# Practice M4: NoSQL

During this practice we will assume that we are working in Linux environment. It could be a physical machine or a virtual one. The distribution of choice is not that important, but it will be better to stick to some of the well supported distributions.

All steps can be executed in Windows and/or macOS environment as well.

## Part 1: Redis

#### Explore Redis on Docker

Let’s first experiment in a pure Docker environment. It can be local, in a VM, or remote.

* Create a Docker container with Redis

docker container run -d --name redis-host redis

* Connect to the container

docker container exec -it redis-host bash

* Start the CLI tool

redis-cli

* Get information about the instance

INFO

* Get information about the current configuration

CONFIG GET \*

* Get information about the number of databases

CONFIG GET databases

* List databases with keys

INFO keyspace

* Select a database to work with, by default is the first one (with index of 0)

SELECT 0

* Let’s create a key and get is value back

SET name Stoyan

GET name

* Now, delete the key, and check if we can get its value

DEL name

GET name

* There is also a combined get-set command. Let’s try it

GETSET name Ivan

GETSET name Stoyan

* There is a conditional set command

SETNX name Petya

* It did not work, or at least it seems so. Let’s first check, and then clean up a little bit

GET name

DEL name

* Now if we re-run it again

SETNX name Gergana

GET name

* Did you know, that we can set expiration of the keys?

EXPIRE name 10

TTL name

* Now if we wait for more than 10 seconds and try to get the value of the name key, there won’t be anything

GET name

* There is an option to set multiple key-value pairs at the same time

MSET cust:001:M Ivan cust:002:F Gergana cust:003:M Petar cust:004:F Dimana

* We can ask for all keys

KEYS \*

* Or with regular expression

KEYS \*M

* We can use keys with integer values as well

SET counter 1

* We can increase the value with 1 or with bigger step

INCR counter

INCRBY counter 10

* There is also command for decrease

DECRBY counter 2

* If we have to repeat an action N times, we can do it

5 INCRBY counter 10

* Now if we ask for the counter value

GET counter

* To get information about the size of the database

DBSIZE

* And finally, we can clean up everything with

FLUSHBD

* Close the tool, leave the container, and remove it

exit

exit

docker container rm --force redis-host

#### Explore Redis on Kubernetes

Now we are ready to repeat some of the steps from the previous section, but this time in Kubernetes environment. Let’s use Minikube for this.

* Start Minikube if not already started
* Launch a Redis pod

kubectl run redis-host --port=6379 --image=redis --generator=run-pod/v1

* Connect to it

kubectl exec -it redis-host redis-cli

* Create multiple key-value pairs with

MSET cust:001:M Ivan cust:002:F Gergana cust:003:M Petar cust:004:F Dimana

* And ask only for those ending with F

KEYS \*F

* Get values of multiple keys

MGET cust:002:F cust:004:F

* Quit the pod and clean up

exit

kubectl delete pod redis-host

Alternative approach could to go declarative, as we always should go:

* Examine the redis-pod.yml file
* Create the pod

kubectl create -f redis-pod.yml

* Connect the same way, we did earlier

kubectl exec -it redis-host redis-cli

* And repeat the steps if you like. Don’t forget to clean up with

kubectl delete -f redis-pod.yml

#### Simple PHP application with Redis backend

The application has two containers – one for the Redis database, and one that provides the PHP and Apache roles.

* Go to folder M4/M4-1/3-docker-simple-app
* Start the Redis container

docker container run -d --name redis-host redis

* Start the other container

docker container run -d --name web-host -v "$(pwd)"/web:/var/www/html:ro --link redis-host:redis-master -p 80:80 shekeriev/php-apache-redis

* Now open a browser tab and navigate to <http://localhost>
* Refresh several times
* Open a connection to the redis-host container

docker container exec -it redis-host redis-cli

* List the databases with keys defined

INFO keyspace

* Ask for the keys

KEYS \*

* Show key value

GET visited

* Set expiration on the key

EXPIRY visited 10

* Then check for the expiration status

TTL visited

* After 10 seconds, return to the browser tab and refresh
* Ask for its value again

GET visited

* Check the expiration status

EXPIRY visited

* Value of -1 means there is no expiration set, and value of -2 means that the key has expired
* Exit and remove both containers

exit

docker container rm --force redis-host web-host

#### Guestbook application on Kubernetes

The following section is borrowed from this article:

<https://kubernetes.io/docs/tutorials/stateless-application/guestbook/>

We must execute the following steps:

* Create the Redis master pod and check the result

kubectl create -f redis-master-deployment.yml

kubectl get pod

* When the pod is in running state, we can check the logs:

kubectl logs -f redis-master-57fc67768d-dh27l

* Then let’s start the service for the Redis master:

kubectl create -f redis-master-service.yml

kubectl get service

* Now we can create the slave nodes for Redis:

kubectl create -f redis-slave-deployment.yml

kubectl get pods

* And of course their service:

kubectl create -f redis-slave-service.yml

kubectl get service

* Last two parts of this application are the application code pods

kubectl create -f frontend-deployment.yml

kubectl get pods

* And their service:

kubectl create -f frontend-service.yml

kubectl get service

* Now, lets’s check how we can access the service:

minikube service frontend –url

* Open a browser tab and navigate to the url. Enter few messages
* Now open the redis-cli on one of the nodes:

kubectl exec -it redis-slave-57f9f8db74-gsjs9 redis-cli

* Show all keys and request current value

KEYS \*

GET messages

* Get information about the node

INFO

exit

* Okay. Let’s scale up the frontend pods:

kubectl scale deployment frontend --replicas=5

kubectl get pods

* In a similar way, we can scale them down if we like:

kubectl scale deployment frontend --replicas=2

kubectl get pods

* It is time to clean up everything:

kubectl delete service -l app=redis

kubectl delete service -l app=guestbook

kubectl delete deployment -l app=redis

kubectl delete deployment -l app=guestbook

* Check if anything left:

kubectl get pods

#### Additional resources (retwis application)

It you are interested in further exploration of Redis as a backend for a php web application, you can check the following links:

* A post with some explanations about the sample application retwis

<https://redis.io/topics/twitter-clone>

* Project’s code:

<https://github.com/antirez/retwis>

## Part 2: MongoDB

#### Explore MongoDB on Docker

Let’s get familiar to basic extent with MongoDB. For this purpose, we will start a Docker container, connect to it, and execute set of commands:

* Start the MongoDB container

docker container run -d --name mongo-host mongo:latest

* Connect to it

docker container exec -it mongo-host bash

* Check what databases are there

show dbs;

* Switch to and create a new database – customers

use customers;

* Create two collections of documents – one for the individual customers and one for the companies

db.createCollection('individuals');

db.createCollection('companies');

* Now if we ask for the collection, we must see them both

show collections;

* Let’s insert one customer in the individuals’ section

db.individuals.insert({customer\_id: "i-001", name: "Ivan Petkov", balance: 1000});

* Now, add two more, but in one command

db.individuals.insert([{customer\_id: "i-002", name: "Tsvetanka Kirilova", balance: 0, sex: "F"}, {customer\_id: "i-002", name: "Gerogi Gospodinov", balance: 100, sex: "M"}]);

* Even though we changed the structure, the insert operation succeeded
* Let’s examine what we have so far:

db.individuals.find();

* We can see the result with different format

db.individuals.find().pretty();

* It appears that we did some mistakes with the third customer. Let’s correct them:

db.individuals.update({\_id: ObjectId("5c2f201ff3e21f82a214c710")}, {"customer\_id" : "i-003", "name" : "Georgi Gospodinov", "balance" : 100, "sex" : "M"});

db.individuals.find();

* If we want to correct just one or two fields, instead of listing all, we can use alternative construction:

db.individuals.update({customer\_id: "i-003"}, {$set: {balance: 2000}});

db.individuals.find();

* We can remove fields. For example, let’s remove the balance field of customer with i-002:

db.individuals.update({customer\_id: "i-002"}, {$unset: {balance: 0}});

db.individuals.find();

* In a similar way, we can add fields to documents:

db.individuals.update({customer\_id: "i-001"}, {$set: {sex: "M"}});

db.individuals.update({customer\_id: "i-002"}, {$set: {balance: 850}});

db.individuals.find();

* If we fire an update command, but there is no matching record:

db.individuals.update({customer\_id: "i-004"}, {customer\_id: "i-004", "name": "Dimana Stoeva", "balance": 5000, "sex": "F"});

* Nothing will happen. In fact, there is one additional option for cases like this:

db.individuals.update({customer\_id: "i-004"}, {customer\_id: "i-004", "name": "Dimana Stoeva", "balance": 5000, "sex": "F"}, {upsert: true});

db.individuals.find();

* There is an alternative approach to insert a record. First set a variable, and then insert the variable:

i1 = {customer\_id: "i-005", "name": "Tanya Dimova", "balance": 1500, "sex": "F"}

db.individuals.insert(i1);

db.individuals.find();

* As with the other databases that we used to work with, here we can do aggregations as well. For example, let’s count all female customers:

db.individuals.aggregate([{$match: {sex: "F"}}, {$group: {\_id: "females", count: {$sum: 1}}}]);

* Now, let’s ask for the total balance per sex:

db.individuals.aggregate([{$group: {\_id: "$sex", totalBalance: {$sum: "$balance"}}}]);

