MongoDB Operations Best Practices

MongoDB v2.2

A 10gen White Paper
February 2013

10gen | The MongoDB Company

Table of Contents

MongoDB Operations Best Practices

MongoDB v2.2

MongoDB is the open-source, document database that is popular among both developers and operations professionals given its agile and scalable approach. MongoDB is used in hundreds of production deployments by organizations ranging in size from emerging startups to Fortune 5 companies. This paper provides guidance on best practices for deploying and managing a MongoDB cluster. It assumes familiarity with the architecture of MongoDB and a basic understanding of concepts related to the deployment of enterprise software. For more information on the architecture of MongoDB, please see the MongoDB Architecture Guide.

Fundamentally MongoDB is a database and the concepts of the system, its operations, policies, and procedures should be familiar to users who have deployed and operated other database systems. While some aspects of MongoDB are different from traditional relational database systems, skills and infrastructure developed for other database systems are relevant to MongoDB and will help to make deployments successful. Typically MongoDB users find that existing database administrators, system administrators, and network administrators need minimal training to understand MongoDB. The concepts of a database, tuning, performance monitoring, data modeling, index optimization and other topics are very relevant to MongoDB. Because MongoDB is designed to be simple to administer and to deploy in large clustered environments, most users of MongoDB find that with minimal training an existing operations professional can become competent with MongoDB, and that MongoDB expertise can be gained in a relatively short period of time.

This document discusses many best practices for operating and deploying a MongoDB system. The MongoDB community is vibrant and new techniques and lessons are shared every day.

This document is subject to change. For the most up-to-date version of the document, please visit 10gen.com. For the most current and detailed information on specific topics, please see the online documentation at mongodb.org. Many links are provided throughout this document to help guide users to the appropriate resources online.

I. Roles and Responsibilities

Applications deployed on MongoDB require careful planning and the coordination of a number of roles in an organization's technical teams to ensure successful maintenance and operation. Organizations tend to find many of the same individuals and their respective roles for traditional technology deployments are appropriate for a MongoDB deployment: Data Architects, Database Administrators, System Administrators, Application Developers, and Network Administrators.

In smaller organizations it is not uncommon to find these roles are provided by a small number of individuals, each potentially fulfilling multiple roles, whereas in larger companies it is more common for each role to be provided 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.

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 tradeoffs with each alternative 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.

Database Administrator (DBA)

As with other database systems, many factors should be considered in designing a MongoDB system for a desired performance 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 goals, the recovery goals, and the desired performance characteristics.

System Administrator (sysadmin)

Sysadmins typically perform a set of activities similar to what is required to manage 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#, 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 across multiple data centers. Network resources are a critical component of a MongoDB system. While MongoDB does not require any unusual 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.

II. Preparing for a MongoDB Deployment

Schema Design

Developers and data architects should work together to develop the right data model, and they should invest time in this exercise early in the project. The 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.

The topic of schema design is significant, and a full discussion is beyond the scope of this document. A number of resources are available online, including conference presentations from 10gen solutions architects and MongoDB users, as well as training provided by 10gen. Briefly, some concepts to keep in mind:

DOCUMENT MODEL
MongoDB stores data as documents in a binary representation called BSON. The BSON encoding extends the popular JSON representation to include additional types such as int, long, and floating point. BSON documents contain one or more fields, and each field contains a value of a specific data type, including arrays 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 data for a given record in a single document, whereas in a relational database information for a given record is usually spread across rows in many tables. In other words, data in MongoDB tends to be more localized.

DYNAMIC SCHEMA
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.

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.

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

TRANSACTIONS
MongoDB guarantees atomic updates to data at the document level. It is not possible to update multiple documents in a single atomic operation. Atomicity of updates may influence the schema for your application.

SCHEMA ENFORCEMENT
MongoDB does not enforce schemas. Schema enforcement should be performed by the application.

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

Document Size

The maximum BSON document size in MongoDB is 16MB. User should avoid certain application patterns that would allow documents to grow unbounded. For instance, applications should not typically update documents in a way that causes them to grow significantly after they have been created, as this can lead to inefficient use of storage. If the document size exceeds its allocated space, MongoDB will relocate the document on disk. This automatic process can be resource intensive and time consuming, and can unnecessarily slow down other operations in the database.

For example, in a blogging application it would be difficult to estimate how many responses a blog post might receive from

readers. Furthermore, it is typically the case that only a subset of comments is displayed to a user, such as the most recent or the first 10 comments. Rather than modeling the post and user responses as a single document it would be better to model each response or groups of responses as a separate document with a reference to the blog post. Another example is product reviews on an e-commerce site. The product reviews should be modeled as individual documents that reference the product. This approach would also allow the reviews to reference multiple versions of the product such as different sizes or colors.

