Performance Best Practices for MongoDB

MongoDB 3.4
November 2016

Table of Contents

Introduction

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

This guide outlines considerations for achieving performance at scale in a MongoDB system across a number of key dimensions, including hardware, application patterns, schema design, and indexing, disk I/O, Amazon EC2, and designing for benchmarks. While this guide is broad in scope, it is not exhaustive. Following the recommendations in this guide will reduce the likelihood of encountering common performance limitations, but it does not guarantee good performance in your application.

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.

MongoDB works closely with users to help them optimize their systems. Users should monitor their systems to identify bottlenecks and limitations. There are a variety of tools available, the most comprehensive of which are MongoDB Ops Manager and Cloud Manager, discussed later in this guide.

For a discussion on the architecture of MongoDB and some of its underlying assumptions, see the MongoDB Architecture Guide. For a discussion on operating a MongoDB system, see the MongoDB Operations Best Practices.

MongoDB Pluggable Storage Engines

MongoDB 3.0 exposed a new 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.

Any of these storage engines can coexist within a single MongoDB replica set, making it easy to evaluate and migrate between 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; if another engine is preferred then start the `mongod` using the `--storageEngine` option. If a 3.2 (or later) `mongod` process is started and one or more databases already exist then MongoDB will use whichever storage engine those databases were created with.

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

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 best practices in this guide apply to all of the supported storage engines. Any differences in recommendations between the storage engines are noted.

Hardware

You can run MongoDB anywhere – from ARM (64 bit) processors through to commodity x86 CPUs, all the way up to IBM Power and zSeries platforms.

Most users scale out their systems by using many commodity servers operating together as a cluster. MongoDB provides native replication to ensure availability; auto-sharding to uniformly distribute data across servers; and in-memory computing to provide high performance without resorting to a separate caching layer. The following considerations will help you optimize the hardware of your MongoDB system.

Ensure your working set fits in RAM. As with most databases, MongoDB performs best when the working set (indexes and most frequently accessed data) fits in RAM. RAM size is the most important factor for hardware; other optimizations may not significantly improve the performance of the system if there is insufficient RAM. If your working set exceeds the RAM of a single server, consider sharding your system across multiple servers. Use the `db.serverStatus()` command to view an estimate of the the current working set size.

Use SSDs for write-heavy applications. 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, PCIe, and NVMe SSDs. Commodity SATA spinning drives are comparable to higher cost spinning drives due to the random 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 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.

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.

Configure compression for storage and I/O-intensive workloads. MongoDB natively supports compression when using the WiredTiger storage engine. Compression reduces storage footprint by as much as 80%, and enables higher IOPscas 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 both documents and indexes. The default Snappy compression algorithm provides a balance between high document and journal compression ratios (typically around 70%, dependent on data types) 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 use prefix compression by default, which serves to reduce the in-memory footprint of index storage, freeing up more of the RAM for frequently accessed documents. Testing has shown a typical 50% compression ratio using the prefix algorithm, though users are advised to test with their own data sets. Administrators can modify the default compression settings for all collections and indexes. Compression is also configurable on a per-collection and per-index basis during collection and index creation.

Combine multiple storage & compression types. 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.

Allocate CPU hardware budget for faster CPUs. 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.

Dedicate each server to a single role in the system. 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. If 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.

The size of the WiredTiger cache is tunable through the `storage.wiredTiger.engineConfig.cacheSizeGB` setting and should be large enough to hold your entire working set. If the cache does not have enough space to load additional data, WiredTiger evicts pages from the cache to free up space. By default, `storage.wiredTiger.engineConfig.cacheSizeGB` is set to 60% of available RAM - 1 GB; caution should be taken if raising the value as it takes resources from the OS, and WiredTiger performance can actually degrade as the filesystem cache becomes less effective.

For availability, multiple members of the same replica set should not be co-located on the same physical hardware or share any single point of failure such as a power supply.

Use multiple query routers. Use multiple `mongos` processes spread across multiple servers. A common deployment is to co-locate the `mongos` process on application servers, which allows for local communication between the application and the `mongos` process.The appropriate number of `mongos` processes will depend on the nature of the application and deployment.

Exploit multiple cores. The WiredTiger storage engine is multi-threaded and can take advantage of many CPU cores. Specifically, the total number of active threads (i.e. concurrent operations) relative to the number of CPUs can impact performance:

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

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

Due to its concurrency model, the MMAPv1 storage engine does not require many CPU cores. As such, increasing the number of cores can help but does not provide significant return.

