A MongoDB White Paper

MongoDB Architecture Guide

MongoDB 2.4

August 2013

mongoDB

Table of Contents

Introduction

"MongoDB wasn't designed in a lab. We built MongoDB from our own experiences building large-scale, high-availability, robust systems. We didn't start from scratch, we really tried to figure out what was broken, and tackle that. So the way I think about MongoDB is that if you take MySQL, and change the data model from relational to document-based, you get a lot of great features: embedded docs for speed, manageability, agile development with dynamic schemas, easier horizontal scalability because joins aren't as important. There are lots of things that work great in relational databases: indexes, dynamic queries and updates to name a few, and we haven't changed much there. For example, the way you design your indexes in MongoDB should be exactly the way you do it in MySQL or Oracle, you just have the option of indexing an embedded field."

— Eliot Horowitz, CTO and Co-founder

MongoDB is designed for how we build and run applications with modern development techniques, programming models, and computing resources.

HOW WE BUILD APPLICATIONS:

- **New and Complex Data Types.** Rich data structures with dynamic attributes, mixed structure, text, media, arrays and other complex types are common in today's applications.
- **Flexibility.** Applications have evolving data models, because certain attributes are initially unknown, and because applications evolve over time to accommodate new features and requirements.
- **Modern Programming Languages.** Object-oriented programming languages interact with data in structures that are dramatically different from the way data is stored in a relational database.
- **Faster Development.** Software engineering teams now embrace short, iterative development cycles. In these projects defining the data model and application functionality is a continuous process rather than a single event that happens at the beginning of the project.

HOW WE RUN APPLICATIONS:

- **New Scalability for Big Data.** Operational and analytical workloads challenge traditional capabilities on one or more dimensions of scale, availability, performance and cost effectiveness.
- **Fast, Real-time Performance.** Users expect consistent, interactive experiences from applications across many types of interfaces.
- **New Hardware.** The relationship between cost and performance for compute, storage, network and main memory resources has changed dramatically. Application designs can make different optimizations and trade offs to take advantage of these resources.
- **New Computing Environments.** The infrastructure requirements for applications can easily exceed the resources of a single computer, and cloud infrastructure now provides massive, elastic, cost-effective computing capacity on a metered cost model.

MONGODB EMBRACES THESE NEW REALITIES THROUGH KEY INNOVATIONS:

- **Document Data Model.** Data is stored in a structure that maps to objects in modern programming languages and is easy for developers to understand.
- **Rich Query Model.** MongoDB is fit for a wide variety of applications. It provides rich index and query support, including secondary, geospatial and text search indexes, the Aggregation Framework and native MapReduce.
- **Idiomatic Drivers.** Developers interact with the database through native libraries that are integrated with their respective environments and code repositories, making MongoDB simple and natural to use.
- **Horizontal Scalability.** As the data volume and throughput grow, developers can take advantage of commodity hardware and cloud infrastructure to increase the capacity of the MongoDB system.
- **High Availability.** Multiple copies of data are maintained with native replication. Automatic failover to secondary nodes, racks and data centers makes it possible to achieve enterprise-grade uptime without custom code and complicated tuning.
- **In-Memory Performance.** Data is read and written to RAM while also persisted to disk for durability, providing fast performance and eliminating the need for a separate caching layer.
- **Flexibility.** From the document data model, to multi-datacenter deployments, to tunable consistency, to operation-level availability options, MongoDB provides tremendous flexibility to the development and operations teams, and for these reasons it is well suited to a wide variety of applications across many industries.

MongoDB Data Model

DATA AS DOCUMENTS

MongoDB stores data as *documents* in a binary representation called **BSON** (Binary JSON). The BSON encoding extends the popular JSON (JavaScript Object Notation) representation to include additional types such as int, long, and floating point. BSON documents contain one or more *fields*, and each field contains a value of a specific data type, including arrays, binary data and sub-documents.

Documents that tend to share a similar structure are organized as *collections*. It may be helpful to think of collections as being analogous to a table in a relational database, documents as similar to rows, and fields as similar to columns.