OPTIMIZING FOR DOCUMENT GROWTH

MongoDB adaptively learns if the documents in a collection tend to grow in size and assigns a padding factor to provide sufficient space for document growth. This factor can be viewed as the `paddingFactor` field in the output of the `db.<collection-name>.stats()` command. For example, a value of 1 indicates no padding factor, and a value of 1.5 indicates a padding factor of 50%.

When a document is updated in MongoDB the data is updated in-place if there is sufficient space. If the size of the document is greater than the allocated space, then the document may need to be re-written in a new location in order to provide sufficient space. The process of moving documents and updating their associated indexes can be I/O-intensive and can unnecessarily impact performance.

SPACE ALLOCATION TUNING

Users who anticipate updates and document growth may consider two options with respect to padding. First, the `usePowerOf2Sizes` attribute can be set on a collection. This setting will configure MongoDB to round up allocation sizes to the powers of 2 (e.g., 2, 4, 8, 16, 32, 64, etc). This setting tends to reduce the chances of increased disk I/O at the cost of some additional storage usage. The second option is to manually pad the documents. If the application will add data to a document in a predictable fashion, the fields can be created in the document before the values are known in order to allocate the appropriate amount of space during document creation. Padding will minimize the relocation of documents and thereby minimize overallocaiton.

GRIDFS

For files larger than 16MB, MongoDB provides a convention called GridFS, which is implemented by all MongoDB drivers. GridFS automatically divides large data into 256KB 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, and capped collections.

TIME TO LIVE (TTL)

If documents in a collection should only persist for a pre-defined period of time, the TTL feature can be used to automatically delete documents of a certain age rather than scheduling a process to check the age of all documents and run a series of deletes. For example, if user sessions should only exist for one hour, the TTL can be set for 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 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 without designing for storage management.

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.

Indexing

Like most database management systems, indexes are a crucial mechanism for optimizing system performance in MongoDB. And while indexes will improve the performance of some operations by one or more orders of magnitude, they have associated costs in the form of slower updates, disk usage, and memory usage. Users should always create indexes to support queries, but should take care not to maintain indexes that the queries do not use. Each index incurs some cost for every insert and update operation: if the application does not use these indexes, then it can adversely affect the overall capacity of the database. This is particularly important for deployments that have insert-heavy workloads.

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. Generally MongoDB makes use of one index in resolving a query. The query optimizer selects the best index to use by periodically running alternate query plans and selecting the index with the lowest scan count 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 was resolved, including:

- The number of documents returned.
- Which index was 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.

Explain plan will show 0 milliseconds if the query was resolved in less than 1ms, which is not uncommon in well-tuned systems. When explain plan is called, prior cached query plans are abandoned, and the process of testing multiple indexes is evaluated to ensure the best possible plan is used.

If the application will always use indexes, MongoDB can be configured 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 100ms). Profiling data is stored in a capped collection where it can easily be searched for interesting events – it may be easier to query this collection than parsing the log files.

PRIMARY AND SECONDARY INDEXES

A unique, index is created for all documents by the **`_id`** field. MongoDB will automatically create the **`_id`** field and assign a unique value, or the value can be specified when the document is inserted. All user-defined indexes are secondary indexes. Any field can be used for a secondary index, including fields with arrays.

COMPOUND INDEXES

Generally queries in MongoDB can only be optimized by one index at a time. It is therefore useful to create compound indexes for queries that specify multiple predicates. For example, consider an application that stores data about customers. The application may need to find customers based on last name, first name, and state of residence. With a compound index on last name, first name, and state of residence, queries could efficiently locate people with all three of these values specified. An additional benefit of a compound index is that any leading field within the index can be used, so fewer indexes on single fields may

be necessary: this compound index would also optimize queries looking for customers by last name.

UNIQUE INDEXES

By specifying an index as unique, MongoDB will reject inserts of new documents or the update of a document with an existing value for the field for which the unique index has been created. By default all indexes are not unique. If a compound index is specified as unique, the combination of values must be unique. If a document does not have a value specified for the field then an index entry with a value of null will be created for the document. Only one document may have a null value for the field unless the sparse option is enabled for the index, in which case index entries are not made for documents that do not contain the field.

ARRAY INDEXES

For fields that contain an array, each array alue is stored as a separate index entry. For example, documents that describe recipes might include a field for ingredients. If there is an index on the ingredient field, each ingredient is indexed and queries on the ingredient field can be optimized by this index. There is no special syntax required for creating array indexes – if the field contains an array, it will be indexed as a array index.

It is also possible to specify a compound array index. If the recipes also contained a field for the number of calories, a compound index on calories and ingredients could be created, and queries that specified a value for calories and ingredients would be optimized with this index. For compound array indexes only one of the fields can be an array in each document.

