M101 - MongoDB for Developers

Date: 2013-05-09

tags: mongodb

In MongoDB, there can be multiple databases. You can switch to a different database using `use <db_name>` command in mongo shell. If the database doesn't exits, it will be automatically created and selected.

A database can have multiple collections. And each collections can have multiple documents which are basically BSON (like JSON) objects (dicts or key-value associations). BSON is a super-script of JSON syntax. Apart from JSON elements, BSON has few other stuff like binary data storage, UTC datetime, unique `ObjectId`, regex, etc.

In Mongo, different documents can have different schema, so it is schemaless.

All the documents in a collection (coll_1) can be retrieved by - `db.coll_1.find();`.

A document can be stored in a collection (coll_1) by - `db.coll_1.save({name:"Lola", favs: ["harry potter", "kill bill"]});`.

One document can be fetched using `findOne()` method. E.g. `db.coll_1.findOne()`.

Output can be prettified using `pretty()` method on any query. E.g. `db.coll_1.find().pretty()`.

When a document is inserted, MongoDB required that every document has an `_id` field in it. It is a primary key field therefore it is unique and immutable. It is an ObjectId instance. If `_id` isn't present while insertion, an ObjectId instance will be created and inserted with it. Any other value such as number, string, etc can be used for `_id` field as well but it has to be unique. If an object with a suplicate `_id` field is inserted, it overwrites the original document.

CRUD Operations

CRUD - Create, Read, Update and Delete.

In MongoDB terms => Insert, Find, Update and Remove.

MongoDB doesn't has its own language analogous to SQL. MongoDB CRUD operations exist as method/functions in programming language APIs.

Mongo shell is an interactive javascript interpreter.

When a statement such as `db.coll_1.findOne()` is run, here `db` is a handle for the currently selected database.

Inserting Docs

```
db.coll_1.insert({name: "Booker DeWitt", mission: "Save Elizabeth"});
```

findOne operation

`db.coll_1.findOne();` gives back a random document from the collection.

We can also give arguments to findOne. E.g. `db.coll_1.findOne({name: "Lola", subject:"CS"});` will find one document with field `name: "Lola"` and `subject:"CS"` in it and nothing if no result is found.

A second argument can also be given, which specifies which fields we want back in the result. E.g. `db.coll_1.findOne({name: "Lola"}, {name: true, profession: true, _id: false});`.

Above query will return a document with `name: "Jones"` field in it and the document will only contain fields `name` and `profession` in it. By default, `_id` is set to true, but we can hide `_id` by setting it to false.

find Operation

`db.coll_1.find();` returns all the documents present in the `coll_1` collection. In mongo shell, 20 documents are returned in batches at a time.

`find` method can also 2 arguments just like `findOne`.

If you want to find a document in a collection and you are sure that only one document for the query exists, then use `findOne` instead, since it will stop searching for database after the first result is found. Whereas, `find` searches the entire database to find all the possible results.

$gt and $lt operations

`db.coll_1.find({ score: { $lt: 100 } });` returns all the documents with `score` field values less than 100.

`db.coll_1.find({ score: { $gt: 90 } });` returns all the documents with `score` field values greater than 90.

`db.coll_1.find({ score: { $gte: 90, $lt: 100 } });` returns all the documents with `score` field values greater than or equal to 90 and less than 100.

`db.coll_1.find({ score: { $lte: 100 }, type: "essay" });` returns all the documents with `score` field values less than or equal to 100 and `type` field value `essay`.

- `$gt` - greater than
- `$lt` - less than
- `$gte` - greater than or equal to
- `$lte` - less than or equal to

These operations also work on strings (uses utf8 encoding values for comparision).

Using regexes, $exists, $type operations

`db.coll_1.find({profession: {$exists: true} });` will return all documents in which `profession` field is present.

`db.coll_1.find({name: {$type: 2} });` will return all documents in which `name` field only has string value.

Numeric encoding for various data types as specified in BSON spec are used for querying for specific data types.

`db.coll_1.find({name: { $regex: "^Ta" });` qill return all documents in which `name` field starts with "Ta" string. Any regex pattern can be used here.

$or, $and operation

`db.coll_1.find({ $or: [ {name: {$regex: "e$"}}, {age: {$exists: true}} ] });` will return all documents where `name` field ends with string "e" or `age` field is present.

