Notes on MongoDB

Sizhe Liu

Version 1.0

Contents

1 Basic CRUD Operation

△ Question
How to get MongoDB running?

In first command window

```
mongod
```

or

```
mongod --dbpath "/path/to/db/"
```

We can also run mongodb using a customized port by

```
mongod --port xxxxx
```

Then open another command window

```
mongo --port xxxxx
```

Now, use "cls" to clear the screen.

△ Question
How to check the occupied space of the data?

```
show dbs
```

△ Question
How to create a new database?

```
use xxxx
```

where the "xxxx" is your database name.

△ Question
How to create a collection of your db and insert one entry?

```
db.collectionname.insertOne({xxxxxxx})
```

where "xxxx" is the content of the inserted entry.

△ Question
A sample JSON file?

```
[
  {
    "departureAirport": "MUC",
    "arrivalAirport": "SFO",
    "aircraft": "Airbus A380",
    "distance": 12000,
    "intercontinental": true
  },
  {
    "departureAirport": "LHR",
    "arrivalAirport": "TXL",
    "aircraft": "Airbus A320",
    "distance": 950,
    "intercontinental": false
  }
]
```

△ Question
How to show all entries elegantly?

```
db.collectionname.find().pretty()
```

△ Question
Do we have to use the same schema in the same collection?

No. That is one big difference between noSQL and SQL.

△ Question
How to add object id manually in JSON file?

```
{
"name1": val1,
"name2": val2,
"_id":xxxxxxx}
```

△ Question
Overview of CRUD operations?

```
//Create
insertOne(data,options)
insertMany(data,options)
//Read
find(filter,options)
findOne(filter,options)
//Update
updateOne(filter,data,options)
updateMany(filter, data, options)
replaceOne(filter,data,options)
//Delete
deleteOne(filter,options)
deleteMany(filter,options)
```

△ Question
How to delete one documents using a filter?

```
db.collectionname.deleteOne({key:val})
```

△ Question
How to delete many documents using a filter?

If the docs do not have common keys, we need to add a common key for the files we want to delete, using update commands

```
db.collectionname.updateOne({distance: 12000}, {$set: {marker: "delete"}})
db.collectionname.updateMany({}, {$set: {marker: "delete"}})
```

where "$set" is the fixed command to setup a new key in the fist file found with key of "distance" equal to 12000.

The second command is to create new field "marker" in all the files.

△ Question
How to add many documents using an array?

```
db.collectionname.insertMany([{xxxxxx},{xxxxxx}])
```

△ Question
How to find many file use filter?

```
db.collectionname.find({key1:{$gt: xxxxxx}})
```

where "$gt: xxxxxx" stands for "value of key1 that is greater than xxxxxx."

△ Question
How to use update, updateOne, and updateMany?

```
db.collectionname.update({"_id":123456}, {delayed:true})
```

This code will replace the entry with id of "123456" with {delayed:true}. Similar to the following command:

```
db.collectionname.replace({"_id":123456}, {delayed:true})
```

For "updateOne", if we want to set a new field in the curly bracket, we can do the following

```
db.collectionname.updateOne({"_id":123456}, {$set delayed:true})
```

Similar rule applies to "updateMany"

△ Question
What 'find()' gives you?

It gives you a "cursor", you can fetch all the entries by using:

```
db.collectionname.find().toArray()
```

Unlike "find()", "findOne()" does not gives a "cursor". Thus, using "pretty()" with "findOne()" will cause error.

△ **Question**
What is projection and how to use it?

Projection is a way to fine tune the data you want to fetch from your database. It creates flags to filter the information in each entry that you want or you do not need. Here is an example:

```
db.passengers.find({},{name: 1, _id:0})
{"name":1xxxx}
{"name":2xxxx}
{"name":3xxxx}
```

The code above does not apply any inter-entry filter as the first argument in "find()" is "{}". The second argument further filter the info inside each entry by allowing "name" of each entry to show and filtering "_id"

△ **Question**
What is embedded document?

An embedded document is the json documents nested inside their parent documents. Using the following code, we can update the "status" using an embedded documents:

```
db.collectionname.updateMany({},{$set:{"status":{description:"UAA", lastUpdated:"2 hours
   ago"}}})
```

where the embedded doc is:

```
{description:"UAA", lastUpdated:"2 hours ago"}
```

△ **Question**
How to fetch an entry if one of its property is an array?

Just use one of the element value in that array. Let's say the property which has an array-like value is "hobbies", we can fetch all the entry which has a hobby of "sports" by:

```
db.collectionname.find({hobbies:"sports"})
```

△ **Question**
How to fetch an entry if one of its property is an embedded doc?

Use "" to wrap the property name. Such name has a format of "xxxx.xxxx.xxxx". **The number of "." indicates layers of nested embedded docs**. The following example has an embedded doc named as "status", with one of its sub property, "details" as another doc. We can find an entry by doing:

```
db.collectionname.find({"status.details.xxxx":"xxxx"})
```

△ **Question**
How to reset your database/collection?

```
use databaseName
db.dropDatabase()
```

Similarly, for collection

```
db.mycollection.drop()
```

1.1 Solution to Some Tasks

We have an array of patients data in a form of:

```
{
"firstName":"xxx",
"lastName":"xxx",
"age":29,
"history":[
{"disease":"cold","treatment":"xxxx"},
{...}
]
}
```

Now we two entries in the collection "patients"

```
db.patients.insertMany([
{
"firstName":"Mat",
"lastName":"McTracy",
"age":49,
"history":[
{"disease":"cold","treatment":"pills"},
{"disease":"PTSD","treatment":"in-house"}
]
},
{
"firstName":"Dan",
"lastName":"Schroski",
"age":29,
"history":[
{"disease":"cold","treatment":"N/A"}
]
}
])
```

Update patient data of 1 patient with new age, name and history entry:

```
db.patients.updateOne({"lastName":"Schroski"}, {$set:{"age":55},{"firstName": Frank},{"
   history":["disease":"cough","treatment":"cough drop"]}})
```

Find all patients who are older than or equal to 30

```
db.patients.find({"age":{$gte: 30}}).pretty()
```

Delete all patients who got a cold as a disease

```
db.patients.deleteMany({"history.disease":"cold"})
```

2 Data Schemas & Data Modeling

2.1 Data types

```
Text-->"MMM"
Boolean-->true
NumberInt(int32)-->NumberInt(55)
NumberLong(int64)-->NumberLong(7489729384792)
NumberDecimal-->NumberDecimal("12.99")
ObjectID-->ObjectID("xx5637")
ISODate-->ISODate("2019-09-20")
Embedded Doc-->{"a":{...}}
Array-->{"b":[...]}
```

△ Question
how to drop the whole database?

```
use dbname
db.dropDatabase()
```

△ Question
How to drop a collection?

```
db.collectionname.drop()
```

△ Question
How to check the statistics of your DB?

```
db.stats()
```

△ Question
Why do we use data type to define values? like the following code?

```
db.numbers.insertOne({a:NumberInt(1)})
```

It returns smaller data size (33 vs. 29).