GEOSPATIAL INDEXES

MongoDB provides geospatial indexes to optimize queries related to location within a two dimensional space, such as projection systems for the earth. Documents must have a field with a two-element array, such as latitude and longitude to be indexed with a geospatial index. These indexes allow MongoDB to optimize queries that request all documents closest to a specific point in the coordinate system.

SPARSE INDEXES

Sparse indexes only contain entries for documents that contain the specified field. Because the document data model of MongoDB allows for flexibility in the data model from document to document, it is common for some fields to be present only in a subset of all documents. Sparse indexes allow for smaller, more efficient indexes when fields are not present in all documents.

By default, the sparse option for indexes is false. Using a sparse index will sometime lead to incomplete results when performing index-based operations such as filtering and sorting. By default, MongoDB will create null entries in the index for documents that are missing the specified field.

For more on indexes, see **Indexing Overview** in the MongoDB Documentation.

INDEX CREATION OPTIONS

Indexes and data are updated synchronously in MongoDB. The appropriate indexes should be determined as part of the schema design process prior to deploying the system.

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. When the background option is enabled, the total time to create the indexes will be greater than if the indexes are created in the foreground, but it will still be possible to use the database while creating indexes.

PRODUCTION APPLICATION CHECKS FOR INDEXES

Make sure that the application checks for the existence of all appropriate indexes on startup and that it terminates if indexes are missing. Index creation should be performed by separate application code and during normal maintenance operations.

INDEX MAINTENANCE OPERATIONS

Background index operations on a replica set primary become foreground index operations on replica set secondaries, which will block all replication. Therefore the best approach to building indexes on replica sets is to:

1. Restart the secondary replica in standalone mode.
2. Build the indexes.
3. Restart as a member of the replica set.
4. Allow the secondary to catch up to the other members of the replica set.
5. Proceed to step one with the next secondary.
6. When all the indexes have been built on the secondaries, restart the primary in standalone mode. One of the secondaries will be elected as primary so the application can continue to function.
7. Build the indexes on the original primary, then restart it as a member of the replica set.
8. Issue a request for the original primary to resume its role as primary replica.

See the MongoDB Documentation for **Build Index on Replica Sets** for a full set of procedures.

INDEX LIMITATIONS
There are a few limitations to indexes 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 128 characters (including its namespace).
- The optimizer generally uses one index at a time.
- Indexes consume disk space and memory. Use them 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 slow update performance.
- 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:

- **Creating multiple indexes in support of a single query:** MongoDB will use a single index to optimize a query. If you need to specify multiple predicates, you need a compound index. For example, if there are two indexes, one on first name and another on last name, queries that specify a constraint for both first and last names will only use one of the indexes, not both. To optimize these queries, a compound index on last name and first name should be used.
- **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.
- **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 male/female is not as beneficial as an index on zip code, or even better, phone number.
- **Regular expressions:** Trailing wildcards work well, but leading wildcards do not because the indexes are ordered.
- **Negation:** Inequality queries are inefficient with respect to indexes.

Working Sets

MongoDB makes extensive use of RAM to speed up database operations. In MongoDB, all data is read and manipulated through memory-mapped files. Reading data from memory is measured in nanoseconds and reading data from disk is measured in milliseconds; reading from memory is approximately 100,000 times faster than reading data from disk. The set of data and indexes that are accessed during normal operations is call the working set.

It should be the goal of the deployment team that the working 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. This process can be time consuming and will be significantly slower than accessing data that is already 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 the RAM on the server, will cause documents to be read into memory and the working set to be written out to disk. Other examples include some 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 the RAM or adding additional servers to the cluster and sharding your database. This topic is discussed in the section on sharding best practices. However, it is far easier to implement sharding before the resources of the system become limited.

MongoDB Setup and Configuration

SETUP

10gen 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 at **http://www.mongodb.org/downloads**.

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. Installations and upgrades should be automated through popular tools such as Chef and Puppet, and the MongoDB community provides and maintains example scripts for these tools.

UPGRADES

User should upgrade software as often as possible so that they can take advantage of the latest features as well as any stability updates or bug fixes. Upgrades should be tested in non-production environments to ensure production applications are not adversely affected by new versions of the software.

Clusters can be upgraded with reduced downtime or no downtime by using a "rolling upgrade" approach: it is possible for each member of a replica set to run under different versions of MongoDB. As a precaution, the release notes for the MongoDB release 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. Customers can deploy rolling upgrades without incurring any downtime, as each member of a replica set can be upgraded individually without impacting cluster availability.

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 files. 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 called **`mongodump`** and **`mongorestore`** are useful for moving data between two MongoDB systems.

Hardware

The following suggestions 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, your queries, your performance SLA, your availability requirements, and the capabilities of the underlying hardware components. 10gen 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.

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 full 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. MongoDB delegates the management of RAM to the operating system. MongoDB will use as much RAM as possible until it exhausts what is 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. 10gen has observed good results and strong price to performance with SATA SSD and with PCI. 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 money may be more effectively spent on more RAM or SSDs. Another benefit of using SSDs is that they provide a more gentle degradation of performance if the working set no longer fits in memory.

