Resource Management Schemes for Distributed Postgres database in cloud native environment

1st Md Awsaf Alam
Dept. of CSE (MSc.)
Bangladesh University of Engineering and Technology
Dhaka, Bangladesh
0421052071@grad.cse.buet.ac.bd

2nd Dr. Anindya Iqbal
Dept. of CSE (Professor)
Bangladesh University of Engineering and Technology
Dhaka, Bangladesh
anindya_iqbal@yahoo.com

***Abstract*—In this paper, we compare different resource management strategies for deploying a Postgresql database. We analyze which workload will be more suitable for**

***Index Terms*—postgresql, kubernetes, docker, aws, distributed database.**

I. Introduction

Managing cloud infrastructure effectively is a big pain point for most companies[1]. Early Infrastructure decisions can have a long term impact for scalability. As new cloud services and frameworks emerge, proper benchmarking comparisons become vital for making effective architecture decisions. In this paper we try to discuss the challenges faced during setting up a distributed Postgres database in a cloud environment, and benchmark different state-of-the-art methods to determine which type of deployment is best suited for each workload. This research work will help Engineers decide very early on what kind of infrastructure support they will need to properly scale a system in production for a given predicted workload.

II. Motivation

Modern computing systems have to be built with scalability in mind. If we abstract out all the details for the sake of simplicity, most software development happens in simple client-server architectures. The client sends a request to the server for a specific data, the server then queries the database for this data, and on some occasions further processes the data. This processed data is then sent to the client as a response for the http request. (Figure 1)

However, we know that in a real live system, there will be multiple clients will could simultaneously send requests to the server. Hence the server needs to be scalable to handle the load, and subsequently, the database also needs to be scaled, in order to handle the enormous load of queries coming from the server. In modern systems therefore, both the server and data base are hosted in the cloud. Cloud services provide the opportunity for developers to scale a system rapidly without having to focus on the hardware complexities. Therefore the hardware is completely abstracted out from the software development process.

In this study we deal with the database layer and identify the best possible methods for managing cloud resources for a distributed database on the cloud. We compare distributed versus non distributed approach and also compare the containerized versus non-containerized approach.

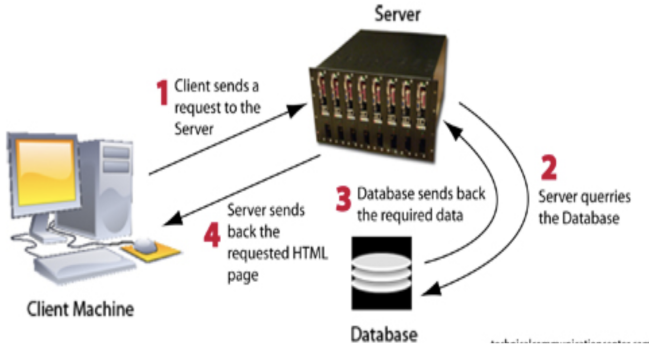

Fig. 1. Basic Client Server Architecture

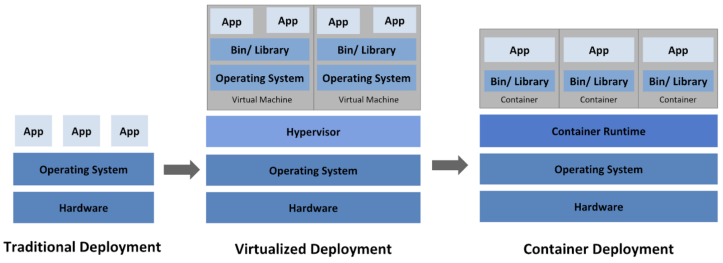

Fig. 2. Containerization

III. Related Work

A. CNSBench

we use the cnsbench tool to benchmark against a set of workloads. we mainly benchmark using two types of workloads. Control Workloads and I/O Workloads.

We may define **Control Workloads** as:

- Combination of actions and rates
- Actions execute operations, for instance create resource (e.g., create Pod or Volume), delete resource (e.g., delete

snapshot), snapshot volume, and scale resource (e.g., scale database deployment)
- Rates trigger associated actions at some interval

CNSBench treats control workloads as first class citizens, and we can run cnsbench inside our deployed kubernetes cluster and define the specification for the different control workloads.

CNSBench allows the usage of external I/O workloads so it's very easy to use existing workload datasets with CNSBench. Control operations impact the I/O workloads.