Disable NUMA, Running MongoDB on a system with Non-Uniform Access Memory (NUMA) can cause a number of operational problems, including slow performance for periods of time and high system process usage.

When running MongoDB servers and clients on NUMA hardware, you should configure a memory interleave policy so that the host behaves in a non-NUMA fashion.

Intra-Cluster Network Compression. As a distributed database, MongoDB relies on efficient network transport during query routing and inter-node replication. MongoDB 3.4 introduces a new option to compress the wire protocol used for intra-cluster communications. 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.

Application Patterns

MongoDB is an extremely flexible database due to its dynamic schema and rich query model. The system provides extensive secondary indexing capabilities to optimize query performance. Users should consider the flexibility and sophistication of the system in order to make the right trade-offs for their application. The following considerations will help you optimize your application patterns.

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

Avoid negation in queries. 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.

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

Test every query in your application with `explain()`. MongoDB provides an `explain` plan capability that shows information about how a query will be, or was, resolved, including:

- The number of documents returned

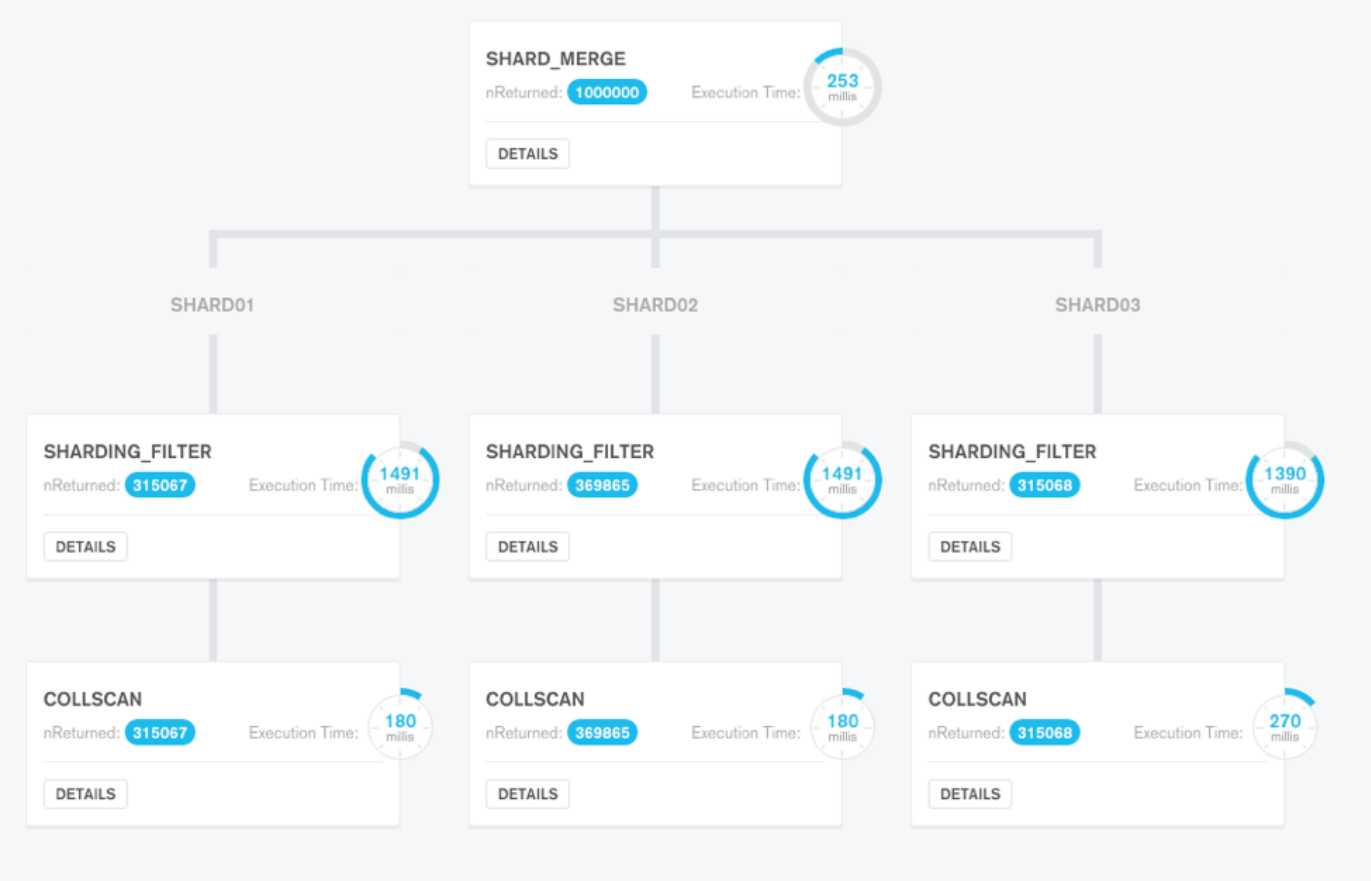

Figure 1: MongoDB Compass visual query plan for performance optimization across distributed clusters

- 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.

