Performance Best Practices for MongoDB

June 2018

Table of Contents

Introduction

MongoDB is designed to meet the demands of modern apps with a technology foundation that enables you through:

1. The document data model – presenting you **the best way to work with data**.
2. A distributed systems design – allowing you to **intelligently put data where you want it**.
3. A unified experience that gives you the **freedom to run anywhere** – allowing you to future-proof your work and eliminate vendor lock-in.

This guide outlines considerations for achieving performance at scale in a MongoDB system across a number of key dimensions, including hardware, application patterns, including multi-document ACID transactions released in MongoDB 4.0, schema design, and indexing, disk I/O, Amazon EC2, and designing for benchmarks. While this guide is broad in scope, it is not exhaustive. You should refer to MongoDB documentation and consider the no cost, online training classes offered by MongoDB University. MongoDB also offers a range of consulting services to work with you at every stage of your application lifecycle.

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

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 exposes the storage engine API, enabling the integration of pluggable storage engines that extend MongoDB with new capabilities, and enable optimal use of specific hardware architectures to meet specific workload

requirements. 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**, which is provided for backwards compatibility only. This engine is deprecated with the MongoDB 4.0 release.

Any of these storage engines can coexist within a single MongoDB replica set, making it easy to evaluate and migrate between them. WiredTiger is the default storage engine for 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.

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 database 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 and encrypted storage engines. Compression reduces storage footprint by as much as 80%, and enables higher IOPs as fewer bits are read from disk. As with any compression algorithm, administrators trade storage efficiency for CPU overhead, and so it is important to test the impacts of compression in your own environment.

MongoDB offers administrators a range of compression options for 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, with the WiredTiger storage engine able to saturate multi-core processor resources.

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.

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.

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

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

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

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
- The number of documents read
- Which indexes were used