`$or` is a prefix operator. It takes separate queries in an array and returns documents which match any of the separate queries. In effect, it performs union of queries.

`$and` works very similar to `$or` operator.

`db.coll_1.find({ $and: [ {name: {$gt: "C"}}, {profession: "Engineer"} ] });` will return all the documents where name is greater than (starts with) letter C and `profession` field is "Engineer".

But `$and` isn't frequently used since the above operation can be writter in a simpler manner such as `db.coll_1.find({name: {$gt: "C"}, profession: "Engineer"});`

Querying inside arrays with $all and $in operators

`db.coll_1.find({favorite: "beer"});` will normally find all the documents in which field `favorite: "beer"` exists. But if the `favorite` field is an array, then it will look inside array if "beer" exists.

If we want to look for multiple values in an array, `$all` operator can be used.

`db.coll_1.find({favorite: {$all: ["beer", "cheese"] }});` will find documents in which `favorite` field is an array and it contains both "beer" and "cheese" elements. It can contain other elements as well.

If we want to look for either of the multiple values in an array, `$in` operator can be used.

`db.coll_1.find({favorite: {$in: ["beer", "cheese"] }});` will find documents in which `favorite` field is an array and it contains either "beer" or "cheese" elements.

Querying for nested documents with dot notation

Suppose we have a document `{name: "Lola", email: {work: "work@lola.com", personal: "me@lola.com"}}` and we want to find documents with work email address to be "work@lola.com", following query with dot notation can be used.

```
db.coll_1.find({ email.work: "work@lola.com"});
```

Cursors

When a query is executed, a cursor is contructed and returned. Interactive shell is configured to print documents by iterating over the cursor. But a cursor can be hold onto with `cur = db.people.find(); null;`. `null;` is tacked on to prevent printing the elements. `cur.hasNext()` returns true if there is document present next. `cur.next()` returns the next document. `find()` method can be implemented as `while (cur.hasNext()) printjson(cur.next());`.

A limit can be imposed on a cursor by appending `cursor()` method over it. For example, on above cursor `cur.limit(5); null;` or on a query `db.coll_1.find().limit(5);`.

We can also get sorted results by appending `sort()` method over the query. For e.g., on above cursor `cur.sort({name: -1}); null;` or on a query `db.coll_1.find().sort({name: -1});`. This will return documents reversely sorted by the value of `name` field.

If we want to skip certain number of elements, `skip()` method can be used. For e.g. on above cursor `cur.skip(5); null;` or on a query `db.coll_1.find().skip(5);`. This will skip first 5 documents and then return the rest.

Sorting, skipping and limiting can be done at the same time e.g., `db.coll_1.find().sort({score: -1}).skip(5).limit(10);`.

All these three operation occur on the server-side in MongoDB.

If we want to count the number of documents we can get as a result for a query, simple append the method `count()` at the end of the query.

```
db.coll_1.find({first_name: "Joe"}).count();
```

Wholesale updating of a document

`update` method in MongoDB can actually do 4 different things. It takes atleast 2 arguments.

`db.coll_1.update({name: "Jon"}, {name: "Lola", job: "Engineer"});` will find a document with key `name: "Jon"`, delete everything in it except `id` field and then insert fields `name: "Lola"` and `job: "Engineer"` in it. This is a dangerous way of updating records since it deletes previous data.

Using $set and $inc command

`db.coll_1.update({name: "Jon"}, {$set: {job: "Engineer"}});` will find a document with key `name: "Jon"` and update just the `job` field (or create if it doesn't exists) leaving rest of the data intact.

`db.coll_1.update({name: "Jon"}, {$inc: {age: 1}});` will find a document with key `name: "Jon"` and increment just the `age` field by 1 (or create if it doesn't exists) leaving rest of the data intact.

These are much safer way to update any field in a document.

Using $unset command

`db.coll_1.update({name: "Jon"}, {$unset: {job: 1}});` will find a document with key `name: "Jon"` and remove just the `job` field, leaving rest of the data intact.

Manipulating arrays inside documents

Suppose we have an object `{_id: 0, a: [1, 2, 3, 4]}`. Following array manipulations can be done.

`db.coll_1.update({_id:0}, {$set: {"a.2": 5}});` --> `{_id: 0, a: [1, 2, 5, 4]}`. Changes `a[2]` element to 5.

`db.coll_1.update({_id:0}, {$push: {"a": 5}});` --> `{_id: 0, a: [1, 2, 5, 4, 6]}`. Pushes value to the end of the array.