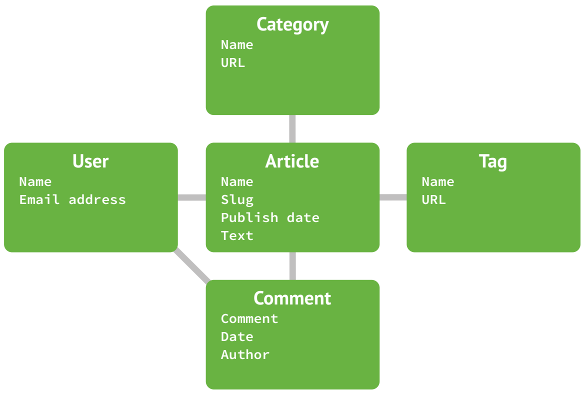

FIGURE 1 // Example relational data model for a blogging application.

For example, consider the data model for a blogging application. In a relational database the data model would comprise multiple tables. To simplify the example, assume there are tables for Categories, Tags, Users, Comments and Articles

In MongoDB the data could be modeled as two collections, one for users, and the other for articles. In each blog document there might be multiple comments, multiple tags, and multiple categories, each expressed as an embedded array.

MongoDB documents tend to have all data for a given record in a single document, whereas in a relational database information for a given record is usually spread across many tables. In other words, data in MongoDB tends to be more localized. In most MongoDB systems, BSON documents also tend to be closely aligned to the structure of objects in the programming language of the application, which makes it easy for developers to understand the realtionship of how the data used in the application maps to the data that is stored in the database.

DYNAMIC SCHEMA

MongoDB documents can vary in structure. For example, all documents that describe users might contain the user id and the last date they logged into the system, but only some of these documents might contain the user's identity for one or more third-party applications. Fields can vary from document to document; there is no need to declare the structure of documents to the system – documents are self-describing. If a new field needs to be added to a document then the field can be created without affecting all other documents in the system, without updating a central system catalog, and without taking the system offline.

SCHEMA DESIGN

Although MongoDB provides robust schema flexibility, schema design is still important. Schema designers should consider a number of topics, including the types of queries the application will need to perform, how objects are managed in the application code, and how documents will change and potentially grow over time. Schema design is an extensive topic that is beyond the scope of this document. For more information, please see **Data Modeling Considerations**.

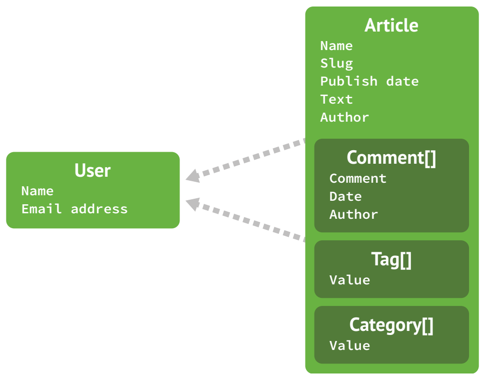

FIGURE 2 // Example document data model for a blogging application.

MongoDB Query Model

IDIOMATIC DRIVERS

MongoDB provides native drivers for all popular programming languages and frameworks to make development natural. Supported drivers include Java, .NET, Ruby, PHP, JavaScript, node.js, Python, Perl, PHP, Scala and others. MongoDB drivers are designed to be idiomatic for the given language. One fundamental difference as compared to relational databases is that the MongoDB query model is implemented as methods or functions within the API of a specific programming language, as opposed to a completely separate language like SQL. This, coupled with the affinity between MongoDB's JSON document model and the data structures used in object-oriented programming, makes integration with applications simple. For a complete list of drivers see the **MongoDB Drivers** page.

MONGO SHELL

The mongo shell is a rich, interactive JavaScript shell that is included with all MongoDB distributions. Nearly all commands supported by MongoDB can be issued through the shell, including administrative operations. The mongo shell is a popular way to interact with MongoDB for ad hoc operations. All examples in the MongoDB Manual are based on the shell. For more on the mongo shell, see the **corresponding page** in the MongoDB Manual.