Most MongoDB deployments should use RAID-10. RAID-5 and RAID-6 do not provide sufficient performance. RAID-0 provides good write performance, but limited read performance and 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.

CPU

MongoDB performance is typically not CPU-bound. As MongoDB rarely encounters workloads able to leverage large numbers of cores, it is preferable to have servers with faster clock speeds than numerous cores with slower clock speeds.

SERVER CAPACITY VS. SERVER QUANTITY

MongoDB was designed with horizontal scale-out in mind using cost-effective, commodity hardware. Even within the commodity server market there are options regarding the number of processors, amount of RAM, and other components. Customers frequently ask whether it is better to have a smaller number of larger capacity servers or a larger number of smaller capacity servers. In a MongoDB deployment it is important to ensure there is sufficient RAM to keep the database working set in memory. While it is not required that the working set fit in RAM, the performance of the database will degrade if a significant percentage of reads and writes are applied to data and indexes that are not in RAM.

PROCESS PER HOST

Users should run one **mongod** process per host. A **mongod** process is designed to run as the single server process on a system; doing so enables it to store its working set in memory most efficiently. Running multiple **mongod** processes on a single host reduces redundancy and risks operational degradation, as multiple instances vie for the same resources. The exception is for **mongod** processes that are acting in the role of arbiter – these may co-exist with other processes or be deployed on smaller hardware.

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 numerous MongoDB users leverage infrastructure-as-a-service (IaaS) products like Amazon Web Services' Elastic Compute Cloud (AWS EC2). IaaS deployments are especially good for initial testing

and development, as they provide a low-risk, low-cost means for getting MongoDB up and running. 10gen has partnerships with a number of cloud and managed services providers, such as AWS, Softlayer, and Microsoft Windows Azure, in addition to partners that provide fully managed instances of MongoDB, like MongoLab and MongoHQ.

SIZING FOR MONGOS AND CONFIG SERVER PROCESSES

For sharded systems, additional processes must be deployed with the **mongod** data storing processes: **mongos** 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 requirements regarding sizing.

Within a shard, MongoDB further partitions documents into chunks. MongoDB maintains metadata about the relationship of chunks to shards in the config server. Three config servers are maintained in sharded deployments to ensure availability of the metadata at all times. To estimate the total size of the shard metadata, multiply the size of the chunk metadata times the total number of chunks in your database – the default chunk size is 64MB. For example, a 64TB database would have 1 million chunks and the total size of the shard metadata managed by the config servers would be 1 million times the size of the chunk metadata, which could range from hundreds of MB to several GB of metadata. Shard metadata access is infrequent: each mongos maintains a cache of this data, and it is periodically updated by background processes when chunks are split or migrated to other shards. The hardware for a config server should therefore be focused on availability: redundant power supplies, redundant network cards, redundant RAID controllers, and redundant storage should be used.

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 shards, and in other cases **mongos** performs aggregation and other tasks. 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 connection to applications and to each **mongos**

While **mongod** instances are typically limited by disk performance and available RAM more than they are limited by CPU speed, **mongos** uses limited RAM and will benefit from fast CPUs and networks.

Operating System and File System Configurations for Linux

Only 64-bit versions of operating systems should be used for MongoDB. Version 2.6.36 of the Linux kernel or later should be used for MongoDB in production.

Because MongoDB preallocates its database files before using them and because MongoDB uses very large files on average, Ext4 and XFS file systems are recommended:

- If you use the Ext4 file system, use at least version 2.6.23 of the Linux Kernel.
- If you use the XFS file system, use at least version 2.6.25 of the Linux Kernel.

For MongoDB on Linux use the following recommended configurations:

- Turn off **atime** for the storage volume with the database files.
- Do not use **hugepages** 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 (16KB) is a good starting point.

- Synchronize time between your hosts. This is especially important in MongoDB clusters.

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 to ensure there is no contention with other processes.

While each deployment has unique requirements, the following settings are a good starting point **`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*): 64000
- **-m** (*memory size*): unlimited
- **-u** (*processes/threads*): 32000

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 pre-defined processes that communicate with a MongoDB system: application servers, monitoring processes, and MongoDB processes.

By default MongoDB processes will bind to all available network interfaces on a system. 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 on the MongoDB Documentation page for **Security Practices and Management**.

Community Recommendations

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

III. High Availability

Under normal operating conditions, a MongoDB cluster 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 to enable fast crash recovery and consistency in the storage engine. Journaling is enabled by default for 64-bit platforms. Users should never disable journaling; journaling helps prevent corruption and increases

operational resilience. Journal commits are issued at least as often as every 100ms by default. In the case of a server crash journal entries will be recovered automatically. Therefore the time between journal commits represents the maximum possible data loss. This setting can be configured to a value that is appropriate for the application.