For example, volume creation often requires:

- Time-consuming file system formatting,
- Volume resizing, which may require data migration and updates to many metadata structures
- Volume reattachment

So, in summary, CNSBench contains Separate I/O workloads and control workloads, and uses existing tools to generate I/O workloads. We can Specify and create realistic control workloads with Easy to define and run benchmarks

B. Resource Management Schemes for Cloud Native Platforms with Computing Containers of Docker and Kubernetes

- Comparative analysis of resource management for Cloud Native Platforms
- Deep learning framework and big data processing

C. Benchmarking geospatial database on Kubernetes cluster

Compare PostgreSQL for clustered vs non-clustered environments.

- PostgreSql
- Clustered and non-clustered based system

IV. Methodology

Our main objective is to understand challenges in deploying a Distributed Database, and develop strategies on how to manage resources effectively in a cloud environment. For this, we will evaluate the trade-offs between docker & kubernetes for distributed PostgreSQL database deployment. During our research work, we compared 3 different types of approaches.

1) Non-clustered approach
2) Dockerized Deployment
3) Using a Kubernetes cluster

Initially we deploy our Postgres database in all 3 environments, and test out different scenarios. We want to find out what kind of workload would be best suitable for each deployment method.

For measurement, we use the existing Benchmarking tools to specify which workloads to use. Then we develop novel resource management strategy for Kubernetes, specifically for PostgreSQL for different workloads.

A. Setting up a cluster in AWS

We use AWS for running all our experiments as it is one of the most widely used cloud providers available, and also provides a free tier for students or research purposes.

In order to setup a cluster in AWS, initially we need to configure a **Virtual Private Cloud (VPC)**. Amazon Virtual Private Cloud (Amazon VPC) enables us to launch AWS resources into a virtual network based on our custom configurations. This virtual network closely resembles a traditional network that we'd normally operate in a data center, with the benefits of using the scalable infrastructure of AWS as well as reducing the hassle of needing our own physical hardware for running different experiments.

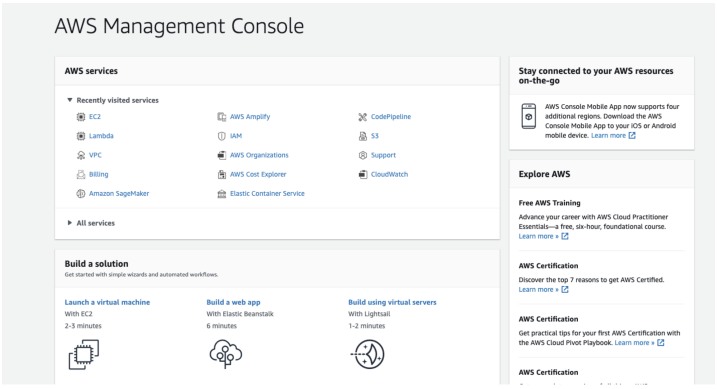

Fig. 3. AWS Management Console

We can create a VPC inside the AWS management console (Fig 3). Once the VPC is created (Fig: 4, we need to do the configurations in order to ensure that the cluster can be setup properly. Next we attach an **Internet Gateway (IGW)** to the VPC instance. An internet gateway is a horizontally scaled, redundant, and highly available VPC component that allows communication between the VPC and the internet. Without the IGW, we would not be able to communicate with the containers inside the VPC.

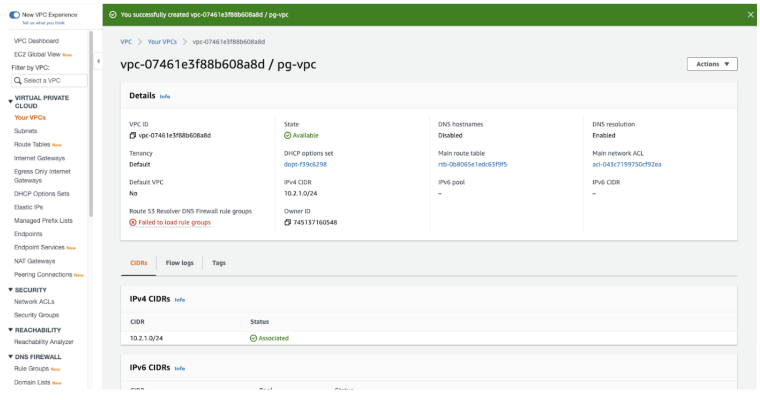

Fig. 4. AWS VPC

Finally, we configure two subnets (Fig: 5) inside the VPC within the same CIDR block as the VPC and launch instances

inside the subnets. We also setup the route tables and make sure all the security groups are properly added so that the different nodes within the cluster do not have any difficulty while communicating. Therefore, we can summarize the steps for setting up a full cluster from scratch as specified below:

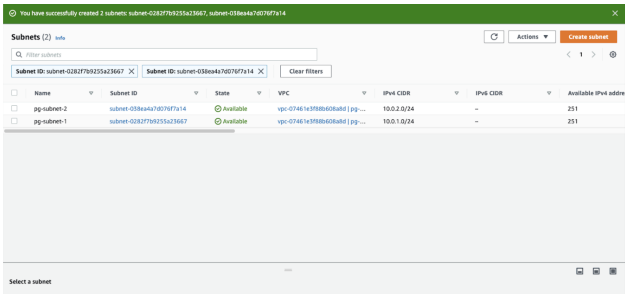

Fig. 5. AWS Subnet

1) Create a VPC
2) Create an IGW
3) Attach IGW to VPC
4) Create a subnet inside a VPC
5) Launch VM inside subnet (Master and worker nodes)
6) Traffic is routed from the igw through subnet into the VM.

For setting up the VMs, we use a *t2-medium* instance as **master** node, and the rrest of the worker nodes of the cluster can be *t2-micro* nodes. Here are the steps we followed for launching a Vm inside our pre-configured VPC instance:

1) Select the type of instance we want
2) Select the Operating System
3) Select the VPC, Subnet, Storage etc other details
4) Specify the Security Group
5) Add a tag to the VM for consistency
6) Finally, add a new key pair for ssh access.

Once the instances are created, we can start installing kubernetes in the virtual machines in order to configure them.

B. Installing Kubernetes and network configurations

Steps for initializing kubernetes in the VM

1) Install Docker: sudo apt update sudo apt install docker.io -y
2) Enable Docker Service sudo systemctl start docker sudo systemctl enable docker
3) Installing dependencies for https and cURL sudo apt install apt-transport-https curl

4. Add Ubuntu Repository to download Kubernetes services $curl - shttps : //packages.cloud.google.com/apt/doc/apt - key.gpg|sudoapt - keyadd$ sudo apt-add-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"

5. Install Kubernetes modules $sudoaptinstallkubeadmkubeletkubectlkubernetes - cni$

6. Disable Swap Memory $sudoswapoff - a$

V. Evaluation

We consider 3 different types of approaches.

1) Installing Posgres on baremetal machine
2) Intalling Postgres using docker container
3) Installing Postgres in a distributed Kubernetes cluster.

For each experiment, we run a script where the operations were carried out in the following order:

- First initialize the database and create a table.
- Populate the table with 10,000 write requests simulatneouly and calculate the latency
- Do this for all the deployment environments.
- Read the data at random intervals, with random number of simultaneous read requests.

For each of the experiments we observed the completion time. The same experiment was carried out multiple times and the average was taken. From here, we can deduce that for Bare metal deployment, the latency increases exponentially a the number of simultaneous requests increases, while in case of a clustered deployment, the increase is linear.

VI. Future Work

The amount of different variations of experiments that can be carried out is almost unlimited. So it is very important to know first which workloads are actually important to the developers. for that we need to carry out an empirical study.

CNSBench does not offer compatibility with all the other postgres operators available on the market. so another work would be Customizing benchmarking tool for applying on other operators to get quantitative data and a more clear picture of the current scenario of postgres kubernetes operators.

Empirical study to understand which factors developers are actually considering before making decisions is also very important. most of the architectural decisions made have a lot of long-term implications. So it is very important that we take those into account and run the experiments accordingly.

References

[1] https://www.packtpub.com/product/hands-on-serverless-computing/9781788836654
[2] MapReduce: Simplified Data Processing on Large Clusters
[3] https://www.cloudflare.com/learning/serverless/what-is-serverless/
[4] Jiffy: A virtual memory abstraction for server- less architectures
[5] https://github.com/resource-disaggregation/jiffy
[6] MapReduce Basics: https://www.youtube.com/watch?v=cvhKoniK5Uo
[7] NIMBLE Task Scheduling Slides: https://www.usenix.org/system/files/nsdi21$_slides_zhang - hong.pdf$