△ Question
How to check the type of a value?

```
typeof db.collectionname.findOne().keyname
```

△ Question
Important hard limits of mongoDB?

A single document in a collection (including all embedded documents it might have) must be ¡= 16mb. Additionally, you may only have 100 levels of embedded documents.

2.2 Relations

△ Question
What are the options for database relations

Embedded doc:

```
{
user:"Lil",
age: 29,
address:{city:'NYC',st:'5th ave'}
}
```

References

```
In User collection:
{
user:'Lil'
favBooks:['id1','id2']
}
In Book collection:
{
_id:'id1',
name: 'Vallery Poetry'
}
```

△ Question
How to build a many-to-many relation with embedded doc?

Use customer-order database as an example: each customer might buy different products, and each product might be bought my different customers.

```
use shop
db.products.insertMany([{_id:"idnum1",title:"Book",price:12.99}, {_id:"idnum2", title:"T-
   shirt",price:25.66}])
db.customers.insertOne({name:"Bronn", age:35})
//Now we update order info for the customer
db.customers.updateOne({name:"Bronn"},{$set:{orders:[{productid:"idnum1", quantity:2},{
   productid:"idnum2", quantity:3}]}})
```

△ Question
How to build a many-to-many relation with reference?

We can use books-authors database as an example

```
use bookEntry
db.books.insertOne({title:"F--Book", author:["author1", "author2"]})
db.authors.insertMany([
{_id:"author1", name:"Max"},
{_id:"author2",name:"Selina"}
])
```

△ Question
How to lookup to merge two collections?

Again, use book-author collections as an example

```
db.books.aggregate([{$lookup:{from:"authors",localField:"authors", foreignField:"_id", as
    :"creators"}}])
```

The code above merges ”books” collection with ”authors” collection. It says: merging the books collection **from** authors. The matching keys between these collections are ”authors” in ”books”(localField) and ”_id” in ”authors”(foreignField). The merged data will store in the field named as ”creators”.

△ Question
Build a simple user-post-comment database.

```
use blog
db.users.insertMany([
{_id:"z3id",name:"z3", age:18, email:"z3@126.com"},
{_id:"l4id",name:"l4", age:29, email:"l4@gmail.com"}
])
db.posts.insertOne({
title:"xxxx",
text:"fefjkhfdghkjkdf kferkg sfjhdnkjg",
tag:["new","tech"],
creator:"z3id",
comments:[{text:"yoyo", author:"l4id"}]
})
```

2.3 Schema validation

△ Question
How to setup validation for collection schema?

We can use ”createCollection” command and ”validator” keyword:

```
db.createCollection("collectionname",{
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      required: ['title', 'text', 'creator', 'comments'],
      properties: {
        title: {
          bsonType: 'string',
          description: 'must be a string and is required'
        },
        text: {
          bsonType: 'string',
          description: 'must be a string and is required'
        },
        creator: {
          bsonType: 'objectId',
          description: 'must be an objectid and is required'
        },
        comments: {
          bsonType: 'array',
          description: 'must be an array and is required',
          items: {
            bsonType: 'object',
            required: ['text', 'author'],
            properties: {
              text: {
                bsonType: 'string',
                description: 'must be a string and is required'
              },
              author: {
                bsonType: 'objectId',
                description: 'must be an objectid and is required'
              }
            }
          }
        }
      }
    }
  }
});
```

△ Question
How to change your validation action?

```
db.runCommand({
  collMod: 'collectionname',
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      required: ['title', 'text', 'creator', 'comments'],
      properties: {
        title: {
          bsonType: 'string',
          description: 'must be a string and is required'
        },
        text: {
          bsonType: 'string',
          description: 'must be a string and is required'
        },
        creator: {
          bsonType: 'objectId',
          description: 'must be an objectid and is required'
        },
        comments: {
          bsonType: 'array',
          description: 'must be an array and is required',
          items: {
            bsonType: 'object',
            required: ['text', 'author'],
            properties: {
              text: {
                bsonType: 'string',
                description: 'must be a string and is required'
              },
              author: {
                bsonType: 'objectId',
                description: 'must be an objectid and is required'
              }
            }
          }
        }
      }
    }
  },
  validationAction: 'warn'
});
```

Now the validation action is changed to "warn". So new entry without following the schema will still be added in the database, but a warning will be written in the log file.

△ Question
Things to consider when you structure your database?

```
//In which Format will you fetch your data
//How often will you fetch and change your data
//How much data will save
//How is your data related
//Will duplicates hurt you(==>many updates)
//Will you hit Data/storage limits
```

△ Question
How to run mongodb as a background service?

```
//In linux
mongod --fork --logpath /path/to/log/folder
//In windows
net start MongoDB
```

With MongoDB running at the background, system info about starting a DB will be written in the log folder.

△ Question
How to shut down mongodb as a background service?

```
//In linux
db.shutdownServer()
//In Windows, open cmd as admin and
net stop MongoDB
```

△ Question
How to start mongodb using a config file?

```
mongod -f /path/to/your/config.cfg
```

3 Create

△ **Question**
what could happen if you use insertMany to insert a doc that is already existing in your db?

```
//The docs before the repeated entry will still be added. A error will be printed out.
```

△ **Question**
How to change the behavior shown above?

Use "ordered", and set it to "false". Then all the new entries will be inserted except for the repeated one.

```
db.collection.insertMany([{_id:"xxx",xx:XXXX},{_id:"xx2",xx:XXXX},{_id:"xx3",xx:XXXX}],{
   ordered:false})
```

3.1 WriteConcern

Use "WriteConcert" to interact your create commands with the Journal of your database. A journal is a "Todo" list for your database.

