P	A	S			
✓	ashard-004		118.42 GB	P	A	R			
✓	ashard-005		98.65 GB	R	P	A			
✓	ashard-006		102.6 GB	A	R	P			

Figure 5: Ops Manager provides real time & historic visibility into the MongoDB deployment.

Hardware Monitoring

Munin node is an open-source software program that monitors hardware and reports on metrics such as disk and RAM utilization. Ops Manager can collect this data from Munin node and provide it along with other data available in the Ops Manager dashboard. While each application and deployment is unique, users should create alerts for spikes in disk utilization, major changes in network activity, and increases in average query length/response times.

mongotop

`mongotop` is a utility that ships with MongoDB. It tracks and reports the current read and write activity of a MongoDB cluster. `mongotop` provides collection-level statistics.

mongostat

`mongostat` is a utility that ships with MongoDB. It shows real-time statistics about all servers in your MongoDB system. `mongostat` provides a comprehensive overview of all operations, including counts of updates, inserts, page faults, index misses, and many other important measures of the system health. `mongostat` is similar to the Linux tool `vmstat`.

MongoDB Compass

Trying to parse text output can significantly increase the time to resolve query performance issues. MongoDB

Compass now been extended to visualize the same real-time performance statistics generated by `mongotop` and `mongostat`, allowing DBAs to generate an instant snapshot of server status and query performance.

Other Popular Tools

There are a number of popular open-source monitoring tools for which MongoDB plugins are available. If MongoDB is configured with the WiredTiger storage engine, ensure the tool is using a WiredTiger-compatible driver:

- Nagios
- Ganglia
- Cacti
- Scout
- Zabbix
- Datadog

Linux Utilities

Other common utilities that should be used to monitor different aspects of a MongoDB system:

- `iostat`: Provides usage statistics for the storage subsystem
- `vmstat`: Provides usage statistics for virtual memory
- `netstat`: Provides usage statistics for the network
- `sar`: Captures a variety of system statistics periodically and stores them for analysis

Windows Utilities

Performance Monitor, a Microsoft Management Console snap-in, is a useful tool for measuring a variety of stats in a Windows environment.

Things to Monitor

Ops Manager and Cloud Manager can be used to monitor 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.

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.

In the event of errors, exceptions or unexpected behavior, the logs should be saved and uploaded to MongoDB when opening a support case. Logs for `mongod` processes running on primary and secondary replica set members, as well as `mongos` and config processes will enable the support team to more quickly root cause any issues.

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. Page faults are part of the normal operation of a MongoDB system, but the volume of page faults should be monitored in order to determine if the working set is growing to the level that it no longer fits in available memory and if alternatives such as more memory or sharding across multiple servers is appropriate. In most cases, the underlying issue for problems in a MongoDB system tends to be page faults. Also use the working set estimator discussed earlier in the guide.

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 RAID configuration may be inadequate for the volume of writes. Other potential issues could be the root cause, but the symptom is typically visible through `iostat` as showing high disk utilization and high queuing for 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 low queues. If metrics start to deviate from baseline performance, caused by a long-running query for example, 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.

System Configuration

It is not uncommon to make changes to hardware and software in the course of a MongoDB deployment. For example, a disk subsystem may be replaced to provide better performance or increased capacity. When components are changed it is important to ensure their configurations are appropriate for the deployment. MongoDB is very sensitive to the performance of the operating system and underlying hardware, and in some cases the default values for system configurations are not ideal. For example, the default `readahead` for the file system could be several MB whereas MongoDB is optimized for `readahead` values closer to 32 KB. If the new storage system is installed without making the change to the `readahead` from the default to the appropriate setting, the application's performance is likely to degrade substantially.

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. By running `db.currentOp()` you will be able to determine what work is currently being performed by the cluster, including rebalancing of documents across the shards.

From MongoDB 3.4, the balancer supports parallel data migrations. Multiple node pairs can perform balancing migrations simultaneously, significantly improving balancing throughput in larger clusters. Additionally, the balancer throttling is now off by default for WiredTiger, speeding up migrations by as much as 10x.

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. Typical causes of replication lag include network latency or connectivity issues, and disk latencies such as the throughput of the secondaries being inferior to that of the primary.

Config Server Availability

In sharded environments it is required to run three or more config servers. Config servers are critical to the system for understanding the location of documents across shards. The database will remain operational in this case, but the balancer will be unable to move chunks and other maintenance activities will be blocked until all three config servers are available.

Config servers are, by default, be deployed as a MongoDB replica set. Config server replica sets can span more than three data centers with up to 50 replica set members supported, providing higher levels of availability and lower cross-region latency. Review the documentation to learn more.

Disaster Recovery: Backup & Recovery

A backup and recovery strategy is necessary to protect your mission-critical data against catastrophic failure, such as a fire or flood in a data center, or human error such as code errors or 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.

