Reproducible Experiments on SkyhookDM Ceph using Popper

Jayjeet Chakraborty, Carlos Maltzahn, Ivo Jimenez, Jeff LeFevre
UC Santa Cruz

I. Introduction

For someone getting started in experimenting with Ceph, it can be a bit overwhelming as there are a lot of steps that one need to execute and get right before they run some actual experiments and get results. In a high-level, the steps that are generally included in a Ceph experimentation pipeline is depicted below.

- Booting up VMs or Bare metal nodes on Cloud providers like AWS, GCP, CloudLab, etc.
- Deploying Kubernetes and baselining the cluster.
- Compiling and deploying Ceph.
- Baselining the Ceph deployment.
- Running experiments and use-case specific benchmarks.
- Writing Jupyter notebooks and plotting graphs.

If done manually, these steps can require typing 100s of commands interactively which can be cumbersome and error-prone. Since Popper is already good at automating experimentation workflows, we felt that automating this complex scenario can be a good use case for Popper and would potentially lower the entry barrier for new Ceph researchers. For our case, we also built workflows to benchmark SkyhookDM Ceph, which is a customization of Ceph to execute queries on tabular datasets stored as objects on a cluster by running queries on large datasets of the order of several hundred GBs.

II. Background

A. Ceph

Ceph is an open-source software storage platform, implements object storage on a single distributed computer cluster, and provides 3 in 1 interfaces for: object, block and file-level storage. Ceph allows decoupling data from physical hardware storage, using software abstraction layers, providing scaling and fault management capabilities. This makes Ceph ideal for cloud, Openstack, Kubernetes and other microservice and container-based workloads as it can effectively address large data volume storage needs.

B. SkyhookDM

SkyhookDM is a cloud storage system that leverages "programmable storage" capabilities to enhance data management directly within the storage layer of a distributed object storage system such as Ceph. The goal of Skyhook is to allow users to transparently grow and shrink their data storage and processing needs as demands change. Skyhook utilizes and extends Ceph distributed object storage with customized C++ "object classes" that enable database operations such as SELECT, PROJECT, AGGREGATE to be offloaded (i.e., pushed down) directly into the object storage layer. We are developing custom user-defined functions (UDFs) to enable domain-specific processing as well. SkyhookDM also enables data management tasks to be executed directly within storage such as local indexing and data redistribution or reformatting (row/col) to support dynamic data management in the cloud. These tasks operate directly on objects at the single object or cross-object level.

C. Kubernetes

Kubernetes is a production-grade open-source container orchestration system written in Golang that automates many of the manual processes involved in deploying, scaling, and managing of containerized applications across a cluster of hosts. A cluster can span hosts across public, private, or hybrid clouds. This makes Kubernetes an ideal platform for hosting cloud-native applications. Kubernetes supports a wide range of container runtimes including Docker, Rkt, and Podman. It was originally developed and designed by engineers at Google and it is hosted and maintained by the CNCF (Cloud Native Computing Foundation). Many cloud providers like GCP, AWS, and Azure provide a completely managed and secure hosted Kubernetes platform.

III. Introduction to Popper

In general, researchers and developers often endup typing a long list of commands in their terminal to build, deploy, and experiment with complex software systems. The process is highly manual and needs a lot of expertise or can lead to frustration because of missing dependencies and errors. The problem of dependency management can be addressed by moving the entire software development life cycle inside software containers. This is known as container-native development. In practice, when developers work following the container-native paradigm, they end up interactively executing multiple docker pull|build|run commands in order to build containers, compile code, test applications, deploy software, etc. Keeping track of which docker commands were executed, in which order, and which flags were passed to each, can

quickly become unmanageable, difficult to document (think of outdated README instructions), and error-prone. While this sounds simple at first, it has significant implications: results in time-savings, improve communication and in general unifies development, testing, and deployment workflows. As a developer or user of "Popperized" container-native projects, users need to learn one tool and leave the execution details to Popper, whether is to build and tests applications locally, on a remote CI server, or a Kubernetes cluster.