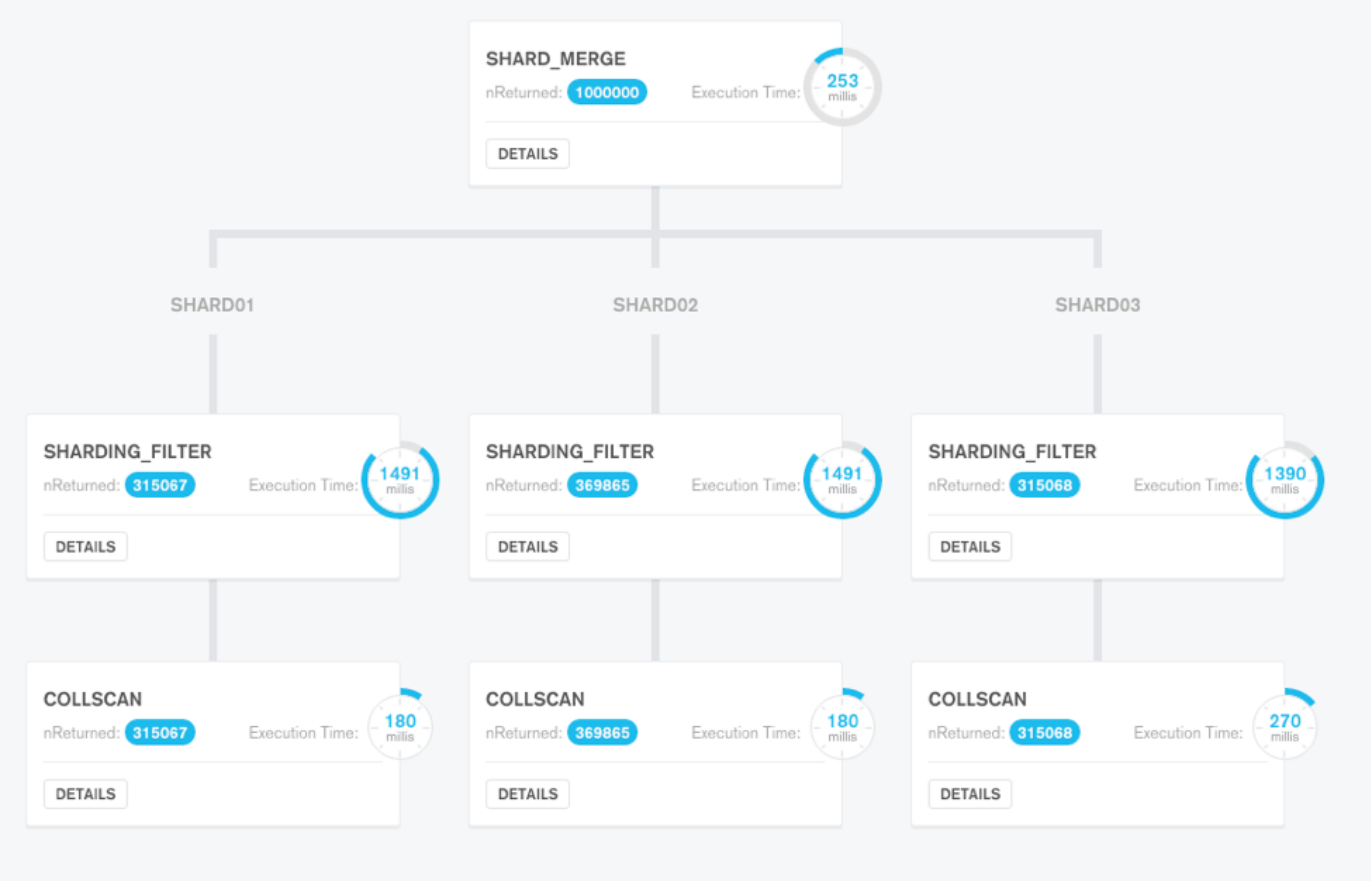

Figure 1: MongoDB Compass visual query plan for performance optimization across distributed clusters

- 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.

Update multiple array elements in a single operation. With fully expressive array updates, developers can perform complex array manipulations against matching elements of an array – including elements embedded in nested arrays – all in a single update operation. Using the `arrayFilters` option, the update can specify which elements to modify in the array field.

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.

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 supports a 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 causal consistency where needed. Introduced in MongoDB 3.6, causal consistency guarantees that every read operation within a client session will always see the previous write operation, regardless of which replica is serving the request. You can minimize any latency impact by using causal consistency only where it is needed.

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.

Multi-Document ACID Transactions

Because documents can bring together related data that would otherwise be modelled across separate parent-child tables in a tabular schema, MongoDB's atomic single-document operations provide transaction semantics that meet the data integrity needs of the majority of applications. One or more fields may be written in a single operation, including updates to multiple sub-documents and elements of an array. The guarantees provided by MongoDB ensure complete isolation as a document is updated; any errors cause the operation to roll back so that clients receive a consistent view of the document. MongoDB's existing document atomicity guarantees will meet 80-90% of an application's transactional needs. They remain the recommended way of enforcing your app's data integrity requirements

MongoDB 4.0 adds support for multi-document ACID transactions, making it even easier for developers to address more use cases with MongoDB. They feel just like the transactions developers are familiar with from relational databases – multi-statement, similar syntax, and easy to add to any application. Through snapshot isolation, transactions provide a consistent view of data, enforce all-or-nothing execution, and do not impact performance for workloads that do not require them. For those operations that do require multi-document transactions, there are several best practices that developers should observe.

Creating long running transactions, or attempting to perform an excessive number of operations in a single ACID transaction can result in high pressure on WiredTiger's cache. This is because the cache must maintain state for all subsequent writes since the oldest snapshot was created. As a transaction always uses the same snapshot while it is running, new writes accumulate in the cache throughout the duration of the transaction. These writes cannot be flushed until transactions currently running on old snapshots commit or abort, at which time the transactions release their locks and WiredTiger can evict the snapshot. To maintain predictable levels of database performance, developers should therefore consider the following:

1. By default, MongoDB will automatically abort any multi-document transaction that runs for more than 60 seconds. Note that if write volumes to the server are low, you have the flexibility to tune your transactions for a longer execution time. To address timeouts, the transaction should be broken into smaller parts that allow execution within the configured time limit. You should also ensure your query patterns are properly optimized with the appropriate index coverage to allow fast data access within the transaction.
2. There are no hard limits to the number of documents that can be read within a transaction. As a best practice, no more than 1,000 documents should be modified within a transaction. For operations that need to modify more than 1,000 documents, developers should break the transaction into separate parts that process documents in batches.
3. In MongoDB 4.0, a transaction is represented in a single oplog entry, therefore must be within the 16MB document size limit. While an update operation only stores the deltas of the update (i.e., what has changed), an insert will store the entire document. As a result, the combination of oplog descriptions for all statements in the transaction must be less than 16MB. If this limit is exceeded, the transaction will be aborted and fully rolled back. The transaction should therefore be decomposed into a smaller set of operations that can be represented in 16MB or less.
4. When a transaction aborts, an exception is returned to the driver and the transaction is fully rolled back. Developers should add application logic that can catch and retry a transaction that aborts due to temporary exceptions, such as a transient network failure or a primary replica election. With retryable writes, the MongoDB drivers will automatically retry the commit statement of the transaction.

You can review all best practices in the MongoDB documentation for multi-document transactions.

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 multi-document 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.

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 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.

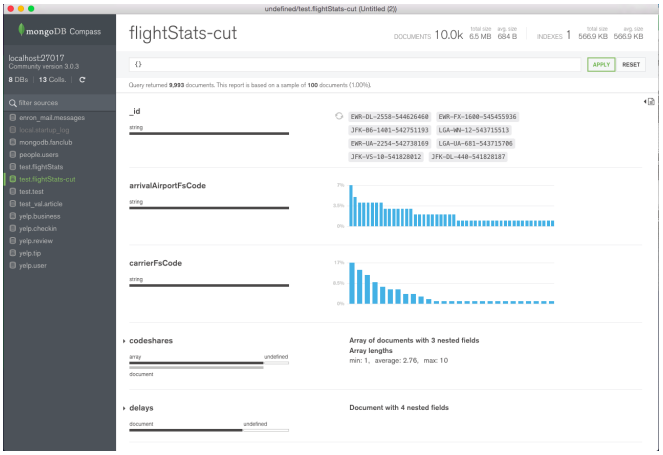

Figure 2: Document structure and contents exposed by MongoDB Compass

Ops Manager offers the Data Explorer to examine the database's schema by running queries to review document structure, viewing collection metadata, and inspecting index usage statistics.

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. Ops Manager includes a performance advisor which continuously highlights slow-running queries and provides intelligent index recommendations to improve performance. Using Ops Manager automation, the administrator can then roll out the recommended indexes automatically, without incurring any application downtime.

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 0 for WiredTiger. Use the `blockdev --setra <value>` command to set the `readahead` block size to 0 when using the WiredTiger storage engine. A `readahead` value of 32 (16 kB) typically works well when using MMAPv1.

If the `readahead` size is 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 which negatively effect performance:

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

Use XFS file systems; avoid EXT3. EXT3 is quite old and is not optimal for most database workloads. 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 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.

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.

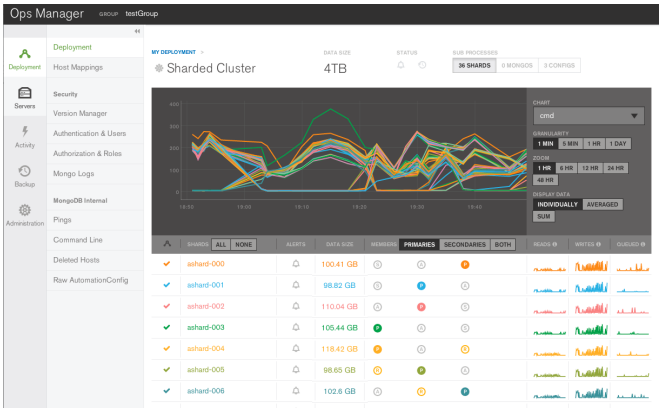

Figure 3: Ops Manager & Cloud Manager provides real time visibility into MongoDB performance.

In addition to monitoring, Ops Manager and Cloud Manager provide automated deployment, upgrades, on-line index builds, data exporation, 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.

Ops Manager also offers the performance advisor which continuously highlights slow-running queries and provides intelligent index recommendations to improve performance. Using Ops Manager automation, the administrator can then roll out the recommended indexes automatically, without incurring any application downtime.

