1. Introduction

This documentation is a cheat sheet for implementing a demo component system of the Kubernetes-aided federated database system. The implementation process follows the architecture of the demo system, as illustrated in Figure 1.1. In particular, Section 3 describes the implementation of read-only database containers, which corresponds to the database layer. Section 4 describes the implementation of a containerized Node. js program that will connect to the database and expose data through RESTful API's, which corresponds to the data access layer. Section 5 describes the deployment and management of the containerized databases in Kubernetes through KIND, which corresponds to the service layer. At last, Section 6 gives a minimal example about the intra-query parallelism experiment.

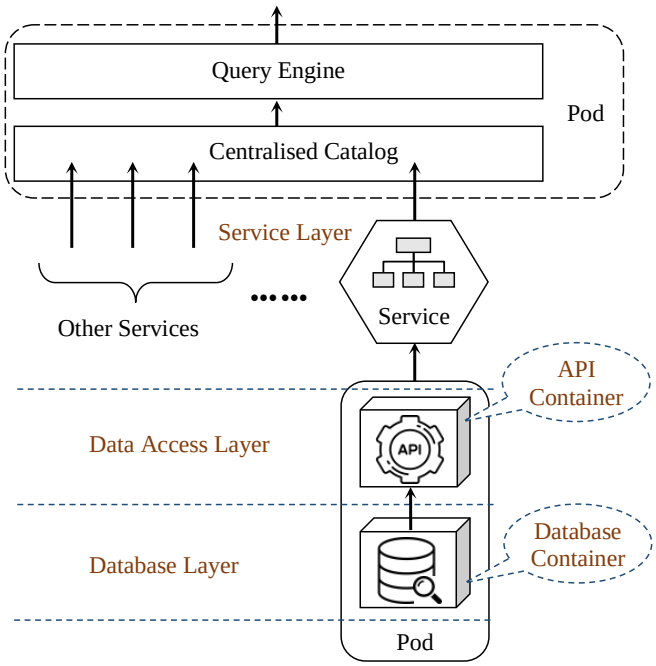

Figura 1.1: The essential parts and architecture of the demo system.

1.1. Environmental Preparation

Among the important points to consider is the operating system used for the implementation, in this case Windows 10 was used, on an Omen HP 17-an003la laptop, which has the following specifications:

Product number	1GX66LA
Product name	OMEN by HP - 17-an003la
Microprocessor	Intel® Core™ i7-7700HQ (base frecuency 2,8 GHz, 6 MB cache, 4 cores)
Standard memory	16 GB SDRAM DDR4-2133 (2 x 8 GB)
Video graphics	NVIDIA® GeForce® GTX 1070 (8 GB GDDR5 dedicated)
Network interface	LAN 10/100/1000 GbE
Operative system	Windows 10 Home 64

Cuadro 1.1: Specifications of the physical machine components.

Additionally, it is necessary to take into account that for the implementation of the database container a dataset to preference was used, more specifically it corresponds to a 2017 Amazon Spot Service instance price history. So if you require testing with other datasets, it will be necessary to make changes to the database table creation and add the dataset to the respective directory.

2. Relevant Tools

It requires the installation and download of a set of tools essential for the development and implementation of the proposed cluster design.

2.1. Node.js

Nodejs is an asynchronous event-driven JavaScript runtime environment designed to create scalable network applications. To install it within Windows, it is necessary to access the following link, which will allow us to download the installer or the binary in its version v16.16.0.

2.2. Docker Desktop

Tool that provides an Integrated Development Environment, or by its abbreviation, IDE, is necessary to install for the management of our images, containers and creation of cluster nodes. To obtain it, you can download the installer from the following link, with version v20.10.16 of Docker Engine.

2.3. Kubectl

It is an interface that simplifies in a standardized way, the execution of command lines on Kubernetes deployments. To download the binary of this tool, you can do it from the following link, in its version v.1.24.0

2.4. KIND

A tool used to run local Kubernetes clusters using Docker container "nodes". To download the binary of this tool, the following link will take us to its download in version v0.14.0. Finally, it will be necessary to rename the file from "kind-windows-amd64"to "kind".

It is important that both the Kubectl and KIND binaries are in the same directory when starting the cluster creation.

