Distributed Apache Cassandra Deployment on Kubernetes with Autoscaling and Monitoring

Bhrugu Kothari
Concordia University
40270224

Jay Patel
Concordia University
40293645

Devanshu Kotadiya
Concordia University
40268999

Mayank Parmar
Concordia University
40269385

Keyur Patel
Concordia University
40154883

ABSTRACT

This work also presents an example of how **Apache Cassandra** is deployed in a **Kubernetes** environment as a way of gaining basic notions regarding distributed systems, such as **scalability**, **fault tolerance**, and **monitoring**. Real-world data processing was simulated using an input data size of **2.4 GB**. The resource usage always adapts itself to the loads because of Horizontal Pod Autoscaler (HPA) [4] and functions optimally. Initially, **Prometheus** incorporated with **Grafana** [4] was used to monitor system health status and throughput in real-time. The project demonstrates how **Kubernetes** and **Cassandra**, along with open-source monitoring tools, can effectively manage large and elastic data in today's decentralized environments [2].

PVLDB Reference Format:
Bhrugu Kothari, Jay Patel, Devanshu Kotadiya, Mayank Parmar, and Keyur Patel. Distributed Apache Cassandra Deployment on Kubernetes with Autoscaling and Monitoring. PVLDB, 14(1): XXX-XXX,2020.DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1. INTRODUCTION

Background:

Due to the explosive amounts of data, distributed systems become more and more important in order to deal with big datasets. High availability, scalability, and fault tolerance are provided by distributed systems that make them a requirement in today's environment [1]. Some examples are the financial management, social networks, and online systems of purchases and sales, which work with considerable volumes of data in real time. In the category of the distributed databases, Apache Cassandra can be considered as an impressive, horizontally scalable, NoSQL database capable of working with immense volumes of data across many nodes, with low latency [2].

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Kubernetes another platform for organizing containers at scale enhances the capability of distributed systems by overseeing containerized applications. These features make Kubernetes ideal for use in the deployment of databases such as Cassandra since Kubernetes provides dynamic scaling and automated resource management, fault tolerance [1][2]. Such a combination of Kubernetes and Cassandra exemplified, what can be realized with distributed systems in practice when dealing with real-world data.

Motivation:

As the volume of data grows and Processing today occurs in a technological world where data can easily be in terabytes or petabytes and still be growing, the systems need to be capable of handling this large data volume while at the same time providing real time results with next to zero delay. Current architecture involve dynamic aspects that provide for suitable patterns of utilizing the available resources for increased utilization and mechanisms for getting back online in case of node failure. This project meets current data management issues by using Apache Cassandra for scaling and handling failures, Kubernetes for the cluster's organization; and Prometheus/Grafana for real- time monitoring [1][2][4].

Key Technologies:

The following key technologies were used in the project:

- **Apache Cassandra:** A distributed NoSQL database characterized by high scalability, fault tolerance and permanent availability.
- **Kubernetes:** It is a container orchestration plane used for a client application, used for packaging and controlling the deployment of multiple applications.
- **Prometheus:** A system for monitoring applications and nodes that gathers and archives current performance data.
- **Horizontal Pod Autoscaler (HPA):** A Kubernetes feature that automatically adjusts pods dependency on resources to meet the emerging workload requirements.

Project Dataset:

To test the system's scalability and fault tolerance, a 2.4 GB dataset was used, which simulates real-world workloads. The dataset, sourced from the THS Stock Competition, contains stock price and volume data. This dataset serves as a realistic example for large-scale data ingestion and querying, allowing the project to evaluate Apache Cassandra's performance under significant workloads [4]. Using Python scripts, the dataset is ingested into Cassandra, ensuring efficient data insertion and storage for subsequent querying [2].

