Scale MobilityDB on AWS cloud User's Manual

COLLABORATORS			
	TITLE : Scale MobilityDB on AWS cloud User's Manual		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Sid Ahmed BOUZOUIDJA	August 3, 2021	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Abstract

MobilityDB is an open source moving object database system. It is based on PostgreSQL and and built on top of PostGIS extension. It provides an effective functionality and new data type in order to store, analyze data trajectory. As MobilityDB deal with a large dataset and it queries is very complex and expensive in term of cpu and memory comsuming. This behavior cannot be supported by a single host machine. There is a way to make the MobilityDB queries more powerful and takes less execution time is to scaling up across a cluster of nodes. We have thinks on the On-premises cluster of machine but this is not enough according to the rapid grows of technologies and 5G network deployment sooner that will makes the MobilityDB dataset drastically big. In this respect the effective way to scale up the MobilityDB is to create our cluster on cloud services and using an orchestration as a control plane that manage the cluster. We have used another tool which is a Citus Data to gain more performance by partitioning tables on shards in single node. All the deployment manifests and configuration files, docker images and the requirements to deploy MobilityDB on AWS is located in this Github repository.

Chapter 1

Introduction

There are many cloud services around the world as AWS, Azure, Google cloud, IBM cloud, Saleforce and so on. In this study we will deploy the MobilityDB on AWS cloud services. The AWS cloud provides a large services, sometime it difficult to make a choice between them in order to adjust it to our application. In order to scaling MobilityDB, we need to keep in mind of two kind of resources. The volume storage and the computation units. In addition to those two resources we need an orchestrator that manage cluster and ensure the availability. Again the AWS provides different orchestrator like ECS for Elastic Container Service, an EKS for Elastic Kubernetes Service, ECR for Elastic Container Registry. The orchestrator used in this study is the EKS one. There is another tool that may increase the MobilityDB performance, this one is away from the cloud services. The citus Data for distribution. Their idea is to partitioning tables on different shards. This mechanism allow the query plan to rooting rapidly the queries to the right data shard based on the hash value. In following we will see the different kind of orchestrator that may support our deployment and than describe the architecture to deploying the scale of MobilityDB.

Chapter 2

Different possibilities to scale MobilityDB on AWS

As we have mentioned, we have several posibilities to deploy MobilityDB on AWS cloud. In order to take advantages of what AWS services provides, we have made a reflection in our choice between ECS (Elastic Container Service) and EKS (Elastic Kubernetes Service). This two environement is very similar and both are linked to AWS services. The only reason to choose the EKS is the portability. If we want for example to migrate our scale MobilityDB environement from AWS to Azure in the future, it will be easy or vice versa.

2.1 Elastic Container Service and Fargate

The ECS is a containers orchestration or a control plane that manage the containers within the EC2 instances. This mode is pretty good because it do not manage the hosting infrastructure. If you want to delegate the host management to AWS services, you can use the Fargate service. The Fargate is a provision server or the capacity provider that provide you resources according to the container demands (CPU and memory). In another words it create automatically a server using the container demand. The advantage is we pay only what our containers are consuming.

2.2 Elastic Kubernetes Service

The EKS is a AWS service that allow us to manage a Kubernetes cluster on AWS ecosystem. The advantage of using the EKS instead the ECS is a portability of your Kubernetes cluster that mean if we want to migrate our cluster to AWS it will be very easy. Even to migrate it to Azure or Google Cloud infrastructure. Another advantage is the popularity of Kubernetes with a large community. The EKS is a control plane that can scheduling and orchestrating a cluster. When you create a EKS, AWS provision a Master node in the background linked with all AWS services as CloudWatch for monitoring, Elastic Load Balancer for load balancing, IAM for users and permissions and VPC for networking. Using the EKS service you can replicate the Master node on other availability zone and regions (link to regions aws) In our deployment guide we have used the region Europe(Paris) eu-west-3, the 3 means that are 3 availability zones. After creating the Cluster(master node) we need to create a workers nodes with EC2 instances and join them to the cluster. EKS service allow us to semi manage the workers nodes using the node group option. If we want to fully manage our workers nodes, again we can use the AWS Fargate provision.

