SOLUTION DOCUMENT

MayaData

Benchmarking 3 million writes with Cassandra on K8s and Kubera Propel

Produced in partnership with

CONTENTS

Introduction 01
- What is Kubera Propel? 03
  - Mayastor NVMe-TCP performance 04
  - Background information on reactor loops 06
  - NVMe-oF TCP performance measurements 08

Solution: MayaData and Intel Together 09
- Hardware Configuration 09
- System configuration 09
  - System tuneables 10
- Baseline device performance 10
- Deploying Kubera Propel 12
  - 2 replicas (mirror) 17
- Running DataStax Enterprise on Kubera Propel 19
  - Getting Started 19
  - Install DataStax Enterprise 20
  - Accessing Cassandra Database 22
  - Now, let's create a Keyspace and add a table with some entries into it 24
    - Creating a Keyspace 24
    - Creating Data Objects 24
    - Inserting and Querying Data 24
- DataStax stress benchmark 25
  - DataStax benchmark results 26

Conclusion 27
- Appendix A 27
- Known issues and future work 27
- Auxiliary buffers 27
- The incoming atomic queue is polled too often 27
- Some threads are polled for no reason 27
- MSP is not cleared as expected 27
- Refcounting could/should be optimized 27
- The first core polls gRPC to often 27

MayaData

INTRODUCTION

Kubera from MayaData is SaaS and on-premise software for the use of Kubernetes as a fast data layer. Capabilities include alerting, visualization, reporting, chaos engineering, per workload back-ups, rolling upgrades, compliance checks, troubleshooting, provisioning and management of underlying storage media and cloud volumes, and more. Kubera Propel is a module of Kubera focused on the use of Kubernetes for high-performance workloads.

The OpenEBS project and MayaData popularized the Container Attached Storage pattern. Much more about this pattern is shared in various blogs on the Cloud Native Computing Foundation site, such as Container Attached Storage is Cloud Native Storage (CAS)

In this solution guide, we present a total solution based upon Kubera Propel and Intel's Optane storage as well as Kubernetes v1.16.0. We demonstrate a configuration that has been shown to deliver in excess of 1 million I/O operations, or IOPS, to a single container. Additionally, we demonstrate the deployment and high-performance operation of a DataStax Enterprise workload, a scale-out NoSQL database built on Apache Cassandra.

Today, OpenEBS and Kubera are used by well-known enterprises as well as thousands of community users world-wide. Popular use cases include:

- Containerized Data Management
- CI/CD
- ML & Data Ops
- Database as Service
- Content Management

OpenEBS is amongst the most broadly used cloud native - or Container Attached Storage - projects per various CNCF and analyst surveys. OpenEBS is the most widely deployed open source example of a category of storage solutions sometimes called Container Attached Storage. OpenEBS is created by MayaData as an open source project and donated to CNCF in early 2019. The open source OpenEBS CNCF Adopters list includes Arista, CNCF, Comcast, KPN, Orange, and others.