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

Choose the appropriate 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.

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

Choose the right read-concern. 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 adds 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.

Use the most recent drivers from MongoDB. MongoDB supports drivers for nearly a dozen languages. These drivers are engineered by the same team that maintains the database kernel. Drivers are updated more frequently than the database, typically every two months. Always use the most recent version of the drivers when possible. Install native extensions if available for your language. Join the MongoDB community mailing list to keep track of updates.

Ensure uniform distribution of shard keys. When shard keys are not uniformly distributed for reads and writes, operations may be limited by the capacity of a single shard. When shard keys are uniformly distributed, no single shard will limit the capacity of the system.

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

Schema Design & Indexes

MongoDB uses a binary document data model based called BSON that is based on the JSON standard. Unlike flat tables in a relational database, MongoDB's document data model is closely aligned to the objects used in modern programming languages, and in most cases it removes the need for complex transactions or joins due to the advantages of having related data for an entity or object contained within a single document, rather than spread across multiple tables. There are best practices for modeling data as documents, and the right approach will depend on the goals of your application. The following considerations will help you make the right choices in designing the schema and indexes for your application.

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

Avoid large documents. The maximum size for documents in MongoDB is 16 MB. In practice, most documents are a few kilobytes or less. Consider documents more like rows in a table than the tables themselves. Rather than maintaining lists of records in a single document, instead make each record a document. For large media items, such as video or images, consider using GridFS, a convention implemented by all the drivers that automatically stores the binary data across many smaller documents.

Avoid unbounded document growth – MMAPv1. When a document is updated in the MongoDB MMAPv1 storage engine, 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. The process of moving documents and updating their associated indexes can be I/O-intensive and can unnecessarily impact performance. To anticipate future growth, the `usePowerOf2Sizes` attribute is enabled by default on each collection. This setting automatically configures MongoDB to round up allocation sizes to the powers of 2. This setting reduces the chances of increased disk I/O at the cost of using some additional storage.

An additional strategy is to manually pad the documents to provide sufficient space for document growth. 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 over-allocation. Learn more by reviewing the record allocation strategies in the documentation.

Avoiding unbounded document growth is a best practice schema design for any database, but the specific considerations above are *not relevant* to the default WiredTiger storage engine which rewrites the document for each update.

Avoid large indexed arrays. Rather than storing a large array of items in an indexed field, store groups of values across multiple fields. Updates will be more efficient.

Avoid unnecessarily long field names. Field names are repeated across documents and consume space. By using smaller field names your data will consume less space, which allows for a larger number of documents to fit in RAM. Note that with WiredTiger's native compression, long field names have less of an impact on the amount of disk space used but the impact on RAM is the same.

Use caution when considering indexes on low-cardinality fields. Queries on fields with low cardinality can return large result sets. Avoid returning large result sets when possible. Compound indexes may include values with low cardinality, but the value of the combined fields should exhibit high cardinality.

Eliminate unnecessary indexes. Indexes are resource-intensive: even with compression enabled they consume RAM, and as fields are updated their associated indexes must be maintained, incurring additional disk I/O overhead.

Remove indexes that are prefixes of other indexes. Compound indexes can be used for queries on leading fields within an index. For example, a compound index on last name, first name can be also used to filter queries that

specify last name only. In this example an additional index on last name only is unnecessary,

Use a compound index rather than index intersection. For best performance when querying via multiple predicates, compound indexes will generally be a better option.

Use partial indexes. Reduce the size and performance of indexes by only including documents that will be accessed through the index. e.g. Create a partial index on the `orderID` field that only includes order documents with an `orderStatus` of "In progress", or only index the `emailAddress` field for documents where it exists.

Avoid regular expressions that are not left anchored or rooted. Indexes are ordered by value. 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.

Use index optimizations available in the WiredTiger storage engine. As discussed earlier, the WiredTiger engine compresses indexes by default. In addition, administrators have the flexibility to place indexes on their own separate volume, allowing for faster disk paging and lower contention.