QUERY TYPES

MongoDB supports many types of queries. A query may return a document or a subset of specific fields within the document:

- **Key-value queries** return results based on any field in the document, often the primary key.
- **Range queries** return results based on values defined as inequalities (e.g, greater than, less than or equal to, between).
- **Geospatial queries** return results based on proximity criteria, intersection and inclusion as specified by a point, line, circle or polygon.
- **Text Search queries** return results in relevance order based on text arguments using Boolean operators (e.g., AND, OR, NOT).
- **Aggregation Framework queries** return aggregations of values returned by the query (e.g., count, min, max, average, similar to a SQL GROUP BY statement).
- **MapReduce queries** execute complex data processing that is expressed in JavaScript and executed across data in the database.

MongoDB supports many types of queries. A query may return a document or a subset of specific fields within the document.

INDEXING

Like most database management systems, indexes are a crucial mechanism for optimizing system performance in MongoDB. And while indexes will improve the performance of some operations by orders of magnitude, they have associated costs in the form of slower writes, disk usage, and memory usage. MongoDB includes support for many types of indexes on any field in the document:

- **Unique Indexes.** By specifying an index as unique, MongoDB will reject inserts of new documents or the update of a document with an existing value for the field for which the unique index has been created. By default all indexes are not set as unique. If a compound index is specified as unique, the combination of values must be unique.
- **Compound Indexes.** It can be useful to create compound indexes for queries that specify multiple predicates. For example, consider an application that stores data about customers. The application may need to find customers based on last name, first name, and state of residence. With a compound index on last name, first name, and state of residence, queries could efficiently locate people with all three of these values specified. An additional benefit of a compound index is that any leading field within the index can be used, so fewer indexes on single fields may be necessary: this compound index would also optimize queries looking for customers by last name.
- **Array Indexes.** For fields that contain an array, each array value is stored as a separate index entry. For example, documents that describe recipes might include a field for ingredients. If there is an index on the ingredient field, each ingredient is indexed and queries on the ingredient field can be optimized by this index. There is no special syntax required for creating array indexes – if the field contains an array, it will be indexed as an array index.
- **TTL Indexes.** In some cases data should expire out of the system automatically. Time to Live (TTL) indexes allow the user to specify a period of time after which the data will automatically be deleted from the database. A common use of TTL indexes is applications that maintain a rolling window of history (e.g., most recent 100 days) for user actions such as clickstreams.
- **Geospatial Indexes.** MongoDB provides geospatial indexes to optimize queries related to location within a two dimensional space, such as projection systems for the earth. These indexes allow MongoDB to optimize queries for documents that contain points or a polygon that are closest to a given point or line; that are within a circle, rectangle, or polygon; or that intersect with a circle, rectangle, or polygon
- **Sparse Indexes.** Sparse indexes only contain entries for documents that contain the specified field. Because the document data model of MongoDB allows for flexibility in the data model from document to document, it is common for some fields to be present only in a subset of all documents. Sparse indexes allow for smaller, more

efficient indexes when fields are not present in all documents.

- **Text Search Indexes.** MongoDB provides a specialized index for text search that uses advanced, language-specific linguistic rules for stemming, tokenization and stop words. Queries that use the text search index will return documents in relevance order. One or more fields can be included in the text index.

QUERY OPTIMIZATION

MongoDB automatically optimizes queries to make evaluation as efficient as possible. Evaluation normally includes selecting data based on predicates, and sorting data based on the sort criteria provided. The query optimizer selects the best index to use by periodically running alternate query plans and selecting the index with the best response time for each query type. The results of this empirical test are stored as a cached query plan and are updated periodically.

COVERED QUERIES

Queries that return results containing only indexed fields are called covered queries. These results can be returned without reading from the source documents. With the appropriate indexes, workloads can be optimized to use predominantly covered queries.

MongoDB Data Management

IN-PLACE UPDATES

MongoDB stores each document on disk as a contiguous block. It allocates space for a document at insert time, and performs updates to documents in-place. By managing data in-place, MongoDB can perform discrete, field-level updates, thereby reducing disk I/O and updating only those index entries that need to be updated. Furthermore, MongoDB can manage disk space more efficiently than designs that manage updates using database compaction at runtime, which requires additional space and imposes a processing overhead, often yielding unpredictable performance.

