Overview of MongoDB

Introduction

- A NO-SQL database
- Conceptualized to be a distributed database
- Easy to scale out
- Allows schema-free storage of data as collections
- Current version 3.2

source: MongoDB: The Definitive Guide

Installation

- Windows install
  - Download the Windows installer
  - Create a C:\data\db directory
  - bin\mongod.exe
    - --dbpath PATH to specify any other directory as the base directory
  - bin\mongo.exe
  - Can also be installed as a service

source: MongoDB: The Definitive Guide

Wire Protocol

- Clients communicate with the MongoDB server using a lightweight TCP/IP Wire protocol; default port is 27017
  - A regular TCP/IP Socket
- Two types of messages
  - Client request
  - Database responses

source: MongoDB: The Definitive Guide

Wire protocol

Standard Message Header

```
struct MsgHeader {
    int32   messageLength; // total message size, including this
    int32   requestID;     // identifier for this message
    int32   responseTo;    // requestID from the original request
                           //   (used in reponses from db)
    int32   opCode;        // request type - see table below
}
```

source: MongoDB: The Definitive Guide

Wire Protocol -opcodes

Opcode Name	opCode value	Comment
OP_REPLY	1	Reply to a client request. responseTo is set
OP_MSG	1000	generic msg command followed by a string
OP_UPDATE	2001	update document
OP_INSERT	2002	insert new document
RESERVED	2003	formerly used for OP_GET_BY_OID
OP_QUERY	2004	query a collection
OP_GET_MORE	2005	Get more data from a query. See Cursors
OP_DELETE	2006	Delete documents
OP_KILL_CURSORS	2007	Tell database client is done with a cursor

source: MongoDB: The Definitive Guide

Mongo Shell

- The shell can connect to daemon running on any other machine
  - bin/mongo www.server.com:port
  - By default starts with “test” databse
  - db variable points to the current database
    - Can use connect to connect with multiple databases
  - bin/mongo localhost:27017/admin – now connects to admin database
  - bin/mongo –nodb - now does not connect with any database

Binary JSON (BSON)

- Binary JSON (BSON): representation of documents that is shared by all drivers, tools, and processes in the MongoDB ecosystem.
- BSON is a lightweight binary format capable of representing any MongoDB document as a string of bytes.
- The database understands BSON, and BSON is the format in which documents are saved to disk.

source: MongoDB: The Definitive Guide

BSON

- **Lightweight**
  - Overhead is minimum
  - Good to use over the network.
- **Traversable**
  - BSON is designed to be traversed easily.
- **Efficient**
  - Encoding data to BSON and decoding from BSON can be performed very quickly in most languages due to the use of C data types.

source: MongoDB: The Definitive Guide

Documents & Collections

- A *document* is the basic unit of data for MongoDB, roughly equivalent to a row in a relational database management system

- A *collection* can be thought of as the schema-free equivalent of a table.

source: MongoDB: The Definitive Guide

Documents

- Documents: an ordered set of keys with associated values
  - Naturally fits with data structures like map, hash etc.
  - E.g. {“greeting” : “Hello”, “foo”: 3}

- Key/value pairs in documents are ordered
  - {“greeting” : “Hello”, “foo”: 3} & {“foo”: 3, “greeting” : “Hello”}

  - Values could be several different data types including embedded documents

source: MongoDB: The Definitive Guide

Keys in Documents

- The keys in a document are strings. Any UTF-8 character is allowed in a key, with a few notable exceptions:
  - Keys must not contain the character \0 (the null character). This character is used to signify the end of a key.
  - The . and $ characters have some special properties and should be used only in certain circumstances, as described in later chapters. In general, they should be considered reserved.
  - Keys starting with _ should be considered reserved;
- MongoDB is type-sensitive and case-sensitive. For example, these documents are distinct:
  - {"foo" : 3} & {"foo" : "3"}
  - {"foo" : 3}   & {"Foo" : 3}
- No duplicate keys
  - {"greeting" : "Hello, world!", "greeting" : "Hello, MongoDB!"}