Ops Manager and Cloud Manager 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. Ops Manager and Cloud Manager are the only MongoDB solutions that offer point-in-time backup of replica sets and cluster-wide snapshots of sharded clusters. You can restore to precisely the moment you need, quickly and safely. Ops teams can automate their database restores reliably and safely using Ops Manager and Cloud Manager. Complete development, test, and recovery clusters can be built in a few simple clicks. Operations teams can configure backups against specific collections only, rather than the entire database, speeding up backups and reducing the requisite storage space.

Queryable Backups (Atlas and Ops Manager 3.6+) allow partial restores of selected data, and the ability to query a backup file in-place, without having to restore it.

Ops Manager 3.6 introduces cross-project restores, allowing users to perform restores into a different Ops Manager Project than the backup snapshot source. This allows DevOps teams to easily execute tasks such as creating multiple staging or test environments that match recent production data, while configured with different user access privileges or running in different regions.

Because Ops Manager only reads the oplog, the ongoing performance impact is minimal – similar to that of adding an additional replica to a replica set.

By using MongoDB Enterprise Advanced you can deploy Ops Manager to control backups in your local data center, or use the Cloud Manager service that offers a fully managed backup solution with a pay-as-you-go model. Dedicated MongoDB engineers monitor user backups on a 24x365 basis, alerting operations teams if problems arise.

Ops Manager and Cloud Manager are not the only mechanisms for backing up MongoDB. Other options include:

- File system copies
- The mongodump tool packaged with MongoDB

File System Backups

File system backups, such as that provided by Linux LVM, quickly and efficiently create a consistent snapshot of the file system that can be copied for backup and restore purposes. For databases with a single replica set it is possible to stop operations temporarily so that a consistent snapshot can be created by issuing the `db.fsyncLock()` command. This will flush all pending writes to disk and lock the entire `mongod` instance to prevent additional writes until the lock is released with `db.fsyncUnlock()`.

For more on how to use file system snapshots to create a backup of MongoDB, please see Backup and Restore with Filesystem Snapshots in the MongoDB Documentation.

Only Ops Manager and Cloud Manager provide an automated method for taking a consistent backup across all shards.

For more on backup and restore in sharded environments, see the MongoDB Documentation page on Backup and Restore Sharded Clusters and the tutorial on Backup a Sharded Cluster with Filesystem 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`.

Integrating MongoDB with External Monitoring Solutions

The Ops Manager API provides integration with external management frameworks through programmatic access to automation features and monitoring data.

In addition to Ops Manager, MongoDB Enterprise Advanced can report system information to SNMP traps, supporting centralized data collection and aggregation via external monitoring solutions. Review the documentation to learn more about SNMP integration.

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.

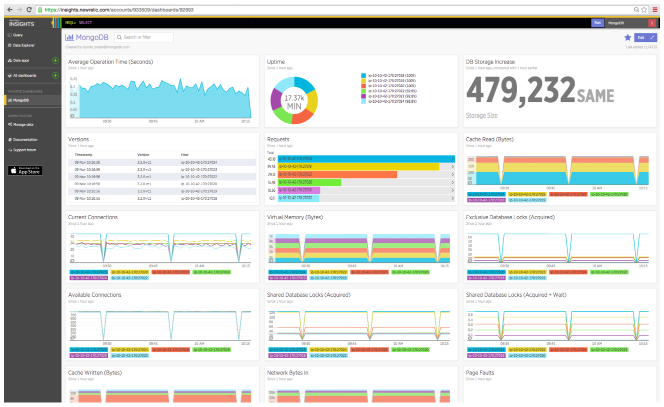


Figure 6: MongoDB integrated into a single view of application performance

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, Ops and Cloud Manager now provide packaged integration with the New Relic platform. Key metrics from Ops Manager 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).

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.

The intention of a Defense in Depth approach is to layer your environment to ensure there are no exploitable single points of failure that could allow an intruder or untrusted party to access the data stored in the MongoDB database. The most effective way to reduce the risk of exploitation is to run MongoDB in a trusted environment, to limit access, to follow a system of least privileges, to institute a secure development lifecycle, and to follow deployment best practices.

MongoDB Enterprise Advanced 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 to the database, collection, and down to the level of individual fields within a document.
- **Auditing.** Ensure regulatory and internal compliance.
- **Encryption.** Protect data in motion over the network and at rest in persistent storage.
- **Administrative Controls.** Identify potential exploits faster and reduce their impact.
- **Network Protection.** Refer to the earlier Networking session for details.

Review the MongoDB Security Reference Architecture to learn more about each of the security features discussed below.

Authentication

Authentication can be managed from within the database itself with challenge/response mechanisms and x.509 PKI certificates, or via MongoDB Enterprise Advanced integration with external security mechanisms including LDAP, Windows Active Directory, and Kerberos.

Authorization

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

MongoDB 3.4 extended existing support for authenticating users via LDAP to now include LDAP authorization. This enables existing user privileges stored in the LDAP server to be mapped to MongoDB roles, without users having to be recreated in MongoDB itself.