△ **Question**
Give an example of using writeConcern in your command.

```
db.collection.insertOne({xxx:"fef","fefef":55},{writeConcern:{w:1, j:true}})
```

"w:1" option above simply indicates that "I need to be sure that the server is acknowledge my writing. And I need to wait till the server is ready to write the data." You can set w:0, but you will not know if the server will generate relevant info for that entry, and the command will be finished faster.

```
//j:true
//will make sure the command above is recorded in the journal first and will be finished
   later when the resource is available.
```

3.2 Atomicity

For atomic writing command (insertOne, updateOne), MongoDB makes sure that each file is rolled back as a whole or is written as a whole.

3.3 Importing data

To import a data file, first go to the location where the data file resides, and open a command prompt to do the following command:

```
mongoimport xxxx.json -d databaseName -c collectionName --jsonArray --drop
```

The "–jsonArray" indicates that the content of the file is an array. "–drop" drops the existing collection if the collection with the same name is already created. Without it, MongoDB appends the data to the collection.

4 Read

First, let's check the names of databases that are already created:

```
db.adminCommand({listDatabases:1, nameOnly:1})
```

△ Question
Some query keywords to remember?

```
$gt//greater than
$gte//greater or equal
$lt//less than
$lte//less or equal
$ne//not equal
$
```

△ Question
How to search the embedded doc?

```
db.collection.find({"key.embeddedkey":{xxx:xxxx}})
```

△ Question
How to search elements in an array?

```
//search entries contain element "qq" in the field of "k"
db.collection.find({k:"qq"})
//search entries only have "qq" in the field of "k"
db.collection.find({k:["qq"]})
```

△ Question
How to search entries with field "kk" equals to 43 or 55?

```
db.collection.find({kk:{$in:[43,55]}})
```

△ Question
How to search entries with field "kk" not equals to 43 or 55?

```
db.collection.find({kk:{$nin:[43,55]}})
```

4.1 Logic operator

△ Question
How to find entries that have field ”kk”>90 or <30?

```
db.collection.find({$or:[{"kk":{$gt:90}}, {"kk":{$lt:30}}]})
```

△ Question
How to count the number of entries you obtained?

```
db.collection.find().count()
```

△ Question
Why it is important to use $and in query?

Because if not, the following two queries will give same results:

```
db.collection.find({kk:"123k",kk:"2345k"})
db.collection.find({kk:"2345k"})
```

The first filter in the first command is overwritten by the second filter. If we use $and, we can apply multiple filters on the same field:

```
db.collection.find({$and:[{kk:"123k"},{kk:"2345k"}]})
```

△ Question
How to use $not in the query?

```
db.collection.find({kk:{$not:{$eq:60}}})
```

The command above gives the entries whose ”kk” field is not equal to 60.

4.2 Element operators

△ Question
How to find entries which do not have field of ”kk”?

The command below finds entries with ”age” field existing and its value is not equal to null.

```
db.collection.find({age: {$exist: true, $ne:null}})
```

△ Question
How to use $type to get entries with field having a sepcific kind of data?

The following code find entries with the type of ”phone” is ”double” or ”string”.

```
db.collection.find({phone:{$type:["double", "string"]}})
```

4.3 Evaluation operator

△ Question
How to find entries with the field ”summary” having string ”musical” in it?

We can use $regex

```
db.collection.find({summary:{$regex:"/musical/"}})
```

△ Question
How to find entries using $expr?

```
//Here we use expression to compare the field of "volume" and "target"
//The following code adds a condition on the value for "volume", where the value is
    subtracted by 30 if its original value is >=190, and then, the new value was used to
    compare
//Only the entries where the updated "volume" value is larger than the "target" are
    returned.
db.collection.find({$expr:{$gt:[{$cond: {if:{$gte:["$volume",190]}, then:{$substract:["
    $volume",30]},else:"$volume"}},"$target"]}})
```

To refer field in entries, we need to use dollar sign. For example, ”$volume”.

4.4 Array query selectors

△ Question
How to make a query for an array of embedded doc?

```
//Use dot reference. For example, if one of your field looks like the following:
"hobbies":[{"title":xxx,"freq":88},{"title":"fedf","freq":8}]
//You can refer to the "title" field by:
"hobbies.title"
```

△ Question
Give an example of using $size

```
//Find entries with the "ff" field having a length equal to 3
db.collection.find({"ff":{$size:3}}).pretty()
```

△ Question
Give an example of using $all

```
//Return movies with genres of "action" and "thriller". The sequence of these two words
    does not matter
db.movies.find({genres:{$all:["action","thriller"]}})
```

△ Question
Give an example of using $elemMatch

```
//Use the following code to return entries where "hobbies" is an array, and each elements
   must satisfy the two conditions:
//title is "Sports", and freq is >=3

//Using \$and won't work here because it only applies to the whole array
db.collection.find({hobbies:{$elemMatch:{title:"Sports", freq:{$gte:3}}}})
```

4.5 Cursors and Filtering

△ Question
Spit out the next document

```
db.collection.find().next()
//Or through defining a cursor
var dataCursor = db.collection.find()
dataCursor.next()
```

We can use "hasNext()" function to check if there is next element left in a cursor.

△ Question
How to sort the fetched results?

Using "sort" function where "-1" is descending and "1" is ascending.

```
db.collection.find().sort({"ddd":-1,"dfe":1}).pretty()
```

△ Question
What is "skip()" and "limit()" for?

"skip(x)" skips the first x results, and "limit(x)" returns x results.

△ Question
How to use projection with arrays?

```
//The following command returns entries that may or may not have "Horror" in "genres"
  field but must have "Drama" genre.
db.movies.find({genres:"Drama"},{genres:{$elemMatch:{$eq:"horror"}}})
```

△ Question
How to use projection with $slice?

```
//The following code will return the entries where the rating is >9, and their first two
  genre tags and names are shown.
db.movies.find({ratings:{$gt:9$}}, {genres:{$slice:2}, name:1})
//The following code skip the first genre tag and show the next two tags
db.movies.find({rating:{$gt:9}},{genre:{$slice:[1,2]},name:1}})
```

4.6 Solutions to some exercise

△ Question
Search all movies that have a avg rating higher than 9.2 and a runtime lower than 100 minutes

```
db.movies.find({$and:[{"rating.average":{$gt:9.2}},{runtime:100}]})
//Or we can use the following one
db.movies.find({{"rating.average":{$gt:9.2}},{runtime:100}})
```

△ Question
Search all movies that have a genre of ”drama” or ”action”

```
db.movies.find({$or:[{genres:"drama"},{genres:"action"}]})
```