`db.coll_1.update({_id:0}, {$pop: {"a": 1}});` --> `{_id: 0, a: [1, 2, 5, 4]}`. Pops value out from the end of the array.

`db.coll_1.update({_id:0}, {$pop: {"a": -1}});` --> `{_id: 0, a: [2, 5, 4]}`. Pops value from the beginning of the array.

`db.coll_1.update({_id:0}, {$pushAll: {"a": [7, 8, 9]}});` --> `{_id: 0, a: [2, 5, 4, 7, 8, 9]}`. Pushes all the values to the end of the array.

`db.coll_1.update({_id:0}, {$pull: {"a": 5}});` --> `{_id: 0, a: [2, 4, 7, 8, 9]}`. Deletes the specified value from the array, no matter the position.

`db.coll_1.update({_id:0}, {$pullAll: {"a": [2, 4, 8]}});` --> `{_id: 0, a: [7, 9]}`. Deletes all the specified values from the array, no matter the position.

`db.coll_1.update({_id:0}, {$addToSet: {"a": 5}});` --> `{_id: 0, a: [7, 9, 5]}`. `addToSet` treats array as a set and adds the value only if it is not already present.

Upserts

Suppose we want to update a document if it exists or if it doesn't exists, then create one with provided information.

```
db.coll_1.update({"name": "Elizabeth"}, {$set: {destination: "Paris"}});
```

The query above will do nothing if there is no document with name field set to "Elizabeth". However, `upsert` flag can be set to true if we want a new collection to be created if it doesn't already exists.

```
db.coll_1.update({"name": "Elizabeth"}, {$set: {destination: "Paris"}}, {upsert: true});
```

The above query will create a new document `{name: "Elizabeth", destination: "Paris"}` if it didn't exist.

Multi-update

By default, all the update operation affects only one document. We can however set `multi` flag to true if we want multiple documents to be updated.

`db.coll_1.update({job: "Doctor"}, {$set: {title: "Dr"}}, {multi: true});` will find all the documents having field `job: "Doctor"` and set field `title: "Dr"` in it.

Removing data

`db.people.remove({name: "Comstock"});` removed all documents which had field `name: "Comstock"` in it.

All documents in a collection can be removed by `db.coll_1.remove();` or a much faster way `db.coll_1.drop();`.

getLastError

`getLastError` is a very helpful command which can give information about the last query whether it failed or succeeded with a document containing the error message and other details such as whether existing documents were affected during an `update` or not, how many documents were affected, etc.

```
db.runCommand({getLastError: 1});
```

Indexes

Indexed can be used to speed up the lookup operation in a collection. For example, if we have a collection with million of user's data and we use `username` as primary key, an index on the collection can be generated by `db.coll_1.ensureIndex({username: 1});`. The value indicates ascending order sorting while index creation.

Multiple keys can also be used to generate compound index. For example `db.coll_1.ensureIndex({a:1, b:1, c:1});`. Following are different combinations of queries and whether index is used for them -

- `a, b`, `a, b, c` - Yes
- `a` - Yes
- `b, c` - No
- `c, b`, `b, a` - No
- `a, c` - Yes (only a part index is used)

Don't generate index for every possible type of query you might make. Instead just create index for most common type of queries. Indexes are not costless, as they take up more memory space and have to be additionally updates everytime something is changed.

All the present indexes in a database can be looked up using `db.system.indexes.find()`. All the present indexes in a collection can be looked up with `db.coll_1.getIndexes()`.

Index can be dropped with `db.dropIndex({username: 1});`.

Multikey Index

If we want to make an index for a key which contains array values, then MongoDB makes indexe for each of the value in the array which is called Multikey index. However, a multikey compound index with two or more parallel arrays cannot be created.

Unique Index

The key in an index can be forced to be unique so that none of the values for the index keys are repeated. It can be done by providing a second argument while creating the index. For example, `_id` key index is a unique one.

```
db.coll_1.ensureIndex({username:1}, {unique: true});
```

If we want to create a unique index on a collection but it already has some duplicates, then we can remove the duplicates while creating index. Remember that this is dangerous since you can't control which documents are deleted.

```
db.coll_1.ensureIndex({username: 1}, {unique: true, dropDups: true});
```

Sparse Index

If we want to create a unique index with a key on a collection but not all the documents in the collection have that key then it would create a problem.