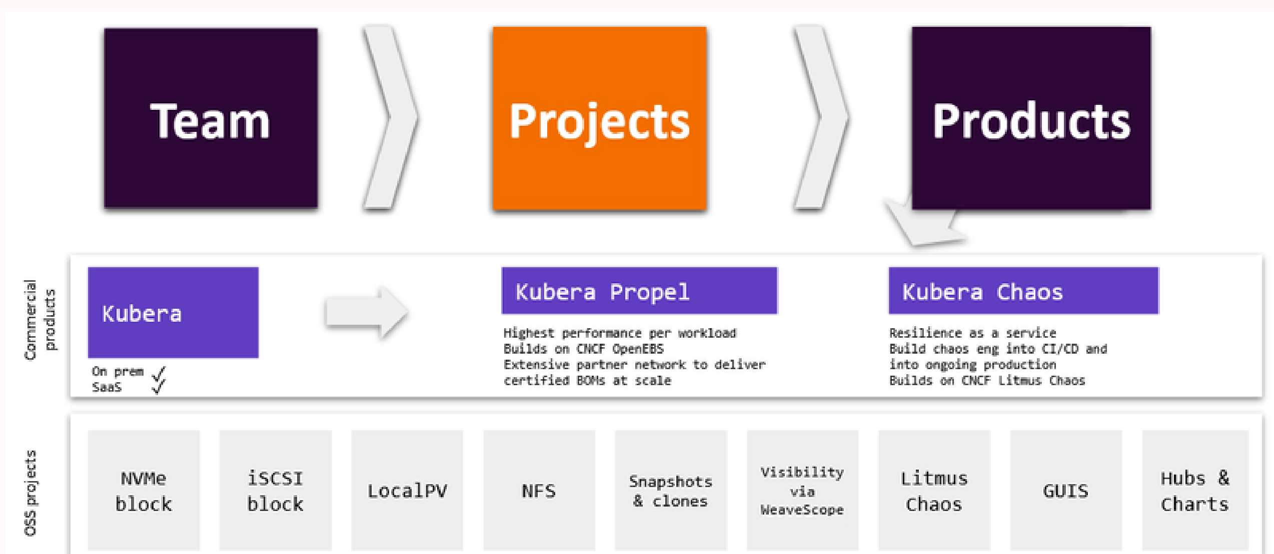
LitmusChaos and the related Litmus Hub enable end to end chaos engineering in Kubernetes environments. LitmusChaos was created by MayaData and donated to CNCF in mid of 2020.

What is Kubera Propel?

Kubera Propel is part of the Kubera product family. Kubera Propel is available as a freely downloadable software and as open source software and works on on-premise and public cloud deployments. Kubera capabilities include:

- Simplified configuration, management, and monitoring of a fast cloud native data plane for stateful workloads on Kubernetes, including Kafka, PostgreSQL, Cassandra, and other workloads.
- Automated lifecycle management of data layer components, including the OpenEBS Mayastor storage engine and underlying storage such as disks and cloud volumes.
- Team and Enterprise plans include support services from MayaData for the entire environment, including the Kubera Enterprise Edition and related components.

The foundation of Kubera Propel, OpenEBS Mayastor, is a breakthrough in per container, open source performance - and approaches theoretical maximum performance measured by various parameters. While OpenEBS Mayastor can pool underlying storage from multiple sources, it is best known for leveraging NVMe-TCP.

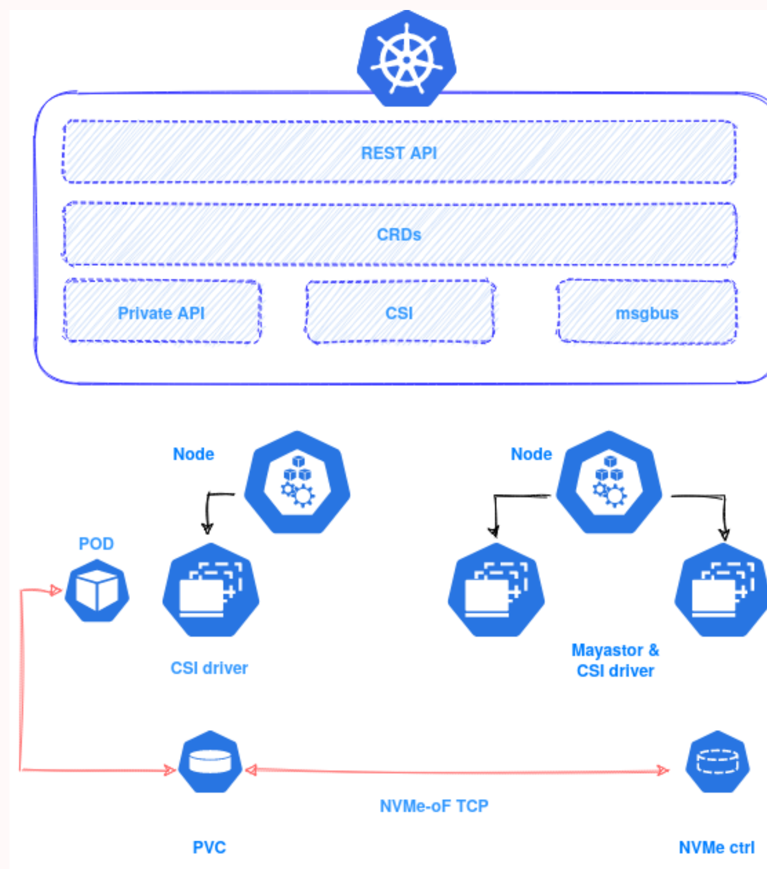


As explained in the Mayastor section of the OpenEBS project, the purpose of Kubera Propel is to deliver a simple to use, easy to provision, and high-performance Container Attached Storage solution.