source: MongoDB: The Definitive Guide

Example

```
{
  "_id" : ObjectId("54c955492b7c8eb21818bd09"),
  "address" : {
    "street" : "2 Avenue",
    "zipcode" : "10075",
    "building" : "1480",
    "coord" : [ -73.9557413, 40.7720266 ],
  },
  “area" : "Manhattan",
  "cuisine" : "Italian",
   "name" : "Vella",
  "restaurant_id" : "41704620"
}
```

source: https://docs.mongodb.org/getting-started/shell/introduction/

Collections

- A group of documents like tables are a group of rows
- Collections are schema-free: documents within a single collection can have any number of different “shapes.”
  - {"greeting" : "Hello, world!"}
  - {"foo" : 5}

source: MongoDB: The Definitive Guide

Collections

- Why do we need separate collections at all?
- Keeping different kinds of documents in the same collection can be a nightmare for developers and admins.
  - Each query should return document of a certain kind
- It is much faster to get a list of collections than to extract a list of the types in a collection.
- Grouping documents of the same kind together in the same collection allows for data locality
- Putting only documents of a single type into the same collection, we can index our collections more efficiently.

source: MongoDB: The Definitive Guide

Collections - Naming

- The empty string ("") is not a valid collection name.
- Collection names may not contain the character \0 (the null character.
- You should not create any collections that start with *system.* - a prefix reserved for system collections.
  - For example, the *system.users* collection contains the database's users, and the *system.namespaces* collection contains information about all of the database's collections.
- User-created collections should not contain the reserved character $ in the name.

source: MongoDB: The Definitive Guide

Databases

- MongoDB groups collections into databases
- A single instance of MongoDB can host several databases – each of which is completely independent
- Each database is stored in separate files on disk

source: MongoDB: The Definitive Guide

Databases - Naming

- The empty string ("") is not a valid database name.
- A database name cannot contain any of these characters: ' ' (a single space), ., $, /, \, or \0 (the null character).
- Database names should be all lowercase.
- Database names are limited to a maximum of 64 bytes.

source: MongoDB: The Definitive Guide

Reserved Databases

- *admin*
  - This is the “root” database, in terms of authentication. If a user is added to the *admin* database, the user automatically inherits permissions for all databases. There are also certain server-wide commands that can be run only from the *admin* database, such as listing all of the databases or shutting down the server.
- *local*
  - This database will never be replicated and can be used to store any collections that should be local to a single server
- *config*
  - When Mongo is being used in a sharded setup, the *config* database is used internally to store information about the shards.

source: MongoDB: The Definitive Guide

Basic Data Types

- Documents are “JSON-like”
  - JSON has only six data types – null, boolean, numeric, string, array, object
- MongoDB has several additional types
  - null, Boolean, string, regular expressions
  - number: default is 64 bit floating, use NumberInt or NumberLong for 4 byte or 8 byte Integer
    - {"x" : NumberInt("3")}
    - {"x" : NumberLong("3")}
  - date: {"x" : **new** Date()}
  - embedded documents: {"x" : {"foo" : "bar"}}
  - object id: a 12 byte ID for documents: {"x" : ObjectId()}
  - binary data and code {"x" : **function**() { */* ... */* }}

source: MongoDB: The Definitive Guide

Basic Data Types

- Arrays
  - {"things" : ["pie", 3.14]}
  - arrays can contain different data types as values
  - MongoDB “understands” their structure and knows how to reach inside of arrays to perform operations on their contents.
    - allows us to query on arrays and build indexes using their contents
    - e.g. MongoDB can query for all documents where 3.14 is an element of the "things" array
- Embedded Documents
  - Documents can be used as the *value* for a key.

```
{
    "name" : "John Doe",
    "address" : {
        "street" : "123 Park Street",
        "city" : "Anytown",
        "state" : "NY"
    }
}
```

  - Same functionality as Arrays

source: MongoDB: The Definitive Guide

_id

- Every document stored in MongoDB must have an "_id" key.
- The "_id" key’s value can be any type, but it defaults to an ObjectId.
- In a single collection, every document must have a unique value for "_id", which ensures that every document in a collection can be uniquely identified.

source: MongoDB: The Definitive Guide

ObjectIds

- ObjectId is the default type for "_id".
- The ObjectId class is designed to be lightweight, while still being easy to generate in a globally unique way across different machines.
  - supports distributed nature of MongoDB
- Uses 12 bytes of storage – displayed as 24 digits in hexadecimal

