Mongo Shell Command Line Interface

1. show dbs or show databases: Shows databases.
2. use <database-name>: Switches to (or creates) DB *database-name*. Newly created DB won't be listed in o/p of show dbs unless it contains at least one collection with data in it.
3. db.dropDatabase(): Drops the current DB.
4. show collections: Displays collections within current DB.
5. db.createCollection('<collection-name>'): Creates new collection within current DB.
6. db.<collection-name>.drop(): Drops the collection *collection_name*.
7. db.<collection-name>.insertOne({field1 : val1, field2 : val2}): Inserts one doc in collection. Even if collection is not created yet, this command creates it when run. MongoDB assigns insertID to every document (as ObjectId) when it is created. Same for insertMany ahead.
8. Collection names, & their field names, are case sensitive.
9. db.<collection-name>.insertMany([{field1 : val1, field2 : val2}, {field1 : val1, field2 : val2}, ...]) : Inserts many docs in the collection as a list (array) of dictionaries.
10. If a field name has special chars/spaces/starts with a number, using quotes is necessary. If field name is reserved keyword in MongoDB, quotes distinguish it from reserved keyword.
11. Default **insert behavior** is ordered (docs are added in order of writing; processing stops when first error occurs).
12. db.<collection-name>.insertMany([doc1, doc2], {ordered : false}) : When executing bulk write operations with flag ordered : false set, MongoDB keeps processing even after encountering error (i.e. erroneous docs are skipped). If flag is not set, it is taken as default insert behavior (which is ordered) and processing will stop post first error.
13. db.collection_name.find({key : value}): Returns all matched pairs.
14. db.collection_name.findOne({key : value}): Returns only FIRST matched pair.
15. **Import JSON file in Mongo Shell through CMD**: In CMD, change directory to folder where json files are present for import. Then type the following commands:
    mongoimport file_name.json --db db_name --collection collection_name
    E.g. mongoimport products.json --db shop --collection products
16. mongoimport file_name.json --db db_name --collection collection_name --jsonArray: Attribute --jsonArray accepts import of data in multiple MongoDB documents within a single JSON array (i.e. list).
17. Both mongoimport and mongoexport are command line database tools of MongoDB.
18. **Importing file from local device into an Atlas collection**: For this, database and collections within it must already be created in Atlas. Then, in Atlas, go to Clusters → Cmd Line Tools → (Scroll down to) Data Import and Export Tools. Copy the mongoimport command there.
    Next, open CMD. Change directory to location where files are present (e.g. *D:\Code\MongoGym*). Paste the copied mongoimport command:
    mongoimport --uri mongodb+srv://echo:<PASSWORD>@cluster-echo.kzjn1ga.mongodb.net/<DATABASE> --collection <COLLECTION> --type <FILETYPE> --file <FILENAME>
    Now, edit the bold fields in the above command (PASSWORD left deliberately). E.g.:
    mongoimport --uri mongodb+srv://echo:<PASSWORD>@cluster-echo.kzjn1ga.mongodb.net/shop --collection products --type json --file products.json
    If file contains multiple lists, append --jsonArray at the end of above command. Hit ↵ and products.json will be imported from device into Atlas. Repeat the command for all files to be imported.
19. Max file size for importing is 16mb.
20. mongoexport –d db_name –c collection_name –o <export location path>: -o is for *out*. Documents are not necessarily exported in the order of their storage in collection.
21. **Exporting from Atlas to local device**: Most of the steps are same as importing to Atlas covered previously, only the command changes from mongoimport to mongoexport:
    mongoexport --uri mongodb+srv://echo:<PASSWORD>@cluster-echo.kzjn1ga.mongodb.net/shop --collection sales --type json --out salesTest.json
22. **Comparison operators**: $eq, $ne, $gt, $gte, $lt, $lte, $in, $nin.
23. db.collection_name.find({'fieldName' : {$operator : value} }): using operators.
24. db.products.find({ 'price' : {$in : [10, 29, 699] }}): example.
25. db.products.find({ 'price' : {$eq : 699} }).count(): Returns count of matched records.