2.3 Citus cluster using the AWS EC2 instance

There is another way to scale our MobilityDB through the Citus cluster using Citus docker image. The idea is simply to create a group of EC2 instances, choose one of them as a Master node and join the worker nodes to the same Master node. Citus docker version provides 3 types of node, the Master, worker and the manager that has as role listening to new worker nodes in the same subnet in order to join them automatically to the Master node. This kind of deployment does not benefit from the AWS services comparing EKS or ECS.

Chapter 3

Deployment using AWS EKS service

In this part we will show you the different layers and components of our deployment in order to scale the MobilityDB using EKS service.

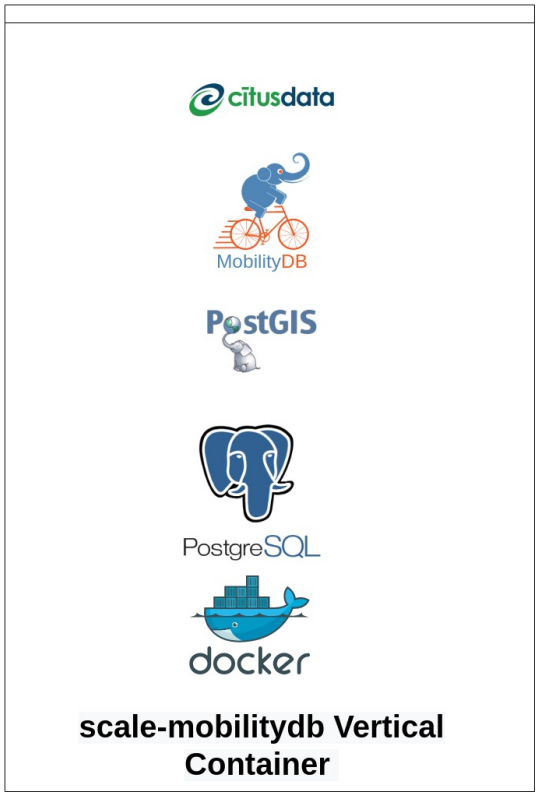
3.1 EKS vertical container view

As you can see in the following figure we have prepared a docker image that contains the MobilityDB environment and Citus environment built on top of it.

3.1.1 Build Citus on top of MobilityDB

This image deploy Citus on top of MobilityDB. The Dockerfile contain both Citus and MobilityDB gist that work adequately. This gist need to be executed in all your cluster nodes if you follow the deployment using Citus cluster. Run it in your host machine if you foollow the deployment on AWS EKS cluster.

```
git clone https://github.com/bouzouidja/scale_mobilitydb.git
cd scale_mobilitydb
docker build -t scalemobilitydb/scalemobilitydb .
```



3.2 Kubernetes Cluster view

After preparing our MobilityDB scale, we can easily deploy it on worker node in Kubernetes cluster using the kubectl command. In the following figure we will show you a small Kubernetes cluster that make MobilityDB scaling. We have a control Plane or the Master node and two workers nodes. The worker in the right it can be seen as storage node, the DS stand for dense storage. The worker in the left it can be seen as compute node, the DC stand for dense compute. As you can see a pod is created within the worker node using the scale-mobilityDB container prepared. Once you have configure this Kubernetes architecture, you can deploy it in any cloud service platform that provide Kubernetes. Maybe you are asking a question why there is a storage node?. In my opinion the data need to be loaded within a cluster before using the MobilityDB queries. Else another AWS services may be explored as EMR for Elastic MapReduce and AWS Apache Spark or AWS S3 for Simple Storage Service to make web-scale computing easier.

