GIT

INITIAL CONFIGURATION:

git init 🡪 to initialize the git repo in the location(pwd)

git -global -user.name “Shirdik” 🡪 to link with github

git remote add origin <https://github.com/shirdik/demo.git> 🡪 to add remote repo

git remote remove origin 🡪 to remove the remote repo(in github)

MOST USED:

git status 🡪 to know the status of the files

git add <filename> 🡪 to stage the files (use ‘.’ For selecting all the modifications to the stage)

git commit -m “TEXT” 🡪 to commit the changes in the local repo

git push 🡪 to upload the application to the repo (github)

git pull 🡪 to download the latest application from the github

git branch b1 🡪 to create branch

git checkout b1 🡪 to switch to the b1 branch

git checkout -b b1 🡪 to create and switch to the b1

* *When we are working in the new branch all the changes made are saved in the new (b1) feature branch*

git rebase <branch name> 🡪 to rebase the branch to the latest version of the application

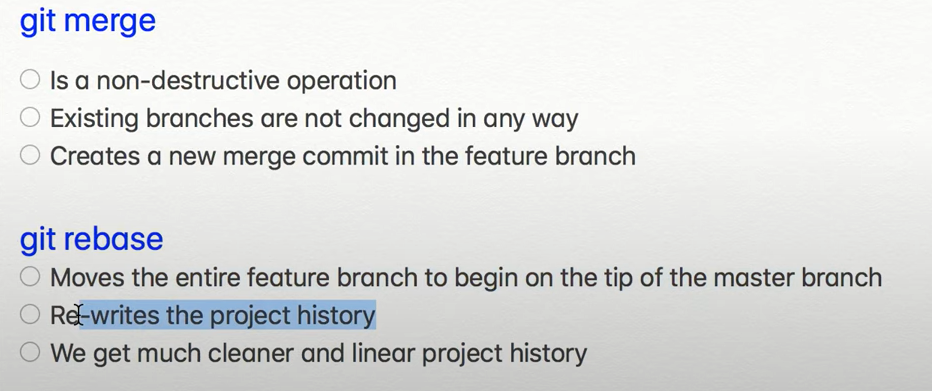
LINK: [Git - git-rebase Documentation (git-scm.com)](https://git-scm.com/docs/git-rebase)

* *Do not do rebase with a shared feature branch*

git log 🡪 to get the commit history of the application

git log –oneline 🡪 for the recent history

🡪 :*wq! --- To write and quit in file*

**

TYPES OF MERGE OPERATIONS:

1. Fast-forward merge:

When the feature branch and master branch are in linear path, this can be done even there are intermediate commits.

1. 3-way merge:

When the feature branch and master branch are not in linear path then new commit is created using the present status of the both branches.

|  |  |  |
| --- | --- | --- |
| **Branch 1** | **Branch 2** | **Result** |
| NO change | Change occurred | Change is done |
| Change occurred | No change | Change is done |
| Change occurred | Change occurred | Conflict (if both changes are different) |

🡪 In order to tackle the conflict, the changes have to be sorted out

**Searching and Rewriting history:**

**Amend:**

This allows the user to stage and commit the latest updates to the last commit. (Adding to the last commit)

🡪git commit –amend –no-edit

**Rewording commit:**

This allows the user to rewrite the comments written for the commits done recently

🡪git rebase -I HEAD~2

(this command means that we have to change the last two commits from the present head)

This will take to the vim editor in which we have to change the pick keyword to reword

**Deleting Commits:**

This feature helps us to delete the commits.

* git rebase -I HEAD~2

(then change to drop from pick in order to delete the commit)

**Reordering Commit:**

To change the order of the commits just change the order of the commits in the vim editor using the same command as the above

**Squash Commits**:

Use the same command and in the interactive rebase session. Use the keyword fixup to the commits to which the commits have to be merged with the above one

**Split Commits:**

Go to the same interactive rebase session, then use the keyword edit and save the file. Then you will be taken to the intermediate session in which you need to unstage the modified files and then stage and commit them separately to split the commit.

🡪git reset HEAD^ (To unstage all the files)

🡪To finish at last use git rebase –continue to get back to the master branch

Mongo DB

Mongo DB is a No sql application used to build the database on the basis of the application. It is schema-less and Non-Relational database It also works with BSON(Binary JSON).