Mayastor NVMe-TCP performance

As explained above, the design goal of Mayastor and Kubera Propel is to achieve <10% performance overhead versus underlying storage. Achieving this goal will be exceptionally differentiating versus traditional shared storage and the other attempts at Container Attached Storage.
Inevitably, Mayastor, even though written in Rust and leveraging the highly efficient SPDK, will impose somewhat of runtime overhead. The question is, how much? Or put differently, how much performance are we willing to sacrifice in return for safer abstractions, data replication, simplified management, and so on? Our goal is to keep the performance difference between local and remote below 10% on an average for a single replica . When using data protection schemes, i.e., one or more synchronous copies,[1] the added latency will depend on network performance.

Currently, the high-level architecture of Mayastor can be depicted thus:



[1]This is dramatically less overhead than that encountered in the most popular shared storage systems

The control plane consists of several components. Most notable are the northbound API and the CRDs the control plane creates to record and share execution state. The control plane handles the creation of PVCs and communicates with Mayastor instances. It is not required that Mayastor run on all nodes; however, it is necessary to run the CSI driver on all nodes intended to consume or mount Mayastor provisioned storage.

The data path (shown in red) is established by the CSI drivers and is what will be benchmarked here. The Mayastor daemonset consumes local storage resources on the node. The consumption is based on storage URIs wherein different schemas will use a different code path. For example, aio:///path/device will direct Mayastor to leverage the AIO subsystem to access the underlying device, whereas uring:/// would cause it to use the recently introduced io_uring subsystem within the kernel. For the purposes of this benchmarking exercise, we will make use of the pcie:/// scheme to access underlying storage devices directly and bypass kernel involvement altogether. The setup of user space PCIe device access is outside of the scope of this document.

Background information on reactor loops

At the core, Mayastor runs in a tight loop to process incoming data. For each CPU core assigned to Mayastor, a reactor is constructed. Within the reactor, there are several data structures for processing incoming Futures and SPDK Threads (not to be confused with a pthread_t). Each thread is a unit of work that, during the loop, can be polled to process any incoming work.

However, the creation of threads is dynamic, so it is crucial that we can safely update this list on a per-core basis; for these updates, we use an additional queue for incoming new threads for the current reactor.

Putting this all together results in a loop that looks roughly like this:

```
impl  Future for Reactor {
    fn poll(self: Pin<&mutSelf>, cx: &mut Context) -> Poll<Self::Output>
{
        match self.get_state() {
            ReactorState::Running => {
                // why 8? why not 9? or 128?
                self.poll_times(8);
            },
            // other states not important now
        }
    }
}
```

Within the poll_times() functions, we can perform various work types and determine what queues we want to poll and the balance between different queues.

```
fn poll_times(&self, times: u32) {
    (0..times).for_each(|_| {
      // poll the storage threads
      self.threads.borrow_mut().iter().for_each(|t| t.poll());



      // poll futures that handle management tasks
      self.run_futures();



      // any new threads for us to add to the poll loop?
      if self.incomming {
           while let Ok(i) = self.incoming.pop() {
               self.threads.borrow_mut().push_back(i);
           }
      }
    }
}
```

We want to answer one question through this benchmarking: What is the cost associated with the reference counting involved? Additionally, our performance analysis can reveal the sweet spot for iterations between each poll(). We will not be answering all such questions within this report, but considering them should give the reader a better understanding of the purpose of such analysis.

NVMe-oF TCP performance measurements

To determine the performance envelope of Mayastor and Kubera Propel, we will first measure the performance of a local NVMe device and use that as the basis of comparison against that same NVMe SSD when connected over NVMe-oF TCP.