△ Question
Search all movies where visitors exceeded expectedVisitors

```
db.movies.find({$expr:{$gt:["$expectedVisitors","$visitors"]}})
```

△ Question
Find all movies with exactly two genres

```
db.movies.find({genre:{$size:2}}).pretty()
```

△ Question
Find all movies which aired in 2018

```
db.movies.find({"meta.aired":2018})
```

△ Question
Find all the movies with ratings higher than 8 and lower than 10.

```
//Ratings is an array, we need to use $elemMatch
db.movies.find({ratings:{$elemMatch:{$gt:8,$lt:10}}}).pretty()
```

5 Update

△ Question
How to use $set to set multiple field?

```
//Use the following example
db.collection.updataMany({last_name:"Liu"}, {$set:{age:40, need:$}})
```

△ Question
How to increment or decrement a number ?

```
db.collection.updateOne({name:"Manuel"}, {$inc:{age:-1}})
db.collection.updateOne({name:"Manuel"}, {$inc:{age:5}})
```

△ Question
How to use $min and $max?

```
//$min lower a value to a specific value or doing nothing if the value is already smaller
    than the intended value
db.collection.updateOne({name:"Sandy"},{$min:{age:35}})
//$max raises a value to a specific value or doing nothing if the value is already larger
    than the intended value
db.collection.updateOne({name:"Sandy"},{$max:{age:35}})
//$mul multiply a number to current value
db.collection.updateOne({name:"Sandy"},{$mul:{age:1.1}})
```

△ Question
How to drop/rename a field

```
db.collection.updateMany({name:"Liu"},{$unset:{age:""}})
db.collection.updateMany({name:"Liu"},{$rename:{age:"nowold"}})
```

△ Question
How to create a doc if it does not exist?

Use "upsert" to append doc if no file is found

```
db.collection.updateOne({name:"Louis"},{$set:{age:55}},{upsert:true})
```

△ Question
How to update one element in an array ?

```
//First use $elemMatch to find the element, then use ".$" to update
db.collection.updateMany({hobbies:{$elemMatch:{freq:3, time:100}},{$set:{"hobbies.$":{
    title:999,freq:999,time:999}}})
//we can even set a new field for that element,haha
db.collection.updateMany({hobbies:{$elemMatch:{freq:3, time:100}},{$set:{"hobbies.$.haha"
    :1})
```

∆ Question
How to update many elements in an array ?

```
//reduce values in field of "frequency" by 1 for all the doc in the hobbies array
db.collection.updateMany({"hobbies.frequency":{$gt:30}},{$inc:{"hobbies.$[].frequency"
    :-1}})
//notice that the elements with frequence smaller 30 still get update, the following one
    only update the elements with frequency>30
db.collection.updateMany({"hobbies.frequency":{$gt:30}},{$inc:{"hobbies.$[el].frequency"
    :-1}},{arrayFilters:[{"el.frequency":{$gt:30}}]})
//Here we define an identifier "el" by using arrayFilters, the condition setup in the
    arrayFilters is the same as the first argument, but it can be different.
```

∆ Question
How to add/drop elements from an array?

```
//use push to add one element
db.collection.updateOne({name:"Yiggritte"}, {$push:{hobbies:{title:"squash",frequency
    :5}}})
//use addToSet to only add if the entry is unique
db.collection.updateOne({name:"Yiggritte"}, {$addToSet:{hobbies:{title:"squash",frequency
    :5}}})

//use push to add many
db.collection.updateOne({name:"Yiggritte"}, {$push:{hobbies:{$each:[{title:"squash",
    frequency:5},{title:"badminton",frequency:7}], $sort:{frequency:-1},$slice:1}}).
//here we sort insertion candidates by frequency in descending manner and only push the
    one with highest frequency

//Use pull to drop elements
db.collection.updateOne({name:"LL"},{$pull:{hobbies:{title:"hiking"}}})
//code above drops element in hobbies array with title of "hiking".
//we can use pop to drop last or first element
db.collection.updateOne({name:"LL"},{$pop:{hobbies:-1}})//first
db.collection.updateOne({name:"LL"},{$pop:{hobbies:1}})//last
```

6 Delete

△ **Question**
How to use deleteOne and deleteMany?

```
db.collection.deleteOne({name:"LL"})
db.collection.deleteMany({totalAge:{$exist:false},isSporty:true})
```

△ **Question**
How to drop an entire collection?

```
db.collection.drop()
```

7 Indexes

△ Question
How to add index for single field?

```
//First check the strategy mongoDB was using to find contacts that are older than 60
db.contacts.explain().find({"dob.age":{$gt:60}})
//we then find the stretegy mongoDB was using is "COLLSCAN"(i.e., full scan)
"winningPlan" : {
                  "stage" : "COLLSCAN",
                  "filter" : {
                          "dob.age" : {
                                  "$gt" : 60
                          }
                  },
                  "direction" : "forward"
          }
//Now, create index for the field "dob.age" in contacts collection using ascending
   sequence
db.contacts.createIndex({"dob.age":1}
//index can be dropped
db.contacts.dropIndex({"dob.age":1})

//Note: an index can only be helpful if the expected return is not huge in size (almost
   around the same size as the while database)
```

△ Question
How to create compound index?

```
//Here we create index for age and gender of the contacts collection
db.contacts.createIndex({"dob.age":1, gender:1})
//If you have the following command
db.contacts.createIndex({"dob.age":1, gender:1, postcode:1, nat:1})
//make sure you always use first N filters. If only the second or the third is used, than
    the index won't be used.
```

△ Question
How to use index for sorting?

```
//The command would be the same. just faster as no memory might be needed to do the
   sorting
db.contacts.find({"dob.age":{$gt:60}}).sort({gender:1})
```

△ Question
How to get indexed info?

```
db.contacts.getIndexes()
```

△ Question
How to get configure index?

```
//Set it as unique values
db.contacts.createIndex({xxx:1},{unique:true})
//The command above will fail if the key itself is not unique.

//Now we use partial filter to narrow down the search results if that filter is used in
   the find function
db.contacts.createIndex({xxx:1},{unique:true, partialFilterExpression:{gender:"male"}})
//Compare to compund index, partial filter has bigger size, and faster for the
   combination you sepecified.

//Now, if the field used to create index might not universal for all the doc, we can also
    use partial filter to apply index only to docs with that field:
db.contacts.createIndex({email:1},{unique:true, partialFilterExpression:{email:{$exists:
   true}}})
```

△ Question
Time-to-live(TTL) index?