0 1 2 3	4 5 6	7 8	9 10 11
Timestamp	Machine	PID	Increment

- if not supplied the _id is generated at the client end

source: MongoDB: The Definitive Guide

Shell

- Shell can run Javascripts
  - can be used to set helper functions, variables etc.
  - mongorc.js is run by the shell at the startup
    - could we used to greet, set global variables, customize prompt etc.

source: MongoDB: The Definitive Guide

Creating Documents

- Inserting and Saving Documents
  - >db.foo.insert({“bar” : “baz”})
  - return nInserted specifying the number of documents inserted.
  - document should be less than 16 MB
    - >for(var i=0; i<100; i++){db.foo.insert({“foo”:”bar”, “b”:i})}

Creating Documents

- Bulk inserts

```
var bulk = db.foo.initializeUnorderedBulkOp();
bulk.insert( { item: "abc123", defaultQty: 100, status: "A", points: 100 } );
bulk.insert( { item: "ijk123", defaultQty: 200, status: "A", points: 200 } );
bulk.insert( { item: "mop123", defaultQty: 0, status: "P", points: 0 } );
bulk.execute();
```

  - allows multiple documents to be inserted at ones
  - only useful if you are inserting multiple documents into a single collection
    - you cannot use batch inserts to insert into multiple collections with a single request
  - Message size should be less than 48 MB

source: MongoDB: The Definitive Guide

Bulk Insert

- Bulk operations builder used to construct a list of write operations to perform in bulk for a single collection
- To initiate the builder

  db.collection.initializeOrderedBulkOp() db.collection.initializeUnorderedBulkOp() method.
- Ordered Operations
  - MongoDB executes the write operations in the list serially.
    - If an error occurs during the processing of one of the write operations, MongoDB will return without processing any remaining write operations in the list.
- Unordered Operations
  - MongoDB can execute in parallel, as well as in a nondeterministic order.
    - If an error occurs during the processing of one of the write operations, MongoDB will continue to process remaining write operations in the list.

source: MongoDB: The Definitive Guide

Removing Documents

- >db.foo.remove({})
  - removes all the documents in the foo collection
  - Collection is not removed and meta information also exists
- >db.foo.remove({“a”:”b”})
  - takes a query document as a parameter
  - only matching documents are removed
- drop(): to drop a collection or database
  - once dropped, all is deleted
  - very fast

source: MongoDB: The Definitive Guide

Updating Documents

- update() is used to change the document
- update parameters
  - an update conditions document to match the documents to update,
  - an update operations document to specify the modification to perform, and
  - an options document
- To change a field value, MongoDB provides update operators, such as $set to modify values.
  - >db.foo.update({“a”:”b”},{$set:{a:”c”}})
  - changes only one document

source: MongoDB: The Definitive Guide

Updating Documents

- Updating multiple documents
  - use the multi option
  - >db.foo.update({“a”:”b”},{$set:{a:”c”}},{multi:true})
- Updating a document is atomic
  - if two updates happen at the same time, whichever one reaches the server first will be applied, and then the next one will be applied
  - conflicting updates can safely be sent in rapid-fire succession without any documents being corrupted: the last update will “win.”

source: MongoDB: The Definitive Guide

Updating Documents

- $inc to update a key
  - {$inc: {“score”:50}}
  - {$inc: {“score”:500}}
- Array modifiers
  - $push adds elements to the end of a array
    - create a new array if array does not exist
  - $ne or $addToSet
  - $pop or $ pull to remove elements

source: MongoDB: The Definitive Guide

Replacing the document

- To replace the entire content of a document except for the _id field, pass an entirely new document as the second argument to update().
  - >db.foo.update({“a”:”c”}, {“b”:”d”})
- If no document matches the update query the update() methos does nothing
- {upsert:true}
  - either update or inserts

source: MongoDB: The Definitive Guide

Querying

- find()
  - matches every document in the collection and returns then
  - add key/value pair to restrict
  - > db.users.find({"age" : 27})
  - > db.users.find({"username" : "joe"})
  - > db.users.find({"username" : "joe", "age" : 27})

source: MongoDB: The Definitive Guide

Specifying which keys to return

- If you do not need all of the key/value pairs, a second argument can be passed specifying the keys
- > db.users.find({}, {"username" : 1, "email" : 1})
- > db.users.find({}, {"username" : 1, "_id" : 0})