2.5. Jupyter Notebook using Anaconda

Jupyter Notebook is an open source web interface that allows code execution through the browser in multiple languages. By following the following tutorial, you will be able to install and configure Jupyter Notebook through Anaconda.

3. Implementation of the Database Layer

The database layer is realised by the implementation of a read-only database container with preloaded data. Since it supports stateless data access at runtime and once the container is deployed, it is able to process read requests.

For its implementation, we start by entering the Windows 10 command console to create a new directory in which to work, such as:

```
> md "Implementations"
> cd "Implementations"
\Implementations> md "Preloaded db"
\Implementations> cd "Preloaded db"
```

For the creation of this read-only database container image, the definition of 4 files is required:

3.1. Dockerfile

It should be noted that, the name of the dataset that was added in the directory is exactly the same in the Dockerfile. Following the example where the dataset is named "2M.txt", the Dockerfile should look like below:

Dockerfile

```
FROM ubuntu:18.04
COPY . /home/diicc/mydockerbuild
WORKDIR /home/diicc/mydockerbuild
RUN apt-get update -y
RUN apt-get install --assume-yes apt-utils
RUN chmod a+x ./install_mysql.sh ./load_data.sh ./select_data.sh
RUN ./install_mysql.sh
RUN ./load_data.sh
RUN rm -f ./install_mysql.sh ./load_data.sh ./2M.txt
CMD ./select_data.sh
```

3.2. Silent MySQL installation

This script will perform a silent installation of MySQL, but without considering the creation of the database or users.

install_mysql.sh

```
export DEBIAN_FRONTEND=noninteractive

MYSQL_ROOT_PASSWORD='1q2w3e4r'

echo debconf mysql-server/root_password password $MYSQL_ROOT_PASSWORD |
    debconf-set-selections
echo debconf mysql-server/root_password_again password $MYSQL_ROOT_PASSWORD |
    debconf-set-selections

apt-get -qq install mysql-server > /dev/null # Install MySQL quietly
service mysql start

apt-get -qq install expect > /dev/null

tee ~/secure_our_mysql.sh > /dev/null << EOF
spawn $(which mysql_secure_installation)
expect "Enter password for user root:"
```

```
send "$MYSQL_ROOT_PASSWORD\r"
expect "Press y|Y for Yes, any other key for No:"
send "y\r"
expect "Please enter 0 = LOW, 1 = MEDIUM and 2 = STRONG:"
send "2\r"
expect "Change the password for root ? ((Press y|Y for Yes, any other key for
    No) :"
send "n\r"
expect "Remove anonymous users? (Press y|Y for Yes, any other key for No) :"
send "y\r"
expect "Disallow root login remotely? (Press y|Y for Yes, any other key for No
    ) :"
send "y\r"
expect "Remove test database and access to it? (Press y|Y for Yes, any other
    key for No) :"
send "y\r"
expect "Reload privilege tables now? (Press y|Y for Yes, any other key for No)
     :"
send "y\r"
EOF

expect ~/secure_our_mysql.sh

rm -v ~/secure_our_mysql.sh # Remove the generated Expect script
echo "MySQL setup completed. Insecure defaults are gone. Please remove this
    script manually when you are done with it (or at least remove the MySQL
    root password that you put inside it."

mysql -version

sed -i 's/127.0.0.1/0.0.0.0/g' /etc/mysql/mysql.conf.d/mysqld.cnf
sed -i '/max_allowed_packet*/c\max_allowed_packet=1073741824' /etc/mysql/mysql.
    conf.d/mysqld.cnf
sed -i '/key_buffer_size*/c\key_buffer_size=100M' /etc/mysql/mysql.conf.d/
    mysqld.cnf
sed -i '/max_connections*/c\max_connections=400' /etc/mysql/mysql.conf.d/
    mysqld.cnf
sed -i '/\[mysqld\]/a\# Skip reverse DNS lookup\nskip-name-resolve' /etc/mysql
    /mysql.conf.d/mysqld.cnf

service mysql stop
```

3.3. Loading the dataset into the database

This script is responsible for creating a database with its respective table, users and associated access permissions, and then load the data from the dataset, which must match the one copied in the directory, to the database, and stop the MYSQL system until deployment.

