A MongoDB C100DBA Exam Study Guide

Information content is from study presentations and v3.2 Manual Documentation at mongodb.com. Study topics are based on MongoDB C100DBA Study Guide and MongoDB Architecture Guide, not actual exam questions.

Table of Contents

1.0 JSON What data types, how many?

MongoDB gives users the ease of use and flexibility of JSON documents together with the speed and richness of BSON, a lightweight binary format.

- JSON spec. is described here.
- Native data types(6): JSON: Number, String, Boolean, Array, Value, Object
- extended mongoDB data types (2): DATETME, GridFS
- The empty object “{}” and empty array “[]” are valid JSON documents.
- Validate JSON documents website.

1.1 BSON

BSON extends the JSON model to provide additional data types and to be efficient for encoding and decoding within different languages. MongoDB's BSON implementation supports embedding objects and arrays within other objects and arrays – MongoDB can even 'reach inside' BSON objects to build indexes and match objects against query expressions on both top-level and nested BSON keys.

- BSON is the binary equivalent of JSON documents. BSON spec is described at bsonspec.org.
- MongoDB uses extended BSON as described here.
- The maximum BSON document size is 16 megabytes. The maximum document size helps ensure that a single document cannot use excessive amount of RAM or, during transmission, excessive amount of bandwidth. To store documents larger than the maximum size, MongoDB provides the GridFS API.

2.0 CRUD (Create, Read, Update, Delete)

Create or insert operations add new *documents* to a *collection*. If the collection does not currently exist, insert operations will create the collection. All write operations in MongoDB are atomic on the level of a single document. See more on *Atomicity and Transactions*.

In MongoDB, documents stored in a collection require a unique *_id* field that acts as a *primary key*. If an _id does not exist in the document, MongoDB will create an ObjectID field. Generally, _id can itself be a complex document, any BSON data type (still, it must be unique). ObjectID(<hexadecimal>) is described here.

2.1 Document Insertion and Update

- Update(): If using Update Operators ($set, $unset, $push, $pop, $inc, $rename, $min, $max…etc.) then only related fields in the document are updated. If the update is composed of field:name arguments only, the entire document is replaced. Once set, you cannot update the value of the _id field nor can you replace an existing document with a replacement document with a different _id field value.
- Upsert(): If db.collection.update(), db.collection.updateOne(), db.collection.updateMany(), or db.collection.replaceOne() includes upsert : true **and** no documents match the specified filter criteria, then a single document is inserted. If both the <query> and <update> fields contain update operator expressions, the update creates a base document from the equality clauses in the <query> parameter and then applies the update expressions from the <update> parameter. If the document exists, an update() is perfomed according to update() behavior.
- findAndModify(): Modifies and returns a single document. By default, the returned document does not include the modifications made on the update. To return the document with the modifications made on the update, use the `new` option.
- Bulk Load, multi-document Insertion: insert as an array of documents. Example: db.collections.insert([{Doc1}, {Doc2}…]). upon completion, a BulkWriteResult() is returned.
- The shell helper **Save()** method uses either the `insert` or the `update` command. This method performs an update with upsert:true.

2.1.1 Document Validation

Starting with version 3.2, MongoDB provides the capability to validate documents during updates and insertions. Validation rules are specified on a per-collection basis using the `validator` option, which takes a document that specifies the validation rules or expressions. Specify the expressions using any *query operators*, with the exception of `$near`, `$nearSphere`, `$text`, and `$where`.

Example: db.createCollection(“contacts”, {validator: { $or: [phone: {type: “string”},email: {$regex:/@mongodb\.com}]}})

- Add document validation to an existing collection using the `collMod` command with the `validator` option. When you add validation to a collection, existing documents do not undergo validation checks until modification (update).
- MongoDB also provides the `validationLevel` option, which determines how strictly MongoDB applies validation rules to existing documents during an update, and the `validationAction` option, which determines whether MongoDB should error (reject) or warn.

2.2 Read

In MongoDB, you read documents using either the db.collection.find() method, or the db.collection.findAndModify() method. You should be familiar with both commands, but the .find() method will receive greater coverage on the exam.

`db.collection.find`(*query*, *projection*) selects documents in a collection and returns a *cursor* to the selected documents. *Query* is the selection filter, *projection* are the returned result data fields. A `projection` *cannot* contain *both* include and exclude specifications, except for the exclusion of the `_id` field. In projections that *explicitly include*

fields, the `_id` field is the only field that you can *explicitly exclude*. An AND condition is implied for multiple selection fields. Use $or operator for OR selections. Combined AND and OR:

```
db.users.find({status: "A", $or: [ { age: { $lt: 30 } }, { type: 1 } ]})
```

Array Query: If a field contains an array and your query has multiple conditional operators, the field as a whole will match if either a single array element meets the conditions or a combination of array elements meet the conditions. Use $elemMatch operator for range matching a single array element. Using an Array Index:

```
db.users.find( { 'points.0.points': { $lte: 55 } } )
```

- $in, $all are particular to Array queries and contain an array argument.

Embedded Document: if you use { name: { first: "Yukihiro", last: "Matsumoto" }} then Exact Matches Only. The query field must match the embedded document exactly. Use Dot notation for more general match: {"name.first": "Yukihiro", "name.last": "Matsumoto"} any array with both.

- Cursor methods: sort(), limit(), skip(). Cursor methods can be combined.
- you *cannot* project specific array elements using the array index; e.g. `{ "ratings.0": 1 }`
- The `{ name : null }` query matches documents that either contain the `name` field whose value is `null` *or* that do not contain the `name` field. Use the $type: 10 operator to return only null fields.
- LibPRC regex library.
- Dot notation allows you to query inside nested documents.

2.2.1 Cursors

A query returns a cursor. If you assign the cursor to a variable you can iterate through the cursor results. E.g.

```
var myCursor = db.users.find( { type: 2 } );

while (myCursor.hasNext()) {
  print(tojson(myCursor.next()));
}

Or,
myCursor.forEach(printjson);

Iterator Index using .toArray() method
var myCursor = db.inventory.find( { type: 2 } );
var documentArray = myCursor.toArray();
var myDocument = documentArray[3];
```

- By default, the server will automatically close the cursor after 10 minutes of inactivity, or if client has exhausted the cursor. The timeout value can be overridden.
- Sort, limit, skip methods operate as ordered in command
- cursor methods list

The db.serverStatus() method returns a document that includes a metrics field. The metrics field contains a metrics.cursor field with the following information:

- number of timed out cursors since the last server restart
- number of open cursors with the option DBQuery.Option.noTimeout set to prevent timeout after a period of inactivity
- number of “pinned” open cursors

- total number of open cursors

2.2.2 Explain Method

MongoDB provides the `db.collection.explain()` method, the `cursor.explain()` method, and the `explain` command to return information on query plans and execution statistics of the query plans. Find out which indexes are used. These queries use indexes: aggregate, find, count, remove, update, group, sort. Note: Explain Methods do not actually remove or update query data.

The mode affects the behavior of `explain()` and determines the amount of information to return. The possible modes are: `"queryPlanner"`, `"executionStats"`, and `"allPlansExecution"`. Default mode is `"queryPlanner"`

The `explain` results present the query plans as a tree of stages. Each stage passes its results (i.e. documents or index keys) to the parent node. The leaf nodes access the collection or the indices. The internal nodes manipulate the documents or the index keys that result from the child nodes. The root node is the final stage from which MongoDB derives the result set.

Stages are descriptive of the operation; e.g.

- COLLSCAN for a collection scan
- IXSCAN for scanning index keys
- FETCH for retrieving documents
- SHARD_MERGE for merging results from shards

2.2.3 explain() Mechanics

The db.collection.explain() method wraps the explain command and is the preferred way to run explain. db.collection.explain().find() is similar to db.collection.find().explain() with the following key differences:

- The db.collection.explain().find() construct allows for the additional chaining of query modifiers. For list of query modifiers, see *db.collection.explain().find().help()*.
- The db.collection.explain().find() returns a cursor, which requires a call to .next(), or its alias .finish(), to return the explain() results. If run interactively in the mongo shell, the mongo shell automatically calls .finish() to return the results. For scripts, however, you must explicitly call .next(), or .finish(), to return the results.

db.collection.explain().aggregate() is equivalent to passing the *explain* option to the db.collection.aggregate() method. For an explanation of explain results, see explain-results

2.3 Update

Save(): Shell helper function. Updates an existing *document* or inserts a new document, depending on its `document` parameter. The `save()` method uses either the `insert` or the `update` command. If the document does **not** contain an *_id* field, then the `save()` method calls the `insert()` method. If the document contains an *_id* field, then the `save()` method is equivalent to an update with the *upsert option* set to `true`.

`findAndModify()`: Modifies and returns a single document. By default, the returned document does not include the modifications made on the update. To return the document with the modifications made on the update, use the `new` option. The `findAndModify()` method is a shell helper around the `findAndModify` command. When using `findAndModify` in a *sharded* environment, the `query` **must** contain the *shard key* for all operations against the shard cluster for the *sharded* collections.

Update():Modifies an existing document or documents in a collection. The method can modify specific fields of an existing document or documents or replace an existing document entirely, depending on the *update parameter*. By default, the `update()` method updates a **single** document. Set the *Multi Parameter* to update all documents that match

the query criteria. When you execute an `update()` with `upsert: true` and the query matches no existing document, MongoDB will refuse to insert a new document if the query specifies conditions on the `_id` field using *dot notation*.
Update Methods

2.4 Delete

Drop a Collection: `db.collection.drop()`.Removes a collection from the database. The method also removes any indexes associated with the dropped collection. The method provides a wrapper around the `drop` command.