Query Conditionals

- "$lt", "$lte", "$gt", and "$gte" are all comparison operators, corresponding to <, <=, >, and >=, respectively.
  - > db.users.find({"age" : {"$gte" : 18, "$lte" : 30}})
    - multiple conditionals can be put with a single key
  - > start = new Date("01/01/2007")
  - > db.users.find({"registered" : {"$lt" : start}})
- $ne not equal

source: MongoDB: The Definitive Guide

OR queries

- $in can be used to query for a variety of values for a single key.
- allows you to specify criteria of different types as well as values
  - > db.raffle.find({"ticket_no" : {"$in" : [725, 542, 390]}})
  - > db.users.find({"user_id" : {"$in" : [12345, "joe"]}})
- opposite of $in is $nin

source: MongoDB: The Definitive Guide

OR queries

- $or allows to query across different key/value pairs
  - > db.raffle.find({"$or" : [{"ticket_no" : 725}, {"winner" : true}]})
  - > db.raffle.find({"$or" : [{"ticket_no" : {"$in" : [725, 542, 390]}}, {"winner" : **true**}]})
- $not can be applied on top of any other criteria
  - > db.users.find({"id_num" : {"$mod" : [5, 1]}})

source: MongoDB: The Definitive Guide

Type specific queries

- null
  - Matches with itself but also matches “does not exist”

```
> db.c.find()
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
```

- > db.c.find({"y" : **null**})
- > db.c.find({"z" : null})
- > db.c.find({"z" : {"$in" : [null], "$exists" : true}})

source: MongoDB: The Definitive Guide

Regular Expressions

- Regular expressions are useful for flexible string matching.
  - For example, if we want to find all users with the name Joe or joe, we can use a regular expression to do caseinsensitive matching:
  - > db.users.find({"name" : /joe/i})
  - > db.users.find({"name" : /joey?/i})
- MongoDB uses the Perl Compatible Regular Expression (PCRE) library to match regular expressions
- Any regular expression syntax allowed by PCRE is allowed in MongoDB.

source: MongoDB: The Definitive Guide

Querying Arrays

- Querying for elements of an array is designed to behave the way querying for scalars
  - > db.food.insert({"fruit" : ["apple", "banana", "peach"]})
  - > db.food.find({"fruit" : "banana"})
    - Will match the document

- $all: allows to match a list of elements
  - > db.food.find({fruit : {$all : ["apple", "banana"]}})
    - order does not matter
  - How about
    - > db.food.find({"fruit" : ["apple", "banana", "peach"]})
    - > db.food.find({fruit : ["apple", "banana"]}})
    - > db.food.find({fruit : ["peach“, "apple", "banana"]}})

source: MongoDB: The Definitive Guide

Querying Arrays

- *key.index:* allowas query for a specific element of an array:
  - > db.food.find({"fruit.2" : "peach"})
  - Arrays are always 0-indexed
- $size: allows query for arrays of a given size
  - > db.food.find({"fruit" : {"$size" : 3}})
  - $size can not be combined with another conditional say for getting a range of sizes
- $slice can be used to return a subset of elements for an array key
  - > db.blog.posts.findOne(criteria, {"comments" : {"$slice" : 10}})
    - return 10 entries.
- range queries
  - > db.test.find({"x" : {"$gt" : 10, "$lt" : 20}})

  {"x" : 5}
  {"x" : 15}
  {"x" : 25}
  {"x" : [5, 25]}
  What will be the output?