3.2.1 Install requirements

3.2.1.1 Install kubectl

```
curl -o kubectl https://amazon-eks.s3-us-west-2.amazonaws.com/1.21.2/2021-07-05/ ↩
    bin/linux/amd64/kubectl
# Check the SHA-256 sum for your downloaded binary.
openssl sha1 -sha256 kubectl

# Apply execute permissions to the binary.
chmod +x ./kubectl

# Copy the binary to a folder in your PATH. If you have already installed a  ↩
    version of kubectl, then we recommend creating a $HOME/bin/kubectl and  ↩
    ensuring that $HOME/bin comes first in your $PATH.
```

```
mkdir -p $HOME/bin && cp ./kubectl $HOME/bin/kubectl && export PATH=$PATH:$HOME/ ↩
    bin

# (Optional) Add the $HOME/bin path to your shell initialization file so that it  ↩
    is configured when you open a shell.
echo 'export PATH=$PATH:$HOME/bin' >> ~/.bashrc

# After you install kubectl , you can verify its version with the following  ↩
    command:
kubectl version --short --client
```

3.2.1.2 Install eksctl

Download and extract the latest release of eksctl with the following command.

```
curl --silent --location "https://github.com/weaveworks/eksctl/releases/latest/ ↩
    download/eksctl_$(uname -s)_amd64.tar.gz" | tar xz -C /tmp
# Move the extracted binary to /usr/local/bin.
sudo mv /tmp/eksctl /usr/local/bin
# Test that your installation was successful with the following command.
eksctl version
```

Install and configure the aws CLI (Command Line Interface) environment

```
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/install
aws --version
# aws-cli/2.1.29 Python/3.7.4 Linux/4.14.133-113.105.amzn2.x86_64 botocore/2.0.0
```

AWS requires that all incoming requests are cryptographically signed.

- Access Key ID
- Secret access Key
- AWS Region
- Output Format

Let configure some mandatory information in order to use the aws services. Navigate to AWS console

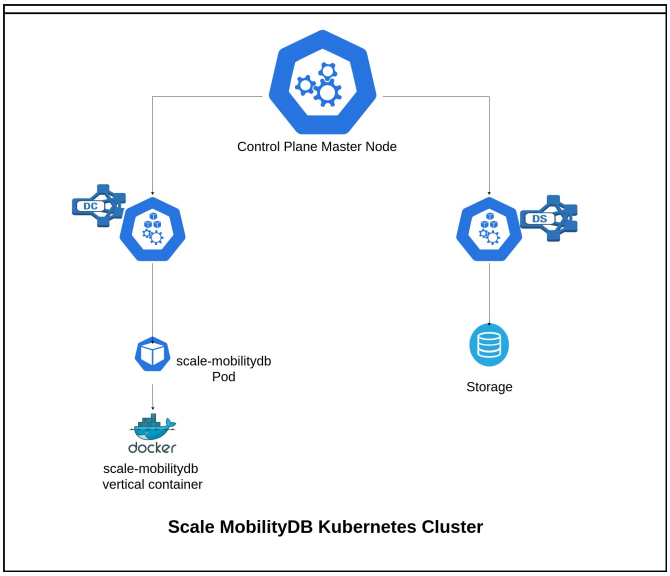
- In the navigation pane, choose Users.
- Choose the name of the user whose access keys you want to create, and then choose the Security credentials tab.
- In the Access keys section, choose Create access key.
- To view the new access key pair, choose Show. You will not have access to the secret access key again after this dialog box closes. Save the access key id and the secret access key somewhere.

Now run aws configure and copy past them in their corresponding parameter

```
aws configure
# AWS Access Key ID [****************FZQ2]:
# AWS Secret Access Key [****************RVKZ]:
# Default region name [eu-west-3]:
# Default output format [None]:
```

You can use the default region as the nearest one from you. In my case i used the eu-west-3 region (Paris). At this stage we can manage our AWS services remotely from our machine through the credentials stored on the file located in:

```
~/.aws/credentials
```



3.3 EKS control plane view

In the following figure we have a Kubernetes cluster created on AWS EKS service using eksctl command. There is a basics parameters needs to pass on eksctl command to create your cluster as which region want to deploy, how many replication in availability zone, the number of node and type of node. Once the EKS cluster is created, all the configuration between nodes and control plane as connectivity, default volume used, EC2 instance creation is automatically set up in the background. In addition the link to other AWS components is set with you cluster like the IAM service for Identity and Access management and Cloud Formation service used to manage the life cycle of AWS resources. We can configure other AWS services with the EKS like Load Balancer used to distribute incoming traffic across multiple targets, such EC2 instances, containers, IP addresses.