Understand any existing document schema – MongoDB Compass. If there is an existing MongoDB database that needs to be understood and optimized then MongoDB Compass is an invaluable tool.

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. By understanding what kind of data is present, you're better placed to determine what indexes might be appropriate.

Without 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.

MongoDB Compass is included with MongoDB Professional and MongoDB Enterprise Advanced.

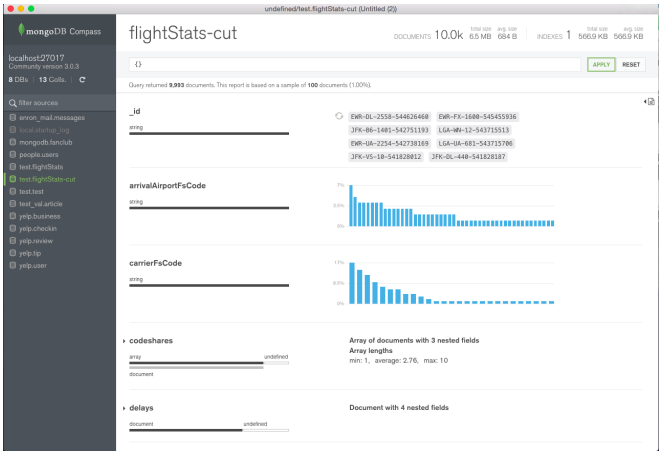

Figure 2: Document structure and contents exposed by MongoDB Compass

Identify & remove obsolete indexes. 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.

Disk I/O

While MongoDB performs all read and write operations through in-memory data structures, data is persisted to disk and queries on data not already in RAM trigger a read from disk. As a result, the performance of the storage sub-system is a critical aspect of any system. Users should take care to use high-performance storage and to avoid networked storage when performance is a primary goal of the system. The following considerations will help you use the best storage configuration, including OS and file system settings.

Readahead size should be set to 32. Use the `blockdev --setra <value>` command to set the `readahead` block size to 32 (16kb) or the size of most documents, whichever is larger.

If the `readahead` size is much larger than the size of the data requested, a larger block will be read from disk – this is wasteful as most disk I/O in MongoDB is random. This has two undesirable consequences:

1. The size of the read will consume RAM unnecessarily.
2. More time will be spent reading data than is necessary.

Both will negatively affect the performance of your MongoDB system. The `readahead` block size should not be set lower than 32.

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.

Disable access time settings. Most file systems will maintain metadata for the last time a file was accessed. While this may be useful for some applications, in a database it means that the file system will issue a write every time the database accesses a page, which will negatively impact the performance and throughput of the system.

Don't use Huge Pages. Do not use Huge Pages virtual memory pages, MongoDB performs better with normal virtual memory pages.

Use RAID10. 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.

By using separate storage devices for the journal and data files you can increase the overall throughput of the disk subsystem. Because the disk I/O of the journal files tends to be sequential, SSD may not provide a substantial improvement and standard spinning disks may be more cost effective.

Use multiple devices for different databases – WiredTiger. Set `directoryForIndexes` so that indexes are stored in separate directories from collections and `directoryPerDB` to use a different directory for each database. The various directories can then be mapped to different storage devices, thus increasing overall throughput.

Note that using different storage devices will affect your ability to create snapshot-style backups of your data, since the files will be on different devices and volumes.

- **Implement multi-temperature storage & data locality using MongoDB 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 adds new helper functions and additional options in Ops Manager and Cloud Manager to configure Zones, essential for managing large deployments.

Considerations for Amazon EC2

Amazon EC2 is an extremely popular environment for MongoDB deployments. MongoDB has worked with Amazon to determine best practices in order to ensure users have a great experience with MongoDB on EC2.

Deploy MongoDB with MongoDB Cloud Manager. The Cloud Manager Service makes it easy for you to provision, monitor, backup, and scale MongoDB. Cloud Manager provisions your AWS virtual machines with an optimal configuration for MongoDB, including configuration of the `readahead` and `ulimits`. You can choose between different instance types, Linux AMIs, EBS volumes, ephemeral storage, and regions. Provisioned IOPS and EBS-optimized instances provide substantially better performance for MongoDB systems. Both can be configured from Cloud Manager.

Considerations for Benchmarks

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

Model your benchmark on your application. The queries, data, system configurations, and performance goals you test in a benchmark exercise should reflect the goals of your production system. Testing assumptions that do not reflect your production system is likely to produce misleading results.