Querying Embedded Documents

- Querying for the whole document: works identically to a normal query

  {
  "name" : {
  "first" : "Joe",
  "last" : "Schmoe"},
  "age" : 45
  }

  - > db.people.find({"name" : {"first" : "Joe", "last" : "Schmoe"}})
- Querying for a specific key or keys of the document
  - > db.people.find({"name.first" : "Joe", "name.last" : "Schmoe"})
  - better than querying the whole document

source: MongoDB: The Definitive Guide

Cursors

```
> for(i=0; i<100; i++) {
... db.collection.insert({x : i});
... }
> var cursor = db.collection.find();
```

- Cursors generally allow you to control a great deal about the eventual output of a query

```
> while (cursor.hasNext()) {
... obj = cursor.next();
... // do stuff
... }
```

source: MongoDB: The Definitive Guide

Limits, Skips, and Sorts

- > db.c.find().limit(3)
  - If there are fewer than three documents matching your query in the collection, only the number of matching documents will be returned
  - limit sets an upper limit, not a lower limit.
- > db.c.find().skip(3)

> db.stock.find({"desc" : "mp3"}).limit(50).skip(50).sort({"price" : -1})

  - This will skip the first three matching documents and return the rest of the matches.
  - If there are fewer than three documents in your collection, it will not return any documents.
- > db.c.find().sort({username : 1, age : -1})
  - sort takes an object: a set of key/value pairs where the keys are key names and the values are the sort directions.
  - Sort direction can be 1 (ascending) or −1 (descending).
  - If multiple keys are given, the results will be sorted in that order.
- > db.stock.find({"desc" : "mp3"}).limit(50).sort({"price" : -1})
- > db.stock.find({"desc" : "mp3"}).limit(50).skip(50).sort({"price" : -1})

source: MongoDB: The Definitive Guide

Aggregation

- Aggregations are operations that process data records and return computed results
- Aggregation operations in MongoDB use *collections* of documents as an input and return results in the form of one or more documents.
- Single Purpose Aggregation Operations
    - returning a count of matching documents, returning the distinct values for a field, and grouping data based on the values of a field etc.
- MongoDB also provides *map-reduce* operations to perform aggregation

source: MongoDB: The Definitive Guide

Single Purpose Aggregation Operations

- **Count:** MongoDB can return a count of the number of documents that match a query.
  - db.records.count()
  - db.records.count({ a: 1 })

- **Distinct:** The distinct operation takes a number of documents that match a query and returns all of the unique values for a field in the matching documents.

- **Group :** The *group* operation takes a number of documents that match a query, and then collects groups of documents based on the value of a field or fields.

source: MongoDB: The Definitive Guide

Indexing

- To support the efficient execution of queries in MongoDB.
  - Without indexes, MongoDB must perform a collection scan, i.e. scan every document in a collection, to select those documents that match the query statement.
- Indexes are special data structures that store a small portion of the collection’s data set in an easy to traverse form
- The index stores the value of a specific field or set of fields, ordered by the value of the field.
- The ordering of the index entries supports efficient equality matches and range-based query operations.
  - In addition, MongoDB can return sorted results by using the ordering in the index.
- MongoDB defines indexes at the *collection* level and supports indexes on any field or sub-field of the documents in a MongoDB collection.

source: MongoDB: The Definitive Guide

Index Creation

- db.collection.createIndex(keys, options)
- Keys : A document that contains the field and value pairs where the field is the index key and the value describes the type of index for that field.
  - For an ascending index on a field, specify a value of 1; for descending index, specify a value of -1.
  - The order of an index is important for supporting sort() operations using the index.
- Options: Optional. A document that contains a set of options that controls the creation of the index.
  - Unique, sparse, expireAfterSeconds, name, v…

Index Examples

- db.collection.createIndex({ orderDate: 1 })
  - an ascending index on the field orderDate.
- db.collection.createIndex({ orderDate: 1, zipcode: -1 })
- db.products.createIndex({ item: 1, quantity: -1 } , { name: "inventory" })
  - To view the name of an index, use the getIndexes() method.

source: MongoDB: The Definitive Guide

createIndex() Behaviours

- To add or change index options you must drop the index using the dropIndex() method and issue another createIndex() operation with the new options.
  - If you create an index with one set of options, and then issue the createIndex() method with the same index fields and different options without first dropping the index, createIndex() will not rebuild the existing index with the new options.
  - If you call multiple createIndex() methods with the same index specification at the same time, only the first operation will succeed, all other operations will have no effect.
- Non-background indexing operations will block all other operations on a database.
  - db.people.createIndex({ zipcode: 1}, {background: true})
- MongoDB will not create an index on a collection if the index entry for an existing document exceeds the Maximum Index Key Length.