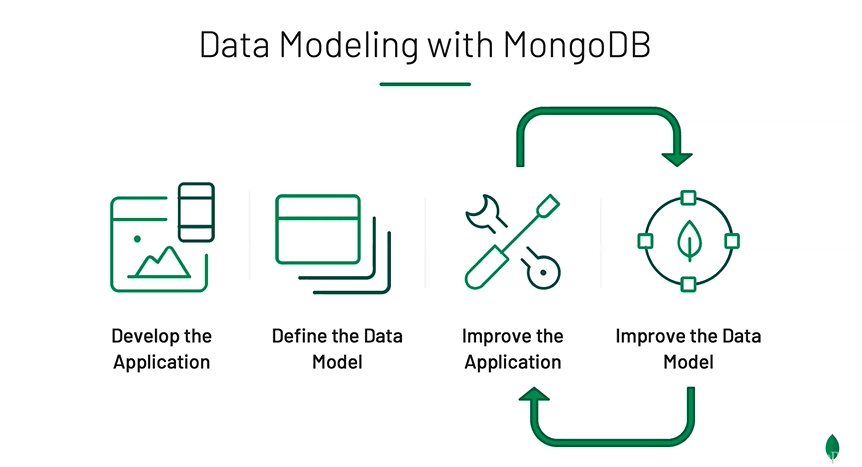
Terms Used:

* Collection – It is a structure which holds the entities
* Entity – It is a real-world object
* Link - Linking two entities with id

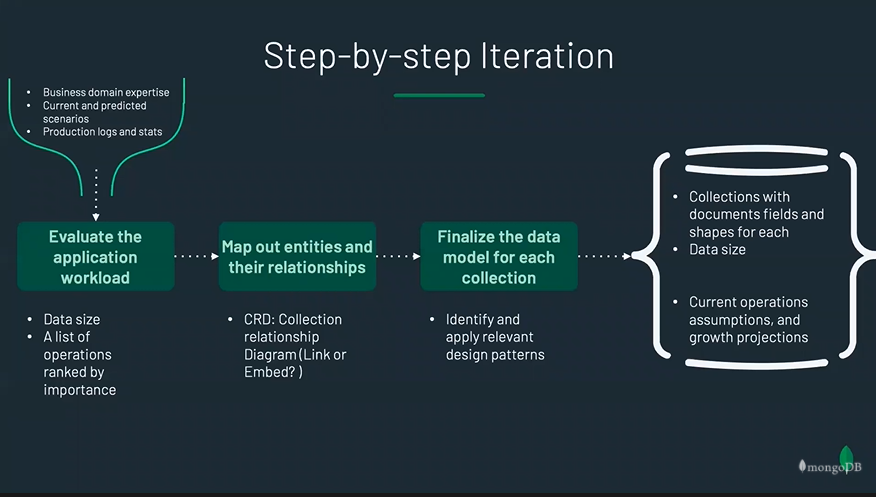
Mongo DB has many advantages like:

1. No schema – One collection can hold different documents in which the fields can be different
2. Scalable – It enables horizontal scalability called sharding(data is distributed and balanced automatically) which helps to overcome the hardware limitations
3. ACID – Atomicity, Consistency, Integrity, Durability
4. Failover Mechanism – It enables the secondary server as primary server when the primary server is down automatically
5. Authentication – It supports common authentication protocols like LDAP, AD

Data Modelling with MongoDB:



Can evolve without any downtime



Design Pattern:

1. The Schema Versioning Pattern:

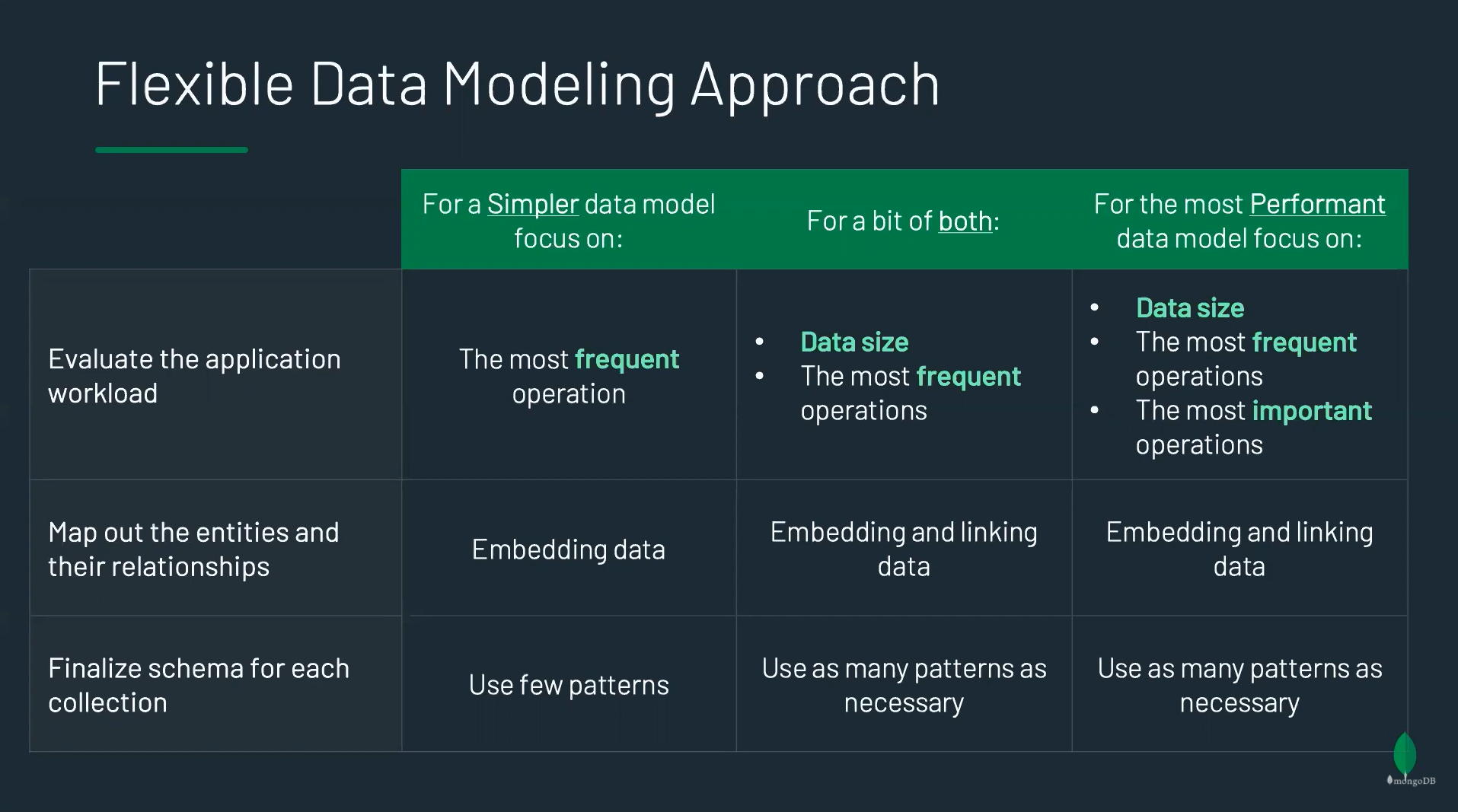
In this pattern when we update the database, we name the versions and according to the version name or number we can keep track of the fields of the document