NVMe supports a variety of transports, the first transport being RDMA. Other benchmark studies have shown that, compared to TCP, RDMA has far lower latency. However, RDMA capable networks are far from ubiquitous, so we choose to run the tests with TCP as the transport.

In this exercise, we will first perform simple baseline tests on the hosts alone. We will then deploy and use Kubernetes, provisioning a PVC and running the same benchmark against it. Finally, we will repeat the benchmark with an additional replica configured to examine the overhead imposed by synchronous mirroring of I/O.

SOLUTION: MAYADATA AND INTEL TOGETHER

By partnering with Intel in the use of Intel Optane and the use of Intel SPDK, MayaData has been able to validate the promise of this approach to deliver extremely high performance per container, per workload performance using its latest storage engine (Mayastor) in Kubera Propel. Much more information about Intel Optane is available from Intel here.

Hardware Configuration

In this solution, the hardware configuration is as follows:

- 3x Intel® Server System R2224WFTZSR
- 2x Intel® Xeon® Gold 6252 CPU 24cores @ 2.10GHz (on each node)
- Intel® Optane™ SSD DC P4800X Series 1.5TB (on each node)
- 256GB DDR4-2933 (on each node)
- Mellanox Technologies MT28908 Family [ConnectX-6]

Note: An operating system using Linux kernel 5.8 is used during the validation process.

System configuration

In this solution, k3s is used to provision a Kubernetes cluster. After deployment, we label two nodes such that only they run Mayastor, and the remaining node will be used for running the workload itself.

System tuneables

kernel modules loaded:

- nvme-tcp

system tunable:

- net.core.rmem_max = 268435456
- net.core.wmem_max = 268435456
- net.ipv4.tcp_rmem = 4096 87380 134217728
- net.ipv4.tcp_wmem = 4096 65536 134217728
- net.core.optmem_max = 16777216

bootflags:

- isolcpus=1-4

Baseline device performance

Before getting into more specific workload performance benchmarks, we will first measure the performance of the device used throughout the test. To do so, we run the following Fio workload:

```
[global]
ioengine=linuxaio
thread=1
group_reporting=1
direct=1
norandommap=1
bs=4k
numjobs=8
time_based=1
ramp_time=0
runtime=300
iodepth=64



[random-read-QD64]
filename=/dev/nvme1n1
rw=randread
stonewall
```

```
[random-write-QD64]
filename=/dev/nvme1n1
rw=randwrite
stonewall


[random-rw-QD64]
filename=/dev/nvme1n1
rw=randrw
stonewall
```

Workload type QD=64	IOPS
Random reads	585K
Random write	516K
Random read/write 50/50	476K

These numbers are extremely high and are provided by a single device[2]. Note that the benchmark itself is rather synthetic in the sense that, in practice, no workload is 100% random. Therefore, in real use cases, these numbers will be even higher.

Regardless, what we are seeking to determine the overall performance penalty between local and remote storage with Mayastor, not the absolute numbers themselves.

[2] This performance is higher than the stated performance for each device per Intel specifications. It appears that in this case Intel specifications are conservative.

Deploying Kubera Propel

To provision MayaData's OpenEBS MayaStor data plane on Kubernetes, we use Kubera Propel. For the most part, we use the same deployment model as found in our online documentation, with the exception of two changes to the YAML configuration. To utilize more than one core, we need to change the limits found within the mayastor-daemonset YAML file. Without it, the container will not be able to consume more than one CPU unit.

```
limits:
            cpu: "4"
            memory: "500Mi"
            hugepages-2Mi: "1Gi"
          requests:
            cpu: "4"
            memory: "500Mi"
            hugepages-2Mi: "1Gi"
```

We have also added the argument **-m -0xF** to instruct Mayastor to start a reactor on the first 4 cores (see the background for more information on what this reactor is). As we are using NVMe-oF TCP, the following appropriate storage classes are created:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: nvmf
parameters:
  repl: '1'
  protocol: 'nvmf'
provisioner: io.openebs.csi-mayastor
---
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: nvmf-mirror
parameters:
  repl: '2'
  protocol: 'nvmf'
provisioner: io.openebs.csi-mayastor
```

This YAML creates two storage classes; one using a single replica and the other using synchronous mirroring (RAID-1) with nvmf (NVMe-oF TCP) as the transport.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ms-volume-claim
spec:
  accessModes:
   - ReadWriteOnce
  resources:
    requests:
      storage: 100G
  storageClassName: nvmf
```