TTL is used for self-destroy data.

```
//Let's create some entries of date first
db.sessions.insertOne({name:"q1", createdAt:new Date()})
db.sessions.insertOne({name:"q2", createdAt:new Date()})
//now, we make these entries self-destroyed after 10 seconds
db.sessions.createIndex({createdAt:1},{expireAfterSeconds:10})

\begin{que}
Query diagnosis \& query planning?
\end{que}
\begin{code}
//show winning plan
db.collection.explain("queryPlanner").find(...)
//show execution stats+possible rejected plans
db.collection.explain("executionStats").find(...)
//show stats+wining plan decision rocess
db.collection.explain("allPlansExecution").find(...)
```

△ Question
Why use multi-key index?

```
//If the value of a field is an array, we need to use multi-key to speed up the code
db.collection.createIndex({"xxxx.xxx":1})
//Array keys cannot be used to build compound index.
```

△ Question
How to use ”text” index

```
//If you have keys with their values are pure text, you can create index based on the
   text as:
db.collection.createIndex({description:"text"})
//Later, you can search on these text as:
db.collection.find({$text:{$search:"xxxx aaaa"}})
//the command above will find all the entries with descriptions containing xxxx and aaaa,
    but not "xxxx aaaa". In order to find entries where the description contains "xxxx
   aaaa" exactly, we use:
db.collection.find({$text:{$search:"\"xxxx aaaa\""})

//We can, of course, create combined text index
db.collection.createIndex({key1:"text",key2:"text"})
```

△ Question
How to sort results obtained based on text index?

```
//Use score option to arrange results based on number of filters each result satisfies
db.collection.find({$text:{$search:"awesone pants"}},{score:{$meta:"textScore"}})
//typically, the command above gives sorted results, but we can also add sort() like:
db.collection.find({$text:{$search:"awesone pants"}},{score:{$meta:"textScore"}}).sort({
   score:{$meta:"textScore"}})
```

△ Question
Exclude words using text index?

```
//Exclude T-shirt in the string
db.collection.find({$text:{$search:"xxxx -t-shirt"})
```

△ Question
Use weight in text index?

```
db.collection.createIndex({key1:"text",key2:"text"},{default_language:"chinese",weights:{
   key1:10,key2:1}})
```

△ **Question**
Why and How of background Indexd

```
//background index does not lock the collection when it is created. Thus, it takes less
   time update when new entry is inserted into the collection
db.collection.createIndex({key:1},{background:true})
```

8 Geospatial Queries

△ Question
An example of GeoJSON object?

```
{
  type : "Polygon",
  coordinates : [
     [ [ 0 , 0 ] , [ 3 , 6 ] , [ 6 , 1 ] , [ 0 , 0 ] ],
     [ [ 2 , 2 ] , [ 3 , 3 ] , [ 4 , 2 ] , [ 2 , 2 ] ]
  ]
}
//Key words for GeoJSON type: "Point", "LineString", "Polygon","MultiPoint","
   MultiLineString","MultiPolygon"
//The following command create an entry by using geospatial data
db.collection.insertOne({name:"MIT", location:{type:"Point",coordinates:[-122.3443443,
   33.9876489]}})
```

△ Question
How to create Geospatial index to track the distance

```
//First, create Index as
db.collection.createIndex({location:"2dsphere"})
//Than, find the location near the point of (-122.3, 77.6)
db.collection.find({location:{$near:{$geometry:{type:"Point", coordinates:[-122.3,
   77.6]}}}})
//Set distance range
db.collection.find({location:{$near:{$geometry:{type:"Point", coordinates:[-122.3,
   77.6]},$maxDistance:500,$minDistance:10}}})
```

△ Question
How to find places inside a certain area

```
//First we define a polygon
const p1=[122,77]
const p2=[133,99]
const p3=[123,99]
const p4=[122,88]
//Find the places in DB that are within the polygon
db.collection.find({location:{$geoWithin:{$geometry:{type:"Polygon", coordinates:[p1,p2,
   p3,p4]}}}})
//We can also save this area and check if a point is inside the area
db.collection.insertOne({name:"region1", area:{type:"Polygon", coordinates:[p1,p2,p3,p4
   ]}})
//Now, reate index
```

```
db.collection.createIndex({area:"2dsphere"})
//check which polygon contains the point (122,45)
db.collection.find({area:{$geoIntersects:{$geometry:{type:"Point", coordinates
   :[122,45]}}}})
```

△ Question
find places within a certain radius

```
//find places within 100 miles from -122,33
db.collection.find({location:{$geoWithin:{$centerSphere:[[-122, 33], 100/3963.2]}}})
//find places within 100 kms from -122,33
db.collection.find({location:{$geoWithin:{$centerSphere:[[-122, 33], 100/6378.1]}}})
```

9 Aggregation Framework

9.1 Fundamentals

△ Question
What is pipeliens?

Pipelines are a composition of stages. Stages are configurable to produce desired transformations. Documents flow through stages like parts in an assembly line or water through a pipe. Finally, with only a few exceptions which we'll cover later, stages can be arranged in any way we like and as many as we require.

```
//think of as a conveyor belt in a factory. Along the line, we have three stages:
//$match: find specific entries
//$project: transform data you found
//$group: combine them into a single file (result)
```

△ Question
What aggregation structure might look like?

```
db.collection.aggregate([{stage 1}, {stage 2}, {...stage N}], {options})

//Now, an example of an aggregation:
db.solarSystem.aggregate([{//pipline is a list of stages
    $match:{
        atmosphericComposition:{$in:[/02/]},
        meanTemperature:{$gte:-40,$lte:40}
        },{
        $project: {
        _id:0,
        name:1,
        hasMoons:{$gt:["$numberOfMoons",0]}
        }
        }], {allowDiskUsr:true}
)
```

△ Question
some important syntax?

```
//Field Path:
"$fieldName"
//System Variable:
"$$UPPERCASE"
//Usert variable
"$$lowercase"
```

△ **Question**
What the $project do?

```
//It bahave like the map function in programming language like Python while it still has
   basic projection functionality.
//The following command reassin the value in gravity.value to a new field, surfacegravity
   :
db.collection.aggregation({$project:{_id:0,name:1,surfacegravity:"$gravity.value"})
//The following command shows my weight on different planets
db.colllection.aggregation({$project:{_id:0,name:1,myweight:{$multiply:[{$divide:["
   $gravity.value",9.8]},70]}})
```