For e.g., `{a: 1, b:2, c: 3}, {a: 4, b: 5}, {a: 6, b: 7}` are the three documents in a collection. If we want to create a unique index for key `c` but last two documents don't have it and MongoDB by default assume them to have `c: null` value. Now, since both of them have `c: null` value, it would be considered as duplicate and thus, unique index can't be created on them.

Sparse Index can solve the problem. `db.coll_1.ensureIndex({username: 1}, {unique: true, sparse: true});`

This index will include only the documents which have the `username` key set to some value.

However, Sparse Index can create some weird artifact with some queries, especially with sort. For e.g. `db.coll_1.find().sort({username: 1});` should include all the documents in the collection but the actual result will include only the ones in the index. Queries in MongoDB always try to use an index when present, so here the sort part of the query uses the sparse index which results in incorrect output.

Background Index Creation

By default, indexes are created in foreground which is faster but it blocks all the other writers. But if it is essential to prevent the blockage of the other writers such as in production environment, then background creation can be preffered which doesn't blocks other writers but it a bit slower. Background creation can be done via providing `background:1` in the second argument.

Using Explain

We can find out whether the queries we perform use an index or not. Appending `explain()` method on the end of a query gives us information about whether index was used or not, which index was used, how many objects were scanned, how many results are there, how many seconds it took, etc.

Choosing an Index*

If there are multiple indexes for a key, then MongoDB has to decide which one to use (only one can be used for a query). What it does is that it runs benchmark for all the indexes internally and uses the one which took least amount of time.

Collection and Index Stats

`db.coll_1.stats()` can be used to get the statistics of collection which includes the total collection size, average document size, total index size and individual index sizes.

`db.coll_1.totalIndexSize()` can be used to get the total index size for a collection.

Hinting an Index

We can manually specify which index to use by appending `hint()` method to the query. For e.g., `db.coll_1.hint({username: 1});` or if we want no index to be used for the query, then `db.coll_1.hint({$natural: 1});`

Profiling

There are 3 levels of logging-

- 1 - Off
- 2 - Log slow queries
- 3 - Log all queries

By default, all the queries that take more than 1000ms are logged into the `mongod` instance's log. This value can be changed to, for e.g. 2ms by running `$ mongod --profile 1 --slowms 2`.

Logs can be checked using `db.systems.profile.find();`.

Logs for specific collections cab be done like this - `db.systems.profile.find({ns: /dbName.collName/}).sort(ts: 1);`. This query will find logs for collName collection in dbName database and sort it by timestamp.

Current profiling level can be checked using `db.getProfilingLevel();` and status using `db.getProfilingStatus`.

Profiling status can be set using `db.setProfilingLevel(2, 10)`. This will change profiling level to 1 and log queries that take more than 10ms.

mongotop and mongostat

Similar to unix's top program which give high level view and gives info about which collection are taking how much resource. mongotop takes one argument which is the time interval to refresh the data in seconds.

mongostat is similar to unix's iostat command. It shows all the different type of operations happening, database size, number of connection. One of the interesting thing it shows is `idx miss %` which tells the % of queries where the index was used but it has to hit the disk since there wasn't enough memory to accomodate the index.

Aggregation

Example of an aggregation query -

```
db.products.aggregate( [ { $group: { _id: "$manufacturer", num_products: {$sum: 1} } } ]
```

Here, the aggregate method takes an array as the argument. It contains different aggregation queries. Here we group by manyfacturer name and then sum the number of products. The value to `$group` key is the schema for the result. The result will contain `_id` field with the manufacturer's name and `num_products` field with number of respective total products. For every product in a group, it will add 1 (which we specified) to the `num_products` field.

There are various stages for any aggregation query and results can be pipelined through these stages -

- $project
- $match
- $group
- $sort
- $skip
- $limit
- $unwind

Compound Grouping

```
db.products.aggregate( [ { $group: { _id: {manufacturer: "$manufacturer", category: "$cat
```

Aggregation Expression Overview

- $sum
- $avg
- $min
- $max
- $push

- $addtoSet
- $first
- $last

Using $sum

```
db.products.aggregate( [ { $group: { _id: "$manufacturer", sum_prices: {$sum: "$price"} }
```

Here, for every product in the group, its `price` field is added to the resulting document's `sum_prices` field. So, the `sum_prices` field will contain the total price for products for every manufacturer.