26. **Cursors** in MongoDB help efficiently retrieve large result sets (usually in batches of 101 docs) from queries. Cursor is a pointer to result set on server & can iterate through query results.
27. Cursor methods: count(), limit(), skip(), sort().
28. db.products.find({ 'price' : {$eq : 699} }).limit(5): Returns 5 documents.
29. db.products.find({ 'price' : {$eq : 699} }).limit(5).skip(1): Return 5 docs post skipping top 1.
30. **Logical operators**: $and, $or, $not, $nor.
31. db.products.find({$and : [{'price':{$gt : 200}}, {'name' : 'Diamond Ring'}]}): using $**and**.
32. Multiple fields within single query document are treated as implicit **AND** operation, so above query is same as db.products.find({'price':{$gt : 200}}, {'name' : 'Diamond Ring'}).
33. $nor returns all records that don't meet any of the conditions specified.
34. **Complex expressions**: The $expr operator allows using aggregation expressions within a query. Useful when you need to compare fields from the same document in a more complex manner. { $expr : {operator : [field, value] } }.
35. E.g. db.products.find({ $expr : {$gt : ['$price', 1340] } })
36. **Exists**: db.products.find({price: {$exists : false}, price:{$gt:1300}}). Here, those records are returned that do not have price field **OR** whose 'price' is greater than 1300. Quite opposite to the AND implicit in point-30 above.
37. **Type**: db.products.find({price: {$type: 'number'}}).count() → Returns count of those documents where price is of data type *number*.
38. **Size**: db.comments.find({'comments' : {$size: 4}}). Returns all documents in 'comments' collection having 'comments' field with 4 documents within an array.
39. **Projection** enables to retrieve only desired fields of a document (and not entire doc). Use projection value 1 for fields to be included (and value 0 to be excluded) from results. Same fields can't be both included & excluded within a query projection. E.g.:
    db.comments.find({'comments' : {$size: 4}}, **{'comments' : 1}**). Here only comments field is projected into results.
    db.comments.find({'comments' : {$size: 4}}, **{'comments' : 1, 'author' : 1}**). Here fields 'comments' and 'author' are projected into results. **{comments : 1, author : 1}** works too.

40. **Embedded docs** within docs are accessed using dot notation. E.g. db.comments.find({'comments' : {$size: 4}}, {'comments.user' : 1}) returns 'user' within 'comments'. In general: db.collection.find({parent.child : value})
41. Finding all comments having users Alice and Vinod in them: db.comments.find({'comments.user': {$in : ['Alice', 'Vinod']}}), this would include other users too. To strictly limit to Alice and Vinod, replace $in by $all.
db.comments.find({'comments.user': {$elemMatch : {'user': 'Vinod', 'text' : 'Thanks for sharing.'}}}). $elemMatch matches with the elements (*'user', 'text'*) of the field (*'comments'*) of the document.
42. Updating docs
db.products.updateOne({filter},{query}) -- updates the topmost matching doc
db.products.updateOne({name : 'Designer Handbag'}, {$set : {isFeatured : true}})
db.products.updateMany({filter},{query}) -- updates the all matching docs
db.products.updateMany({price : 120}, {$set : {isFeatured : true}})
43. **Removing fields**: updateOne({filter}, {$unset : {fieldName : 1}})
44. **Renaming fields**: updateOne({filter}, {$rename : {oldFieldName : 'newFieldName'}})
45. Above two queries work for updateMany as well.
46. **Updating arrays & embedded docs**: ...({filter}, {$push : {arrayField : 'newElement'}}), ...({filter}, {$pop : {arrayField : value}}), ...({filter}, {$set : {'arrayField.$.text' : 'updatedText'}}).
$set adds new field-value pair; $push adds new elements within field.
Add new entry 'user': 'Jessy' in 'comments' field: db.comments.updateOne({'_id':3}, {$set : {'user': 'Jessy'}}). Following line removes this newly added entry:
db.comments.updateOne({'_id':3}, {$unset : {'user': 1}}). (Same as **Deletions** below)
Here's how new elements are added within 'comments' field:
...updateOne({'_id':3}, {$push : {'comments': {'user':'Eva', 'text': 'THIS IS EVA.'}}}).