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 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

An increasing number of companies are moving to the public cloud to not only reduce the operational overhead of

managing infrastructure, but also provide their teams with access to on-demand services that give them the agility they need to meet faster application development cycles. This move from building IT to consuming IT as a service is well aligned with parallel organizational shifts including agile and DevOps methodologies and microservices architectures. Collectively these seismic shifts in IT help companies prioritize developer agility, productivity and time to market.

MongoDB offers the fully managed, on-demand and elastic MongoDB Atlas service, in the public cloud. Atlas enables customers to deploy, operate, and scale MongoDB databases on AWS, Azure, or GCP in just a few clicks or programmatic API calls. MongoDB Atlas is available through a pay-as-you-go model and billed on an hourly basis. It's easy to get started – use a simple GUI to select the public cloud provider, region, instance size, and features you need. MongoDB Atlas provides:

- Automated database and infrastructure provisioning so teams can get the database resources they need, when they need them, and can elastically scale whenever they need to.
- Security features to protect your data, with network isolation, fine-grained access control, auditing, and end-to-end encryption, enabling you to comply with industry regulations such as HIPAA.
- Built in replication both within and across regions for always-on availability.
- Global clusters allows you to deploy a fully managed, globally distributed database that provides low latency, responsive reads and writes to users anywhere, with strong data placement controls for regulatory compliance.
- Fully managed, continuous and consistent backups with point in time recovery to protect against data corruption, and the ability to query backups in-place without full restores.
- Fine-grained monitoring and customizable alerts for comprehensive performance visibility.
- Automated patching and single-click upgrades for new major versions of the database, enabling you to take advantage of the latest and greatest MongoDB features.
- Live migration to move your self-managed MongoDB clusters into the Atlas service or to move Atlas clusters between cloud providers.
- Widespread coverage on the major cloud platforms with availability in over 50 cloud regions across Amazon Web Services, Microsoft Azure, and Google Cloud Platform. MongoDB Atlas delivers a consistent experience across each of the cloud platforms, ensuring developers can deploy wherever they need to, without compromising critical functionality or risking lock-in.

MongoDB Atlas can be used for everything from a quick Proof of Concept, to dev/test/QA environments, to powering production applications. The user experience across MongoDB Atlas, Cloud Manager, and Ops Manager is consistent, ensuring that you easily move from on-premises to the public cloud, and between providers as your needs evolve.

Built and run by the same team that engineers the database, MongoDB Atlas is the best way to run MongoDB in the cloud. Learn more or deploy a free cluster now.

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.

MongoDB Stitch

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

Stitch represents the next stage in the industry's migration to a more streamlined, managed infrastructure. Virtual Machines running in public clouds (notably AWS EC2) led the way, followed by hosted containers, and serverless offerings such as AWS Lambda and Google Cloud Functions. These still required backend developers to implement and manage access controls and REST APIs to provide access to microservices, public cloud services, and of course data. Frontend developers were held back by needing to work with APIs that weren't suited to rich data queries.

The Stitch serverless platform addresses these challenges by providing four services:

- **Stitch QueryAnywhere**. Brings MongoDB's rich query language safely to the edge. An intuitive SDK provides full access to your MongoDB database from mobile and IoT devices. Authentication and declarative or programmable access rules empower you to control precisely what data your users and devices can access.
- **Stitch Functions**. Stitch's HTTP service and webhooks let you create secure APIs or integrate with microservices and server-side logic. The same SDK that accesses your database, also connects you with popular cloud services, enriching your apps with a single method call. Your custom, hosted JavaScript functions bring everything together.
- **Stitch Triggers**. Real-time notifications let your application functions react in response to database changes, as they happen, without the need for wasteful, laggy polling.
- **Stitch Mobile Sync** (coming soon). Automatically synchronizes data between documents held locally in MongoDB Mobile and your backend database, helping resolve any conflicts – even after the mobile device has been offline.

Whether building a mobile, IoT, or web app from scratch, adding a new feature to an existing app, safely exposing your data to new users, or adding service integrations, Stitch can take the place of your application server and save you writing thousands of lines of boilerplate code.

We Can Help

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

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

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

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

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

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

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

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

Resources

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

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

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