△ **Question**
Using expression in project?

```
//It's just like normal query. The following command calculates approx. radius for each
   planet
db.collection.aggregation({$project:{_id:0,name:1,radius:{$floor:{$add:["$radius.value"
   ,0.5]}}}})
//We can also use :
db.collection.aggregation({$project:{_id:0,name:1,radius:{$floor:{$add:["$$CURRENT.radius
   .value",0.5]}}}})
```

△ **Question**
Give an example of building pipeline in pymongo?

```
import pymongo
course_cluster_uri = "mongodb://agg-student:agg-password@cluster0-shard-00-00-jxeqq.
   mongodb.net:27017,cluster0-shard-00-01-jxeqq.mongodb.net:27017,cluster0-shard-00-02-
   jxeqq.mongodb.net:27017/test?ssl=true&replicaSet=Cluster0-shard-0&authSource=admin"
course_client = pymongo.MongoClient(course_cluster_uri)
movies = course_client['aggregations']['movies']

counting = {
    "$count": "one_word_titles"
}
shaping = {
"$project":{
       "title":1,
       "spltil":{"$size":{"$split":["$title"," "]}}
    }
}
matching = {
    "$match":{"spltil":{"$eq":1}}
}
```

```
pipeline = [
    shaping,
    matching,
    counting
]

display(list(movies.aggregate(pipeline)))
//Note here that we can have multiple stages for "match", "project"
//Make sure each stage follows the format of {key1:{expr1},key2:{expr2}}
```

△ Question
What is $match in the command above

```
//It acts like a find query
```

△ Question
What's the advantage of using sort early in the pipeline?
How to speed up your sort functionality?

```
//we can take advantage of using indexes later in the pipeline
//use {allowDiskUse:true} in your pipeline in order to use more than 100MB memory
```

△ Question
What is $group stage?

```
//It just like the "GROUP BY" in SQL but much more flexible
```

△ Question
How to use $group to calculate an average over the whole/part of database?

```
//The command below calculates the average of the field "metacritic" and renamed the
   results as "avgMeta"
db.collection.aggregation([
{$match:{"metacritic":{$gte:0}},
{$group:{_id:null,avgMeta:{$avg:"$metacritic"}}}
])
//Notice that "null" is used for _id to make sure the calculation is over the whole
   database.

//The command below group the data by the year:
db.collection.aggregation([
{$match:{"metacritic":{$gte:0}},
{$group:{_id:"$year",avgMeta:{$avg:"$metacritic"}}}
])
```

△ Question
How to count how many files are in each group?

```
db.collection.aggregation([
{$group:{_id:"$year",count:{$sum:1}}}
])
```

△ Question
What you will get when you use accumulator function in $project

```
//accumulator function will work with the array within each file, so you will get a
   result for each file.
```

△ Question
Give an example of using $reduce and accumulator functions

```
//The following calculate the maximum of "avg_high_tmp"
db.collection.aggregation({
   $project:{
            _id:0,
            max_temp:{
            $reduce:{
               input:"$trends",
               initialValue:-Infinity,
               $cond:[{$gt:["$$this.avg_high_tmp","$$value"]},
                     "$$this.avg_high_tmp",//if avg_high_tmp>Value, return
                        avg_high_tmp
                     "$$value"//else, return the value.
                    ]
            }
         }
      })
//Notice that the $$value has an initialValue is set to be -Infinity
//$$variable are temporary for this query only.
```

A much easier way to do the same thing is to use $max:

```
db.collection.aggregation([{$project:{_id:0, max_temp:{$max:"$trends.avg_high_tmp"}}}])
```

△ Question
What are the available accumulator functions?

```
$max,$min,$avg,$stdDevPop//standard deviation over whole dataset
$stdDevSamp//standard deviation over sampled subset.
```

△ Question
What is $unwind stage and why is it useful?

```
//we use the following entry as an example
{
   "name":"book",
   "genre":["fiction","advanture"]

}
//After unwind, we have two entries:
{
   "name":"book",
   "genre":"fiction"

}
{
   "name":"book",
   "genre":"advanture"

}
//Unwind stage is useful when we use array-like field as one of the conditions in group
   stage. For example:
{"genre":["advanture","fiction"]}
{"genre":["fiction","advanture"]}
//are not equivalent, if we use group stage directly on this field. However, with unwind,
    we might be able to work around such condtion.

//to unwind the field above, we can do the following:
{"$unwind":"$genre"}
```

△ Question
How to use unwind with group? give an example.

```
//Let's use our increasing understanding of the Aggregation Framework to explore our
   movies collection in more detail. We'd like to calculate how many movies every cast
   member has been in, and get an average imdb.rating for each cast member. Which cast
   member has the been in the most movies with English as an available language?
predicate = {
   "$match":{"languages":"English"}
}
unwinding = {//Use detailed unwind is important here!!!
   "$unwind":{"path":"$cast","includeArrayIndex": "arrayIndex","
      preserveNullAndEmptyArrays": False }
}
grouping = {
    "$group":{
        "_id":"$cast",
        "numFilms":{"$sum":1},
        "average":{"$avg":"$imdb.rating"}
    }
}
sorting = {
    "$sort":{"numFilms":-1}
}
limiting = {
    "$limit":5
}
pipeline = [
    predicate,
    unwinding,
    grouping,
    #shaping,
    sorting,
    limiting
]
display(list(movies.aggregate(pipeline)))
//The command above gives the following result:
[{'_id': 'John Wayne', 'numFilms': 107, 'average': 6.424299065420561},
 {'_id': 'Michael Caine', 'numFilms': 83, 'average': 6.506024096385542},
 {'_id': 'Christopher Lee', 'numFilms': 76, 'average': 6.132894736842106},
 {'_id': 'Robert De Niro', 'numFilms': 75, 'average': 6.690140845070423},
 {'_id': 'Bette Davis', 'numFilms': 68, 'average': 7.310294117647059}]
```

∆ Question
What is $lookup stage

```
//It is effectively "LEFT JOIN" in SQL, it has the following structure
{
   $lookup:
     {
       from: <collection to join in current database>,
       localField: <field from current collection>,
       foreignField: <field from the documents of the "from" collection>,
       as: <output array field>//any string can be used, if the string already exists in
           current collection, it will be overwritten.
     }
}
//Notice that the lookup will retrive the whole document once it finds a matched entry.
```