`Remove()`:Removes documents from a collection. By default, remove() removes all documents that match the query expression.

- Specify the justOne option to limit the operation to removing a single document. To delete a single document sorted by a specified order, use the *findAndModify()* method.
- You cannot use the remove() method with a *capped collection*.
- All remove() operations for a sharded collection that specify the justOne option must include the *shard key* or the _id field in the query specification.
- To remove all documents use db.collection.remove({}). However to remove all documents from a collection, it may be more efficient to use the drop() method to drop the entire collection, including the indexes, and then recreate the collection and rebuild the indexes.

2.5 Write Concern

Write concern describes the level of acknowledgement requested from MongoDB for write operations to a standalone `mongod` or to *replica sets* or to *sharded clusters*. In sharded clusters, `mongos` instances will pass the write concern on to the shards.

W: 1 (default) Requests acknowledgement that the write operation has propagated to the standalone `mongod` or the primary in a replica set.

W: 0 Requests no acknowledgment of the write operation. However, `w:0` may return information about socket exceptions and networking errors to the application.

W: “majority” Requests acknowledgment that write operations have propagated to the majority of voting nodes. implies *j: true*, if journaling is enabled. Journaling is enabled by default.

- Use “majority” whenever possible with wtimeout[ms]: {writeConcern:{w:"majority",wtimeout:5000}}

2.6 Capped Collections

Capped collections are fixed-size collections that support high-throughput operations that insert and retrieve documents based on insertion order. Capped collections work in a way similar to circular buffers: once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection.

- You cannot Shard a Capped Collection.
- You cannot delete documents from a capped collection.
- If an update or a replacement operation changes the document size, the operation will fail.
- Capped collections have an `_id` field and an index on the `_id` field by default.
- *TTL Collections* are not compatible with capped collections.

use `db.createCollection()` to create a *capped collection*, or to create a new collection that uses *document validation*. This command creates a capped collection named `log` with a maximum size of 5 megabytes and a maximum of 5000 documents:

```
db.createCollection("log", { capped : true, size : 5242880, max : 5000 } )
```

For a *capped collection*, a tailable cursor is a cursor that remains open after the client exhausts the results in the initial cursor. As clients insert new documents into the capped collection, the tailable cursor continues to retrieve documents.

2.7 mongoImport

The `mongoimport` tool imports content from an *Extended JSON*, CSV, or TSV export created by `mongoexport`, or potentially, another third-party export tool.

TSV: A text-based data format consisting of tab-separated values. This format is commonly used to exchange data between relational databases, since the format is well-suited to tabular data. You can import TSV files using mongoimport.

3.0 Indexes: Single, Compound, Multikey (array elements)

MongoDB defines indexes at the *collection* level and supports indexes on any field or sub-field of the documents in a MongoDB collection. MongoDB indexes use a B-tree data structure. See this reference to Btree. Indexes slow writes but vastly speed reads/queries.

3.1 Index Types

- **Default _id Index:** MongoDB creates a *unique index* on the *_id* field during the creation of a collection. The `_id` index prevents clients from inserting two documents with the same value for the `_id` field. You cannot drop this index on the `_id` field.
- **Single Field**: In addition to the MongoDB-defined _id index, MongoDB supports the creation of user-defined ascending/descending indexes on a *single field of a document*.
- **Embedded Field**: You can create indexes on fields within embedded documents, just as you can index top-level fields in documents. Indexes on embedded fields differ from *indexes on embedded documents*, which include the full content up to the maximum `index size` of the embedded document in the index. Instead, indexes on embedded fields allow you to use a "dot notation," to introspect into embedded documents.
- **Compound**: MongoDB also supports user-defined indexes on multiple fields, i.e. *compound indexes*.
    - **Order of fields** listed in a compound index has significance. For instance, if a compound index consists of { userid: 1, score: -1 }, the index sorts first by userid and then, within each userid value, sorts by score. Sort order can matter in determining whether the index can support a sort operation.
    - **Index Prefixes**: If a Compound Index is defined in the order of A,B,C then Index Prefixes are A and A,B. The Index can support queries on A only, or A,B or A,B,C, or A,C but not B only, or C only or B, C since without the A field, none of the listed fields correspond to a prefix index. If 2 Indexes are defined as A,B and A, then the A only field Index is redundant and will not be used.
- **Multikey**: 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. These *multikey indexes* allow queries (such as keyword search) to select documents that contain arrays by matching on element or elements of the arrays. MongoDB automatically determines whether to create a multikey index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.
- **Geospatial**: To support efficient queries of geospatial coordinate data, MongoDB provides two special indexes: *2d indexes* that uses planar geometry when returning results and *2sphere indexes* that use spherical geometry to return results. See *2d Index Internals* for a high level introduction to geospatial indexes.
- **Text Indexes**: MongoDB provides a text index type that supports searching for string content in a collection. These text indexes do not store language-specific *stop* words (e.g. "the", "a", "or") and *stem* the words in a collection to only store root words. See *Text Indexes* for more information on text indexes and search.
- **Hashed Indexes**: To support *hash based sharding*, MongoDB provides a *hashed index* type, which indexes the hash of the value of a field. These indexes have a more random distribution of values along their range, but *only* support equality matches and cannot support range-based queries.

3.2 Index Properties

- **Partial Indexes**: *Partial indexes* only index the documents in a collection that meet a specified filter expression. By indexing a subset of the documents in a collection, partial indexes have lower storage requirements and reduced performance costs for index creation and maintenance. Partial indexes offer a superset of the functionality of sparse indexes and should be preferred over sparse indexes.

  Example: db.restaurants.createIndex({ cuisine: 1, name: 1 }, { partialFilterExpression: { rating: { $gt: 5 } } })

  - MongoDB will not use the partial index for a query or sort operation if using the index results in an incomplete result set.
  - MongoDB cannot create multiple versions of an index that differ only in the options. As such, you cannot create multiple partial indexes that differ only by the filter expression.
  - Partial indexes only index the documents in a collection that meet a specified filter expression. If you specify both the `partialFilterExpression` and a *unique constraint*, the unique constraint only applies to the documents that meet the filter expression.
  - Shard key indexes cannot be partial indexes.
- **Sparse Indexes**: The *sparse* property of an index ensures that the index only contain entries for documents that have the indexed field. The index skips documents that *do not* have the indexed field. You can combine the sparse index option with the unique index option to reject documents that have duplicate values for a field but ignore documents that do not have the indexed key.
- *TTL indexes*: are special indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time. This is ideal for certain types of information like machine generated event data, logs, and session information that only need to persist in a database for a finite amount of time. See: *Expire Data from Collections by Setting TTL* for implementation instructions. On *replica set* members, the TTL background thread *only* deletes documents when a member is in state *primary*. *Secondary* members replicate deletion operations from the primary.

  ```
  db.eventlog.createIndex( { "lastModifiedDate": 1 }, { expireAfterSeconds: 3600 } )
  ```

- Background Index Creation**:** For potentially long running index building operations, consider the background operation so that the MongoDB database remains available during the index building operation. As of MongoDB version 2.4, a mongod instance can build more than one index in the background concurrently. The background index operation uses an incremental approach that is slower than the normal "foreground" index builds. If the index is larger than the available RAM, then the incremental process can take *much* longer than the foreground build. Queries will not use partially-built indexes: the index will only be usable once the index build is complete.

  ```
  db.people.createIndex( { zipcode: 1}, {background: true} )
  ```

  - Unique Index: To create a unique index, use the `db.collection.createIndex()` method with the `unique` option set to `true`. You can also enforce a **unique** constraint on *compound indexes*. If you use the unique constraint on a *compound index*, then MongoDB will enforce uniqueness on the *combination* of the index key values. You cannot specify a unique constraint on a *hashed index*.

    db.collection.createIndex(<key and index type specification>, { unique: true })

    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. Because of the unique constraint, MongoDB will only permit one document that lacks the indexed field.

- Text Indexes: support text search queries on string content. `text` indexes can include any field whose value is a string or an array of string elements. Case insensitive. Diacritic insensitive. Can also create a compound index

except other special index types multi-key, geospatial. If the compound `text` index includes keys **preceding** the `text` index key, to perform a `$text` search, the query predicate must include **equality match conditions** on the preceding keys. A collection can have at most **one** `text` index.

- **db.collection.createIndex({ textfield: "text"})** to create an Index on key textfield. Can also include wildcards for indexes on multiple text fields, e.g. createIndex({ te*tfield: "text"})
- **db.collection.find({text: {search: "trees cat" }}**, {score:{$meta: "test score"}}) to search text with scoring. Returns any document with tree, trees, cat, cats, tree cat, cat tree, cats trees, trees cats, etc. Supports text search for various languages, Ignores stop words (a, the, an, and, etc.)
- `text` indexes can be large. They contain one index entry for each unique post-stemmed word in each indexed field for each document inserted.
- Building a `text` index is very similar to building a large multi-key index and will take longer than building a simple ordered (scalar) index on the same data.
- When building a large `text` index on an existing collection, ensure that you have a sufficiently high limit on open file descriptors. See the *recommended settings*.
- `text` indexes will impact insertion throughput because MongoDB must add an index entry for each unique post-stemmed word in each indexed field of each new source document.
- Additionally, `text` indexes do not store phrases or information about the proximity of words in the documents. As a result, phrase queries will run much more effectively when the entire collection fits in RAM.
- For each indexed field in the document, MongoDB multiplies the number of matches by the weight and sums the results. Using this sum, MongoDB then calculates the score for the document. See `$meta` operator for details on returning and sorting by text scores.