source: MongoDB: The Definitive Guide

Index Types

- Default _id
  - All MongoDB collections have an index on the _id field that exists by default.
- Single Field
  - MongoDB supports the creation of user-defined ascending/descending indexes on a single field of a document.
- Compound Index
  - MongoDB also supports user-defined indexes on multiple fields

source: MongoDB: The Definitive Guide

Index Types

- Multikey Index
  - MongoDB uses multikey indexes to index the content stored in arrays.
  - If you index a field that holds an array value, MongoDB creates separate index entries for every element of the array.
  - Multikey indexes allow queries to select documents that contain arrays by matching on element or elements of the arrays.

source: MongoDB: The Definitive Guide

Other Index types

- Geospatial Index
- Text Index
- Hashed Index

source: MongoDB: The Definitive Guide

Index Properties

- TTL Indexes
  - The TTL index is used for TTL collections, which expire data after a period of time.
- Unique Indexes
  - A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.
- Sparse Indexes
  - A sparse index does not index documents that do not have the indexed field.

source: MongoDB: The Definitive Guide

TTL Index

- Special single-field indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time.
  - Data expiration is useful for certain types of information like machine generated event data, logs, and session information
  - db.eventlog.createIndex({ "lastModifiedDate": 1 }, { expireAfterSeconds: 3600 })
  - Expiration of Data
    - TTL indexes expire documents after the specified number of seconds has passed since the indexed field value; i.e. the expiration threshold is the indexed field value plus the specified number of seconds.
  - Restrictions
    - TTL indexes are a single-field indexes. Compound indexes do not support TTL and ignores the expireAfterSeconds option.
    - The _id field does not support TTL indexes

source: MongoDB: The Definitive Guide

Unique Indexes

- A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.
  - *db.members.createIndex({ "user_id": 1 }, { unique: true })*
- The unique constraint applies to separate documents in the collection.
- If a document does not have a value for the indexed field in a unique index, the index will store a null value for this document.

source: MongoDB: The Definitive Guide

Sparse Indexes

- Sparse indexes only contain entries for documents that have the indexed field, even if the index field contains a null value.
- The index skips over any document that is missing the indexed field.
- The index is “sparse” because it does not include all documents of a collection.
  - By contrast, non-sparse indexes contain all documents in a collection, storing null values for those documents that do not contain the indexed field.
  - *db.addresses.createIndex({ "xmpp_id": 1 }, { sparse: true })*
- Sparse *compound indexes* that only contain ascending/descending index keys will index a document as long as the document contains at least one of the keys.

source: MongoDB: The Definitive Guide

Sparse Indexes

```
{ "_id" : ObjectId("523b6e32fb408eea0eec2647"), "userid" : "newbie" }
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : "abby", "score" : 82 }
{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : "nina", "score" : 90 }
```

```
db.scores.createIndex( { score: 1 } , { sparse: true } )
```

```
db.scores.find().sort( { score: -1 } )
```

source: MongoDB: The Definitive Guide

Index Management

- To remove an index, use the *db.collection.dropIndex()* method
  - db.accounts.dropIndex({ "tax-id": 1 })
    - The operation returns a document with the status of the operation:
    - { "nIndexesWas" : 3, "ok" : 1 }
- To remove *all* indexes
  - db.collection.dropIndexes()
- To modify an Index
  - First drop the old index
  - Then create the new index
- To Rebuild Indexes
  - db.accounts.reIndex()
- To return a list of Indexex
  - db.collection.getIndexes()

source: MongoDB: The Definitive Guide

Index Management

- Return Query Plan with explain()
  - Use the db.collection.explain() or the cursor.explain() method in executionStats mode to return statistics about the query process
  - Gives details including the index used, the number of documents scanned, and the time the query takes to process in milliseconds.
- Control Index Use with hint()
  - To force MongoDB to use a particular index for a db.collection.find() operation, specify the index with the hint() method.