Using $avg

```
db.products.aggregate( [ { $group: { _id: "$category", avg_price: {$avg: "$price"} } } ]
```

This will show the average price for each category.

Using $addToSet

```
db.products.aggregate( [ { $group: { _id: "$manufacturer", categories: {$addToSet: "$cate
```

The resulting document will contain keys named `categories` which will contain the name of the categories (in a set (array with unique elements)) for the corresponding manufacturer.

Using $push

```
db.products.aggregate( [ { $group: { _id: "$manufacturer", categories: {$push: "$category
```

Similar to `%addToSet` but doesn't makes a set with unique items, rather makes a normal array which might contain duplicates.

Using $max and $min

```
db.products.aggregate( [ { $group: { _id: "$manufacturer", max_price: {$max: "$price"} }
```

Finds the maximum price for the product by the manufacturers.

Double $group stages

```
db.grades.aggregate([ {$group: {_id: {class_id: "$class_id", student_id: "$student_id"},
```

This will pipe the result from first aggregation to the second one.

$project

`$project` is used for reshaping/projecting the document as they come through the pipeline. Its a 1:1 stage of the pipeline, same number of document leave the `$project` phase which come in. It can -

- remove keys
- add new keys
- reshape keys
- **use some simple functions on keys**
    - $toUpper
    - $toLower
    - $add
    - $multiply

```
db.products.aggregate([{
    $project: {
        _id: 0,
```

```
        maker: {$toLower: "$manufacturer"}, // manufacturer name to lowercase
        details: {category: "$category",
                  price: {$multiply: ["$price", 10]}} // 10 x'es the price
        },
        item: "$name"
}]);
```

$match

It acts as a filter for the incoming documents. E.g. -

$sort

`$sort` operation happens inside the memory.

$limit and $skip

Very similar to how how sort, skip and limit work with `find` method queries.

$first and $last

As the names suggest, these will help get the the first and the last documents. E.g., if we want to find the largest city in each state -

```
db.zips.aggregate([
    {$group: {_id: {state: "$state", city: "$city"}, population: {$sum: pop}}},
    {$sort: {"_id.state": 1, population: -1}},
    // since the document was sorted by descending order of population, highest population city is at the top
    {$group: {_id: "_id.state", city {$first: "$_id.city"}}} // get the top/first city
]);
```

$unwind

Unwind can be used to unwind an array inside the document. After an unwind operation on a key containing array, new documents are formed with the remainder of the document and each elements from the array.

`db.coll_1.aggregate([{$unwind: $b}]);`

`{a: 1, b: [2, 3]}` will get unwinded to `{a:1, b:2}` and `{a:1, b:3}`.

Limitations of the Aggregation Framework

- Result document limited to 16MB of memory
- Limited to use only 10% of the memory of the machine
- In a sharded environment, any aggregation query is brought back to `mongos` instance after the first grouping by a `mongod` instance which might affect performance on the machine running `mongos` if the data set is huge.

Alternatives to the aggregation framework - mapreduce, hadoop.

Application Engineering

Write Concern

In the mongo shell, every time a query is made, the shell calls the `getLastError` method to see if it succeeded or failed. But while using drivers such as PyMongo, this is not the default behavior. We have to set the safe mode values to make the driver check if there was any error. It takes two parameter - w and j. Following are the different cases -

- w 0, j 0 - fire and forget, doesn't checks for errors
- w 1, j 0 - acknowledge that the query was received but doesn't checks if it succeeded (default for the drivers)

- w 1, j 1 - commit to journal, means that query was received, there is no error and it is fault tolerant in case of power loss since the query stored in the journal can be used for recovery (recommended)

Network Error

If the application sends the query to mongod over the network then there is a possibility of uncertainity whether the query was completed or not due to network error. For e.g., if the query was made, mongod received it but then suddenly the network went down. The mongod may have executed the query and but the application didn't receive any acknowledgement so it has no idea whether the query succeeded or not. Thogu application can later the database to see if the change was made or not but still there is a certain level of uncertainity.

Replication

To maintain availability and fault tolerance in case of downtime of a mongod server, replica sets are used. They mirror the same data and synchronize it asynchronously. At least 3 mongod servers are required in a replica set. One of them is primary to which the application or mongos talks and rest are secondary. If the primary one goes down, then election is done by the rest and one is chosen are primary. When the ex-primary server comes back up, it joins the set as a secondary server.