3.3 Index Use

When developing your indexing strategy you should have a deep understanding of your application's queries. Before you build indexes, map out the types of queries you will run so that you can build indexes that reference those fields. Indexes come with a performance cost, but are more than worth the cost for frequent queries on large data set. Indexes support queries, update operations, sorts, and some phases of the *aggregation pipeline*.

- **Index Intersection**: MongoDB can use the *intersection of indexes* to fulfill queries. For queries that specify compound query conditions, if one index can fulfill a part of a query condition, and another index can fulfill another part of the query condition, then MongoDB can use the intersection of the two indexes to fulfill the query. Whether the use of a compound index or the use of an index intersection is more efficient depends on the particular query and the system.
- **Background/Foreground Operation**: By default, creating an index blocks all other operations on a database. When building an index on a collection, the database that holds the collection is unavailable for read or write operations until the index build completes. For potentially long running index building operations, consider the background operation so that the MongoDB database remains available during the index building operation.
- **Build Indexes on Replica Sets**: For replica sets, secondaries will begin building indexes *after* the *primary* finishes building the index. In *sharded clusters*, the mongos will send createIndex() to the primary members of the replica set for each shard, which then replicate to the secondaries after the primary finishes building the index.
- **Covered Queries**: When the query criteria and the *projection* of a query include *only* the indexed fields, MongoDB will return results directly from the index *without* scanning any documents or bringing documents into memory. These covered queries can be *very* efficient. To determine whether a query is a covered query, use the `db.collection.explain()` or the `explain()` method and review the *results*.

  An index cannot cover a query if:

  - any of the indexed fields in any of the documents in the collection includes an array. If an indexed field is an array, the index becomes a *multi-key index* and cannot support a covered query.

- any of the indexed fields in the query predicate or returned in the projection are fields in embedded documents.
- **Restrictions on Sharded Collection**: An index cannot cover a query on a *sharded* collection when run against a `mongos` if the index does not contain the shard key, with the following exception for the `_id` index: If a query on a sharded collection only specifies a condition on the `_id` field and returns only the `_id` field, the `_id` index can cover the query when run against a `mongos` even if the `_id` field is not the shard key.
- To determine whether a query is a covered query, use the `db.collection.explain()` or the `explain()` method and review the *results*.

Generally, MongoDB only uses *one* index to fulfill most queries. However, each clause of an `$or` query may use a different index, and starting in 2.6, MongoDB can use an *intersection* of multiple indexes.

You can force MongoDB to use a specific index using the `hint()` method.

3.4 Index Restrictions

- **Index Key Limit**: The *total size* of an index entry, which can include structural overhead depending on the BSON type, must be *less than* 1024 bytes. You cannot insert, update, mongodump etc. an index exceeding the index key limit.
- For a *compound* multikey index, each indexed document can have *at most* one indexed field whose value is an array. As such, you cannot create a compound multikey index if more than one to-be-indexed field of a document is an array. Or, if a compound multikey index already exists, you cannot insert a document that would violate this restriction.
- A single collection can have *no more* than 64 indexes.
- Fully qualified index names, which includes the namespace and the dot separators (i.e. `<database name>.<collection name>.$<index name>`), cannot be longer than 128 characters.
- There can be no more than 31 fields in a compound index.
- A *multikey index* cannot support a *covered query*.
- Other restrictions apply to Geospatial and 2dsphere indexes.

4.0 Data Modeling

The key challenge in data modeling is balancing the needs of the application, the performance characteristics of the database engine, and the data retrieval patterns. When designing data models, always consider the application usage of the data (i.e. queries, updates, and processing of the data) as well as the inherent structure of the data itself.

There are two tools that allow applications to represent these relationships: *references* and *embedded documents*.

- References store the relationships between data by including links or *references* from one document to another. Applications can resolve these *references* to access the related data. Broadly, these are *normalized* data models.
- Embedded documents capture relationships between data by storing related data in a single document structure. MongoDB documents make it possible to embed document structures in a field or array within a document. These *denormalized* data models allow applications to retrieve and manipulate related data in a single database operation.

4.1 Document Growth

Some updates, such as pushing elements to an array or adding new fields, increase a *document's* size. For the MMAPv1 storage engine, if the document size exceeds the allocated space for that document, MongoDB relocates the document on disk. When using the MMAPv1 storage engine, growth consideration can affect the decision to normalize or denormalize data. See *Document Growth Considerations*.

4.2 Data Use and Performance

When designing a data model, consider how applications will use your database. For instance, if your application only uses recently inserted documents, consider using *Capped Collections*. Or if your application needs are mainly read operations to a collection, adding indexes to support common queries can improve performance. See *Operational Factors and Data Models*.

4.3 Working Set

The *working set* is the portion of your data that clients access most often. This includes Documents and Indexes. working set should stay in memory to achieve good performance. Otherwise many random disk IO's will occur, and unless you are using SSD, this can be quite slow. Solid state drives (SSDs) can outperform spinning hard disks (HDDs) by 100 times or more for random workloads.

One area to watch specifically in managing the size of your working set is index access patterns. If you are inserting into indexes at random locations (as would happen with id's that are randomly generated by hashes), you will continually be updating the whole index. If instead you are able to create your id's in approximately ascending order (for example, day concatenated with a random id), all the updates will occur at the right side of the b-tree and the working set size for index pages will be much smaller.

The default WiredTiger cache size value assumes that there is a single `mongod` instance per machine. If a single machine contains multiple MongoDB instances, then you should decrease the setting to accommodate the other `mongod` instances.

4.4 Modeling Tree Structures

Use tree data structures to model large hierarchical or nested data relationships. Multiple queries may be required to retrieve subtrees. Parent, Child, Ancestor, Paths, Nested Set references can be used to describe a hierarchy.

4.5 Data Storage Engines

The *storage engine* is the component of the database that is responsible for managing how data is stored, both in memory and on disk. MongoDB supports multiple storage engines, as different engines perform better for specific workloads.

- *WiredTiger* is the default storage engine starting in MongoDB 3.2. It is well-suited for most workloads and is recommended for new deployments. WiredTiger provides a document-level concurrency model, checkpointing, and compression, among other features.
- *MMAPv1* is the original MongoDB storage engine and is the default storage engine for MongoDB versions before 3.2. It performs well on workloads with high volumes of reads and writes, as well as in-place updates.
- For the MMAPv1 storage engine, if an update operation causes a document to exceed the currently allocated *record size*, MongoDB relocates the document on disk with enough contiguous space to hold the document. Updates that require relocations take longer than updates that do not, particularly if the collection has indexes.
- The *In-Memory Storage Engine* is available in MongoDB Enterprise. Rather than storing documents on-disk, it retains them in-memory for more predictable data latencies.

5.0 Aggregation

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. MongoDB provides three ways to perform aggregation: the *aggregation pipeline*, the *map-reduce function*, and *single purpose aggregation methods*.

- **Aggregation Pipeline:** MongoDB's *aggregation framework* is modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated result. The

most basic pipeline stages provide *filters* that operate like queries and *document transformations* that modify the form of the output document. Can operate on a *sharded collection*. Can use indexes to improve its performance during some of its pipeline stages. In addition, the aggregation pipeline has an internal optimization phase. See *Pipeline Operators and Indexes* and *Aggregation Pipeline Optimization* for details.

  - **Result Size Restrictions:** 16MB. When returning a cursor or storing the results in a collection, each document in the result set is subject to the `BSON Document Size` limit, currently 16 megabytes; if any single document that exceeds the `BSON Document Size` limit, the command will produce an error. The limit only applies to the returned documents; during the pipeline processing, the documents may exceed this size.
  - **Memory Restrictions:** Pipeline stages have a limit of 100 megabytes of RAM. If a stage exceeds this limit, MongoDB will produce an error. To allow for the handling of large datasets, use the `allowDiskUse` option to enable aggregation pipeline stages to write data to temporary files.
  - In the `db.collection.aggregate` method, pipeline stages appear in an array. Documents pass through the stages in sequence.
  - **Pipeline Operators and Indexes:** The $match and $sort pipeline operators can take advantage of an index when they occur at the **beginning** of the pipeline.

- **Map-Reduce:** MongoDB also provides *map-reduce* operations to perform aggregation. In general, map-reduce operations have two phases: a *map* stage that processes each document and *emits* one or more objects for each input document, and *reduce* phase that combines the output of the map operation.

Single Purpose Aggregation Operations: MongoDB also provides `db.collection.count()`, `db.collection.group()`, `db.collection.distinct()`. special purpose database commands. All of these operations aggregate documents from a single collection. While these operations provide simple access to common aggregation processes, they lack the flexibility and capabilities of the aggregation pipeline and map-reduce.

Operator expressions are similar to functions that take arguments. In general, these expressions take an array of arguments

Boolean expressions evaluate their argument expressions as booleans and return a boolean as the result.

`$not`	Returns the boolean value that is the opposite of its argument expression. Accepts a single argument expression.