After creating the PVC, Mayastor's control plane creates a CRD, "Mayastor Volume" (MSV), that will contain additional information about the corresponding volume. Using kubectl describe msv -n mayastor, we get:

```
Name:         ba081dc3-46db-445b-969c-7e5245aba146
Namespace:    mayastor
Labels:       <none>
Annotations:  <none>
API Version:  openebs.io/v1alpha1
Kind:         MayastorVolume
Metadata:
  Creation Timestamp:  2020-09-11T08:49:30Z
  Generation:          1
  Managed Fields:
    API Version:  openebs.io/v1alpha1
    Fields Type:  FieldsV1
    fieldsV1:
      f:spec:
        .:
        f:limitBytes:
        f:preferredNodes:
        f:replicaCount:
        f:requiredBytes:
        f:requiredNodes:
      f:status:
        .:
        f:nexus:
```

```
.:
          f:children:
          f:deviceUri:
          f:state:
        f:node:
        f:reason:
        f:replicas:
        f:size:
        f:state:
    Manager:         unknown
    Operation:       Update
    Time:            2020-09-11T08:51:18Z
  Resource Version:  56571
  Self Link:
 /apis/openebs.io/v1alpha1/namespaces/mayastor/mayastorvolumes/ba081dc3
-46db-445b-969c-7e5245aba146
  UID:               94e11d58-fed9-44c9-9368-95b6f0712ddf
Spec:
  Limit Bytes:  0
  Preferred Nodes:
  Replica Count:   1
  Required Bytes:  100000000000
  Required Nodes:
Status:
  Nexus:
    Children:
      State:     CHILD_ONLINE
      Uri:       bdev:///ba081dc3-46db-445b-969c-7e5245aba146
    Device Uri:  nvmf://x.y.z.y:8420/nqn.2019-05.io.openebs:nexus-
ba081dc3-46db-445b-969c-7e5245aba146
    State:       NEXUS_ONLINE
  Node:          atsnode3
  Reason:
  Replicas:
    Node:     node3
    Offline:  false
    Pool:     pool-atsnode3
    Uri:      bdev:///ba081dc3-46db-445b-969c-7e5245aba146
  Size:       100000000000
  State:      healthy
Events:       <none>
```