AUTO-SHARDING

MongoDB provides horizontal scale-out for databases using a technique called *sharding*, which is transparent to applications. Sharding distributes data across multiple physical partitions called shards. Sharding allows MongoDB deployments to address the hardware limitations of a single server, such as bottlenecks in RAM or disk I/O, without adding complexity to the application.

MongoDB supports three types of sharding:

- **Range-based Sharding.** Documents are partitioned across shards according to the shard key value. Documents with shard key values “close” to one another are likely to be co-located on the same shard. This approach is well suited for applications that need to optimize range-based queries.
- **Hash-based Sharding.** Documents are uniformly distributed according to an MD5 hash of the shard key value. Documents with shard key values “close” to one another are unlikely to be co-located on the same shard. This approach guarantees a uniform distribution of writes across shards, but is less optimal for range-based queries.
- **Tag-aware Sharding.** Documents are partitioned according to a user-specified configuration that associates shard key ranges with shards. Users can optimize the physical location of documents for application requirements such as locating data in specific data centers.

MongoDB automatically balances the data in the cluster as the data grows or the size of the cluster increases or decreases. For more on sharding see the **Sharding Introduction**.

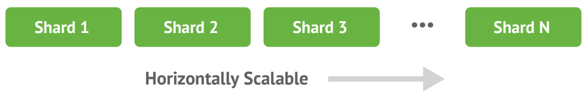

FIGURE 3 // Automatic sharding provides horizontal scalability in MongoDB.

QUERY ROUTER

Sharding is transparent to applications; whether there is one or one hundred shards, the application code for querying MongoDB is the same. Applications issue queries to a query router that dispatches the query to the appropriate shards.

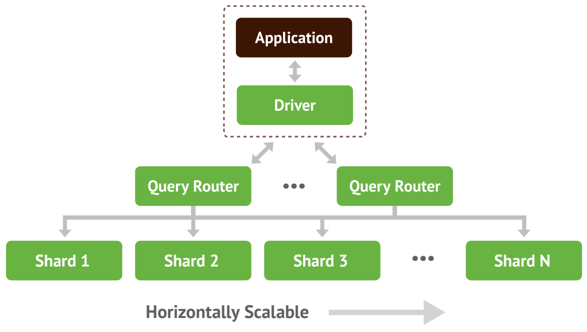

FIGURE 4 // Sharding is transparent to applications.

For key-value queries that are based on the shard key, the query router will dispatch the query to the shard that manages the document with the requested key. When using range-based sharding, queries that specify ranges on the shard key are only dispatched to shards that contain documents with values within the range. For queries that don't use the shard key, the query router will dispatch the query to all shards and aggregate and sort the results as appropriate. Multiple query routers can be used with a MongoDB system, and the appropriate number is determined based on performance and availability requirements of the application.

Consistency & Durability

TRANSACTION MODEL

MongoDB guarantees atomic updates to data at the document level. Because data for each record tends to exist in a single document, this level of granularity is sufficient for most applications. One or more fields may be updated in a single operation, including push operations and inserts into capped arrays.

MongoDB uses a readers-writer lock that allows concurrent reads by multiple readers but exclusive access to a single writer. When a read lock exists, many reads may share this lock. However, when a write lock exists, a single write operation holds the lock exclusively, and no other read or write operation may share the lock. Locks are managed at the shard level; there is no global coordination of locks across the entire cluster. For more on locks and lock yielding, see the entry on **Concurrency**.

JOURNALING

MongoDB implements write-ahead journaling of operations to enable fast crash recovery and durability in the storage engine. Journaling helps prevent corruption and increases operational resilience. Journal commits are issued at least as often as every 100ms by default. In the case of a server crash, journal entries are recovered automatically.

REPLICA SETS

MongoDB maintains multiple copies of data called replica sets using native replication. A replica set is a fully self-healing shard that helps prevent database downtime. Replica failover is fully automated, eliminating the need for administrators to intervene manually.