2. The System

2.1 System Architecture

This way the system is aimed to deploy Apache Cassandra on a Kubernetes cluster both teaching and demonstrating important aspects of a distributed system such as fault tolerance, scalability, and monitoring. Apache Cassandra explained below is a high scalable and fault-tolerant NoSQL database With the help of Kubernetes StatefulSets for Cassandra, Cassandra will provide a stable network identity and provide persistent disk storage for each node [2] [4]. Orchestration, scalable computing, and failure resiliency are achieved through the Kubernetes cluster as controlled by Minikube for the Cassandra pods. To achieve better utilization of resources at different workloads, the Horizontal Pod Autoscaler (HPA) scales the number of Cassandra pods with respect to resource usage which includes CPU. Monitoring layers tackle an information gathering stack with Prometheus to compile vital system stats and Grafana that displays lively performance and system health via dynamic panels. Last but not least, a 2.4 GB of stock data set is loaded into Cassandra using a Python script to perform an actual workload and verify the system performance and extensibility.

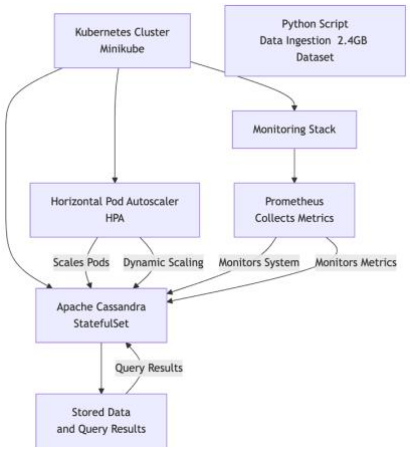
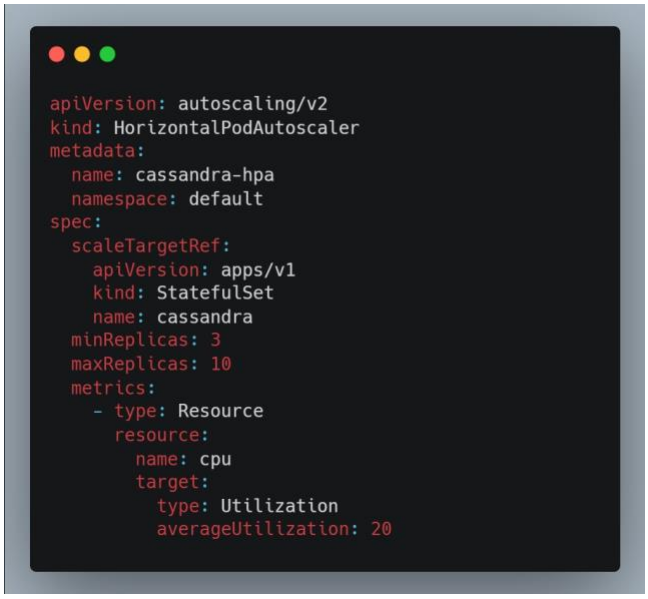


Figure 2.1 Flowchart of system architecture

2.2 Cassandra Deployment

Apache Cassandra runs on the Kubernetes cluster as Stateful Set to guarantee High Availability and Scalability with Fault Tolerance. Cassandra is an example of a stateful application and the StatefulSets are perfect for management because they provide every pod a unique identity as well as persistent storage.

Configuration of Cassandra StatefulSet:

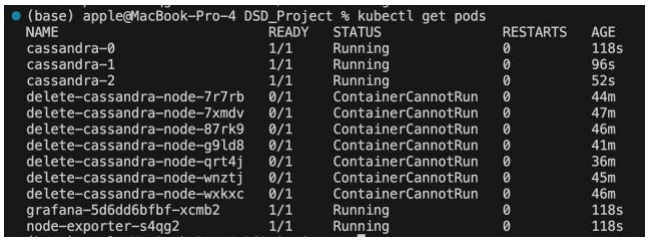


```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: cassandra-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: StatefulSet
    name: cassandra
  minReplicas: 3
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 20
```

Figure 2.2 Configuration of Cassandra statefulset

Deployment Steps:

1. **Save the configuration**
   Save the above YAML configuration as `final.yaml`.
2. Apply the configuration using the following command:
   `kubectl apply -f final.yaml`
3. Verify the pods are running:
   `kubectl get pods`



```
(base) apple@MacBook-Pro-4 DSD_Project % kubectl get pods
NAME                                READY   STATUS                RESTARTS   AGE
cassandra-0                         1/1     Running               0          118s
cassandra-1                         1/1     Running               0          96s
cassandra-2                         1/1     Running               0          52s
delete-cassandra-node-7r7rb         0/1     ContainerCannotRun    0          44m
delete-cassandra-node-7xmdv         0/1     ContainerCannotRun    0          47m
delete-cassandra-node-87rk9         0/1     ContainerCannotRun    0          46m
delete-cassandra-node-g9ld8         0/1     ContainerCannotRun    0          41m
delete-cassandra-node-qrt4j         0/1     ContainerCannotRun    0          36m
delete-cassandra-node-wnztj         0/1     ContainerCannotRun    0          45m
delete-cassandra-node-wxkxc         0/1     ContainerCannotRun    0          46m
grafana-5d6dd6bfbf-xcmb2            1/1     Running               0          118s
node-exporter-s4qg2                 1/1     Running               0          118s
```

Figure 2.2.1 Verifying pods status

2.3 Load Balancing

Node affinity ensures that workloads of a client are distributed evenly across the Cassandra pods thus avoiding server overload. The Kubernetes services regulate traffic allowing the client requests to be directed to suitable pod while partitioning at Cassandra level is done by a token ring which splits the data evenly between the nodes. It provides for the best use of resources, compensation for latency, and maximum availability when the program enlarges in quantity of work it needs to carry out.

3. Performance Evaluation and Analysis

3.1 Scalability

Scalability is a critical attribute of distributed systems, referring to the system's ability to handle increasing workloads by dynamically

adapting its resources without compromising performance. In this project, **Apache Cassandra** and **Kubernetes** work in tandem to achieve horizontal scalability, ensuring the system can meet varying resource demands effectively [1][2].

Horizontal Scalability with Kubernetes HPA
Horizontal scalability is achieved through **Kubernetes Horizontal Pod Autoscaler (HPA)**. HPA monitors resource usage (e.g., CPU utilization) and adjusts the number of Cassandra pods dynamically based on the workload [3].

The Horizontal Pod Autoscaler (HPA) only scales by monitoring the pod's CPU utilization of the Cassandra pods by consuming metrics from the Kubernetes metrics server. Another advantage of HPA is that when the utilization of the CPU reaches a certain percentage, for instance, 50% then HPA intervenes by replicating more Cassandra pods to cater for high demand. When the workload reduces or nominally rises, HPA shrinks the pods to save the resources required in optimum condition for the best performance [3][4].

Key Advantages of Horizontal Scaling:

Horizontal Scaling comes with several advantages when it comes to resources and performance. It Adopt resources are optimized by applying the critical resources needed, hence minimizing cost and avoiding over purchasing. Its benefits include sustaining the performance as and when the workload is scaled up where new pods are added to accommodate query rate and ingestion. Also, horizontal scaling ensures no downtime as the process of scaling doesn't affect the DB operations but it happens concurrently [1][3].

3.2 Fault Tolerance

The next test was to check the fault tolerance of the Cassandra nodes with one of the Cassandra pods manually deleted. System response in failure conditions is properly controlled, which proves the effectiveness of the developed system. The meaningless shows that Kubernetes quickly acknowledged the pod as failed and immediately recreated it in 30-40 second [2]. At the same time, Cassandra's data replication mechanism ensured that no data was ever lost whilst replicating data. The system is clearly visible and running all through the pod recovery and this is evidence enough that the pod can actually sustain pod recovery without necessarily requiring any external human intervention. This will demonstrate the effectiveness of having a high availability of the system due to any given node failure.

3.3 Monitoring and Visualization

Prometheus is used for real time performance measurement to get metrics of Cassandra pods and from Kubernetes node. Prometheus enhances observability of the distributed system by offering valuable information about specific resources for example, CPU usage, memory consumption and performance of the system [4].

These metrics are collected and written to disk by Prometheus, and can be queried to provide insight into the overall health of the system as well as the performance when under different loads [3].