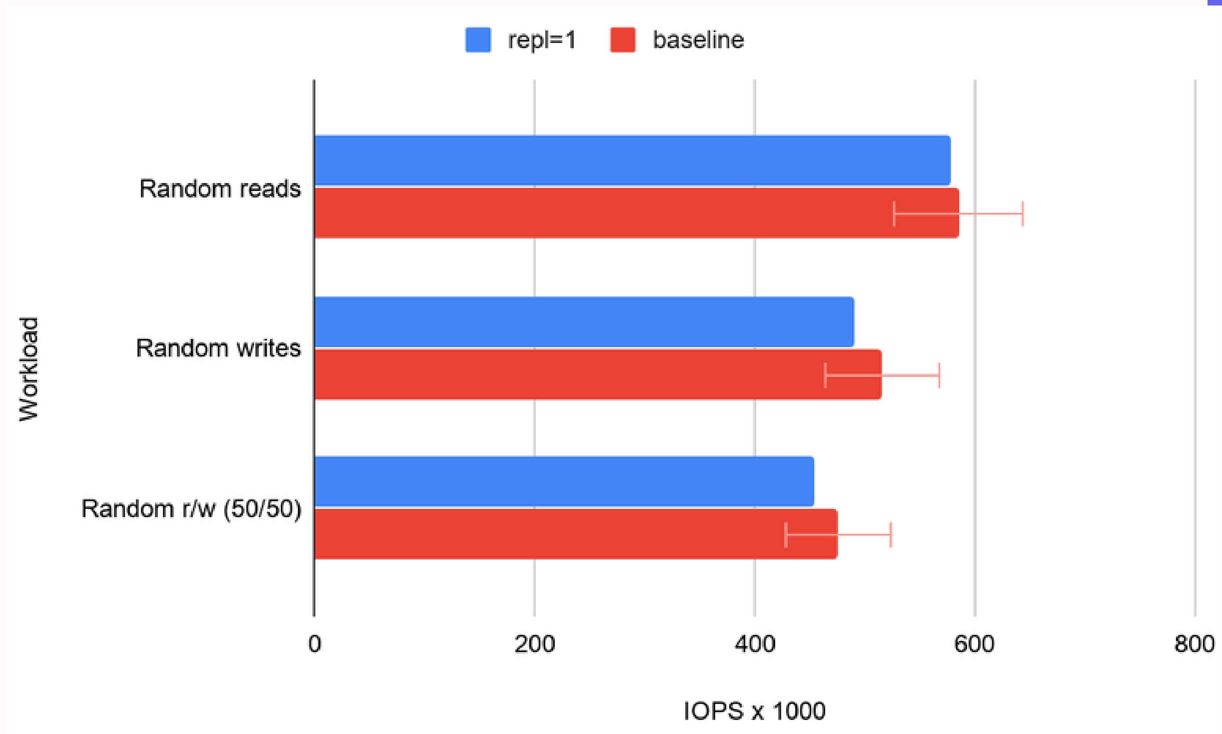
We can see from the MSV that a replica is created on node3. Lastly, we deploy a workload again, this time starting Fio (our workload for this benchmark) within its own container:

```
Name:          fio
Namespace:     default
Priority:      0
Node:          atsnode4/10.40.0.94
Start Time:    Fri, 11 Sep 2020 10:51:18 +0200
Labels:        <none>
Annotations:   <none>
Status:        Running
IP:            172.16.2.3
IPs:
  IP:  172.16.2.3
Containers:
  fio:
    Container ID:
 docker://17b4b6f4a2d8335c8374c56d5b1d9a0f17cf8bfca993e0c06a8b2a6d4a030
2b2
    Image:          nixery.dev/shell/fio/tini
    Image ID:       docker-
pullable://nixery.dev/shell/fio/tini@sha256:1b3a999c230d43b50ed867bd269
b9f6637aa34176b92ed0ff2a7348c5cece233
    Port:           <none>
    Host Port:      <none>
    Command:
      tini
      --
    Args:
      sleep
      1000000
    State:          Running
      Started:      Fri, 11 Sep 2020 10:51:38 +0200
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-
qgvqf (ro)
      /volume from ms-volume (rw)
Conditions:
  Type              Status
  Initialized       True
 Ready              True
  ContainersReady   True
  PodScheduled      True
Volumes:
```

```
ms-volume:
    Type:       PersistentVolumeClaim (a reference to a
PersistentVolumeClaim in the same namespace)
    ClaimName:  ms-volume-claim
    ReadOnly:   false
  default-token-qgvqf:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-qgvqf
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason                  Age        From
 Message
  ----    ------                  ----       ----                     --
-----
  Normal  Scheduled               <unknown>  default-scheduler
 Successfully assigned default/fio to atsnode4
  Normal  SuccessfulAttachVolume  54m        attachdetach-controller
 AttachVolume.Attach succeeded for volume "pvc-ba081dc3-46db-445b-969c-
7e5245aba146"
  Normal  Pulling                 54m        kubelet, atsnode4
 Pulling image "nixery.dev/shell/fio/tini"
  Normal  Pulled                  54m        kubelet, atsnode4
 Successfully pulled image "nixery.dev/shell/fio/tini"
  Normal  Created                 54m        kubelet, atsnode4
 Created container fio
  Normal  Started                 54m        kubelet, atsnode4
 Started container fio
```

Notice that we use sleep as the container's start command so that we can quickly start Fio when we are ready, using a simple script found in the Mayastor repository.