3.3.1 Create an Amazon EKS cluster (control plane)

Run the following eksctl in order to create a cluster using Elastic Kubernetes Service

```
 eksctl create cluster \
--name mobilitydb-cluster \
--version 1.20 \
--region eu-west-3 \
--nodegroup-name linux-nodes \
--node-type m5.large \
--ssh-access \
--nodes 3
```

In the region option you can use the nearest region from your location. In the node-type option you can define the type of the ressource for the created node. AWS provide a lot of ressource type. In my case i defined a m5.large type, which is 2 CPUs, 8G

of RAM, 10G of storage. You can find the entire list of node type. here You can customize your cluster creation using according to you needs, Run eksctl create cluster --help to see all the options. The creation process take about a 20 minutes of time. If you want to delete the cluster with all the ressources created just use:

```
eksctl delete cluster --name mobilitydb-cluster
```

View the cluster's ressources

```
  kubectl get nodes
# You should see your EC2 node as this:
# NAME                                           STATUS   ROLES    AGE     VERSION
# ip-192-168-47-163.eu-west-3.compute.internal   Ready    none   8m56s   v1.20.4- ↩
    eks-6b7464
# ip-192-168-9-100.eu-west-3.compute.internal    Ready    none   8m48s   v1.20.4- ↩
    eks-6b7464
# ip-192-168-95-188.eu-west-3.compute.internal   Ready    none   8m52s   v1.20.4- ↩
    eks-6b7464
```

You should see three nodes created in the terminal and the AWS interface for EC2 instances here

3.3.2 Deploy our scaleMobilityDB image using the kubectl

We have prepared a manifest yaml file that define the environment of our workload mobilityDB. It contain the basics information and configuration in order to configure our Kubernetes cluster. The deployment instance used to specify the scale-mobilitydb docker image and mount volume path. Finnaly the number of replications to our deployment in order to increase the availability. configMap instance defined the environement information (postgres user, password, database name). The most important instances is the PersistentVolume and PersistentVolumeClaim. The PersistentVolume parameter allows to define the class of storage, device and file system allow that store our mobilitydb data, it simply a workers nodes that store data. AWS provides different classes of storages, for more information see. this The PersistentVolumeClaim parameter defines the type of request, access to use in order to interogate our PersistentVolume. A PersistentVolumeClaim has an access type policy – ReadWriteOnce, ReadOnlyMany, or ReadWriteMany. It simply a pod that manage the accesses to storage. When you create a EKS cluster, by default the PersistentVolume is set to gp2 (General Purpose SSD driver). It's an Amazon EBS (Elastic Block Store) class. Use this command to see the default storage class.

```
kubectl get storageclass
# NAME            PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE       ↩
    ALLOWVOLUMEEXPANSION   AGE
# gp2 (default)   kubernetes.io/aws-ebs   Delete          WaitForFirstConsumer   false  ↩
                     15d
```

If you want to create your own storage class and set it as default, follow this guides Finnaly the service instance used to expose our MobilityDB workload. All thoses configuration can be updated according to your workload needs. Putting it all together in mobilitydb-workload.yaml file. Run this command to initialize all the instances.

```
kubectl apply -f mobilitydb-workload.yaml

# deployment.apps/scale-mobilitydb created
# persistentvolume/postgres-pv-volume unchanged
# persistentvolumeclaim/postgres-pv-claim unchanged
# configmap/postgres-config unchanged
# service/scale-mobilitydb created
```