Aggregation Quick Reference

Aggregation Pipeline and Sharded Collections

6.0 Replication: PRIMARY, SECONDARY, ARBITER

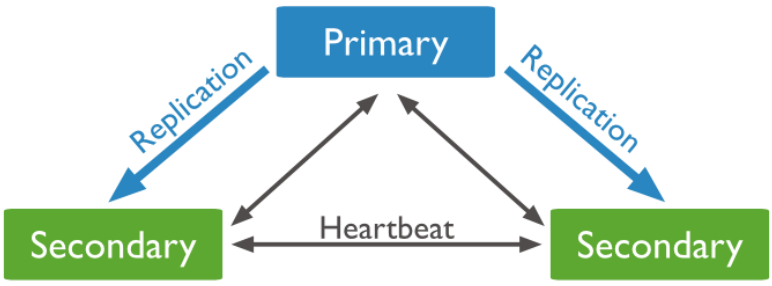

Figure 1: Replica Set

A *replica set* in MongoDB is a group of mongod processes that maintain the same data set on different servers thereby providing redundancy and high availability. The members of a replica set are:

- Primary. The primary receives all write operations. MongoDB applies write operations on the *primary* and then records the operations on the primary's *oplog*. *Secondary* members replicate this log and apply the operations to their data sets.
- *Secondaries* replicate operations from the primary to maintain an identical data set. Secondaries may have additional configurations for special usage profiles. For example, secondaries may be *non-voting* or *priority 0*.
- You can also maintain an *arbiter* as part of a replica set. Arbiters do not keep a copy of the data. However, arbiters play a role in the elections that select a primary if the current primary is unavailable. Arbiters can resolve elections in cases of an even number of replica set members and unexpected network partitions.

The minimum requirements for a replica set are: A *primary*, a *secondary*, and an *arbiter*. Most deployments, however, will keep three members that store data: A *primary* and two *secondary members*.

6.1. Write Concern

Write concern describes the level of acknowledgement requested from MongoDB for write operations to a standalone mongod or to *replica sets* or to *sharded clusters*. In sharded clusters, mongos instances will pass the write concern on to the shards.

- 0 = Requests no acknowledgment of the write operation. However, w: 0 may return information about:
    - socket exceptions and networking errors to the application.
    - If you specify w: 0 but include j: true, the j: true prevails to request acknowledgement from the standalone mongod or the primary of a replica set.
- 1 = Primary only (default) Requests acknowledgement that the write operation has propagated to the
    - standalone mongod or the primary in a replica set.
- "majority" = Requests acknowledgment that write operations have propagated to the majority of voting nodes [1],
    - including the primary, and have been written to the on-disk journal for these nodes.
- <tag> = Requests acknowledgement that the write operations have propagated to a replica set member with the specified tag.
- J option = Journal Option. Requests acknowledgement that the mongod instances, as specified in the w: <value>,
- have written to the on-disk journal. j: true does not by itself guarantee that the write will not be rolled
- back due to replica set primary failover.
- w option = wtimeout. This option specifies a time limit, in milliseconds, for the write concern.
- wtimeout is only applicable for w values greater than 1.

6.2 Read Preference, Example: db.collection.find().readPref({mode: 'nearest', tags: [{'dc': 'east'}]})

- primary = (Default). All operations read from the current replica set primary.
- primaryPreferred = In most situations, operations read from the primary but if it is unavailable, operations read from secondary members.
- secondary = All operations read from the secondary members of the replica set.
- secondaryPreferred = read from secondary members but if no secondary members are available, operations read from the primary.
- nearest = read from member of the replica set with the least network latency, irrespective of the member's type (P/S).

- The following Read Preference may return “stale” data: **primaryPreferred, secondary, secondaryPreferred, nearest**. i.e. only **primary** Read Preference ensures “fresh” data.

6.3 Read Concern

- "local" = Default. The query returns the instance’s most recent copy of data. Provides no guarantee that the data has been written to a majority of the replica set members.
- "majority" = The query returns the instance’s most recent copy of data confirmed as written to a majority of members in the replica set.
    - Note: To use a read concern level of "majority", you must use the WiredTiger storage engine and start the mongod instances with the --enableMajorityReadConcern command line option(or the replication.enableMajorityReadConcern setting if using a configuration file).

    - To ensure that a single thread can read its own writes, use "majority" read concern and "majority" write concern against the primary of the replica set.

6.4 Elections/Voting

Replica sets use elections to determine which set member will become *primary*. Elections occur after initiating a replica set, and also any time the primary becomes unavailable. The primary is the only member in the set that can accept write operations. If a primary becomes unavailable, elections allow the set to recover normal operations without manual intervention. Elections are part of the *failover process*.

When a primary does not communicate with the other members of the set for more than 10 seconds, an eligible secondary will hold an election to elect itself the new primary. The first secondary to hold an election and receive a majority of the members’ votes becomes primary. A member’s priority affects both the timing and the outcome of elections; secondaries with higher priority call elections relatively sooner than secondaries with lower priority, and are also more likely to win. Replica set members continue to call elections until the highest priority member available becomes primary.

- A replica set can have up to `50 members`, but only `7 voting members`, non-voting members allow a replica set to have more than seven members.
- To configure a member as non-voting, set its `members[n].votes` value to `0`.
- If your replica set has an even number of members, add an *arbiter* to ensure that members can quickly obtain a majority of votes in an election for primary.
- In general and when possible, all members should have only 1 vote. This prevents intermittent ties, deadlocks, or the wrong members from becoming primary.

A *network partition* may segregate a primary into a partition with a minority of nodes. When the primary detects that it can only see a minority of nodes in the replica set, the primary steps down as primary and becomes a secondary. Independently, a member in the partition that can communicate with a majority of the nodes (including itself) holds an election to become the new primary.

6.5 Priority

The `priority` settings of replica set members affect both the timing and the outcome of *elections* for primary. Higher-priority members are more likely to call elections, and are more likely to win. Use this setting to ensure that some members are more likely to become primary and that others can never become primary. The higher the number, the higher the priority.

- To modify priorities, you update the `members` array in the replica configuration object. The array index begins with `0`. Do **not** confuse this index value with the value of the replica set member’s `members[n]._id` field in the array.

- The value of priority can be any floating point (i.e. decimal) number between 0 and 1000. The default value for the priority field is 1.
- To block a member from seeking election as primary, assign it a priority of 0. *Hidden members* and *delayed members* have priority set to 0.
- For *arbiters*, the default priority value is 1; however, arbiters cannot become primary regardless of the configured value.

6.6 Failover/Rollover

A rollback reverts write operations on a former *primary* when the member rejoins its *replica set* after a *failover*. A rollback is necessary only if the primary had accepted write operations that the *secondaries* had **not** successfully replicated before the primary stepped down. When the primary rejoins the set as a secondary, it reverts, or "rolls back," its write operations to maintain database consistency with the other members.

A rollback does ***not*** occur if the write operations replicate to another member of the replica set before the primary steps down *and* if that member remains available and accessible to a majority of the replica set.

- When a rollback does occur, MongoDB writes the rollback data to *BSON* files in the `rollback/` folder under the database's `dbPath` directory.
- To prevent rollbacks of data that have been acknowledged to the client, use *w: majority write concern* to guarantee that the write operations propagate to a majority of the replica set nodes before returning with acknowledgement to the issuing client.

A `mongod` instance will not rollback more than 300 megabytes of data. If your system must rollback more than 300 megabytes, you must manually intervene to recover the data. In this situation, save the data directly or force the member to perform an initial sync. To force initial sync, sync from a "current" member of the set by deleting the content of the `dbPath` directory for the member that requires a larger rollback.

6.7 Hidden Replica Set Member

A hidden member maintains a copy of the *primary's* data set but is **invisible** to client applications. Hidden members are good for workloads with different usage patterns from the other members in the *replica set*. Hidden members must always be *priority 0 members* and so **cannot become primary**. The `db.isMaster()` method does not display hidden members. Hidden members, however, **may vote** in *elections*. The most common use of hidden nodes is to support *delayed members*.

- **Read Operations:** Clients will not distribute reads with the appropriate *read preference* to hidden members. As a result, these members receive no traffic other than basic replication. Use hidden members for dedicated tasks such as reporting and backups.
- *Delayed members* should be hidden. In a sharded cluster, `mongos` do not interact with hidden members.
- **Voting:** Hidden members *may* vote in replica set elections. If you stop a voting hidden member, ensure that the set has an active majority or the *primary* will step down.

6.8 Delayed Replica Set Member

Delayed members contain copies of a *replica set's* data set. However, a delayed member's data set reflects an earlier, or delayed, state of the set.

Because delayed members are a "rolling backup" or a running "historical" snapshot of the data set, they may help you recover from various kinds of human error. For example, a delayed member can make it possible to recover from unsuccessful application upgrades and operator errors including dropped databases and collections.

- **Must be** *priority 0* members. Set the priority to 0 to prevent a delayed member from becoming primary.
- **Should be** *hidden* members. Always prevent applications from seeing and querying delayed members.
- *do* vote in *elections* for primary, if `members[n].votes` is set to 1.

- Delayed members copy and apply operations from the source *oplog* on a delay. When choosing the amount of delay, consider that the amount of delay:
- must be equal to or greater than your expected maintenance window durations.
- must be *smaller* than the capacity of the oplog. For more information on oplog size, see *Oplog Size*. The length of the secondary `members[n].slaveDelay` must fit within the window of the oplog. If the oplog is shorter than the `members[n].slaveDelay` window, the delayed member cannot successfully replicate operations.
- In sharded clusters, delayed members have limited utility when the *balancer* is enabled. Because delayed members replicate chunk migrations with a delay, the state of delayed members in a sharded cluster are not useful for recovering to a previous state of the sharded cluster if any migrations occur during the delay window.