Workload type QD=64	IOPS repl=1	Baseline	% difference (Mayastor overhead)
Random reads	579K	585K	1.04
Random write	490K	516K	5.31
Random read/write	454K	476K	5.62



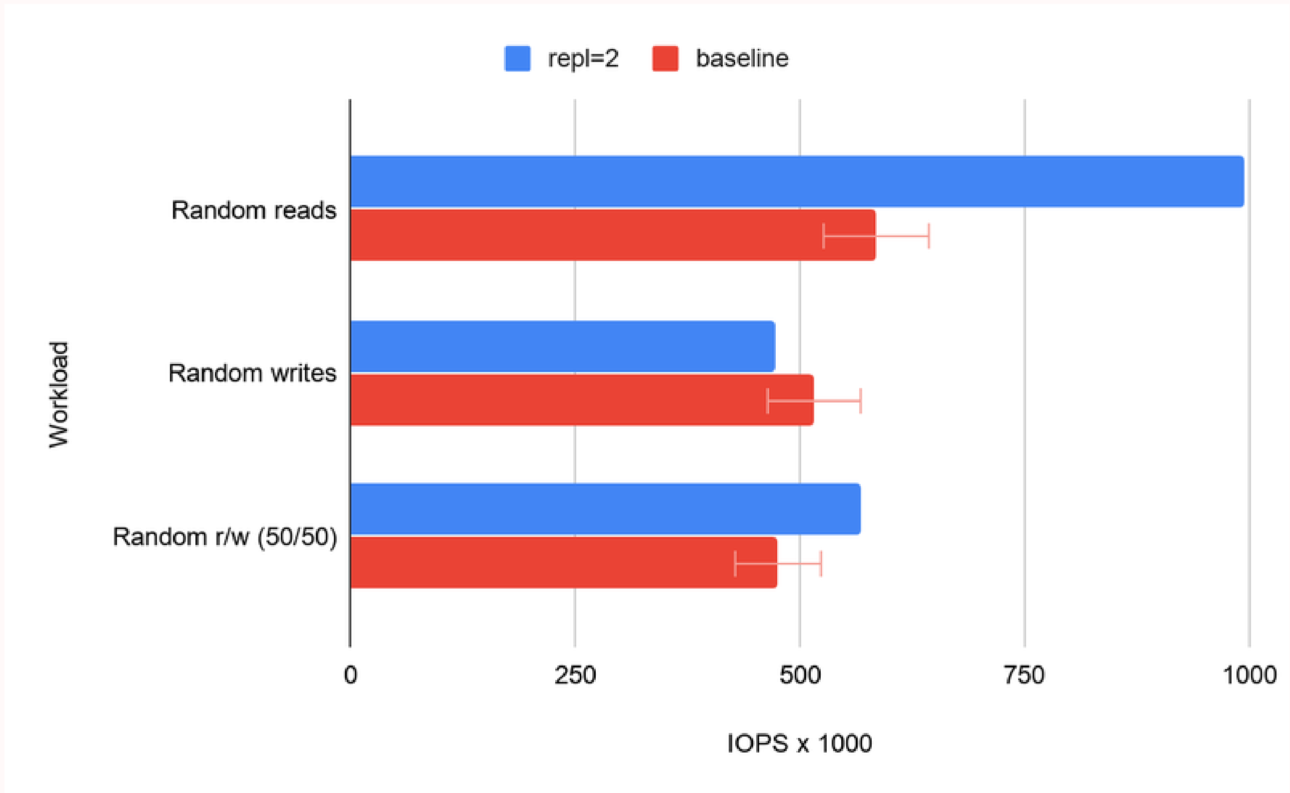
As we can see from the results, the initial analysis shows performance impact well within the bounds of our 10% goal.

Benchmarking with 2 replicas (mirror)

To determine the overhead of Kubera Propel / Mayastor mirroring, we repeat the same test with two replicas. As we now have double the disk bandwidth, we expect to see that the read performance will improve. For writes, however, we can expect a drop in performance because we must do each write to both disks before we acknowledge the IO. As an aside, there is no caching in Mayastor, which reduces the risk of data loss when using Mayastor.

Workload type QD=64	IOPS repl=1	Baseline	% difference
Random reads	993K	585K	+41.09%
Random write	473K	516K	-9.09%
Random read/write	568K	476K	+16.2%

With this run, we are close to our 10% goal. However, as the 10% goal is intended to apply to single replicas only, achieving only a 9% impact is a success.