Now you should see all instances running.

```
kubectl get all

# NAME                                  READY   STATUS    RESTARTS   AGE
# pod/scale-mobilitydb-7d745544dd-dkm7k   1/1     Running   0          43s
```

```
# NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
# service/kubernetes            ClusterIP   10.100.0.1      none        443/TCP          15d
# service/scale-mobilitydb      NodePort    10.100.38.140   none         5432:30200/TCP   69m

# NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
# deployment.apps/scale-mobilitydb        1/1     1            1           69m

# NAME                                               DESIRED   CURRENT   READY   AGE
# replicaset.apps/scale-mobilitydb-7d745544dd        1         1         1       69m
```

At this stage you can run your psql client to confirm that the scale-mobilitydb is deployed successfully. To run the psql, we need to know on which node the MobilityDB pod is running. the following command show details informations including the ip address that host the scale-mobilitydb.

```
  kubectl get pod -owide

# NAME                                  READY   STATUS    RESTARTS   AGE     IP  ↩
                NODE                                            NOMINATED NODE   READINESS  ↩
   GATES
# scale-mobilitydb-7d745544dd-dkm7k   1/1     Running   0          100s   192.168.45.32    ↩
   ip-192-168-60-10.eu-west-3.compute.internal   none           none
```

In my case, scale-mobilitydb have pod name as scale-mobilitydb-7d745544dd-dkm7k and is running in the node 192.168.45.32. As we have the host ip and the name of pod that run our scale MobilityDB environement instance, we can use this command to connect to postgres database, the password for postgres user is postgres. We can run our psql client within the pod scale-mobilitydb to confirm that citus and mobilitydb extension it's well created.

```
  kubectl exec -it  scale-mobilitydb-7d745544dd-dkm7k -- psql -h 192.168.45.32 -U postgres  ↩
      -p 5432 postgres

# Password for user postgres:
# psql (13.3 (Debian 13.3-1.pgdg100+1))
# SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression ↩
   : off)
# Type "help" for help.

# postgres=# \dx
#                                     List of installed extensions
#     Name    | Version |   Schema   |                              Description
#  ↩
    ------------+---------+------------+-------------------------------------------------#----------

# citus      | 10.1-1  | pg_catalog | Citus distributed database
#  mobilitydb | 1.0     | public     | Temporal datatypes and functions
# plpgsql    | 1.0     | pg_catalog | PL/pgSQL procedural language
# postgis    | 2.5.5   | public     | PostGIS geometry, geography, and raster spatial types ↩
     and functions
#(4 rows)

#postgres=#
```

Run MobilityDB queries In order to make the MobilityDb queries more powerfull, we have used the single node citus that create shards for distributed table, Auto scaling AWS service As we have a complex MobilityDB queries, we may use the Vertical Autoscaler and the Horizontal Autoscaler that AWS provides to optimize the cost according to the query needs. Vertical Pod scaling using the Autoscaler The vertical scaling that provide aws it's a mechanism allows us to adjust automatically the pods

ressources. This adjustment decrease the cluster cost and can free up cpu and memory to other pods that may need it. The vertical autoscaler analyze the pods demand in order to see if the CPU and memory requirements are appropriate. If adjustments are needed, the vpa-updater relaunches the pods with updated values. To deploy the vertical autoscaler, as following is the steps:

```
  git clone https://github.com/kubernetes/autoscaler.git
cd autoscaler/vertical-pod-autoscaler/
```