It may be beneficial for performance to locate MongoDB's journal files and data files on separate storage arrays. 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.

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 at inopportune times.

A replica set typically consists of multiple replicas. 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 CPU failure), one of the secondary members is automatically elected to primary and begins to process all writes. 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. When a node is down, MongoDB has less resiliency and the DBA or sysadmin should work to recover the failed replica in order to mitigate the temporarily reduced resiliency of the system.

Replica sets also provide operational flexibility by providing sysadmins with an avenue for performing hardware and software maintenance without taking down the entire system. For instance, if one needs to perform a hardware upgrade on all members of the replica set, one can perform the upgrade on each secondary replica, one at a time, without impacting the replica set. When all secondaries have been upgraded, one can temporarily demote the primary replica to secondary to upgrade that server. Similarly, the addition of indexes and other operational tasks can be carried out on replicas one at a time without interfering with the uptime of the system.

For more details, please see the MongoDB Documentation on **Rolling Upgrades**.

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. Run an odd number of members or run an arbiter (a replica that exists solely for participating in election of the primary) on one of your application servers if you have an even number of members.
- With geographically distributed members, know where the majority of members will be in the case of any network partitions. Attempt to ensure that the set can elect a primary among the members in the primary data center.
- Consider including a hidden member (a replica that cannot become a primary) or delayed member (a replica that applies changes on a fixed time delay to provide recovery from unintentional transactions) in your replica set to support dedicated functionality, like backups, reporting, and testing.
- Consider keeping one or two members of the set in an off-site data center, but make sure to configure the priority to prevent them from becoming primaries.
- The number of nodes in the cluster should be odd, including arbiters and other types of nodes.
- There should be at least three replicas with copies of the data in a replica set, or two replicas with an arbiter.

More information on replica sets can be found on the **Replication Fundamentals** MongoDB Documentation page.

Availability of Writes

MongoDB allows one to specify the level of availability when issuing writes to the system, which is called write concern. The following options can be configured on a per connection, per database, per collection, or per operation basis:

- **Errors Ignored:** Write operations are not acknowledged by MongoDB, and may not succeed in the case of connection errors that the client is not yet aware of, or if the mongod produces an exception (e.g., a duplicate key exception for unique indexes.) While this operation is efficient because it does not require the database to respond to every write operation, it also incurs a significant risk with regards to the persistence and durability of the data. ***Warning: Do not use this option in normal operation.***
- **Unacknowledged:** MongoDB does not acknowledge the receipt of write operation as with a write concern level of ignore; however, the driver will receive and handle network errors as possible given system networking configuration. Sometimes this configuration is called "fire and forget" and it was the default global write concern for all drivers before late 2012.
- **Write Acknowledged:** The `mongod` will confirm the receipt of the write operation, allowing the client to catch network, duplicate key, and other exceptions. This is the default global write concern.
- **Journal Safe (journaled):** The `mongod` will confirm the write operation only after it has flushed the operation to the journal. This confirms that the write operation can survive a `mongod` crash and ensures that the write operation is durable on disk. While `receipt acknowledged` provides the fundamental basis for write concern, there is an up-to-100-millisecond window between journal commits where the write operation is not fully durable. Require `journaled` as part of the write concern to provide this durability guarantee.
- **Replica Safe:** The options mentioned above confirm writes to the replica set primary. It is also possible to wait for acknowledgement of writes to other replicas. MongoDB supports writing to a specific number of replicas, or waiting for acknowledgement of the write to a special mode called "majority." Because replicas can be deployed across racks within data centers and across multiple data centers, ensuring writes to additional replicas can provide extremely robust durability for updates.
- **Data Center Awareness:** sophisticated policies can be created to ensure data is written to specific combinations of replica sets for each write operation prior to acknowledgement of success using tag sets. 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**.

For more on the subject of configurable availability of writes see the MongoDB Documentation on **Write Concern**.

Read Preferences

By default, all reads are made against the primary node. This ensures applications see the most recent state of the database and guarantees the ability to read your own writes. It is also possible to specify that reads be allowed against non-primary replicas where data is eventually consistent. The read throughput of a database can be increased by allowing reads from the secondaries. In some cases non-primary reads may be appropriate, such as applications where the volume of updates is relatively low, where updates occur within a predictable window, or for specific workloads such as reporting where small delays in replicated data may be acceptable. One of the available options is `primaryPreferred`, which issues reads to a secondary replica only if the primary is unavailable. This configuration allows for reads during the failover process which can take time to complete. Another option directs reads to the replica closest to the user, which can significantly decrease the latency of read and write operations.

For more on the subject of configurable reads, see the MongoDB Documentation page on **Replica Set Read Preference**.