1. The Bucket Pattern:

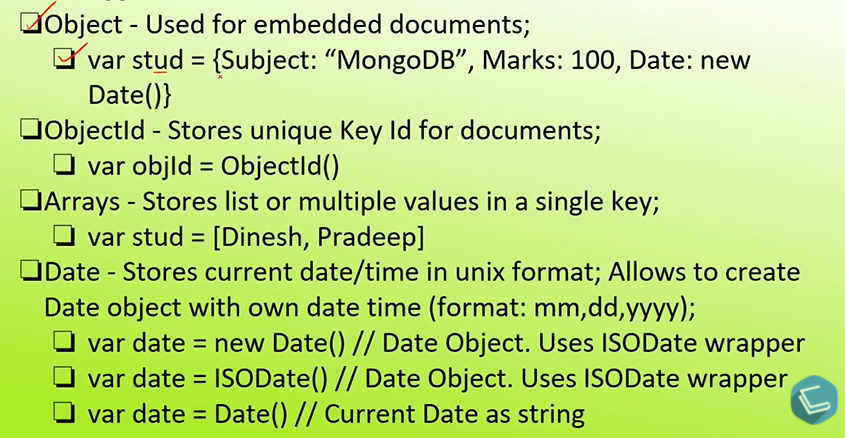
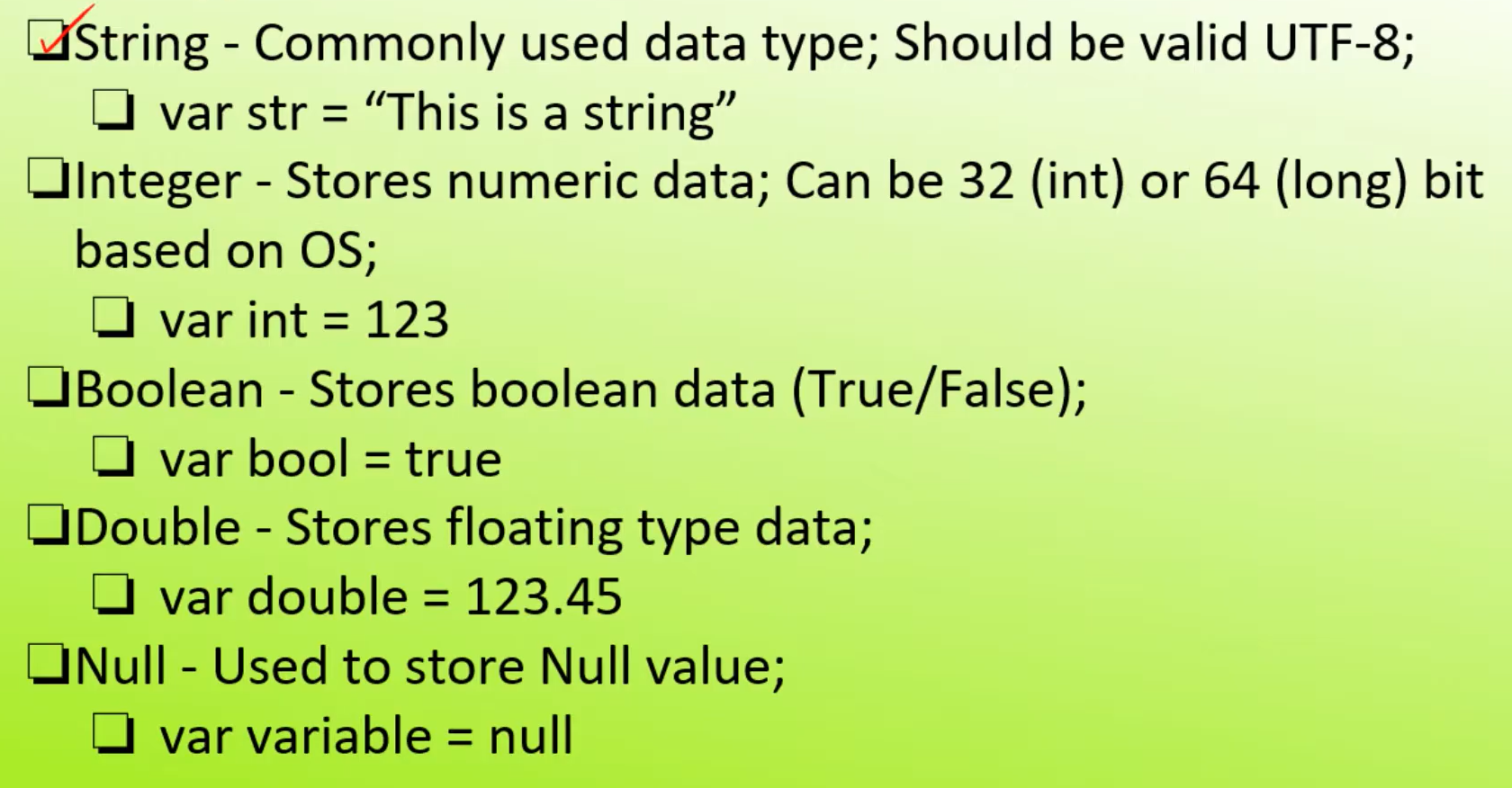
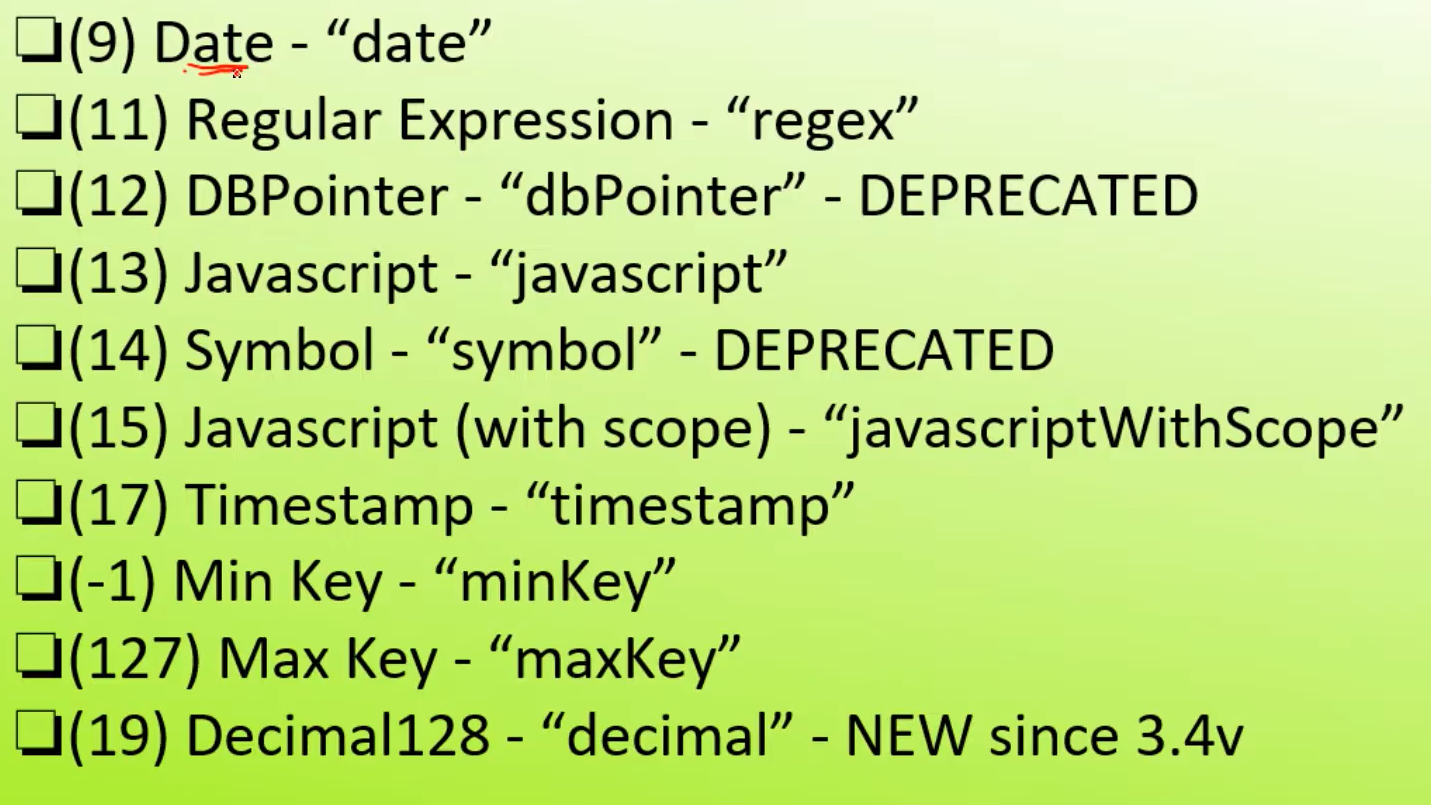
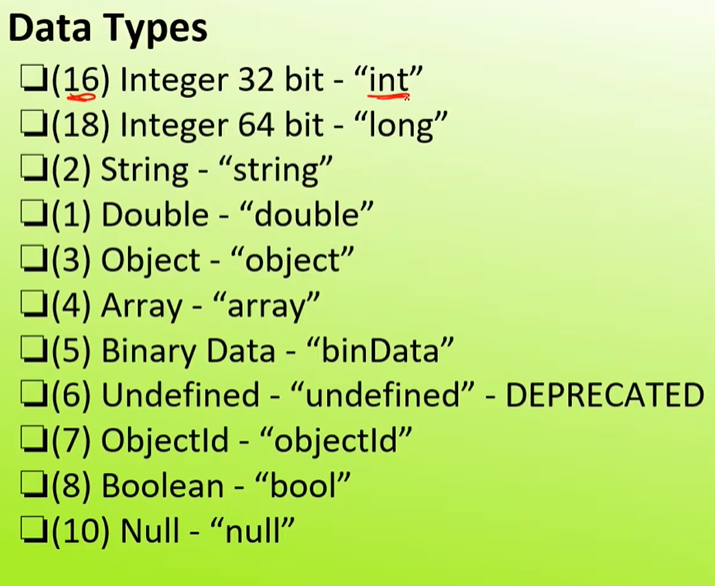
In this pattern we create a document for every entity and we update the document concatenating the new values instead of creating or overwriting the data. This reduces the size taken for the entity and increases the retrieval time