A replica set consists of multiple replicas. At any given time one member acts as the primary replica and the other members act as secondary replicas. MongoDB is

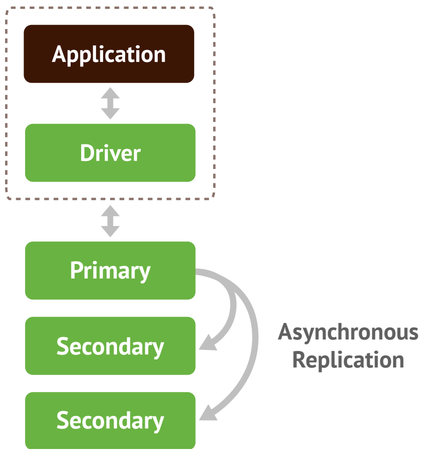

FIGURE 5 // A replica set is a fully self-healing shard. Data across replicas can be eventually consistent or strictly consistent.

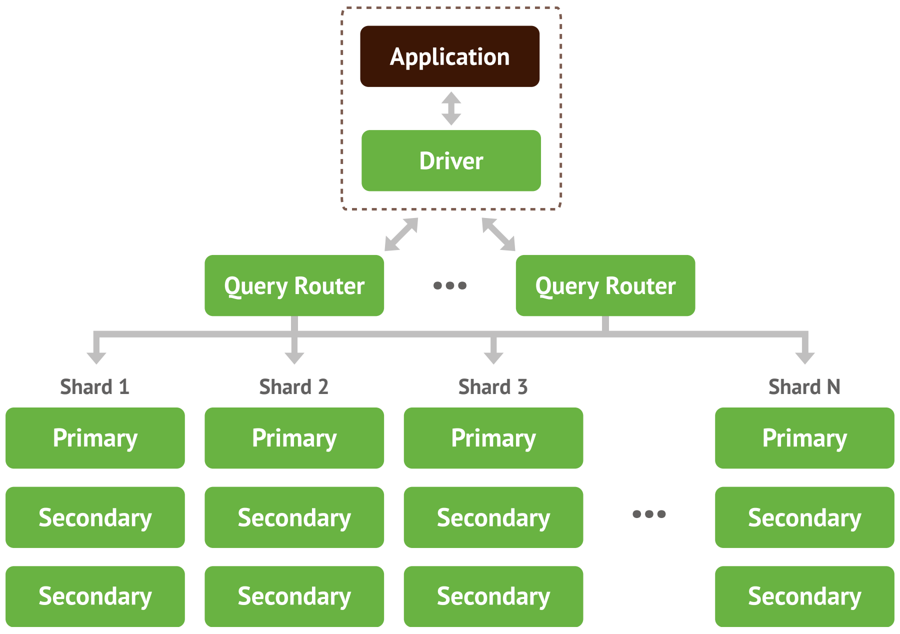

FIGURE 6 // Sharding and replica sets: automatic sharding provides horizontal scalability; replica sets help prevent database downtime.

consistent by default: reads and writes are issued to a primary copy of the data. If the primary member fails for any reason (e.g., overheating, hardware failure or network partition), one of the secondary members is automatically elected to primary and begins to process all writes.

The number of replicas in a MongoDB replica set is configurable, and a larger number of replicas provides increased data durability and protection against database downtime (e.g., in case of multiple machine failures, rack failures, data center failures, or network partitions). Optionally, operations can be configured to write to multiple replicas before returning to the application, thereby providing functionality that is similar to synchronous replication. For more on this topic, see the section on Configurable Availability of Writes later in this guide.

Applications can optionally read from secondary replicas, where data is eventually consistent by default. Reads from secondaries can be useful in scenarios where it is acceptable for data to be slightly out of date, such as some reporting applications. Applications can also read from the closest copy of the data as measured by ping distance when latency is more important than consistency. For more on reading from secondaries see the entry on **Read Preference**.

Replica sets also provide operational flexibility by providing a way to upgrade hardware and software without requiring the database to go offline. This is an important features as these types of operations can account for as much as one third of all downtime in traditional systems. For instance, if one needs to perform a hardware upgrade on all members of the replica set, one can sequentially upgrade each secondary without impacting the replica set. When all secondaries have been upgraded, one can temporarily demote the primary replica to secondary to upgrade that server. Similarly, other operational tasks, like adding indexes, can be carried out on replicas without interfering with uptime. For more on replica sets, see the entry on **Replication**.

Replica sets also provide operational flexibility by providing a way to upgrade hardware and software without requiring the database to go offline.