- *db.people.find({ name: "John Doe", zipcode: { $gt: "63000" } }).hint({ zipcode: 1 }).explain("executionStats")*

source: MongoDB: The Definitive Guide

Index Strategies

- Create a Single-Key Index if All Queries Use the Same, Single Key
- Create Compound Indexes to Support Several Different Queries
- If an ascending or a descending index is on a single field, the sort operation on the field can be in either direction.
    - If the query planner cannot obtain the sort order from an index, it will sort the results in memory.
- Ensure Indexes Fit in RAM
    - db.collection.totalIndexSize() - returns in bytes

source: MongoDB: The Definitive Guide

Indexing considerations

- Each index requires at least 8KB of data space.
- Adding an index has some negative performance impact for write operations. For collections with high write-to-read ratio, indexes are expensive since each insert must also update any indexes.
- Collections with high read-to-write ratio often benefit from additional indexes. Indexes do not affect un-indexed read operations.
- When active, each index consumes disk space and memory. This usage can be significant and should be tracked for capacity planning, especially for concerns over working set size.

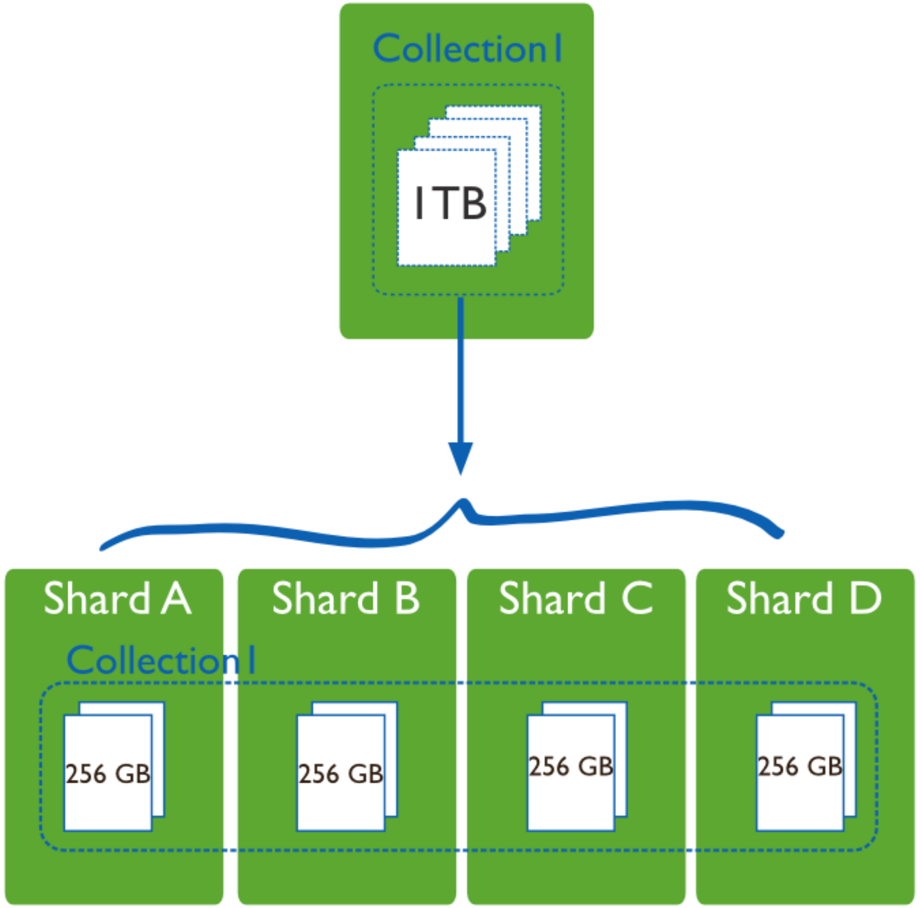
source: MongoDB: The Definitive Guide

Sharding

- Sharding is a method for storing data across multiple machines.
- Purpose of Sharding
  - Database systems with large data sets and high throughput applications can challenge the capacity of a single server.
  - High query rates can exhaust the CPU capacity of the server.
  - Larger data sets exceed the storage capacity of a single machine.
  - Working set sizes larger than the system’s RAM stress the I/O capacity of disk drives.
- Vertical Scaling: adds more CPU and storage resources to increase capacity.
- Horizontal Scaling: divides the data set and distributes the data over multiple servers, or shards