IV. Kubernetes Cluster Setup

Kubernetes clusters from any cloud provider like Google Kubernetes Engine, Elastic Kubernetes Service, etc. can be used. Popper workflows for spawning nodes in CloudLab, the NSF sponsored experimentation testbed and setting up Kubernetes clusters on them were implemented. The workflows leverage Geni-Lib to programmatically allocate nodes in CloudLab and use setup production ready Kubernetes clusters using Kubespray. Kubernetes clusters should have monitoring infrastructure setup to monitor several system parameters in real time and recoed them while running experiments. Popper workflows are also present for the acheiving the same using standard tools like Prometheus and Grafana.

V. Kubernetes Cluster Baselines

The Kubernetes cluster was baselined to measure the Disk and Network bandwidth of the underlying nodes. This step is necessary to have a baseline with which the results of the Ceph benchmarks can be compared. Kubestone, which is Kubernetes benchmarking operator was used for this purpose. Kubestone provides operators for running blockdevice benchmarks with Fio and Network benchmarks using Iperf. We launched client pods in different nodes and benchmarked the SEQ RW and RAND RW of the blockdevices. The fio workflow also performs sweeps on
parameters like IO depth, job count, blocksize to help analyze how the performance variability with different parameters which can be mapped to and compared with actual workloads while running Ceph benchmarks. THE SEQ READ bandwidth of the blockdevices were on avg. 410 MB/s while keeping the CPU busy with 8 fio jobs and an IO depth 32.

The network bandwidth between pods present in distinct were measured using Iperf and was found to be around 8 - 8.5 Gb/s between different set of nodes. The observed bandwidth was lower then the theoritical bandwidth of 10 Gb/s, due to the underlying Kubernetes networking stack Calico.

VI. RADOS Benchmarks

Ceph was deployed through Rook, which is a cloud-native storage orchestrator Kubernetes to make storage systems self-healing, self-managing and self-scaling. Rook deploys all the Ceph daemons as Pods. Our Ceph deployment comprised of 3 MONs and 4 OSDs, where each OSD was on a distinct node using a single blockdevice. We benchmarked RADOS, which is the Object store interface of Ceph to measure the througput of object READs and WRITEs and find out the overhead it incurs on top of raw blockdevice. We used the `rados bench` utility provided by Ceph and ran benchmarks with object sizes of 10MB using all the cores of the client. We began with a single OSD setup and gradually scaled it up to 5 OSDs to capture how the througput improves with more OSDs and also the peak throughput. As shown in Figure [], we observed significant throughput increases as we scaled from 1 to 4 OSDs. But, as we scaled to 5 OSDs the throughput increase was insignificant and was stagnant at around 1000 MB/s. Since the network capacity was around 8–8.5 Gb/s, it was derived that the bottleneck was shifted from the storage to network. The CPU usage, memory pressure and network traffic were also recorded. Only the network between the client and the OSDs were saturated.

VII. Case Study: SkyhookDM Benchmarks on UChicago SSL Cluster

We benchmarked the SkyhookDM cluster to measure the overhead that its processing libraries incur while querying tabular data. The vanilla Ceph cluster was converted to a SkyhookDM cluster with the tabular libraries loaded by running the `setup-skyhook-ceph` step in the `rook.yml` workflow. Upon running this step, the Ceph image is replaced with the SkyhookDM image and all the daemons are updated eventually. 10000 objects each of size 10MB containing tabular data in Flatbuffer format were loaded into a pool with replication disabled. The resulting dataset size was 210GB comprising of 750 Million rows. Queries were executed on the dataset to select 1%, 10% and 100% of the data with the query execution once on the client side and once on the storage. As shown in Figure [], The total query execution time for processing 1% and 10% of the data on the storage side was significantly less then that of client side. This was quite expected as only a subset of the rows needed to be transferred through the network in case of storage side processing. For the 100% query scenario, since the entire dataset needed to be transferred through the network for both client and storage side processing, both the durations were almost equal.

Since the total dataset size was 210 GB and the time spent in fetching and querying the entire dataset was approx. 210 s, it can be seen that the throughput was approx. 1000 MB/s (8 Gb/s). So, the network was again the bottleneck as expected. It can be seen from Figure x and Figure y, that the overhead of querying the tabular data was found to be negligible as compared to the data transfer latency since both vanilla Ceph and SkyhookDM had nearly the same througput.

VIII. References