load_data.sh

```
service mysql start

rootpsw='1q2w3e4r'
usertest='mydb'
passtest='#1A2b%3C4d5E!'
tabletest='mytab'

mysql -uroot -p$rootpsw <<MYSQL_SCRIPT
CREATE DATABASE $usertest;
CREATE USER '$usertest'@'localhost' IDENTIFIED BY '$passtest';
CREATE USER '$usertest'@'%' IDENTIFIED BY '$passtest';
GRANT ALL PRIVILEGES ON *.* TO '$usertest'@'localhost' WITH GRANT OPTION;
GRANT ALL PRIVILEGES ON *.* TO '$usertest'@'%' WITH GRANT OPTION;
FLUSH PRIVILEGES;
MYSQL_SCRIPT

echo "MySQL user created."
echo "Username:   $usertest"
echo "Password:   $passtest"

mysql -u$usertest -p$passtest -D$usertest <<MYSQL_SCRIPT
CREATE TABLE $tabletest (SPOTINSTANCEPRICE VARCHAR(18), PRICE VARCHAR(15),
    FECHA VARCHAR(30), INSTANCE VARCHAR(25), MACHINE VARCHAR(25), ZONE VARCHAR
    (25) );
LOAD DATA LOCAL INFILE '/home/diicc/mydockerbuild/2M.txt' INTO TABLE
    $tabletest CHARACTER SET utf8 FIELDS TERMINATED BY '\t' LINES TERMINATED
    BY '\n';
MYSQL_SCRIPT

service mysql stop
```

3.4. Internal start of the service

With all the above ready, this will be the script that will basically be in charge of enabling the start of the MySQL service when the container is deployed.

select_data.sh

```
service mysql start
tail -F /var/log/mysql/error.log
```

Each time you want to make a new deployment of the container with a different dataset, it is necessary to take the precaution of making the respective name changes in the Dockerfile and load_data.sh files.

With the base files ready and being located in the "Preloaded db"directory path, we can finally create our Docker image and upload it to the Docker Hub repository using the command lines:

```
\Implementations\Preloaded db> docker build -t nicosaldias/preloadeddata:2m .
\Implementations\Preloaded db> docker push nicosaldias/preloadeddata:2m
```

4. Implementation of the Data Access Layer

When the database container is ready, it is necessary to define an application that takes care of the process of generating query traffic to the database. For this, we propose the implementation of a containerized program in Node.js, so that through a RESTful API.

Starting, inside the CMD console, we will create the directory in which we will work:

```
\Implementations\Preloaded db> cd ..
\Implementations> md Node-app
\Implementations> cd Node-app
```

4.1. App definition

We will create a new Node.js app and we will install 3 dependencies to establish connections with MySQL databases and to allow the return of data as an API.

```
\Implementations\Node-app> npm init -y
\Implementations\Node-app> npm install express mysql promise-mysql
```

Modify the "package.json"file in the scripts variable, as follows:

package.json

```
{
  "name": "Node",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node single_connection.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.1",
    "mysql": "^2.18.1",
    "promise-mysql": "^3.3.2"
  }
}
```

Thus, our application will be able to run using the "npm startçommand. Now we only have to define which queries our application will make, these were defined in the following file:

sigle_connection.js

```
var http = require('http'),
    express = require('express'),
    app = express(),
    server = http.Server(app),
    mysql = require('mysql');

const { url } = require('inspector');
const os = require('os');

console.log('Server Started');
```

```
const max = 25000;
const min = 0;



app.use(express.json());

server.listen(3000)

function get_connection(){
    return mysql.createConnection({
        host:'localhost',
        user:'mydb',
        password:'#1A2b%3C4d5E!',
        database:'mydb',
        port:3306
    })
}



function aleatorio(min,max){
    var num = Math.floor(Math.random() * ((max+1) - min) + min);
    return num;
}



app.get('/message', function(req,res){
    res.send('Hola mundo!');
});

app.get('/pod', function(req,res){
    res.send(os.hostname());
});

app.get('/count', function(req,res){
    var conn = get_connection()
    conn.connect(function(err){
        if(!err){
            conn.query('SELECT COUNT (*) FROM mytab',function(err2,records,
    fields){
                if(!err2){
                   res.send(records)
                   console.log(os.hostname());
```

```
                }
                conn.end()
            })
        }
        else{
            res.send(err)
        }
    })
});

app.get('/select', function(req, res){
    var url_min = parseInt(req.query.min);
    var url_max = parseInt(req.query.max);
    var conn = get_connection()
    console.log(url_min);
    console.log(url_max);
    conn.connect(function(err){
        if(!err){
            conn.query('SELECT * FROM mytab LIMIT ?,?',[url_min,url_max],
    function(err2,records,fields){
                if(!err2){
                    res.send(records)
                    console.log(url_min);
                    console.log(url_max);
                }
                conn.end()
            })
        }
        else{
            res.send(err)
        }
     })

});

app.get('/selectall', function(req,res){
     var conn = get_connection()
     conn.connect(function(err){
         if(!err){
             conn.query('SELECT * FROM mytab',function(err2,records,fields){
                 if(!err2){
                    res.send(records)
```

```
                    console.log(os.hostname());
                }
                conn.end()
            })
        }
        else{
            res.send(err)
        }
    })
});


app.get('/randomselect', function(req,res){
    var conn = get_connection()
    var ran = aleatorio(min,max);
    console.log(ran);
    conn.connect(function(err){
        if(!err){
            conn.query('SELECT * FROM mytab LIMIT ?',[ran],function(err2,
    records,fields){
                if(!err2){
                    res.send(records)
                    console.log(os.hostname());
                }
                conn.end()
            })
        }
        else{
            res.send(err)
        }
    })
});
```

To test that the application runs correctly, we test the commands:

```
\Implementations\Node-app> npm start
\Implementations\Node-app> CTRL + C
```

This application is capable of responding to the following requests:

- **http://localhost:3000/selectall**: To read all the data in the table
- **http://localhost:3000/randomselect**: To read a random number of rows

from the table

- **http://localhost:3000/pod**: Responses with the name of the host of the app
- **http://localhost:3000/count**: Returns the number of rows of the table has
- **http://localhost:3000/select?min='x'&max='y'**: Returns the y rows after skip reading the first x rows.

4.2. Dockerfile

To containerize this application, it is necessary to define the Dockerfile that will allow us to create the image and later upload it to the repository.

Dockerfile

```
FROM node:14
WORKDIR /app
COPY package.json /app
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm","start"]
```

To prevent the Docker image that our application includes the dependencies and installations performed, we can create this '.dockerignore' file. **.dockerignore**

```
node_modules
```

Finally, we create the image of our Node.js app and upload it to the Docker Hub repository.

```
\Implementations\Node-app> docker build -t nicosaldias/node-db:api .
\Implementations\Node-app> docker push nicosaldias/node-db:api
```

5. Implementation of the Service Layer

In order to deploy and manage the database container and the API container together, it is necessary to make use of Kubernetes clusters and its deployment and services mechanisms.

We start by exiting the directory we use for the Node.js app, and create a new one for the cluster launch and its deployments. It is important to remember that inside this directory must also be the kubectl and kind binaries.

```
\Implementations\Node-app> cd ..
\Implementations> md Cluster
\Implementations> cd Cluster
```

5.1. Cluster configuration

We will define a cluster composed of 3 nodes, 1 node that will play the role of control-plane, to manage the complete state of the cluster and 2 worker nodes, which will be in charge of running the containers with our applications inside pods.

kind-cluster.yaml

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
- role: worker
```

We launch the cluster using the command:

```
\Implementations\Cluster> kind create cluster --name mycluster --config kind-
    cluster.yaml
```

In case you need to delete the newly created clusters, you can use the following command, paying attention to the name parameter that specifies which cluster will be deleted.