deploy the autoscaler pods to your cluster

```
  ./hack/vpa-up.sh
```

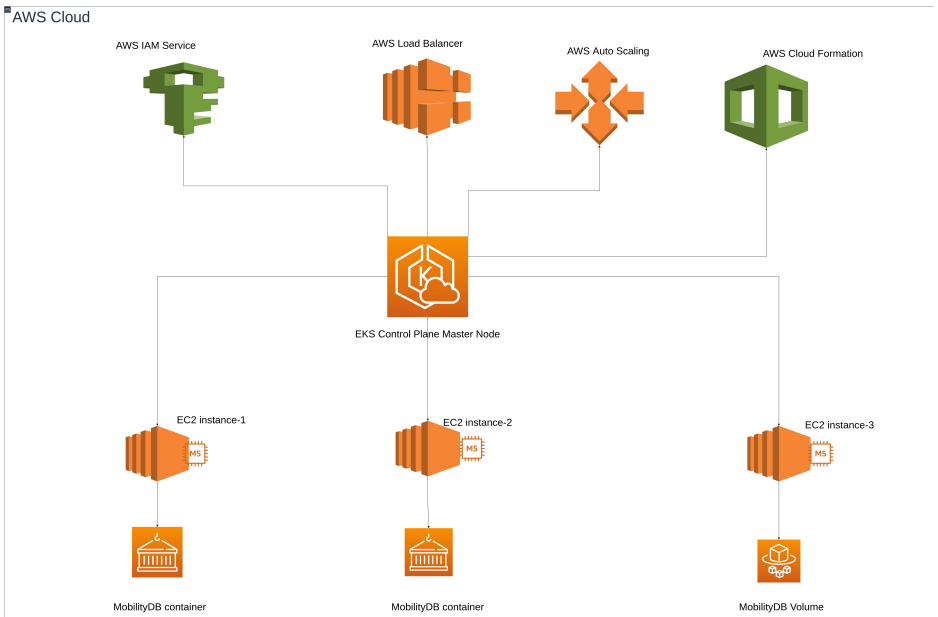
Check the vertical autoscaler pods

```
Kubectl get pods -n kube-system

# NAME                                        READY   STATUS    RESTARTS   AGE
# aws-node-rx54z                              1/1     Running   0          17d
# aws-node-ttf68                              1/1     Running   0          17d
# coredns-544bb4df6b-8ccvm                    1/1     Running   0          17d
# coredns-544bb4df6b-sbqhz                    1/1     Running   0          17d
# kube-proxy-krz8w                            1/1     Running   0          17d
# kube-proxy-lzm4g                            1/1     Running   0          17d
# metrics-server-9f459d97b-vtd6n              1/1     Running   0          3d11h
# vpa-admission-controller-6cd546c4f-g94vr    1/1     Running   0          38h
# vpa-recommender-6855ff754-f4blx             1/1     Running   0          38h
# vpa-updater-9fd7bfbd5-n9hpn                 1/1     Running   0          38h
```

Horizontal Pod scaling using the Autoscaler The horizontal Autoscaler that provides AWS allows to increase the number of pods within the cluster, it's a replication controller. This can help the application scale out to meet increased demand or scale in when resources are not needed, the Horizontal Pod Autoscaler makes application to meet the resources target. Before deploying the Horizontal autoscaler, we need the Kubernetes Metric server. The metric server is an API that collect the ressources statistics from the cluster and expose them for the use of the autoscaler. For more information about the metric server see. here Deploy the metric server

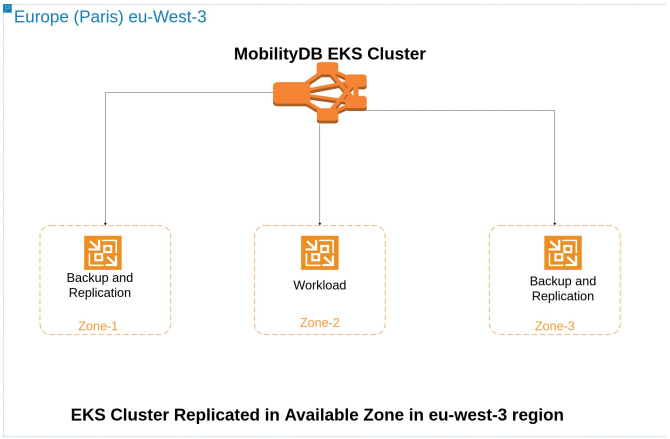
```
  kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/ ↩
      download/components.yaml
```