* What if we want to order the result from lowest to highest balance value:

db.individuals.aggregate([{$group: {\_id: "$sex", totalBalance: {$sum: "$balance"}}}, {$sort: {totalBalance: 1}}]);

* We can have more than one aggregate function:

db.individuals.aggregate([{$group: {\_id: "$sex", totalBalance: {$sum: "$balance"}, recordCount: {$sum: 1}}}]);

* Now, let’s delete a record:

db.individuals.remove({customer\_id: "i-003"});

db.individuals.find();

Now let’s add two documents in the other collection:

* Add one with the following structure:

db.companies.insert({customer\_id: "c-001", name: "Company A Ltd.", VAT: "BG1234567890", balance: 10000});

* And another one with extended (nested) structure:

db.companies.insert({customer\_id: "c-002", name: "Company B Ltd.", VAT: "BG1234567890", balance: 15000, phones: [{mobile: "0888/11-22-33"}, {landline: "02/123-45-67"}]});

* We can ask for the documents the same way:

db.companies.find().pretty();

Now, we can clean up the data, and the container:

* Instead of deleting documents one by one, there is faster way:

db.companies.drop();

db.individuals.drop();

* Now, exit from the CLI tool and from the container:

exit

exit

* As a last step, remove the container:

docker container rm --force mongo-host

#### MongoDB + Mongo Express on Docker

If we want to interact in a better and easier way with the MongoDB, then we can spin-up MongoDB Express as well. We can do it in two steps, but it can be done with a docker-compose in one step as well:

* If we have docker-compose installed, we can run:

docker-compose -f docker-compose.yml up -d

* Now, we can open a browser tab and navigate to: <http://localhost:8081>
* Let’s explore a little bit
* Why not create a new database, collection, and few documents
* Query for the documents
* Close the browser
* Delete the service

docker-compose down

#### Simple PHP application with MongoDB backend

Let’s check a simple web application with MongoDB backend split in two containers:

* Start the MongoDB container and seed the database:

docker container run -d --name mongo-host -v "$(pwd)"/db:/docker-entrypoint-initdb.d mongo:latest

* Start the second container:

docker run -d --name web-host -v "$(pwd)"/web:/var/www/html:ro --link mongo-host:mongo-host -p 80:80 shekeriev/php-apache-mongo

* Now open a browser tab and navigate to <http://localhost>
* Refresh several times
* Open a connection to the mongo-host container

docker container exec -it mongo-host mongo

* List the databases and pick up the one named test

show dbs

use test

* List collections

show collections

* Ask for all documents in counters collection

db.counters.find();

* Reset the counter or set a different value

db.counters.update({counter: 'visits'}, {$set: {total: 100}});

* Return to the browser tab and refresh again
* Close the connection

exit

* Remove both containers

docker container rm --force web-host mongo-host

#### MongoDB on Kubernetes

//TO DO

## Part 3: Apache Cassandra

#### Explore Cassandra on Docker

As usual, it is a good option first to get familiar with Cassandra in a Docker environment:

* Start a Cassandra container:

docker container run -d --name cassandra-host cassandra:latest

* And let’s connect to it:

docker container exec -it cassandra-host bash

* The CQL shell itself is started with

cqlsh

* Now, that we are in, let’s get some help

help

* Let’s get familiar with the environment

SHOW HOST

SHOW VERSION

DESCRIBE KEYSPACES

* It is time to create our own keyspace:

CREATE KEYSPACE catalog WITH replication = {'class': 'SimpleStrategy', 'replication\_factor': 1};

* Now, enter the keyspace

USE catalog;

* And create a column family or table to store our books for example:

CREATE TABLE books ( title text, authors list<text>, year int, PRIMARY KEY (title));

* We can examine the table just created:

DESC TABLE catalog.books;

* Let’s insert two rows:

INSERT INTO catalog.books (title , authors , year ) VALUES ( 'Book 1', ['Author 1', 'Author 2'], 2018);

INSERT INTO catalog.books (title , authors , year ) VALUES ( 'Book 2', [], 2018);

* We can ask for the data:

SELECT \* FROM catalog.books ;

* Let’s insert one more row:

INSERT INTO catalog.books (title , authors , year ) VALUES ( 'Book 3', ['Author 3', 'Author 2'], 2017);

* Let’s do few queries:

SELECT \* FROM catalog.books ;

SELECT \* FROM catalog.books WHERE title IN ('Book 1');

SELECT \* FROM catalog.books WHERE authors CONTAINS 'Author 2';

SELECT \* FROM catalog.books WHERE authors CONTAINS 'Author 2' ALLOW FILTERING ;

* Of course, we can do aggregation queries as well:

SELECT COUNT(\*) FROM catalog.books;

* If try to see counts by year, we will face an error:

SELECT year, COUNT(\*) FROM catalog.books GROUP BY year;