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[0].hidden = true
cfg.members[0].slaveDelay = 3600
rs.reconfig(cfg)
```

6.9.0 Replica Set Oplog

The *oplog* (operations log) is a special *capped collection* in the PRIMARY "local" collection that keeps a rolling record of all operations that modify the data stored in your databases. MongoDB applies database operations on the *primary* and then records the operations on the primary's oplog. The *secondary* members then copy and apply these operations in an asynchronous process. All replica set members contain a copy of the oplog, in the `local.oplog.rs` collection, which allows them to maintain the current state of the database.

- Whether applied once or multiple times to the target dataset, each operation in the oplog produces the same results, i.e. each operation in the oplog is *idempotent*.
- Idempotent: The quality of an operation to produce the same result given the same input, whether run once or run multiple times.

When you start a replica set member for the first time, MongoDB creates an oplog of a default size.
For UNIX and Windows systems. The default oplog size depends on the storage engine:

Storage Engine	Default Oplog Size	Lower Bound	Upper Bound
In-Memory Storage Engine	5% of physical memory	50 MB	50 GB
WiredTiger Storage Engine	5% of free disk space	990 MB	50 GB
MMAPv1 Storage Engine	5% of free disk space	990 MB	50 GB

To view oplog status, including the size and the time range of operations, issue the `rs.printReplicationInfo()` method. To resize the oplog after replica set initiation, use the *Change the Size of the Oplog* procedure.

Replication lag is a delay between an operation on the *primary* and the application of that operation from the *oplog* to the *secondary*. To check the current length of replication lag: call the `rs.printSlaveReplicationInfo()` method. Minimum lag times are essential to overall database consistency.

A larger *oplog* can give a replica set a greater tolerance for lag, and make the set more resilient. Increase Oplog storage under the following usage patterns: 1. Updates to multiple documents at once 2. Deletions equal the same amount of data as Inserts 3. Significant number of in-place Updates

The oplog should be long enough to hold all transactions for the longest downtime you expect on a secondary. At a minimum, an oplog should be able to hold minimum 24 hours of operations. Some prefer 72 hours.

7.0 Sharded Clusters

Vertical scaling adds more CPU and storage resources to increase capacity. Scaling by adding capacity has limitations: high performance systems with large numbers of CPUs and large amount of RAM are disproportionately *more expensive* than smaller systems. Additionally, cloud-based providers may only allow users to provision smaller instances. As a result there is a *practical maximum* capability for vertical scaling.

Sharding, or *horizontal scaling*, by contrast, divides the data set and distributes the data over multiple servers, or **shards**. Each shard is an independent database, and collectively, the shards make up a single logical database. MongoDB uses sharding to support deployments with very large data sets and high throughput operations.

Use *sharded clusters* if:

- Your data set approaches or exceeds the storage capacity of a single MongoDB instance.
- The size of your system's active *working set* *will soon* exceed the capacity of your system's *maximum* RAM.
- A single MongoDB instance cannot meet the demands of your write operations, and all other approaches have not reduced contention.

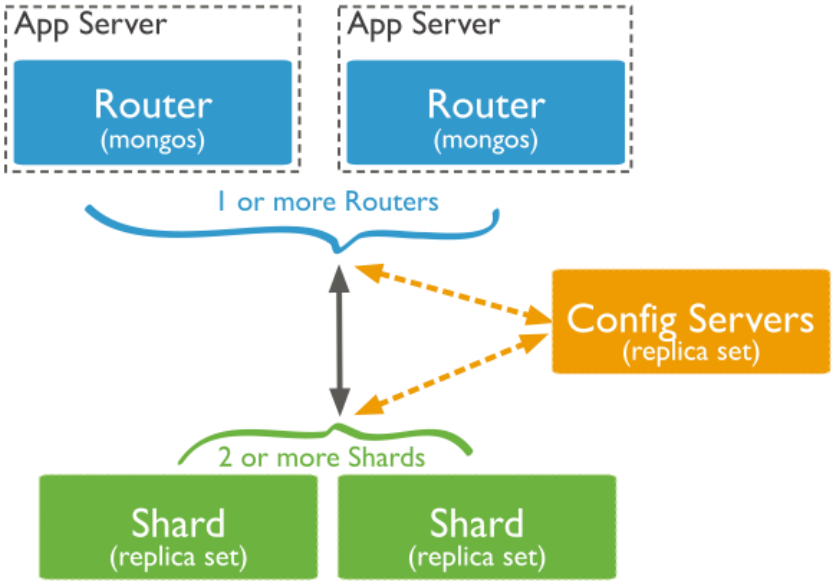

Figure 2, Typical Sharded Cluster Architecture

7.1 Sharded Cluster Components

- **Shards** store the data. To provide high availability and data consistency, in a production sharded cluster, each shard is a *replica set* [1]. 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. A client sends requests to a mongos, which then routes the operations to the shards and returns the results to the clients. A sharded cluster can contain more than one mongos to divide the client request load, and most sharded clusters have more than one mongos for this reason.
- **Config servers** store the cluster's metadata. This 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.

Note: Changed in version 3.2: Starting in MongoDB 3.2, config servers for sharded clusters can be deployed as a *replica set*. The replica set config servers must run the *WiredTiger storage engine*. MongoDB 3.2 deprecates the use of three

mirrored mongod instances for config servers. Use the `sh.status()` method in the `mongo` shell to see an overview of the cluster.

7.2 Data Partitioning/Shard Key

MongoDB distributes data, or shards, at the collection level. Sharding partitions a collection's data by the **shard key**. Every database has a *primary shard* that holds all the un-sharded collections for a database.

To shard a collection, you need to select a **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. To divide the shard key values into chunks, MongoDB uses either **range based partitioning** or **hash based partitioning**. See the *Shard Key* documentation for more information. The index on the shard key can be a compound index or hashed index but **cannot** be a *multikey index*, a *text index* or a *geospatial index*.

- If you shard a collection without any documents and *without* such an index, `sh.shardCollection()` creates the index on the shard key. If the collection already has documents, you must create the index before using `sh.shardCollection()`.

For *range-based sharding*, MongoDB divides the data set into ranges determined by the shard key values to provide **range based partitioning**. For *hash based partitioning*, MongoDB computes a hash of a field's value, and then uses these hashes to create chunks.

MongoDB allows administrators to direct the balancing policy using **tag aware sharding**. Administrators create and associate tags with ranges of the shard key, and then assign those tags to the shards. Then, the balancer migrates tagged data to the appropriate shards and ensures that the cluster always enforces the distribution of data that the tags describe.

Tags are the primary mechanism to control the behavior of the balancer and the distribution of chunks in a cluster. Most commonly, tag aware sharding serves to improve the locality of data for sharded clusters that span multiple data centers.

The addition of new data or the addition of new servers can result in data distribution imbalances within the cluster, such as a particular shard contains significantly more chunks than another shard or a size of a chunk is significantly greater than other chunk sizes.

MongoDB ensures a balanced cluster using two background processes: splitting and the balancer.

- The chunk size is user configurable. The default chunk size is 64MB.
- **Splitting** only creates metadata changes, **Balancing** migrates data between Shards.
- Finally, the metadata regarding the location of the chunk on *config server* is updated.

The *balancer* will not begin moving data across shards until the imbalance of chunks among the shards exceeds the *migration threshold*. An insufficiently granular shard key can result in chunks that are "unsplittable". Splits cannot be "undone". If you increase the chunk size, existing chunks must grow through inserts or updates until they reach the new size.

Shard Keys are immutable: After you insert a document into a sharded collection, you cannot change the document's value for the field or fields that comprise the shard key. The `update()` operation will not modify the value of a shard key in an existing document.

MongoDB does not support creating new unique indexes in sharded collections, because insert and indexing operations are local to each shard, and will not allow you to shard collections with unique indexes on fields other than the `_id` field. MongoDB *can* enforce uniqueness on the *shard key*. MongoDB enforces uniqueness on the *entire* key combination, and not specific components of the shard key. In most cases, the best shard keys are compound keys that include elements

that permit *write scaling* and *query isolation*, as well as *high cardinality*. These ideal shard keys are not often the same keys that require uniqueness and enforcing unique values in these collections requires a different approach.

7.3 Config Servers

Config servers hold the metadata about the cluster, such as the shard location of the data. The `mongos` instances cache this data and use it to route reads and writes to shards.

Changed in version 3.2: Starting in MongoDB 3.2, config servers for sharded clusters can be deployed as a *replica set*. Using a replica set for the config servers improves consistency across the config servers. In addition, using a replica set for config servers allows a sharded cluster to have more than 3 config servers since a replica set can have up to 50 members. To deploy config servers as a replica set, the config servers must run the *WiredTiger storage engine*.

The following restrictions apply to a replica set configuration when used for config servers:

- Must have zero *arbiters*.
- Must have no *delayed members*.
- Must build indexes (i.e. no member should have `buildIndexes` setting set to false).

7.4 Mongos

`mongos` stands for "MongoDB Shard," is a routing service for MongoDB shard configurations that processes queries from the application layer, and determines the location of this data in the *sharded cluster*, in order to complete these operations. From the perspective of the application, a `mongos` instance behaves identically to any other MongoDB instance.

Generally, the fastest queries in a sharded environment are those that `mongos` will route to a single shard, using the *shard key* and the cluster meta data from the *config server*. For queries that don't include the shard key, `mongos` must query all shards, wait for their responses and then return the result to the application. These "scatter/gather" queries can be long running operations.

Pre-Splitting Data: Only pre-split an empty collection.

Manual Chunk Splitting

Merge Chunks

8.0 Security

8.1 Authentication Mechanism

MongoDB supports multiple authentication mechanisms that can be integrated into your existing authentication environment.

- *SCRAM-SHA-1*
- *MongoDB Challenge and Response (MONGODB-CR)*

Changed in version 3.0: New challenge-response users created in 3.0 will use SCRAM-SHA-1. If using 2.6 user data, MongoDB 3.0 will continue to use the MONGODB-CR.

- *x.509 Certificate Authentication*.

In addition to supporting the aforementioned mechanisms, MongoDB Enterprise also supports the following mechanisms:

- *LDAP proxy authentication*, and
- *Kerberos authentication*.

In addition to verifying the identity of a client, MongoDB can require members of replica sets and sharded clusters to *authenticate their membership* to their respective replica set or sharded cluster. See *Internal Authentication* for more information.

8.2 Users

To add a user, MongoDB provides the `db.createUser()` method. When adding a user, you can assign *roles* to the user in order to grant privileges. MongoDB stores all user information, including `name`, `password`, and the `user's authentication database`, in the *system.users* collection in the `admin` database.

- When adding a user, you create the user in a specific database. This database is the authentication database for the user.
- A user can have privileges across different databases; i.e. a user's privileges are not limited to the authentication database. By assigning to the user roles in other databases, a user created in one database can have permissions to act on other databases. For more information on roles, see *Role-Based Access Control*.
- The user's name and authentication database serve as a unique identifier for that user. That is, if two users have the same name but are created in different databases, they are two separate users. If you intend to have a single user with permissions on multiple databases, create a single user with roles in the applicable databases instead of creating the user multiple times in different databases.

To authenticate a user, either

- Use the command line authentication options (e.g. -u, -p, --authenticationDatabase) when connecting to the mongod or mongos instance, or
- Connect first to the mongod or mongos instance, and then run the authenticate command or the db.auth() method against the authentication database.

8.3 Sharded Cluster Users

To create users for a sharded cluster, connect to the `mongos` instance and add the users. Clients then authenticate these users through the `mongos` instances.

However, some maintenance operations, such as `cleanupOrphaned`, `compact`, `rs.reconfig()`, require direct connections to specific shards in a sharded cluster. To perform these operations, you must connect directly to the shard and authenticate as a *shard local* administrative user.

To create a *shard local* administrative user, connect directly to the shard and create the user. MongoDB stores *shard local* users in the `admin` database of the shard itself.

These *shard local* users are completely independent from the users added to the sharded cluster via `mongos`. *Shard local* users are local to the shard and are inaccessible by `mongos`.

Direct connections to a shard should only be for shard-specific maintenance and configuration. In general, clients should connect to the sharded cluster through the `mongos`.

8.3.1 Locahost Exception

The localhost exception allows you to enable access control and then create the first user in the system. With the localhost exception, after you enable access control, connect to the localhost interface and create the first user in the `admin` database. The first user must have privileges to create other users, such as a user with the `userAdmin` or `userAdminAnyDatabase` role.

- The localhost exception applies only when there are no users created in the MongoDB instance.
- In the case of a sharded cluster, the localhost exception applies to each shard individually as well as to the cluster as a whole. Once you create a sharded cluster and add a user administrator through the mongos instance, you must still prevent unauthorized access to the individual shards. Follow one of the following steps for each shard in your cluster:
    - Create an administrative user, or
    - Disable the localhost exception at startup. To disable the localhost exception, set the enableLocalhostAuthBypass parameter to 0.

8.4 Add Users

MongoDB employs role-based access control (RBAC) to determine access for users. A user is granted one or more *roles* that determine the user's access or privileges to MongoDB *resources* and the *actions* that user can perform. A user should have only the minimal set of privileges required to ensure a system of *least privilege*. Each application and user of a MongoDB system should map to a distinct user. This *access isolation* facilitates access revocation and ongoing user maintenance.

8.5 Enable Client Access Control

Enabling access control requires authentication of every user. Once authenticated, users only have the privileges as defined in the roles granted to the users.

To enable access control, use either the command line option `--auth` or `security.authorization` configuration file setting.

With access control enabled, ensure you have a user with `userAdmin` or `userAdminAnyDatabase` role in the `admin` database.

Note

The tutorial enables access control and uses the *default authentication mechanism*. To specify a different authentication mechanism, see *Authentication Mechanisms*.

You can also enable client access control by *enforcing internal authentication* for replica sets or sharded clusters.

8.5.1 Considerations

You can create users before enabling access control or you can create users after enabling access control. If you enable access control before creating any user, MongoDB provides a *localhost exception* which allows you to create a user administrator in the `admin` database. Once created, authenticate as the user administrator to create additional users as needed.

8.6 Built-in Roles

MongoDB provides built-in roles that provide the different levels of access commonly needed in a database system. Built-in *database user roles* and *database administration roles* roles exist in *each* database. The `admin` database contains additional roles.

8.6.1 User Roles

Every database includes the following roles:

read	Provides the ability to read data on all *non*-system collections and on the following system collections: system.indexes, system.js, and system.namespaces collections.
readWrite	Provides all the privileges of the read role and the ability to modify data on all non-system collections and the system.js collection.

8.6.2 Database Admin Roles

Every database includes the following database administration roles:

dbAdmin	Provides the ability to perform administrative tasks such as schema-related tasks, indexing, gathering statistics. This role does not grant privileges for user and role management. For the specific privileges granted by the role, see dbAdmin.
dbOwner	Provides the ability to perform any administrative action on the database. This role combines the privileges granted by the readWrite, dbAdmin and userAdmin roles.
userAdmin	Provides the ability to create and modify roles and users on the current database. Since the userAdmin role allows users to grant any privilege to any user, including themselves, the role also indirectly provides *superuser* access to either the database or, if scoped to the admin database, the cluster.

8.6.3 Cluster Admin Roles

The `admin` database includes the following roles for administering the whole system rather than a specific database. These roles include but are not limited to *replica set* and *sharded cluster* administrative functions.

clusterAdmin	Provides the greatest cluster-management access. This role combines the privileges granted by the clusterManager, clusterMonitor, and hostManager roles. Additionally, the role provides the dropDatabase action.
clusterManager	Provides management and monitoring actions on the cluster. A user with this role can access the config and local databases, which are used in sharding and replication, respectively. For the specific privileges granted by the role, see clusterManager.
clusterMonitor	Provides read-only access to monitoring tools, such as the MongoDB Cloud Manager and Ops Manager monitoring agent.
hostManager	Provides the ability to monitor and manage servers.

8.6.4 Backup and Restoration Roles

The `admin` database includes the following roles for backing up and restoring data:

`backup`	Provides privileges needed to back up data. This role provides sufficient privileges to use the MongoDB Cloud Manager backup agent, Ops Manager backup agent, or to use `mongodump`. For the specific privileges granted by the role, see `backup`.
`restore`	Provides privileges needed to restore data with `mongorestore` without the *--oplogReplay* option or without `system.profile` collection data.

8.6.5 All-Database Roles