Availability

REPLICATION

Operations that modify a database on the primary are replicated to the secondaries with a log called the *oplog*. The oplog contains an ordered set of idempotent operations that are replayed on the secondaries. The size of the oplog is configurable and by default 5% of the available free disk space. For most applications, this size represents many hours of operations and defines the recovery window for a secondary should this replica go offline for some period of time and need to catch up to the primary.

If a secondary is down for a period longer than is maintained by the oplog, it must be recovered from the primary using a process called initial synchronization. During this process all databases and their collections are copied from the primary or another replica to the secondary as well as the oplog, then the indexes are built. Initial synchronization is also performed when adding a new member to a replica set. For more information see the page on **Replica Set Data Synchronization**.

ELECTIONS AND FAILOVER

Replica sets reduce operational overhead and improve system availability. If the primary replica for a shard fails, secondary replicas together determine which replica should become the new primary in a process called an *election*. Once the new primary has been determined, remaining secondaries are configured to receive updates from the new primary. If the original primary comes back online, it will recognize that it is no longer the primary and will configure itself to become a secondary. For more on failover elections see the entry on **Replica Set Elections**.

ELECTION PRIORITY

MongoDB considers a number of criteria when electing a new primary, and a configuration called the *election priority* allows users to influence their deployments to achieve certain operational goals. Every replica set member has a priority that determines its eligibility to become primary. In an election, the replica set elects an eligible member with the highest priority value as primary. By default, all members have a priority of 1 and have an equal chance of becoming primary; however, it is possible to set priority values that affect the likelihood of a replica becoming primary.

In some deployments, there may be operational requirements that can be addressed with election priorities. For instance, all replicas located in a secondary data center could be configured with a priority so that they would only become primary if the primary data center fails. Similarly, a replica can be configured to act as a backup by setting the priority so that it never becomes primary.

CONFIGURABLE WRITE AVAILABILITY

MongoDB allows users to specify write availability in the system, which is called the *write concern*. The default write concern acknowledges writes from the application, allowing the client to catch network exceptions and duplicate key errors. Each query can specify the appropriate write concern, ranging from unacknowledged to acknowledgement that writes have been committed to multiple replicas, a majority of replicas, or all replicas. It is also possible to configure the write concern so that writes are only acknowledged once specific policies have been fulfilled, such as writing to at least two replicas in one data center and at least one replica in a second data center. For more on configurable availability see the entry on **Write Concern**.

Performance

IN-MEMORY PERFORMANCE WITH ON-DISK CAPACITY

MongoDB makes extensive use of RAM to speed up database operations. Reading data from memory is measured in nanoseconds, whereas reading data from spinning disk is measured in milliseconds; reading from memory is approximately 100,000 times faster than reading data from disk. In MongoDB, all data is read and manipulated through memory-mapped files. Data that is not accessed is not loaded into RAM. While it is not required that all data fit in RAM, it should be the goal of the deployment team that indexes and all data that is frequently accessed should fit in RAM. For example it may be the case that a fraction of the entire database is most frequently accessed by the application, such as data related to recent events or popular products. If the volume of data that is frequently accessed exceeds the capacity of a single machine, MongoDB can scale horizontally across multiple servers using automatic sharding.

Because MongoDB provides in-memory performance, for most applications there is no need for a separate caching layer.

Conclusion

MongoDB provides a powerful, innovative database platform, architected for how we build and run applications today. In this guide we have explored the fundamental concepts and assumptions that underly the architecture of MongoDB. Other guides on topics such as Operations Best Practices can be found at **mongodb.com**.

About MongoDB

MongoDB (from *humongous*) is reinventing data management and powering big data as the leading NoSQL database. Designed for how we build and run applications today, it empowers organizations to be more agile and scalable. MongoDB enables new types of applications, better customer experience, faster time to market and lower costs. It has a thriving global community with over 4 million downloads, 100,000 online education registrations, 20,000 user group members and 20,000 MongoDB Days attendees. The company has more than 600 customers, including many of the world's largest organizations.

Resources

For more information, please visit **mongodb.com** 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**

mongoDB

Published by MongoDB, Inc. / 2013