△ **Question**
Give an example of using $lookup?

```
//An entry in air_routes looks like:
{"_id":{"$oid":"56e9b39b732b6122f877fa31"},"airline":{"id":410,"name":"Aerocondor","alias
   ":"2B","iata":"ARD"},"src_airport":"CEK","dst_airport":"KZN","codeshare":"","stops":0,
   "airplane":"CR2"}
//An entry in air_alliances looks like:
{"name": "Star Alliance",
    "airlines":["Air Canada",
      "Adria Airways",
      "Avianca",
      "Scandinavian Airlines",
      "All Nippon Airways",
      "Brussels Airlines",
      "Shenzhen Airlines",
      "Air China",
      "Air New Zealand",
      "Asiana Airlines",
      "Brussels Airlines",
      "Copa Airlines",
      "Croatia Airlines",
      "EgyptAir",
      "TAP Portugal",
      "United Airlines",
      "Turkish Airlines",
      "Swiss International Air Lines",
      "Lufthansa",
      "EVA Air",
      "South African Airways",
      "Singapore Airlines"
      ]}
//Which alliance from air_alliances flies the most routes with either a Boeing 747 or an
   Airbus A380 (abbreviated 747 and 380 in air_routes)?
predicate = {
  "$match": {
      "airplane": {"$regex": "747|380"}
  }
}
lookup = {
    "$lookup":{
        "from":"air_alliances",
        "localField":"airline.name",
        "foreignField":"airlines",
        "as":"alliance"
    }
}
unwinding = {
    "$unwind":{"path":"$alliance.name", "includeArrayIndex": "arrayIndex","
       preserveNullAndEmptyArrays": True}
}
grouping = {
    "$group":{
            "_id":"$alliance.name",
             "count":{"$sum":1}
             }
}
sorting = {
    "$sort":{"count":-1}
}
pipeline = [
    predicate,
    lookup,
    unwinding,
    grouping,
    sorting
]
display(list(routes.aggregate(pipeline)))
//The commands above give:
[{'_id': [], 'count': 51},
 {'_id': ['SkyTeam'], 'count': 16},
 {'_id': ['Star Alliance'], 'count': 11},
 {'_id': ['OneWorld'], 'count': 11}]
```

△ Question
Give an example of using $graphlookup?

```
//Here we are trying to find the connection between the actor "Woody Harrelson" and the
   director "Steven Spielberg". Now, we build stages one by one:
//1. Find all the movies made by Steven, and only get the directors data
{
        "$match": {
            "directors": "Steven Spielberg"
        }
    },
    {
        "$project": {
            "directors": 1
        }
    },
//2. Treat each director as a node on a graph, trasverse over the graph by using
   graphlookup:
```

```
graph_lookup = {
    "$graphLookup": {
        "from":"movies",
        "startWith": "$directors",
        "connectFromField": "directors",
        "connectToField": "directors",
        "maxDepth":6,
        "depthField": "network_level",
        "as":"network"
    }
}
//Now, each movie entry has a new field called "network", within which is a linked list
   starting from that movie, and each node on the list has its depth value(from 1 to 6).
//3. Next, we unwind the network and only save the information of graph depth and cast
   lists that might or might not contain "Woody Harrelson".
{
        "$unwind": "$network"
    },
    {
        "$project": {
            "cast": "$network.cast",
            "level": "$network.network_level"
        }
    },
//4. group by the levels on the graph, combine the cast lists of all the nodes with
   identical network_levels.
    {
        "$group": {
            "_id": "$level",
            "cast": {"$addToSet": "$cast"}
        }
    },
//5. because the resulted combined cast list is a 2d array, we can transform it into 1d
   array by using \$reduce:
project_cast = {
    "$project": {
        "cast":{"$reduce":{
            "input":"$cast",
            "initialValue":[""],
            "in":{"$concatArrays":["$$value", "$$this"] }
                    }
               }
    }
}
//6. Finally, find the cast lists with "Woody Harrelson" in it, and order their
   network_level ascendingly
```

```
{
       "$match": {
          "cast": "Woody Harrelson"
       }
   },
   {
       "$sort": {
          "_id": 1
       }
    },
   {
       "$project": {
          "_id": 0,
          "answer": "$_id"
       }
   },
   {
       "$limit": 1
   }
//This pipeline gives the following results:
[{'answer': 2}]
//meaning Woody is 2 person away from Steven.
```

9.2 Import data using CSV

MongoDB cluster/command line tools is used here.

```
//First go to your cluster on ATLAS, click the "..." button to find command line tools,
   copy the following command
mongoimport --host hostName --ssl --username xxxx --password xxxxx --
   authenticationDatabase admin --db databasename --collection collectionName --type CSV
   --file xxxx.csv

//If you have header line in your csc file, add --headerline to specify the names for all
    the fields.

//After the data is imported to your cluster on ATLAS, connect your ATLAS cluster with
   MongoDB compass by clicking your cluster, and hit the connect button to copy the
   connect command string for compass.

//Now, open your compass, and it will automatically detect the string and autofill the
   field for connection.

//Type your password, and then click the connect button.
```

△ Question
How to force data type for each column of your raw data?

```
//Go to your raw data, add header name with data type, for example
columnName.string()
columnName.int32()
columnName.decimal()
//One special case is the date format, you need to provide an example date in order to
   get it working. In this case, the example must be "Mon Jan 2 15:04:05 MST 2006"

//Let's say your date format is smt like "12/1/10 8:26", then your header should be:
columnName.date(1/2/06 15:04)

//Now we need to add --columnsHaveTypes at the end of the mongoimport command
```

△ Question
How to drop, insert to, or merge to current data from imported data?

```
//use --drop to delete existing data
//use --mode=insert to insert to
//use --mode=upsert to overwrite or insert entries, with upsert, we can specify which
   field to replace by using --upsertFields=fieldName,fieldName2
//use --mode=merge means to replace only the fields of the new data that we're importing
   and keep other fields the same.
```

9.3 On Schema

△ Question
Why have a shcema

- ∼ If you don't have a schema, look harder-
- Because you want to group similar things together
- Identify common characteristics through aggregation

△ Question
What does a schema consist of?

- In relational databases you start with the data and its relationships
- in MongoDB, schema begins with data access patterns
  Performance focused

△ Question
What are the entity relationship

```
//1 to N relation
//N to N relation
```

△ Question
What is the flexibility of schema validation?

Schema Validation
- You can insert documents without defining any validation!
- You can define a validation and only insert those types!

