A MongoDB White Paper

# Performance Considerations for MongoDB

MongoDB 2.4

October 2013

# Table of Contents

# Introduction

MongoDB is an open-source, high-performance, scalable, general purpose database. It is used by organizations of all sizes to power online applications where low latency and high availability are critical requirements of the system.

This guide outlines considerations for achieving performance at scale in a MongoDB system in a number of key areas, 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.

MongoDB works closely with users to help them optimize their systems. Users should monitor their systems to identify bottlenecks and limitations using tools such as the free **MongoDB Management Service** (MMS), which provides monitoring, visualization and alerts on over 100 system metrics. **MongoDB Subscribers** can optionally run MMS on-premise as part of their larger performance monitoring strategy.

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**.

# Hardware

MongoDB is designed for horizontal scale out. Rather than scaling up with ever larger and more expensive servers, users are instead encouraged to scale 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.** MongoDB performs best when the working set fits in RAM. Sufficient RAM 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 **serverStatus** command to view an estimate of the the current working set size.

**Use SSD for write-heavy applications.** Because most disk I/O in MongoDB is random, SSD can provide a significant performance improvement for write-heavy systems. Data access is dominated by seek time in disk subsystems, and for spinning disks seek times tend to be around ~5ms, while for SSD seek times tend to be around 0.1ms, around 50x faster than spinning disks. SSD can also mitigate the latencies associated with working sets that do not fit in RAM, however in general it is best to ensure your working set fits in RAM.

**Allocate CPU hardware budget for faster CPUs.** The benefits for faster CPUs outweigh those for many cores on a per-server basis in the current version of MongoDB.

**Dedicate each server to a single role in the system.** Each mongod server should manage a single replica rather than multiple replicas. For example, in a sharded system, each server should manage a primary or secondary replica rather than managing both primary and secondary replicas.

**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.

# 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 flexilitiy 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 as find and set operations.** Rather than retrieving the entire document in your application, updating fields, then saving the document back to the database, instead use find and set operators to issue the update to specific fields. This has the advantage of 1) less network usage, 2) fewer round trips, 3) fewer unnecessary index updates.

**Avoid negation in queries.** Like most database systems, MongoDB does not index the absence of values and negation conditions will require scanning all records in a result set. If negation is the only condition, all records will need to be scanned.

**Use explain() on every query in your application.** MongoDB provides the ability to view how a query will be evaluated in the system, including which indexes are used and whether the query is covered. This capability is similar to Explain Plan and similar features in relational databases. The **feedback from explain()** will help you understand whether your query is performing optimally.

**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 the index, and all the fields returned by the query must be present in the 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.

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

**Only read from primaries.** You may read from secondaries if your application can tolerate eventaual 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. To increase read capacity in your system **consider sharding**.

**Use the most recent drivers from MongoDB, Inc.** MongoDB supports drivers for 13 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 driver 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 operations, range-based sharding is beneficial because queries 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 data for a record contained within a single document. 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 your schema and indexes for your application.

**Store all data for a record in a single document.** MongoDB provides atomic operations 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 16MB. 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 documents, such as video, consider using **GridFS**, a convention implemented by all the drivers that stores the binary data across many smaller documents.

**Avoid unbounded document growth.** MongoDB allocates space for a document at creation time. If the document grows, MongoDB will rewrite the document in a larger space and update all index entries for the document. While this operation is supported and necessary in some cases, to the extent possible design your schema so that documents do not grow unbounded. If documents will need to grow over time, consider padding the document at creation time, or use the **powerOf2Sizes** option to minimize document rewrites.

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

**Use small 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.

**Use sparse indexes.** Sparse indexes will only maintain entries for documents that include the fields specified for the index. Fewer index entries will result in smaller indexes, and updates to documents are more efficient when there are fewer indexes to maintain. Using a sparse index will sometimes 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.

**Avoid indexes on low-cardinality fields.** Queries on fields with low cardinality will 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: they consume RAM, and as fields are updated their associated indexes must be maintained, including potentially significant disk I/O.

**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 used to filter queries that specify last name only. In this example an additional index on last name only is unnecessary; the compound index is sufficient for queries on last name as well as last name and first name.

**Avoid regular expressions.** 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.

# Disk I/O

While MongoDB performs all read and write operations through in-memory data structures, data is persisted to disk and 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 filesystem settings.

**Readahead size should be set to 32.** Set readahead to 32 or the size of most documents, whichever is larger. Most disk I/O in MongoDB is random. If the readahead size is much larger than the size of the data requested, a larger block will be read from disk. This has two undesirable consequences: 1) the size of the read will consume RAM unnecessarily, and 2) more time will be spent reading data than is necessary. Both will negatively affect the performance of your MongoDB system. Readahead should not be set lower than 32.

**Use EXT4 or XFS file systems; avoid EXT3.** EXT3 is quite old and is non-optimal for most database workloads. For example, MongoDB pre-allocates space for data. In EXT3 pre-allocation will actually write 0s to the disk to allocate the space, which is time consuming. In EXT4 and XFS pre-allocation is performed as a logical operation, which is much more efficient.

**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 hugepages.** Do not use hugepages virtual memory pages, MongoDB performs better with normal virtual memory pages.

**Use RAID10.** The performance of RAID5 is not ideal for MongoDB. RAID0 performs well but does not provide sufficient fault tolerance. RAID10 is the best compromise for performance and fault tolerance.

**Place journal, log and data files on different devices.** 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.

# Considerations for Amazon EC2

Amazon EC2 is an extremely popular environment for MongoDB. MongoDB has worked with Amazon to determine these best practices in order to ensure users have a great experience with MongoDB on EC2. The following considerations will help you get the best performance from **MongoDB on EC2**.

**Use MongoDB's AMIs.** MongoDB has worked with Amazon to pre-configure **AMIs that are optimized for EC2**. Use this AMI rather than installing the standard binaries to ensure the environment is optimized for MongoDB.

**Use provisioned IOPS and EBS-optimized instances.** Provisioned IOPS and EBS-optimized instances provide substantially better performance for MongoDB systems. Both are included in MongoDB's AMI.

# 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 into the appropriate shard. 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**.

**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. The MongoDB Blog provides a good overview of **how to use the tool**.

# Resources

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

| Resource | Website URL |
|---|---|
| MongoDB Enterprise Download | **mongodb.com/download** |
| Free Online Training | **education.mongodb.com** |
| Webinars and Events | **mongodb.com/events** |
| White Papers | **mongodb.com/white-papers** |
| Case Studies | **mongodb.com/customers** |
| Presentations | **mongodb.com/presentations** |
| Documentation | **docs.mongodb.org** |

US 866.237.8815 • INTL +1 650 440 4474 • info@mongodb.com