Type of Replica Set Nodes -

- Regular - Normal ones which can take place of the primary one if it goes down
- Arbiter - Just present for voting purposes.
- Delayed/Regular - Is usually a few hours behind the primary node and is present for disaster recovery. It cannot participate in voting and cannot become a primary node.
- Hidden - Used for different purposes, e.g. analytics. It cannot become the primary node but can participate in the voting.

Write Consistency

For a strong consistency between read and writes, it is recommended to read from and write to only the primary node. Though it is possible to do read operation from secondary nodes (by running `rs.slaveOk();` on them) but there is a chance that you might get stale data if the synchonization didn't occur fast enough.

Create Replica Set

Here for the sake of example, we will make the replicas sets using the following command on our single machine-

```
mkdir -p /data/rs1 /data/rs2 /data/rs3
mongod --replSet rs1 --logpath "1.log" --dbpath /data/rs1 --port 27017 --fork
mongod --replSet rs1 --logpath "2.log" --dbpath /data/rs2 --port 27018 --fork
mongod --replSet rs1 --logpath "3.log" --dbpath /data/rs3 --port 27019 --fork
```

`--replSet rs1` makes sure that all the three instances belong to same replica set and `--fork` makes it run in the background so that 3 different shells are not required. At this point, all 3 instances do not know about each other.

Now we need to create a configuration to make sure that they work in co-operation.

```
config = {
    _id: "rs1", // replSet we used to run the instances
    members: [
        // We don't want host 0 to be the primary node, so setting priority 0
        // And we are delaying it by 5 seconds.
```

```
        {_id: 0, host: "vivekagr.local: 27017", priority: 0, slaveDelay: 5},
        {_id: 1, host: "vivekagr.local: 27017"},
        {_id: 2, host: "vivekagr.local: 27019"},
    ]
};
rs.initiate(config);
rs.status();
```

We need to open mongo shell and connect to either port 27018 or 27019 and not 27017 since we can't run the configuration step on a node that cannot become a primary node.

Replication internally works by using a capped collection `oplog.rs` in `local` database. Secondary nodes query primary to check if any new data has been added to the `oplog.rs` and then it is synchronized.

Failover and Rollback

Suppose if the secondary nodes are lagging few seconds behind the primary node and suddenly primary node goes offline then one of the secondary servers will be promoted to primary position but it won't have the writes for last few seconds. Now, when the ex-primary node comes online and while synchronizing data, it sees that it has writes which were not synchronized with other nodes, it will rollback that data and write it to a rollback log.

If an application is writing or reading data during a failover and election situation, exception will occur so it necessary to catch such exception and handle the situation accordingly.

Revisiting Write Concern

Suppose we have 3 nodes in total - 1 primary and 2 secondary. Now consider the following vaues of w and j.

- `w = 1` - wait for just nodes (primary) to acknowledge the write
- `w = 2` - wait for two nodes (primary & one secondary) to acknowledge the write
- `w = 3` - wait for three nodes (all) to acknowledge the write
- `j = 1` - wait for only primary to write it all the way down to disk and journal

`wtimeout` refers to the maximum time to wait.

These values can be set in either of the three ways - while configuring the replica set, while making connection to the mongod or inside the collection itself.

`w: majority` can also be set and is considered to be the best practice. It avoids having the data rolled back in case of a single node failure. Take the three node example. If you set w:majority, then at least one other node will have the date at the time of failover. That node will be preferred to take over as primary. The node that is furthest ahead will be preferred in the election of a new primary.

Read Preferences

There are various read preferences that can be set -

- primary
- secondary
- secondary preferred
- primary preferred
- nearest
- tagging

Sharding

To get horizontal scalability, we break up the database on to multiple logical hosts and that is done according to a shard key. Shard key is some part of the document itself (usually a unique key for the collection). There has to be an index present beforehand for the key which is going to be used as the shard key.

Building a Sharding Environment

See `init_sharded_env.sh` file for the process and code.

Implications of sharding on development

- Every query should include a shard key
- Shard key is immutable
- Index that starts with the shard key is required
- For update commands, eithe rshard key has to be provided or multi key set to true (which results in the broadcast of the query to all the nodes)
- No shard key in query means that query is broadcasted to all nodes which is inefficient
- No unique key unless part of the shard key because mongos has no way of knowing that whether the copy exists on other shards since each shard has its own set of unique keys