1. The Computed Pattern:

In this pattern while we update the documents we update with some summary records too like the average rating which in turn reduces the load on cpu and for faster retrieval of the data



Data Types:



Commands:

mongo –version 🡪 To know the version of the MongoDB server.

mongo 🡪 to initialize the server.

show dbs 🡪 shows the databases created/available.

show collections 🡪 shows the collections available in the database.

🡪This command shows the databases which have at least one collection.

use <database\_name> 🡪 This command opens the database. If there is no database with the given name it creates one.

**CRUD**

CREATE:

db.createCollection(“<collection\_name>”) 🡪 to create a new collection in the database.

db 🡪 shows current working database.

db.dropDatabase() 🡪 to delete database

db.<collection\_name>.drop() 🡪 to delete the collection

db.<collection\_name>.insertOne({“field” : “data”}) 🡪for inserting one document

db.<collection\_name>.insertMany({“field” : “data”} ,{“field” : “data”}) 🡪for inserting one document

READ:

db.<collection\_name>.find({<query>},{<projection>}) 🡪 to show the documents in the collection

db.<collection\_name>.find( { name: {$in[“shirdi”, “chandrika”]}} ) 🡪 to find specific

db.<collection\_name>.find().pretty() 🡪 for better readability

db.student.find({name: {$eq: "shirdi"}},{"\_id":0}).pretty()

Relational:

$lt 🡪less than

$lte 🡪 less than or equal to

$eq 🡪 equal to

$gt 🡪 greater than

$gte 🡪 greater than or equal to

$in 🡪 in the array

$nin 🡪 not in the array

Logical:

db.<collection\_name>.find({ $and: { filed1 : data1 },{ field2 : data2 } })

$and 🡪 and operation

$not 🡪not operation

$or 🡪 or operation

db.<collection\_name>.find({name: {$in: [“shirdi”,”chandrika”]}, {age: 0}}) 🡪 use projection tag as in order to show that field and 0 to hide the field

db.<collection\_name>.find().count() 🡪 used to get the count

db.<collection\_name>.find().sort({age: 1}) 🡪 1 for ascending and -1 for descending order and the field can be changed

db.<collection\_name>.find().limit(2) 🡪 to limit no:of results

db.<collection\_name>.find(synopsis:{$regex: “bilbo”})

🡪to find the text in a documents

Update:

db.<collection\_name>.updateOne({\_id:1},{$set: { age:23}}) 🡪 to update the fields

🡪$push – to add to the list

Delete:

db.<collection\_name>.deleteOne({\_id:4}) 🡪 to delete the field

db.dropDatabase()🡪to delete active database

AGGREGATION:

In aggregation we use stages in order to append the conditions one by another.

$match🡪to get the matched results

$group🡪 to group by distinct values of the field

🡪 db.zipcodes.aggregate([ {$match:{city:"ATLANTA"}}, {$group:{\_id:"city",count:{$sum:"$pop"}}} ])

for getting total pop in the city

🡪db.zipcodes.aggregate([

... {$match:{city:"ATLANTA"}},

... {$group:{\_id:"city",count:{$sum:1}}}

... ])

to count the no:of zipcodes in the city