The `admin` database provides the following roles that apply to all databases in a `mongod` instance and are roughly equivalent to their single-database equivalents:

readAnyDatabase	Provides the same read-only permissions as read, except it applies to *all* databases in the cluster. The role also provides the listDatabases action on the cluster as a whole. For the specific privileges granted by the role, see readAnyDatabase.
readWriteAnyDatabase	Provides the same read and write permissions as readWrite, except it applies to *all* databases in the cluster. The role also provides the listDatabases action on the cluster as a whole. For the specific privileges granted by the role, see readWriteAnyDatabase.
userAdminAnyDatabase	Provides the same access to user administration operations as userAdmin, except it applies to *all* databases in the cluster. Since the userAdminAnyDatabase role allows users to grant any privilege to any user, including themselves, the role also indirectly provides *superuser* access. For the specific privileges granted by the role, see userAdminAnyDatabase.
dbAdminAnyDatabase	Provides the same access to database administration operations as dbAdmin, except it applies to *all* databases in the cluster. The role also provides the listDatabases action on the cluster as a whole

8.6.6 Other Roles

The following role provides full privileges on all resources:

SuperUser Roles

Warning: Several roles provide either indirect or direct system-wide superuser access.

The following roles provide the ability to assign any user any privilege on any database, which means that users with one of these roles can assign *themselves* any privilege on any database:

- dbOwner role, when scoped to the admin database
- userAdmin role, when scoped to the admin database
- userAdminAnyDatabase role

root	Provides access to the operations and all the resources of the readWriteAnyDatabase, dbAdminAnyDatabase, userAdminAnyDatabase and clusterAdmin roles *combined*.
__system	Provides privileges to take any action against any object in the database. Do not assign this role to user objects representing applications or human administrators, other than in exceptional circumstances.

Manage Users and Roles

Collection-Level Access Control: allows administrators to grant users privileges that are scoped to specific collections.

Enable authentication tutorial

9.0 Encryption

9.1 Transport Encryption

You can use TLS/SSL (Transport Layer Security/Secure Sockets Layer) to encrypt all of MongoDB's network traffic. TLS/SSL ensures that MongoDB network traffic is only readable by the intended client.

Before you can use SSL, you must have a `.pem` file containing a public key certificate and its associated private key. MongoDB can use any valid SSL certificate issued by a certificate authority or a self-signed certificate. If you use a self-signed certificate, although the communications channel will be encrypted, there will be *no* validation of server identity. Although such a situation will prevent eavesdropping on the connection, it leaves you vulnerable to a man-in-the-middle attack. Using a certificate signed by a trusted certificate authority will permit MongoDB drivers to verify the server's identity.

9.2 Encryption at Rest Wired-Tiger Storage Engine only, Enterprise Edition Only

There are two broad classes of approaches to encrypting data at rest with MongoDB: Application Level Encryption and Storage Encryption. You can use these solutions together or independently. Encryption at rest, when used in conjunction with transport encryption and good security policies that protect relevant accounts, passwords, and encryption keys, can help ensure compliance with security and privacy standards, including HIPAA, PCI-DSS, and FERPA.

10.0 Administration

10.1 Backup Methods

10.1.1 Filesystem Snapshot

You can create a backup by copying MongoDB's underlying data files. File systems snapshots are an operating system volume manager feature, and are not specific to MongoDB. To get a correct snapshot of a running `mongod` process, you must have journaling enabled and the journal must reside on the same logical volume as the other MongoDB data files. Without journaling enabled, there is no guarantee that the snapshot will be consistent or valid.

10.1.2 Copy Underlying Data Files

Another way to copy the underlying files is to make a copy of the entire –dbpath directory. First lock the database then flush all the "dirty" files using the command below.

db.fsyncLock()

While the DB is locked, any further writes will be queued until unlocked

Use the copy command

cp –R /data/db/* /mnt/externaldevice/backup

unlock DB and pending writes will be processed

db.fsyncUnlock()

Important: If you use authentication, don't close the shell because you can't login again and may have to restart mongod. Also, fsyncLock() does not persist and a new mongod will start unlocked.

10.1.3 mongodump/mongorestore

The `mongodump` tool reads data from a MongoDB database and creates high fidelity BSON files. The `mongorestore` tool can populate a MongoDB database with the data from these BSON files.

Mongodump/mongorestore can be used to create an entirely new database using the –d –c commandline options.

If you have a unique index other than "_id" do not mongodump/restore because indexes are not dumped. Instead "freeze" the data and copy the directory.

10.2 Incremental Backups Using mongooplog

Perform a complete backup initially then incremental backups of the oplog since the last full backup. This technique is much more complex than other methods.

10.3 Journal Files

With journaling enabled, MongoDB creates a subdirectory named `journal` under the `dbPath` directory. The `journal` directory contains journal files named `j._<sequence>` where `<sequence>` is an integer starting from `0` and a "last sequence number" file `lsn`.

Journal files contain the write ahead logs; each journal entry describes the bytes the write operation changed in the data files. Journal files are append-only files. When a journal file holds 1 gigabyte of data, MongoDB creates a new journal file. If you use the `storage.smallFiles` option when starting `mongod`, you limit the size of each journal file to 128 megabytes.

MongoDB uses *write ahead logging* to an on-disk *journal* to guarantee *write operation* durability. The MMAPv1 storage engine also requires the *journal* in order to provide crash resiliency.

The WiredTiger storage engine does not require journaling to guarantee a consistent state after a crash. The database will be restored to the last consistent *checkpoint* during recovery. However, if MongoDB exits unexpectedly in between checkpoints, journaling is required to recover writes that occurred after the last checkpoint.

MongoDB configures WiredTiger to use snappy compression for the journaling data.

To enable journaling, start `mongod` with the *--journal* command line option. For 64-bit builds of `mongod`, journaling is enabled by default. Do not disable journaling on production systems.

10.4 Server Logs

`Log messages have the form: <timestamp> <severity> <component> [<context>] <message>`

Severity Levels:

Level	Description
F	Fatal
E	Error
W	Warning
I	Informational, for Verbosity Level of 0
D	Debug, for All Verbosity Levels > 0

The default format for the <timestamp> is iso8601-local. To modify the timestamp format, use the *--timeStampFormat* runtime option or the systemLog.timeStampFormat setting.

10.4.1 Log Message Components

Log messages now include components, providing functional categorization of the messages:

ACCESS	related to access control, such as authentication
COMMAND	related to *database commands*, such as count.
CONTROL	related to control activities, such as initialization
FTDC (New in v3.2)	related to the diagnostic data collection mechanism, such as server statistics and status messages.
QUERY	related to queries, including query planner activities.

INDEX	related to indexing operations, such as creating indexes
NETWORK	related to network activities, such as accepting connections
REPL	related to replica sets, such as initial sync and heartbeats
SHARDING	related to sharding activities, such as the startup of the mongos.
STORAGE	related to storage activities, such as processes involved in the fsync command.
JOURNAL	related specifically to journaling activities
WRITE	related to write operations, such as update commands
GEO	related to the parsing of geospatial shapes, such as verifying the GeoJSON shapes

See *Configure Log Verbosity Levels*. To set verbosity log level per component use e.g. db.setLogLevel(-1, "query") where loglevel {0-5, -1=inherit} . To view the current verbosity levels, use the db.getLogComponents() method.

10.4.2 Log Rotation

When used with the *--logpath* option or `systemLog.path` setting, `mongod` and `mongos` instances report a live account of all activity and operations to a log file. By default, MongoDB only rotates logs in response to the `logRotate` command, or when the `mongod` or `mongos` process receives a `SIGUSR1` signal from the operating system.

MongoDB's standard log rotation approach archives the current log file and starts a new one. The current log file is renamed by appending a UTC timestamp to the filename, in *ISODate* format. It then opens a new log file

You can also configure MongoDB to support the Linux/Unix logrotate utility.

Also, configure `mongod` to send log data to the system's `syslog`, using the *--syslog* option. In this case, you can take advantage of alternate logrotation tools.

Mongo Shell Log Rotate Command:

use admin
db.runCommand({ logRotate : 1 })

10.5 Database Profiling

MongoDB's "Profiler" is a database profiling system that can help identify inefficient queries and operations.

Profile Levels:

Level	Setting
0	Off, No profiling
1	On, Only indicates slow operations, time settable
2	On, all operations included

Enable the profiler by setting the `profile` value using: db.setProfilingLevel(level, slowms)

You can view the output of the profiler in the system.profile collection of your database by issuing the show profile command in the mongo shell, or with the following operation:

db.system.profile.find({millis: {$gt: 100}}) : for operations greater than 100ms

10.6 Monitoring

Monitoring is a critical component of all database administration. A firm grasp of MongoDB's reporting will allow you to assess the state of your database and maintain your deployment without crisis.

Monitoring Strategies: Each strategy can help answer different questions and is useful in different contexts. These methods are complementary.

- First, there is a set of utilities distributed with MongoDB that provides real-time reporting of database activities.
- Second, *database commands* return statistics regarding the current database state with greater fidelity.
- Third, MongoDB Cloud Manager, a hosted service, and Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced, provide monitoring to collect data from running MongoDB deployments as well as providing visualization and alerts based on that data.

10.6.1 MongoDB Reporting Tools

The MongoDB distribution includes a number of utilities that quickly return statistics about instances' performance and activity.

mongostat	`mongostat` captures and returns the counts of database operations by type (e.g. insert, query, update, delete, etc.). These counts report on the load distribution on the server. Use `mongostat` to understand the distribution of operation types and to inform capacity planning. See the *mongostat manual* for details.
mongotop	`mongotop` tracks and reports the current read and write activity of a MongoDB instance, and reports these statistics on a per collection basis. Use `mongotop` to check if your database activity and use match your expectations. See the *mongotop manual* for details.

10.6.2 Monitor Commands

MongoDB includes a number of commands that report on the state of the database..These data may provide a finer level of granularity than the utilities discussed above.

`db.serverStatus()`	from the shell, returns a general overview of the status of the database, detailing disk usage, memory use, connection, journaling, and index access.
`db.stats()`	from the shell, returns a document that addresses storage use and data volumes at the database level. The `dbStats` reflect the amount of storage used, the quantity of data contained in the database, and object, collection, and index counters.
`db.collection.stats()`	provides statistics that resemble `dbStats` at the collection level, including a count of the objects in the collection, the size of the collection, the amount of disk space used by the collection, and information about its indexes.
`rs.status()`	returns an overview of your replica set's status. The *replSetGetStatus* document details the state and configuration of the replica set and statistics about its members.
`db.collection.totalIndexSize()`	return the index size in bytes

11. MongoDB and Relational Databases

Highly Scalable, limited Function: Memcached, Key Value Store

Limited Scaling, High Funtionality: Traditional Relational Databases

MongoDBDoes does not support Joins or Transactions

See MongoDB limits and thresholds

Database Concepts

SQL Terms/Concepts	MongoDB Terms/Concepts
database	*database*
table	*collection*

SQL Terms/Concepts	MongoDB Terms/Concepts
row	*document* or *BSON* document
column	*field*
index	*index*
table joins	embedded documents and linking
primary key Specify any unique column or column combination as primary key.	*primary key* In MongoDB, the primary key is automatically set to the *_id* field.
aggregation (e.g. group by)	aggregation pipeline

Flexible (Schema-less) Schema

Data in MongoDB has a *flexible schema*. Unlike SQL databases, where you must determine and declare a table's schema before inserting data, MongoDB's *collections* do not enforce *document* structure.

References store the relationships between data by including links or *references* from one document to another. Applications can resolve these *references* to access the related data. Broadly, these are *normalized* data models.

Embedded documents capture relationships between data by storing related data in a single document structure. MongoDB documents make it possible to embed document structures in a field or array within a document. These *denormalized* data models allow applications to retrieve and manipulate related data in a single database operation.

Atomocity of Write Operations

In MongoDB, write operations are atomic at the *document* level, and no single write operation can atomically affect more than one document or more than one collection. A denormalized data model with embedded data combines all related data for a represented entity in a single document. This facilitates atomic write operations since a single write operation can insert or update the data for an entity. Normalizing the data would split the data across multiple collections and would require multiple write operations that are not atomic collectively.

Appendix

A.1 BSON data types

BSON supports the following data types as values in documents. Each data type has a corresponding number and string alias that can be used with the `$type` operator to query documents by BSON type.

Type	Number	Alias	Notes
Double	1	"double"	
String	2	"string"	
Object	3	"object"	
Array	4	"array"	
Binary data	5	"binData"	
Undefined	6	"undefined"	Deprecated.

Type	Number	Alias	Notes
ObjectId	7	“objectId”	
Boolean	8	“bool”	
Date	9	“date”	
Null	10	“null”	
Regular Expression	11	“regex”	
DBPointer	12	“dbPointer”	
JavaScript	13	“javascript”	
Symbol	14	“symbol”	
JavaScript (with scope)	15	“javascriptWithScope”	
32-bit integer	16	“int”	
Timestamp	17	“timestamp”	
64-bit integer	18	“long”	
Min key	-1	“minKey”	
Max key	127	“maxKey”	

A.2 Unique Keys for Sharded Collections

The `unique` constraint on indexes ensures that only one document can have a value for a field in a *collection*. For *sharded collections these unique indexes cannot enforce uniqueness* because insert and indexing operations are local to each shard.

MongoDB does not support creating new unique indexes in sharded collections and will not allow you to shard collections with unique indexes on fields other than the `_id` field.

If you need to ensure that a field is always unique in a sharded collection, there are three options:

Enforce uniqueness of the *shard key*.

MongoDB *can* enforce uniqueness for the *shard key*. For compound shard keys, MongoDB will enforce uniqueness on the *entire* key combination, and not for a specific component of the shard key.

You cannot specify a unique constraint on a *hashed index*.

- Use a secondary collection to enforce uniqueness.

Create a minimal collection that only contains the unique field and a reference to a document in the main collection. If you always insert into a secondary collection *before* inserting to the main collection, MongoDB will produce an error if you attempt to use a duplicate key.

If you have a small data set, you may not need to shard this collection and you can create multiple unique indexes. Otherwise you can shard on a single unique key.

- Use guaranteed unique identifiers.

  Universally unique identifiers (i.e. UUID) like the `ObjectId` are guaranteed to be unique.

Procedures

Unique Constraints on the Shard Key

Process

To shard a collection using the `unique` constraint, specify the `shardCollection` command in the following form:

db.runCommand({ shardCollection : "test.users" , key : { email : 1 } , unique : true });

Remember that the `_id` field index is always unique. By default, MongoDB inserts an `ObjectId` into the `_id` field. However, you can manually insert your own value into the `_id` field and use this as the shard key. To use the `_id` field as the shard key, use the following operation:

db.runCommand({ shardCollection : "test.users" })

Limitations

- You can only enforce uniqueness on one single field in the collection using this method.
- If you use a compound shard key, you can only enforce uniqueness on the *combination* of component keys in the shard key.

A.3 Deploy a Sharded Cluster Test Environment, tutorial

Sets up Sharded Clusters for testing.

A.3.1 Use shell script "init_sharded_env_raw.sh"

Linux script sets up 3 shards with 3-member replica sets each. User edit for database, collection and shard key.

init_sharded_env_raw.sh

A.3.2 ShardingTest Platform

Deploys Sharding Cluster with mongods on ports 30000, 30001, 30002

> mongo –nodb

> cluster = new ShardingTest({"shards":3, "chunksize": 1})

A.3.3 Insert some data using shell script

use students
for (var i = 0; i<1000; i++) {db.grades.insert({student_id:i})}

A.4 Compound Multikey Indexes and Multikey Index Bounds

For a *compound* multikey index, each indexed document can have *at most* one indexed field whose value is an array. As such, you cannot create a compound multikey index if more than one to-be-indexed field of a document is an array. Or, if a compound multikey index already exists, you cannot insert a document that would violate this restriction.

For example, consider a collection that contains the following document:

```
{ _id: 1, a: [ 1, 2 ], b: [ 1, 2 ], category: "AB - both arrays" }
```

You cannot create a compound multikey index `{ a: 1, b: 1 }` on the collection since both the `a` and `b` fields are arrays.

But consider a collection that contains the following documents:

```
{ _id: 1, a: [1, 2], b: 1, category: "A array" }
{ _id: 2, a: 1, b: [1, 2], category: "B array" }
```

A compound multikey index `{ a: 1, b: 1 }` is permissible since for each document, only one field indexed by the compound multikey index is an array; i.e. no document contains array values for both `a` and `b` fields. After creating the compound multikey index, if you attempt to insert a document where both `a` and `b` fields are arrays, MongoDB will fail the insert.

A.5 Enabling Authentication Example

This section describes native SCRAM-SHA-1 authorization set-up where administrator rossw is omnipotent for all admin, user, database, cluster, restore roles.

User type	roles	Database (db)
Administrator (omnipotent, unique)	• `root` • `userAdminAnyDatabase` • `dbAdminAnyDatabase`	"admin"
normal db user	• `readWrite`	Per database

See Authorization Tutorial and Built-in Role Reference

Method: Add db admin user before enabling Authentication:

1. Before enabling server with Authentication enabled, start a mongod **without** –-auth option.
mongod --port 27017 --dbpath c:\data\db

2. Create the database admin user with omnipotent privileges
mongo –-port 27017
use admin

MongoDB Enterprise > db.createUser({ user: "rossw", pwd: "xxxx", roles: [{ role: "root", db: "admin" }]})
Successfully added user: {
"user" : "rossw",