A few words on developing a schema:

- MongoDB supports very strict and very loose schemas
- ldentify important queries to support (access patterns)
- After considering queries, then consider relationships (opposite of relational schema design)
- Schemas change over time!

△ Question
How to migrate your schema specified by a pipeline?

```
//Use "$out":"collectionName" at the end of your pipeline.
```

△ Question
What is view and its features?

· Views contain no data themselves. They are created on demand and reflect the data in the source collection
Views are read only. Write operations to views will error
Views have some restrictions. They must abide by the rules of the Aggregation Framework, and cannot contain find() projection operators
Horizontal sicing is performed with the Smatch stage, reducing the number of documents that are returned.
Vertical slicing is performed with a Sproject or other shaping stage, modifying individual documents.

10 Numeric Data

△ **Question**
How to defina a value as integer?

```
db.collection,insertOne({age:NumberInt(29)})//int32
db.collection,insertOne({code:NumberLong(2243564908)})//int64
db.collection,insertOne({code:NumberDecimal(22435.64908)})//decimal 128
```

△ **Question**
How to do math with int32/int64/float?

```
//make sure you declare the number type before using $inc,$mul etc.
db.collection.updateOne({}, {$inc:{amount:NumberLong("234")}})
```

11 Security

△ Question
How to create and editing users?

```
//First, start your DB with auto option
mongod --auth
//Now, go to MongoDB shell and create an user as:
db.createUder({user:"Liu", pwd:"fesf", roles:["userAdminAnyDatabase"]})
//Now, use this new user to go to your db
db.auth('Liu','fesf')
//Remember, the users must be attached to a database.
//You can also create an user for multiple database
db.createUser({user:"dev", pwd:"ddd", roles:[{role:"readWrite",db:"customers"},{role:"
   readWriter",db:"shop"}]})
```

△ Question
Built-in roles?

```
//user:
"read","readWrite"
//admin
"dvAdmin","userAdmin","dbOwner"
//All database roles
"readAnyDatabase","readWriteAnyDatabase","userAdminAnyDatabase","dbAdminAnyDatabase"
//superuser
"dbOwner","userAdmin","userAdminDatabase", "root"
```

△ Question
how to use user/pwd pair to login a db?

```
//instead using db.auth, we can use
mongo -u Liu -p fesf --authenticationDatabase dbName
```

△ Question
How to update user so that it can edit another DB?

```
//First go to a DB
use shope
//then, make sure your user can edit the collection named "blog" in "shop" db
db.updateUser("userName",{roles:["readWrite",{role:"readWrite",db:"blog"}]})
//Afterwards, make sure you log out the user and re-login to "shop" db. Because the user
   might be attached to another db
mongo -u userName -p xxxxx --autheticationDatabase shop
```

12 MongoDB with Machine Learning

12.1 Using MongoDB to Do Associative Role Learning

12.1.1 Terminology

- Relationship Terms
  - Antecedent
  - Consequent
- Characteristic Terms
  - Support
  - Confidence
  - Lift

First,If we have the following relationships between two items/variables:

A implies B
A ⇒ B
Chips ⇒ Beer

where we use chips/beer as an example (i.e., the purchase of beer implies the purchase of chips). Then Chips is the **Antecedent**, and the beer is **Consequent**.

Support is defined as:

Support(A) =
Transactions where A appears / Total Transactions

In a associative role analysis, if we have multiple items/variables, the support has to be calculated for all possible combinations. For example, if we have a list of items/variables: $\{A, B, C\}$, we need to calculate the support of $\{A\}, \{B\}, \{C\}, \{A, B\}, \{A, C\}, \{B, C\}$, that is:

$$\mathbb{P}\{A \cap B\}, \mathbb{P}\{A \cap C\}, \mathbb{P}\{C \cap B\}, \mathbb{P}\{A\}, \mathbb{P}\{B\}, \mathbb{P}\{C\}$$

Confidence is defined as the likelihood (possibility) of item B to be bought with item A:

Confidence (A ⇒ B) =
Support([A ∩ B]) / Support([A])

Lift is defined as how likely item B is purchased when A is purchased.Unlike confidence, lift take into account the popularity of item A:

Confidence(A ⇒ B) =
Support([A ∪ B] /(Support([A]) " Support(IB)))

A general rule for the value of lift is:

- $lift > 1$, positive association;
- $lift = 1$, no association;
- $lift < 1$, negative association;

12.1.2 Titanic data for association rule (in Python)

12.2 Principal Component Analysis (PCA) with MongoDB

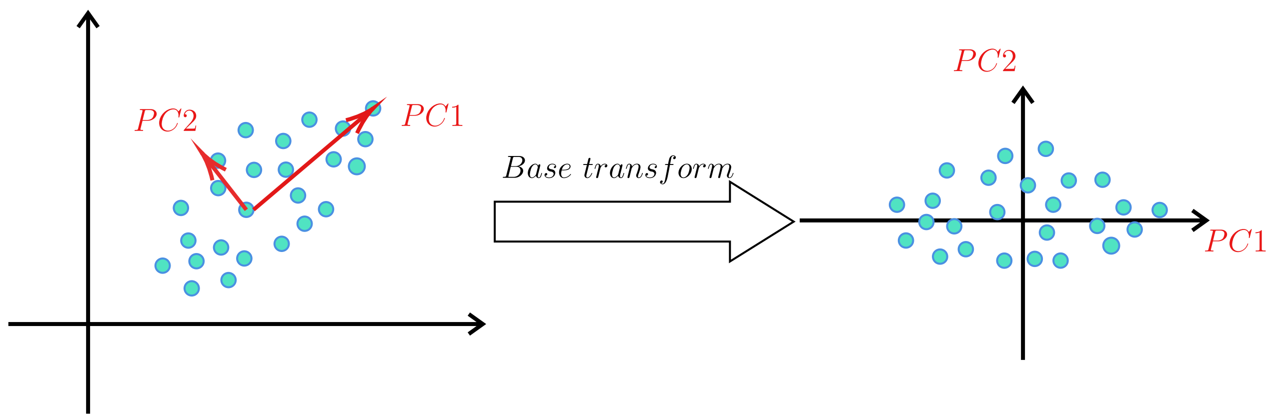

Figure 1: PCA on 2D data points

Important notes on PCA: PCA will be applied to each dimension; each principal vector will be perpendicular to each other.

12.2.1 A Python example

github link

12.3 Linear Regression with MongoDB

github link

12.4 Decision tree with MongoDB+Python

github link

12.5 K-mean with MongoDB+Python

github link