Below, you can see the Prometheus code for the Cassandra cluster:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prometheus
  labels:
    app: prometheus
spec:
  replicas: 1
  selector:
    matchLabels:
      app: prometheus
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      containers:
        - name: prometheus
          image: prom/prometheus:v2.46.0
          ports:
            - containerPort: 9090
          volumeMounts:
            - name: prometheus-config
              mountPath: /etc/prometheus
      volumes:
        - name: prometheus-config
          configMap:
            name: prometheus-config
```

Figure 3.3 Prometheus code for the Cassandra cluster

3.4 Data Ingestion Performance

To assess the performance of the designed system when dealing with large volumes and real comparable workloads, the Cassandra cluster was loaded with a 2.4 GB payload. The last phase of ingestion was performed successfully, which proved that Cassandra is capable of handling large volumes of data. In the course of ingest, query performance was flat in comparison to the dynamic resources the system can bring in as needed. These outcomes confirm the capability of Cassandra in handling the vast amount of data with fast query response time and operational stability for which it was designed for [4].

```
prometheus-65d5b5d846-t6b94   1/1     Running            0          23m
(base) apple@MacBook-Pro-4 DSD_Project % kubectl top pods
NAME                          CPU(cores)   MEMORY(bytes)
cassandra-0                   383m         1491Mi
cassandra-1                   352m         1638Mi
cassandra-2                   390m         1655Mi
cassandra-3                   400m         1497Mi
grafana-5d6dd6bfbf-mwfth      1m           39Mi
node-exporter-wxkxj           0m           3Mi
prometheus-65d5b5d846-t6b94   1m           29Mi
(base) apple@MacBook-Pro-4 DSD_Project %
```

Figure 3.4 CPU Performance

4. Demo Scenarios

4.1 Data Querying and Analytics

The first scenario shows how the used system can exhibit the ingested information and process it for analysis in a short span of time. Finally after loading the 2.4 GB data set into the Cassandra database, some basic queries have been performed to fetch stock price trends, high volume trades and some queries with date filters. These queries mimic common analytics activities which are present in real-world scenarios as are financial analytics. The operations ran freely and at a steady pace, which proved that Cassandra has low latency, and the results demonstrated that it performs analytical workloads at a very high level with no considerable slow down [4].

4.2 Data Consistency and Availability

The second scenario focuses on evaluating the system's data consistency and availability during operations. Apache Cassandra ensures high availability and fault tolerance through its **data**

replication mechanism, where data is replicated across multiple nodes. To validate this, queries were executed on different Cassandra pods to verify the consistency of results under normal operating conditions.

During the test, data written to one pod was quickly replicated to other nodes, ensuring that the same data could be queried from multiple pods without discrepancies. Even when one of the pods was temporarily unavailable, the system remained operational, and data requests were seamlessly handled by the remaining pods. This highlights Cassandra's ability to maintain continuous availability and provide reliable query results through its distributed architecture [4].

The results confirmed that Cassandra's replication strategy guarantees both data consistency and system availability, making it suitable for applications requiring reliable access to large-scale datasets [4].

4.3 Simulated System Recovery

In this instance, a fault was introduced by powering off a node running the Kubernetes in order to assess the system's recuperation capabilities with Cassandra pod. Cassandra pod kill test was done by forcibly terminating a running Cassandra pod to see the response from the Kubernetes self-healing process. Kubernetes acted quickly and noticed that one pod has failed and started a new pod within 30 to 40 seconds. Retention process of Cassandra kept no data loss and the database was fully functioning during the recovery process. This scenario supported the fault tolerance and very high reliability of the system as it stood under unforeseen failure conditions ensuring that on-going work was not interrupted or impeded and that data was not corrupted [4].

5. References

[1] P. V. Krishna and M. S. S. R. P. Rao, *Kubernetes Architecture, Best Practices, and Patterns*. IEEE Communications Standards Magazine, 2022. DOI: 10.1109/MCOMSTD.2022.9930690.

[2] Kubernetes Documentation, *Kubernetes Cassandra Stateful Application*, 2022. Available: https://kubernetes.io/docs/tutorials/stateful-application/cassandra/.

[3] Kubernetes Documentation, *Horizontal Pod Autoscaler*, 2022. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[4] Portworx, *Cassandra Kubernetes Experts Guide*, 2020. Available: https://portworx.com/wp-content/uploads/2020/08/cassandra-kubernetes-experts-guide.pdf.