source: MongoDB: The Definitive Guide

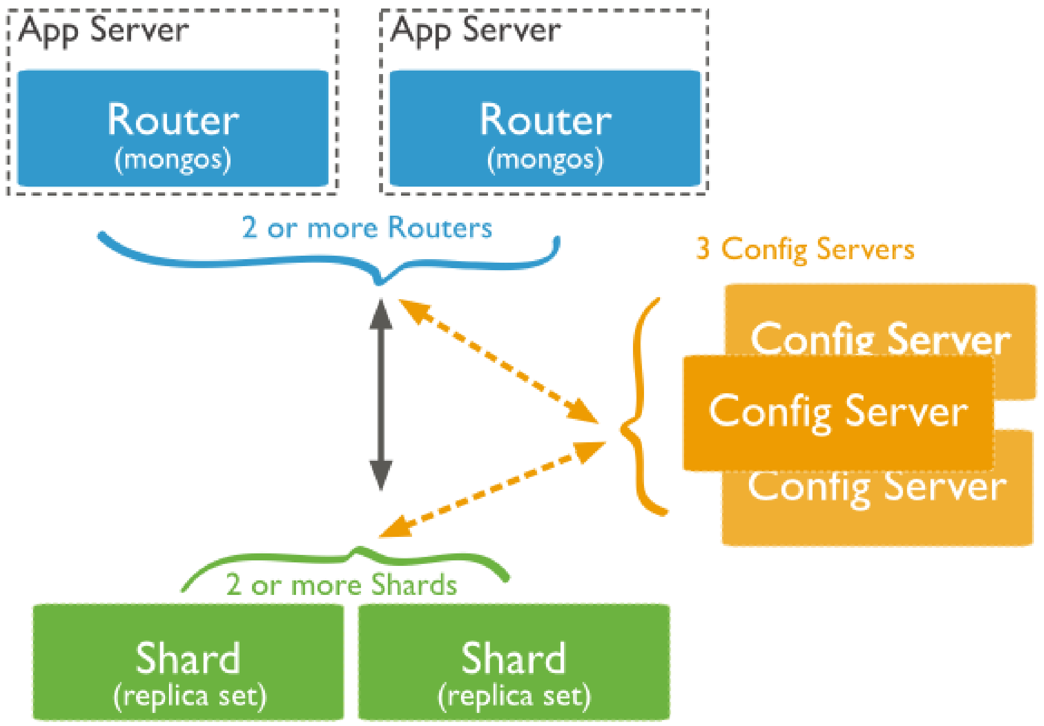
Sharding



- Sharding reduces the number of operations each shard handles. Each shard processes fewer operations as the cluster grows.
  - As a result, a cluster can increase capacity and throughput horizontally.
- Sharding reduces the amount of data that each server needs to store. Each shard stores less data as the cluster grows.

source: MongoDB: The Definitive Guide

Sharding in MongoDB



- Shards store the data. To provide high availability and data consistency, in a production sharded cluster, each shard is a replica set. For more information on replica sets, see Replica Sets.
- Query Routers, or mongos instances, interface with client applications and direct operations to the appropriate shard or shards.
  - The query router processes and targets operations to shards and then returns results to the clients.
  - A sharded cluster can contain more than one query router to divide the client request load.
  - A client sends requests to one query router. Most sharded clusters have many query routers.
- Config servers store the cluster's metadata.
  - The data contains a mapping of the cluster's data set to the shards.
  - The query router uses this metadata to target operations to specific shards. Production sharded clusters have exactly 3 config servers.

source: MongoDB: The Definitive Guide

Data Partitioning

- MongoDB distributes data, or shards, at the collection level. Sharding partitions a collection’s data by the shard key
- A shard key is either an indexed field or an indexed compound field that exists in every document in the collection.
- MongoDB divides the shard key values into chunks and distributes the chunks evenly across the shards
- Range Based Sharding: MongoDB divides the data set into ranges determined by the shard key values to provide range based partitioning
  - Range based partitioning supports more efficient range queries
- Hash Based Sharding: For hash based partitioning, MongoDB computes a hash of a field’s value, and then uses these hashes to create chunks.
  - With hash based partitioning, two documents with “close” shard key values are unlikely to be part of the same chunk. This ensures a more random distribution of a collection in the cluster.

source: MongoDB: The Definitive Guide