* The reason is that GROUP BY is allowed only on the PRIMARY KEY. It means, that we should know upfront what queries will be executed, and structure the table accordingly
* Let’s create new table, this time for magazines:

CREATE TABLE magazines (title text, year int, copies int, price decimal, categories list <text>, PRIMARY KEY (year, title));

* And insert several rows:

INSERT INTO catalog.magazines (title, year, copies, price, categories) VALUES ('Magazine 1', 2018, 150, 5.40, ['IT', 'Development']);

INSERT INTO catalog.magazines (title, year, copies, price, categories) VALUES ('Magazine 2', 2018, 450, 4.80, ['Lifestyle', 'Women']);

INSERT INTO catalog.magazines (title, year, copies, price, categories) VALUES ('Magazine 3', 2018, 236, 3.80, ['Lifestyle', 'Fashion', 'Women']);

INSERT INTO catalog.magazines (title, year, copies, price, categories) VALUES ('Magazine 1', 2017, 50, 5.40, ['IT', 'Development']);

INSERT INTO catalog.magazines (title, year, copies, price, categories) VALUES ('Magazine 2', 2017, 410, 4.80, ['Lifestyle', 'Women']);

* Now, we have enough data to do two more aggregation queries:

SELECT year, COUNT(\*), SUM(copies) FROM catalog.magazines GROUP BY year;

SELECT year, COUNT(\*), SUM(copies) FROM catalog.magazines WHERE categories CONTAINS 'Women' GROUP BY year ALLOW FILTERING;

* Finally, we can exit and remove the container:

exit

exit

docker container rm --force cassandra-host

#### Simple web application

Let’s examine a small web application with Cassandra backend and PHP+Apache as frontend, distributed between two containers:

* Go to folder M4/M4-3/ 2-simple-app
* Start the Cassandra container:

docker container run -d --name cassandra-host cassandra:latest

* Start the PHP+Apache container:

docker container run -d --name=web-host -p 80:80 -v "$(pwd)"/web:/var/www/html:ro --link=cassandra-host:cassandra-host shekeriev/php-apache-cassandra

* Now open a new tab in the browser and navigate to <http://localhost/init.php>
* Then, either in the same tab, or in a new one, navigate to <http://localhost/>
* When we are ready, we can clean up as usual:

docker container rm –force web-host cassandra-host

#### Cassandra cluster on local Docker

We can try to create a Cassandra cluster on our local Docker host:

* Create the master node:

docker run -d --name cassandra-1 cassandra:latest

* If we have a limited amount of available resources, we can alter the configuration by:

docker run -d --name cassandra-1 -e MAX\_HEAP\_SIZE=512M -e HEAP\_NEWSIZE=512M cassandra:latest

* We can start second node:

docker run -d --name cassandra-2 -e CASSANDRA\_SEEDS="$(docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' cassandra-1)" -e MAX\_HEAP\_SIZE=512M -e HEAP\_NEWSIZE=512M cassandra:latest

* And add a third node, by changind the approach of linking:

docker container run -d --name cassandra-3 -e MAX\_HEAP\_SIZE=512M -e HEAP\_NEWSIZE=512M --link cassandra-1:cassandra cassandra:latest

* While nodes are added, we can monitor the status of the cluster and the process as a whole:

docker container exec -it cassandra-1 sh -c 'nodetool status'

* Then we can log on to one of the nodes and try few commands, or just clean up everything

docker container rm --force cassandra-1 cassandra-2 cassandra-3

#### Cassandra cluster on Kubernetes

This section is borrowed from: <https://kubernetes.io/docs/tutorials/stateful-application/cassandra/>

The steps to build a Cassandra cluster on Kubernetes, running on Minikube are:

* If you have a running Minikube, stop it and recreate it with the following command:

minikube start --memory 5120 --cpus=4

* Now, create the Cassandra service and check if everything is okay:

kubectl create -f cassandra-service.yaml

kubectl get svc

* Next step is to create the deployment:

kubectl create -f cassandra-statefulset.yaml

kubectl get statefulset cassandra

* Now, monitor the pods creation. It can take several minutes:

kubectl get pods -l="app=cassandra"

* After a while, we can ask for status of cluster nodes:

kubectl exec -ti cassandra-0 -- nodetool status

* Once everything is up, we can increase or modify the replicas count:

kubectl edit statefulset cassandra

* Then check if the changes are applied:

kubectl get statefulset cassandra

* And finally remove everything:

kubectl delete statefulset -l app=cassandra

kubectl delete pvc -l app=cassandra

kubectl delete service -l app=cassandra

* As last step, delete the Minikube if you do not need it any more:

minikube delete

#### Additional resources (twissandra application)

It you are interested in further exploration of Cassandra as a backend for a python web application, you can check the following link: <https://github.com/twissandra/twissandra>