IV. Scaling a MongoDB System

Horizontal Scaling with Shards

MongoDB provides horizontal scale-out for databases using a technique called sharding, which is transparent to applications. MongoDB distributes data across multiple physical partitions called shards. MongoDB ensures data is equally distributed across

shards as data is updated or the size of the cluster increases or decreases. Sharding allows MongoDB deployments to address the limitations of a single server, such as bottlenecks in RAM or disk I/O, without adding complexity to the application.

However, sharding can add operational complexity to a MongoDB deployment and it has its own 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 will soon exceed the capacity of the maximum amount of RAM in the system.
- **Disk I/O Limitation:** The system has a large amount of write activity, and the operating system cannot write data fast enough to meet demand, and/or I/O bandwidth limits how fast the writes can be flushed to disk.
- **Storage Limitation:** The data set approaches or exceeds the storage capacity of a single node in the system.

MongoDB users that meet these criteria or that think they are likely to in the future should plan on sharding in advance rather than waiting until they run out of capacity. MongoDB users 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.

Selecting a Shard Key

Sharding uses range-based partitioning to distribute a collection of documents based on a user-specified shard key. Because shard keys and their values cannot be updated, and because the values of the shard key determine the physical storage of the documents as well as how queries are routed across the cluster, it is very important to select a good shard key.

As an example to illustrate good shard key selection, consider an email repository. Each document includes a unique key, in this case created by MongoDB automatically, a user identifier, the date and time the email was sent, email subject, recipients, email body, and any attachments. Emails can be large, in this case up to 16MB, and most users have lots of email, several GB each. Finally, we know that the most popular query in the system is to retrieve all emails for a user, sorted by time. We also know the second most popular query is on recipients of emails. The indexes needed for this system are `on _id`, `user+time`, and `recipients`.

Each email document is as follows:

```
{
  _id ObjectId(),
  user: 123,
  time: Date(),
  subject: ". . . ",
  recipients: [….],
  body: " . . . . ",
  attachments: [.…]
}
```

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

- **Cardinality:** Data partitioning is managed in 64MB chunks by default. Low cardinality (e.g., the attribute "size") will tend to group documents together on a small number of shards, which in turn will require frequent rebalancing of the chunks. Instead, a shard key should exhibit high cardinality.
- **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.
- **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.

In some cases it may not be possible to achieve all three goals with a natural field in the data. It may therefore be required to create a new field for the sake of sharding and to store this in your documents, potentially in the `_id` field. Another alternative is to create a compound shard key by combining multiple fields in the shard key index. If neither of these options is viable, it may be necessary to make compromises in performance, either for reads or for writes. If reads are more important, then prioritize the shard key selection to the extent the shard key provides query isolation. If writes are more important, then prioritize the shard key selection to the extent the shard key provides write distribution.

Considering the email repository example, the quality of a few different candidate shard keys is assessed in the table below:

	Cardinality	Insert Scaling	Query Isolation
`_id`	Doc level	One shard[1]	Scatter/gather[2]
`hash(_id)`	Hash level[3]	All shards	Scatter/gather
`User`	Many docs[4]	All shards	Targeted
`User,time`	Doc level	All shards	Targeted

Sharding Best Practices

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

- **Select a good shard key.** For more on selecting a shard key, see the section 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 configuration servers to provide redundancy.** Production deployments must use three config servers. Config servers should be deployed in a topology that is robust and resilient to a variety of failures.
- **Use replica sets.** Replica sets provide data redundancy and allow MongoDB to continue operating in a number of failure scenarios. Replica sets should be used in any deployment.
- **Use multiple `mongos` instances.**
- **For bulk inserts, there are specific best practices.** Pre-split data into multiple chunks so that no balancing is required during the insert process. Alternately, disable the balancer. Also, use multiple `mongos` instances to load in parallel for greater throughput. For more information see **Strategies for Bulk Inserts in Sharded Clusters** in the MongoDB Documentation.

Dynamic Data Balancing

As data is loaded into MongoDB, the system may need to dynamically rbalance 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 the balancer or to configure when balancing is performed to further minimize the impact on performance. For more information on the balancer and scheduling the balancing process, see the MongoDB Documentation page on **Sharding Fundamentals**.

1 MongoDB's auto-generated `ObjectId()` is based on timestamp and is therefore sequential in nature and will result in all documents being written to the same shard.

2 Most frequent query is on `userId+time`, not `_id`, so all queries will be scatter/gather.

3 Cardinality will be reflective of extent of hash collisions: the more collisions, the worse the cardinality.

4 Each user has many documents, so the cardinality of the user field is not very high.

Sharding and Replica Sets

Sharding and replica sets are absolutely compatible and both features should be utilized in most deployments. 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.