```
\Implementations\Cluster> kind delete cluster --name mycluster
```

5.2. Deployments

For the deployment of our applications we considered defining a namespace under which our pods and services will be selected, 2 multi-container pod deployments, composed of a read-only database container and the Node.js app, and finally, 2 NodePort services to expose these replicas later on.

node-db.yaml

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: node

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-db-deploy1
  namespace: node
  labels:
    app: node1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: node1
  template:
    metadata:
      labels:
        app: node1
    spec:
      containers:
        - name: node
          image: nicosaldias/node-db:api
          ports:
            - containerPort: 3000
        - name: db
          image: nicosaldias/preloadeddata:2.8k
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: 1q2w3e4r
          ports:
            - containerPort: 3306

---
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: node-db-deploy2
  namespace: node
  labels:
    app: node2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: node2
  template:
    metadata:
      labels:
        app: node2
    spec:
      containers:
        - name: node
          image: nicosaldias/node-db:api
          ports:
            - containerPort: 3000
        - name: db
          image: nicosaldias/preloadeddata:2.8k
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: 1q2w3e4r
          ports:
            - containerPort: 3306
---
apiVersion: v1
kind: Service
metadata:
  namespace: node
  name: node-service1
spec:
  ports:
    - port: 3000
      targetPort: 3000
      protocol: TCP
  type: NodePort
  selector:
    app: node1

---
```

```
apiVersion: v1
kind: Service
metadata:
  namespace: node
  name: node-service2
spec:
  ports:
    - port: 3000
      targetPort: 3000
      protocol: TCP
  type: NodePort
  selector:
    app: node2
```

We apply this configuration to the cluster and wait about 4 minutes for the deployment to complete and the pods to be running:

```
\Implementations\Cluster> kubectl apply -f node-db.yaml
\Implementations\Cluster> kubectl get all -o wide -n node
```

5.3. Exposing the deployments

For the external traffic to be processed by our application instances, it is necessary to expose the services we have just deployed to the localhost of our physical machine, for this we will expose individually the 2 services, assigning them different access ports.

It is important to consider that each of these commands must be executed in different command consoles.

```
\Cluster> kubectl port-forward service/node-service1 3000:3000 -n node
\Cluster> kubectl port-forward service/node-service2 3001:3000 -n node
```

6. The Intra-query Parallelism Experiment

If we want to compare the use of sequential read processes, to one that makes use of multiple concurrent threads by splitting the request into 2, we can use the following codes written in Python to measure the data read latency on the instances within the newly implemented cluster.

In this code we make use of a single worker node, which is represented by the service 'node-service-1' and is assigned port 3000. So we start by making a query to find out how many rows the database table has and then make a complete read of the existing data, returning finally the time it took to read the data.

sigle-process-request.ipynb

```
import time
import threading
import requests
import math


def count():
    #start=time.perf_counter()
    receive = requests.get('http://localhost:3000/count')
    wjdata = receive.json()
    rows = int(wjdata[0]['COUNT (*)'])
    #thread_end = time.perf_counter() - global_start
    #end=time.perf_counter()
    #print('Count global:',thread_end,'\n')
    #print('Count:',end-start,'\n')
    return rows

def select_all(num):
    #start=time.perf_counter()
    receive = requests.get('http://localhost:3000/selectall')
    thread_end = time.perf_counter() - global_start
    #end=time.perf_counter()
    print('Thread global',num,':',thread_end,'\n')
    #print('Thread',num,':',end-start,'\n')


global_start = 0

global global_start
global_start = time.perf_counter()

count()
select_all(0)
```

On the other hand, for this version we tried to make use of 2 concurrent threads

in order to divide the number of rows to read within the table, that is why the function test1 is assigned port 3000, which refers to the service 'node-service-1' that will receive the requests to read the first half of the table and the function test2, which is assigned port 3001, referring to the service 'node-service-2' that will process the final half of the data in the table.

multithreaded-requests.ipynb

```
import time
import threading
import requests
import math



def count():
    receive = requests.get('http://localhost:3000/count')
    data = receive.json()
    rows = int(data[0]['COUNT (*)'])
    return rows

def test1(num,half):
    parametros = {'min': '0','max': half}
    receive = requests.get('http://localhost:3000/select', params=parametros)
    thread_end = time.perf_counter() - global_start
    print('Thread global',num,':',thread_end,'\n')



def test2(num,half,total):
    parametros = {'min': half,'max': total}
    receive = requests.get('http://localhost:3001/select', params=parametros)
    thread_end = time.perf_counter() - global_start
    print('Thread global',num,':',thread_end,'\n')



global_start = 0

global global_start
global_start = time.perf_counter()

#get the number of rows in the table
number_rows =count()
```

```
first_half = int(number_rows/2)
second_half = int(number_rows - first_half)

#run the threads
t1 = threading.Thread(target=test1 , args=(1,first_half))
t2 = threading.Thread(target=test2 , args=(2,second_half,number_rows))
t1.start()
t2.start()
```