Auditing

MongoDB Enterprise Advanced enables security administrators to construct and filter audit trails for any operation against MongoDB, whether DML, DCL or DDL. For example, it is possible to log and audit the identities of users who retrieved specific documents, and any changes made to the database during their session. The audit log can be written to multiple destinations in a variety of formats including to the console and `syslog` (in JSON format), and to a file (JSON or BSON), which can then be loaded to MongoDB and analyzed to identify relevant events

Encryption

MongoDB data can be encrypted on the network and on disk.

Support for SSL allows clients to connect to MongoDB over an encrypted channel. MongoDB supports FIPS 140-2 encryption when run in FIPS Mode with a FIPS validated Cryptographic module.

Data at rest can be protected using:

- The MongoDB Encrypted storage engine
- Certified database encryption solutions from MongoDB partners such as IBM and Vormetric
- Logic within the application itself

With the Encrypted storage engine, protection of data at-rest now becomes an integral feature of the database. By natively encrypting database files on disk, administrators eliminate both the management and performance overhead of external encryption mechanisms. This new storage engine provides an additional level of defense, allowing only those staff with the appropriate database credentials access to encrypted data.

Using the Encrypted storage engine, the raw database “plain text” content is encrypted using an algorithm that takes a random encryption key as input and generates "ciphertext" that can only be read if decrypted with the decryption key. The process is entirely transparent to the application. MongoDB supports a variety of encryption algorithms – the default is AES-256 (256 bit encryption) in CBC mode. AES-256 in GCM mode is also supported. Encryption can be configured to meet FIPS 140-2 requirements.

The storage engine encrypts each database with a separate key. The key-wrapping scheme in MongoDB wraps all of the individual internal database keys with one external master key for each server. The Encrypted storage engine supports two key management options – in both cases, the only key being managed outside of MongoDB is the master key:

- Local key management via a keyfile
- Integration with a third party key management appliance via the KMIP protocol (recommended)

Read-Only, Redacted Views

From MongoDB 3.4, 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. Permissions granted against the view are specified separately from permissions granted to the underlying collection(s).

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.

MongoDB Atlas: Database as a Service For MongoDB

MongoDB Atlas is a cloud database service that makes it easy to deploy, operate, and scale MongoDB in the cloud by automating time-consuming administration tasks such as database setup, security implementation, scaling, patching, and more.

MongoDB Atlas is available on-demand through a pay-as-you-go model and billed on an hourly basis.

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

- Security features to protect your data, with fine-grained access control and end-to-end encryption
- Built in replication for always-on availability. Cross-region replication within a public cloud can be enabled to help tolerate the failure of an entire cloud region.
- Fully managed, continuous and consistent backups with point in time recovery to protect against data corruption, and the ability to query backups in-place without full restores
- Fine-grained monitoring and customizable alerts for comprehensive performance visibility
- One-click scale up, out, or down on demand. MongoDB Atlas can provision additional storage capacity as needed without manual intervention.

- Automated patching and single-click upgrades for new major versions of the database, enabling you to take advantage of the latest and greatest MongoDB features
- Live migration to move your self-managed MongoDB clusters into the Atlas service with minimal downtime

MongoDB Atlas can be used for everything from a quick Proof of Concept, to test/QA environments, to powering production applications. The user experience across MongoDB Atlas, Cloud Manager, and Ops Manager is consistent, ensuring that disruption is minimal if you decide to manage MongoDB yourself and migrate to your own infrastructure.

MongoDB Stitch: Backend as a Service

MongoDB Stitch is a backend as a service (BaaS), giving developers a REST-like API to MongoDB, and composability with other services, backed by a robust system for configuring fine-grained data access controls. Stitch provides native SDKs for JavaScript, iOS, and Android.

Built-in integrations give your application frontend access to your favorite third party services: Twilio, AWS S3, Slack, Mailgun, PubNub, Google, and more. For ultimate flexibility, you can add custom integrations using MongoDB Stitch's HTTP service.

MongoDB Stitch allows you to compose multi-stage pipelines that orchestrate data across multiple services; where each stage acts on the data before passing its results on to the next.

Unlike other BaaS offerings, MongoDB Stitch works with your existing as well as new MongoDB clusters, giving you access to the full power and scalability of the database. By defining appropriate data access rules, you can selectively expose your existing MongoDB data to other applications through MongoDB Stitch's API.

Take advantage of the free tier to get started; when you need more bandwidth, the usage-based pricing model ensures you only pay for what you consume. Learn more and try it out for yourself.

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 10 million downloads, thousands of customers, and over 1,000 technology and service partners. MongoDB users rely on the best practices discussed in this guide to maintain the highly available, secure and scalable operations demanded by organizations today.

We Can Help

We are the MongoDB experts. Over 4,300 organizations rely on our commercial products, including startups and more than half 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 Stitch is a backend as a service (BaaS), giving developers full access to MongoDB, declarative read/write controls, and integration with their choice of services.

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)
MongoDB Stitch backend as a service (mongodb.com/cloud/stitch)