Geographic Distribution

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

It is possible to combine the features of replica sets, tag-aware sharding, read preferences, and write concern in order to provide a deployment that is geographically distributed in which users to read and write to their local data centers. Tag-aware sharding enables users to ensure that particular data in a sharded system is always on specific shards. This can be used for global applications to ensure that data is geographically close to the systems that use it; it can also fulfill regulatory requirements around data locality. One can restrict sharded collections to a select set of shards, effectively federating those shards for different uses. For example, one can tag all 'USA' data and assign it to shards located in the United States.

More information on sharding can be found in the MongoDB Documentation under **Sharding Fundamentals** and **Data Center Awareness**.

V. Disaster Recovery

Many projects must address disaster recovery in order to fulfill obligations related to compliance, service level agreements or internal legal standards. Organizations tend to establish a recovery point objective and recovery time objective for their MongoDB deployment. This section describes effective strategies for recovering from a catastrophic disaster that affects a MongoDB system.

Multi-Data Center Replication

MongoDB's mature replication capabilities provide continuous database availability. Replica sets allow for flexible deployment designs that account for failure at the server, rack, and data center levels. Rather than performing backups and maintaining multiple backup copies of the database offsite from a datacenter, MongoDB recommends deploying a MongoDB database across multiple datacenters. In the case of a natural or human-induced disaster, the failure of a single datacenter can be accommodated with no downtime for a MongoDB database when deployed across data centers.

Backup and Restore

There are two commonly used approaches to backing up a MongoDB database: file system copies and a tool packaged with MongoDB called **`mongodump`**. File system backups are recommended, such as that provided by Linux LVM, which 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 shard 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 **Using Block Level Backup Methods** in the MongoDB Documentation.

As distributed systems, sharded environments complicate backup and restore operations. Currently MongoDB does not provide an automated method for locking all shards in a cluster for backup purposes. The process for creating the backup follows these approximate steps:

1. Stop the balancer so that chunks are consistent across shards in the cluster.

2. Stop one of the config servers to prevent all metadata changes.
3. Lock one replica of each of the shards using **db.fsyncLock()**.
4. Create a backup of one of the config servers.
5. Create the file system snapshot for each of the locked replicas.
6. Unlock all the replicas.
7. Start the config server.
8. Start the balancer.

For more on backup and restore in sharded environments, see the MongoDB Documentation page on **Backups with Sharding and Replication** and the tutorial on **How to Create a Backup of a Sharded Cluster with File System Snapshots**.

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** and then replaying it during **mongorestore**, a tool that imports content from BSON database dumps produced by **mongodump**. **mongodump** can also work against an inactive set of database files.

More information on creating backups can be found on the **Backup and Restoration Strategies** MongoDB Documentation page.

VI. Capacity Planning

System performance and capacity planning are two important topics that should be addressed in 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 is a partial list of popular tools for monitoring MongoDB as well as different aspects of the system that should be monitored.

Monitoring Tools

MONGODB MONITORING SERVICE (MMS)
Users should monitor their deployments proactively. 10gen provides a free, cloud-based solution called MongoDB Monitoring Service (MMS). MMS is also available as an on-premise solution from 10gen.

MMS features charts, custom dashboards, and automated alerting. MMS runs in the cloud and requires minimal setup and configuration. Users install a local agent on all **mongod** instances that tracks hundreds of key health metrics on database utilization, including:

- **Op Counters:** Count of operations executed per second.
- **Memory:** Amount of data MongoDB is using.
- **Lock Percent:** Percent of time spent in write lock.
- **Background Flush:** Average time to flush data to disk.
- **Connections:** Number of current open connections to MongoDB.
- **Queues:** Number of operations waiting to run.

- **Page Faults:** Number of page faults from disk.
- **Replication:** `Oplog` length (for primary) and replication delay to primary (on secondary).
- **Journal:** Amount of data written to journal.

These metrics are securely reported to MMS where they are processed, aggregated, alerted, and visualized in a browser. Users can easily determine the health of their clusters on a variety of performance metrics, and 10gen support can more effectively understand and help diagnose the health of a cluster in real-time or based on some point in the past.

The MMS agent transmits all metrics to the MMS servers over SSL (128-bit encryption), and agent traffic is exclusively outbound from your data center. The monitoring agent is a script, whose source code you are free to examine. MMS pushes no data to the agent. The agent is subject to firewalls and contains support for MongoDB database authentication. Monitored services are only discovered when the agent fetches information provided to MMS or from other members of the cluster. The agent only records server metrics: the collection of hardware and application data must be explicitly enabled. The web interface is secured over SSL, and extensive data access controls and audits are in place to ensure that the safety of your data. Each collection of cluster data is visible by the customer and by 10gen support.

The manual for MMS is available online at **http://mms.10gen.com/help/index.html**.

HARDWARE MONITORING

Munin node is an open-source software program that monitors hardware and reports on metrics like disk and RAM usage. MMS can collect this data from Munin node and provide it along with other data available in the MMS 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.