Drops last item in 'comments' field: ...updateOne ({'_id':3}, {$pop : {'comments': 1}}).
Access document where id = 7 and 'user' in 'comments' field is Alice: db.comments.find({'_id':7, 'comments.user' : 'Alice'}).
Update the 'text' within 'comments' field for Alice:
...updateOne({'_id':7, 'comments.user' : 'Alice'}, {$set : {'comments.$.text' : 'Awesome Alice.'}}).
47. **Deletions:** Delete doc with _id=1: db.comments.deleteOne({'_id' : 1})
Delete all docs with price=55: db.comments.deleteMany({'price' : 55})
48. **Indexes**: store fraction of data in more searchable format; speedy data retrieval. Indexes are separate from collections and multiple indexes can exist per collection; facilitate sorting based on specific fields. Efficient aggregation operations with optimized indexes. Can do multiple fields with indexes.
49. Glance at **query plan**: db.comments.find({'price': {$gt : {100}}}).explain('executionStats'). Can run without specifying executionStats too.
50. **Create index**: ...createIndex({field : 1}) [1/-1 for storing index in asc/desc orders.]
See indexes on a collection: ...getIndexes(). Note: _id is default index.
Drop indexes: ...dropIndex({field : 1}) OR ...dropIndex('index_name')
Finding using **regex**: db.products.find({'name': {$regex : 'air'}}).count()
51. **Aggregations** consists of multiple pipeline stages, each stage operating on its input data. Aggregations help transforming/reshaping data and make complex calculations possible.
$match: filters docs based on specified conditions. ...aggregate([$match : {name: 'Tom'}])
$group: group by field to perform aggregation (like GROUP BY in SQL). E.g.
...aggregate([{$group : { _id : '$company', totalSales : {$sum : 1 } } }])
...aggregate([{$group : { _id : '$company', totalSales : {$sum : '$price' } } }])
Find sum of prices for each company having price > 900:
...aggregate([{ $match : { price : { $gt : 900 } } }, { $group : { '_id' : '$company', totalPrice : { $sum : '$price' } } }])
Another query: ...aggregate([{ $match : { price : { $gt : 900 }}},
{ $group: { '_id' : '$company', totalPrice : { $sum : '$price' }, totalProducts : { $sum : 1}, avgPrice : { $avg : '$price' } } }])
52. **Sorting**: ...aggregate([{ $group : { '_id' : '$quantity', totalPrice : { $sum : '$price'} },
{ $sort : { totalPrice : 1 } }])
53. **Project**: db.products.aggregate([{$project : {discountedPrice : {$subtract : ['$price', 5] }}}]).
This subtracts 5 from each 'price' value. [*$sum, $subtract, $multiply, $avg*]
Another e.g.: ...aggregate([{$project : {price:1, '_id': false}}, {$sort: {price:-1}}])
54. **Push**: db.products.aggregate([{ $match : { price : { $gt : 1300 }}},
{ $group : { _id : '$price', allColors : {$push : '$colors'} }}])
55. **$Unwind**: deconstructs an array field and produces multiple documents.
db.products.aggregate([{ $unwind : '$colors'}, { $group : { _id : '$company', allColors : {$push : '$colors'} }}]) → allColors may have duplicated values. To keep distinct values, use $addToSet instead of $push.
56. **$Size**: returns length of an array field. db.products.aggregate([{$unwind:'$colors'}, {$match : {price : {$gt : 1200}}}, {$group: {_id:'$company', allColors : {$addToSet : '$colors'}}}, {$project: {colorLength : {$size: '$allColors'}}}])
57. MongoDB Atlas is MongoDB's fully managed cloud database service.
It offers an easy way to deploy, manage, and scale MongoDB databases in the cloud.
Atlas eliminates the need for manual setup & maintenance, allowing devs to focus on their applications; it scales automatically to accommodate growing workloads.
Atlas supports global clusters, enabling databases to be deployed across multiple regions for better availability and reduced latency. Hierarchy of Atlas deployment:
Organization → Clusters → Projects → Databases → Collections → Documents
Use Atlas for free by signing-in into it.
58. MongoDB Compass provides GUI for doing whatever can be done in Mongo shell.