Create chunks before loading, or use hash-based sharding. If range queries are part of your benchmark use range-based sharding and create chunks before loading. Without pre-splitting, data may be loaded into a shard then moved to a different shard as the load progresses. By pre-splitting the data, documents will be loaded in parallel into the appropriate shards. If your benchmark does not include range queries, you can use hash-based sharding to ensure a uniform distribution of writes.

Disable the balancer for bulk loading. Prevent the balancer from rebalancing unnecessarily during bulk loads to improve performance.

Prime the system for several minutes. In a production MongoDB system the working set should fit in RAM, and all reads and writes will be executed against RAM. MongoDB must first page the working set into RAM, so prime the system with representative queries for several minutes before running the tests to get an accurate sense for how MongoDB will perform in production.

Monitor everything to locate your bottlenecks. It is important to understand the bottleneck for a benchmark. Depending on many factors any component of the overall system could be the limiting factor. A variety of popular tools can be used with MongoDB – many are listed in the manual.

The most comprehensive tool for monitoring MongoDB is Ops Manager, available as a part of MongoDB Enterprise Advanced. Featuring charts, custom dashboards, and automated alerting, Ops Manager tracks 100+ key database and systems metrics including operations counters, memory, and CPU utilization, replication status, open connections, queues, and any node status. The metrics are securely reported to Ops Manager where they are processed, aggregated, alerted, and visualized in a browser, letting administrators easily determine the health of MongoDB in real-time. The benefits of Ops Manager are also available in the SaaS-based Cloud Manager, hosted by MongoDB in the cloud. Organizations that run on MongoDB Enterprise Advanced can choose between Ops Manager and Cloud Manager for their deployments.

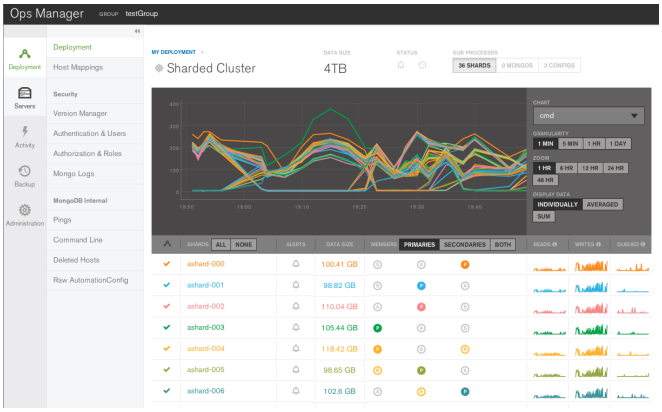

Figure 3: Ops Manager & Cloud Manager provides real time visibility into MongoDB performance.

From MongoDB 3.4, Ops Manager allows telemetry data to be collected every 10 seconds, up from the previous minimum 60 seconds interval.

In addition to monitoring, Ops Manager and Cloud Manager provide automated deployment, upgrades, on-line index builds, and cross-shard on-line backups.

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 can be used

to visualize output from the profiler when identifying slow queries.

Ops Manager and Cloud Manager include a Visual Query Profiler that 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 4) 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.

Figure 4: Visual Query Profiling in MongoDB Ops & Cloud Manager

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

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

Use mongoperf to characterize your storage system. `mongoperf` is a free, open-source tool that allows users to simulate direct disk I/O as well as memory mapped I/O, with configurable options for number of threads, size of documents, and other factors. This tool can help you to understand what sort of throughput is possible with your system, for disk-bound I/O as well as memory-mapped I/O.

Follow configuration best practices. Review the MongoDB production notes for the latest guidance on packages, hardware, networking, and operating system tuning.

MongoDB Atlas: Database as a Service For MongoDB

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

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

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

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

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

This paper is aimed at people managing their own MongoDB instances, performance best practices for MongoDB Atlas are described in a dedicated paper – MongoDB Atlas Best Practices.

We Can Help

We are the MongoDB experts. Over 2,000 organizations rely on our commercial products, including startups and more than a 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 Cloud Manager is a cloud-based tool that helps you manage MongoDB on your own infrastructure. With automated provisioning, fine-grained monitoring, and continuous backups, you get a full management suite that reduces operational overhead, while maintaining full control over your databases.

MongoDB Professional helps you manage your deployment and keep it running smoothly. It includes support from MongoDB engineers, as well as access to MongoDB Cloud Manager.

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

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

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

Resources

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

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

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