SNMP

For those customers that have institutional policies that may prevent them from using a cloud-based monitoring solution, the Subscriber Edition of MongoDB includes SNMP support, which can be used to integrate MongoDB with external monitoring solutions.

DATABASE PROFILER

MongoDB provides a profiling capability called the Database Profiler that 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. Profiling data is stored in a capped collection where it can easily be searched for interesting events – it may be easier to query this collection than to try to parse the log files.

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 stats.

MONGOSTAT

`mongostat` is a utility that ships with MongoDB. It shows real-time stats 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.

OTHER POPULAR TOOLS

There are several popular open-source monitoring tools for which MongoDB plugins are available:

- Nagios
- Ganglia
- Cacti
- Scout

- Munin
- Zabbix

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`:** Provide 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

APPLICATION LOGS AND DATABASE LOGS

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

PAGE FAULTS

When a working set ceases to fit in memory, or other operations have moved other data into 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 doesn't fit in 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.

DISK

Your MongoDB system should be designed so that its working set fits in memory (see the section on working sets for more information on this topic). However, disk I/O is still a key performance consideration for a MongoDB system because writes are flushed to disk every 60 seconds and commits to the journal every 100ms. 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 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 the available 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, they rely 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 things start to go wrong, such as a high degree of page faults, a high degree of write locks due to update volume, or a long-running query, requests from applications will begin to queue up. 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 32KB. 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 evenly 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 **`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.

In order to ensure data is evenly distributed across all shards in a cluster, it is important to select a good shard key. 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 shard keys and shard values are immutable. Shard keys and shard values are immutable, so in addition to reloading all the data 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 to replicate to a secondary. Some 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 config servers. Config servers are critical to the system for understanding the location of documents across shards. If one config server goes down then the other two will go into read-only mode. The database will remain operational in this case, but the balancer will be unable to move chunks until all three config servers are available.

For more information on monitoring tools and things to monitor, see the **Monitoring Database Systems** page in the MongoDB Documentation.

VII. Security

As with all software, administrators of MongoDB must consider security and risk exposures 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 un-trusted 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 follow a secure development lifecycle, and to follow deployment best practices.

Secure environments use the following strategies to control access:

- Network filter (e.g. firewalls, ACL rules on routers) rules that block all connections from unknown systems to MongoDB components. Firewalls should limit both incoming and outgoing traffic to/from a specific port to trusted and untrusted systems.
- Binding **`mongod`** and **`mongos`** instances to specific IP addresses to limit accessibility.
- Limiting MongoDB programs to non-public local networks and virtual private networks.
- Running the process in a **`chroot`** environment.
- Requiring authentication for access to MongoDB instances.
- Mandating strong, complex, single purpose authentication credentials. This should be part of the internal security policy (but is not currently enforceable in MongoDB).
- Deploying a model of least privileges, where all users and processes only have the amount of access they need to accomplish required tasks.
- Following application development and deployment best practices, which include: validating all inputs, managing sessions and application-level access control.

In addition to the above security controls, it is advisable to monitor your MongoDB logs on a continuous basis for anything unusual that may indicate a security incident.

Access Control

MongoDB provides basic authentication capabilities. Access is provisioned on a per-database basis. Users either have read-only access or normal access which permits reads, writes, and the creation of additional users.

SSL

Many deployments will require encrypting data in motion over the network. Many of the MongoDB drivers support SSL connections, including the drivers for Java, Ruby, Python, node.js, and C#. The subscriber edition of MongoDB provides support for SSL connections.

Data Encryption

To support regulatory requirements, some customers may need to encrypt data stored in MongoDB. One approach is to encrypt field-level data within the application layer using the requisite encryption type that is appropriate for the application. Another option is to use third-party libraries that provide disk-level encryption as part of the operating system kernel. For example, 10gen has a partnership with Gazzang, which has a certified product for encrypting and securing sensitive data within MongoDB. The solution encrypts data in real time and Gazzang provides advanced key management that ensures only authorized processes can access this data. Gazzang software ensures that the cryptographic keys remain safe and ensures compliance with standards such as HIPPA, PCI-DSS and FERPA.

Query Injection

As a client program assembles a query in MongoDB, it builds a BSON object, not a string. Thus traditional SQL injection attacks should not pose a risk to the system for queries submitted as BSON objects.

However, several MongoDB operations permit the evaluation of arbitrary Javascript expressions and care should be taken to avoid malicious expressions. Fortunately most queries can be expressed in BSON and for cases where Javascript is required, it is possible to mix Javascript and BSON so that user-specified values are evaluated as values and not as code.

10gen | The MongoDB Company

New York • Palo Alto • Washington, DC • London • Dublin • Barcelona • Sydney
US 646.237.8815 • INTL 650.440.4474 • info@10gen.com

10gen | The MongoDB Company

Published by 10gen, Inc.
February 2013