EKS AWS Cloud Cluster Linked with Other Services

3.4 EKS cluster replication view

In this figure we have our MobilityDB environment replicated in all available zones in the Europe (Paris) eu-West-3 region. We can also deploy our application in other regions as Europe (Frankfurt) eu-central-1 or Europe (Milan) eu-south-1



EKS Cluster Replicated in Available Zone in eu-west-3 region

Chapter 4

Deployment MobilityDB using Citus Cluster

In this part we defined a simple cluster of machine (EC2 instances) created by hand on AWS cloud as on premise version. So we have deployed the scale-mobilitydb in all nodes and we choose one from them as Master and then make the others as worker by joining them to the Master by the sql command citus_add_node('IP/host of new worker', 5432);

4.1 Deploy scale-mobilitydb as standalone

Before doing this step you need to connect within your aws EC2 machine known as master node. We assume that we have already create and configure one aws EC2 host master node and some aws EC2 host worker node. - You can run the image as standalone using docker run command, Execute this on all cluster's nodes.

```
  sudo ssh -i YourKeyPairGenerated.pem ubuntu@EC2_Public_IP_Address

docker run --name scaledb_standalone -p 5432:5432 -e POSTGRES_PASSWORD=postgres  ↩
    scalemobilitydb/scalemobilitydb
```

You can specify the mount volume option in order to fill the mobilityDB dataset from your host machine by adding -v /path/on/host_mobil
After running the scalemobilitydb instance, you can add and scale manually your database using the citus query.

```
select * from citus_add_node('new-node', port);
```
Check wether if the new-node is added correctely in the cluster.
```sql
select master_get_active_worker_nodes();
-  master_get_active_worker_nodes
-- --------------------------------
--  (new-node,5432)
-- (1 row)
```

Let create MobilityDB table and distribute it on column_dist in order to create shards by hashing the column_dist values. If no nodes added on the cluster than the distribution is seen as single node citus otherwise is multi nodes citus.

```
  CREATE TABLE mobilitydb_table(
column_dist integer,
T timestamp,
Latitude float,
Longitude float,
Geom geometry(Point, 4326)
);
```

```
SELECT create_distributed_table('mobilitydb_table', 'column_dist');
```

fill free to fill the table mobilitydb_table before or after the distribution. At this stage you can run MobilityDB queries on the citus cluster.

4.2 Deploy scalemobilitydb using citus manager

This deployment is similar to the last one except that there is a manager node. It simply listens for new containers tagged with the worker role, then adds them to the config file in a volume shared with the master node. In the same repository scale_mobilitydb run the command - Running the image as Citus cluster using this following

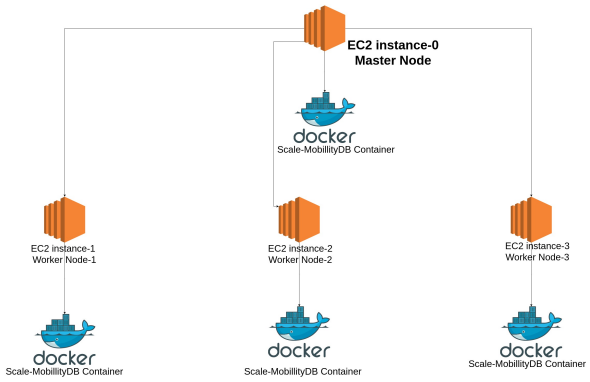
```
  docker-compose -p scale-mobilitydb up

# Creating network "citus_default" with the default driver
# Creating citus_worker_1
# Creating citus_master
# Creating citus_config
# Attaching to citus_worker_1, citus_master, citus_config
# worker_1    | The files belonging to this database system will be owned by user "postgres ↩
    ".
# worker_1    | This user must also own the server process.
# ...
```

You can run more workers in order to scale the citus cluster by running:

```
docker-compose -p scale-mobilitydb scale worker=5

# Creating and starting 2 ... done
# Creating and starting 3 ... done
# Creating and starting 4 ... done
# Creating and starting 5 ... done
```



Scaling MobilityDB using Citus daya cluster