```
    "roles" : [
        {
            "role" : "root",
            "db" : "admin"
        }
    ]
}
```

3. Update admin users. Notice that db.updateUser() replaces, not updates, the user role document

```
use admin
su = {roles: [
{role: "root", db:"admin"},
{role: "dbAdminAnyDatabase", db: "admin"},
{role: "userAdminAnyDatabase", db: "admin"}]}

db.updateUser("rossw", su)
```

```
MongoDB Enterprise > db.system.users.find()
{ "_id" : "admin.rossw", "user" : "rossw", "db" : "admin", "credentials" : { "SCRAM-SHA-1" : { "iterationCount" : 10000,
"salt" : "/UMdZS9aWXK4mzJROcnKJQ==", "storedKey" : "mKKIzo0vZ+qZMKCzGUhYVGQVjBA=", "serverKey" :
"3ECzmLjWo3Wr1XmICGTRdj9lZLY=" } }, "roles" : [ { "role" : "root", "db" : "admin" }, { "role" : "dbAdminAnyDatabase",
"db" : "admin" }, { "role" : "userAdminAnyDatabase", "db" : "admin" } ] }
{ "_id" : "admin.raw", "user" : "raw", "db" : "admin", "credentials" : { "SCRAM-SHA-1" : { "iterationCount" : 10000, "salt" :
"2UfpNEFcC3J68E47puNPmw==", "storedKey" : "9Yz2QlAGqNS9UKyskpMTq52fuwU=", "serverKey" :
"Tj3U6eac1BwAu4V+B9BFsJYXEps=" } }, "roles" : [ { "role" : "readWrite", "db" : "week6" } ] }
```

Method: Enable Authorization and add User

4. Exit mongo, kill then restart mongod with --auth

```
mongod –-port 27017 –-dbpath c:\data\db –-auth
```

5. start mongo shell and login as db admin user

```
mongo --port 27017 -u "rossw" -p "xxxx" --authenticationDatabase "admin"
```

6. Add database user raw with readWrite privileges on db week6

```
MongoDB Enterprise > db.createUser({ user: "raw", pwd: "xxxx", roles: [{ role: "readWrite", db: "week6" }]})
Successfully added user: {
    "user" : "raw",
    "roles" : [
        {
            "role" : "readWrite",
            "db" : "week6"
        }
    ]
}
```

Shell Authentication

When logged into mongo shell, use db.auth() method to authenticate:

```
use admin
db.auth( "myUserAdmin", "PWDabc123" )
```

Validate User Roles and Privileges

Use the usersInfo command or db.getUser() method to display user information.

```
use admin
db.getUser("reportsUser")
```

Identify the privileges granted by a role

For a given role, use the db.getRole() method, or the rolesInfo command, with the showPrivileges option: For example, to view the privileges granted by read role on the products database, use the following operation,

- use week6
- db.getRole("read", { showPrivileges: **true** })
- db.getRole("readWrite", { showPrivileges: **true** })
- db.getRole("root", { showPrivileges: **true** })

Document End

Disclaimer: This document is a "best effort" based on contemporaneous MongoDB website documentation. It is not intended as a comprehensive, authoritative reference. I'm not responsible if you use this Guide and fail the certification exam. Use entirely at your own risk.

By Ross Winters, 7/12/16