Running DataStax Enterprise on Kubera Propel

Having examined the underlying performance of Kubera Propel and OpenEBS Mayastor, we would now like to examine a popular workload, Datastax Cassandra.

To do so, we again use an Intel based (Hardware configuration) bare metal Kubernetes cluster. This time we install DataStax Cassandra to use storage from OpenEBS Mayastor as provisioned and managed by Kubera Propel. This guide will help you to install DataStax Enterprise using the **kubectl** method. An explanation of how to provision Mayastor volumes is available in the previous section Deploying Kubera Propel, where the Storage Classes nvmf and **nvmf-mirror** were created.

Getting Started

Verify that Kubera Propel related pods are running properly:

```
$ kubectl get pods -n kube-system -l role=kubera-propel
NAME                       READY   STATUS    RESTARTS   AGE
kubera-controller-0        5/5     Running   0          24s
abc-node-24jkd             2/2     Running   0          17s
abc-node-bj4xz             2/2     Running   0          17s
abc-node-ltvzn             2/2     Running   0          17s
```

Verify that Kubera Propel storage classes are created:

```
$ kubectl get sc
NAME                     PROVISIONER          AGE
nvmf (default)   mayadata.io/mayastor   8d
nvmf-mirror      mayadata.io/mayastor   8d
local-pv         mayadata.io/openebs    8d
```

Install DataStax Enterprise

In this solution, we use DataStax Cass Operator to install Cassandra and configure it to utilize Kubera Propel. Installing the Cass Operator itself is a straightforward process. There are different manifests for each Kubernetes version from 1.13 through 1.17. Apply the relevant manifest to your cluster as follows:

```
$ K8S_VER=v1.16
kubectl apply -f

https://raw.githubusercontent.com/datastax/cass-
operator/v1.4.1/docs/user/cass-operator-manifests-$K8S_VER.yaml
```

Note that when this solution guide was written, the latest version of DataStax Cass Operator was v.1.4.1. Before you install the operator, make sure to change the version above with the latest version.

The above installs the Cass Operator in your Kubernetes cluster, with 1.16 manifests. Specify your Kubernetes version for K8S_VER and apply the command directly. Since our Kubernetes version is 1.16, we used the above command to install the DataStax Cass Operator.

Verify that the Cass Operator is installed successfully:

```
$ kubectl -n cass-operator get pods --selector name=cass-operator

NAME                              READY    STATUS     RESTARTS    AGE
cass-operator-78c9999797-sxn54    1/1      Running    0           32s
```

Now, let's download the YAML spec of Cassandra and update the Storage Class name with the one we created above.

```
wget
https://raw.githubusercontent.com/datastax/cass-
operator/v1.4.1/operator/example-cassdc-yaml/cassandra-3.11.x/example-
cassdc-minimal.yaml
```

Update the Storage Class name with the one which you have created above. The change has to be done in
spec.storageConfig.cassandraDataVolumeClaimSpec.storageClassName

In our setup, we have updated the Storage Class name as nvmf and volume size of 50Gi.

After the required modification, apply the DataStax Cassandra StatefulSet YAML spec in the following way.

```
$ kubectl -n cass-operator create -f example-cassdc-minimal.yaml
```

Verify DataStax Cassandra pods are running successfully:

```
$ kubectl -n cass-operator get pods --selector
cassandra.datastax.com/cluster=cluster1

NAME                             READY   STATUS    RESTARTS   AGE
cluster1-dc1-default-sts-0       2/2     Running   0          3m33s
cluster1-dc1-default-sts-1       2/2     Running   0          3m33s
cluster1-dc1-default-sts-2       2/2     Running   0          3m32s
```

Verify PVCs are created successfully for each Cassandra pod:

```
$ kubectl -n cass-operator get pvc

NAME                                        STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS   AGE
server-data-cluster1-dc1-default-sts-0      Bound    pvc-55f6a23f-4947-
49de-96ff-d507391c5888   50Gi       RWO             nvmf    3m57s
server-data-cluster1-dc1-default-sts-1      Bound    pvc-1a3a1357-ce91-
4584-a738-b8c02924aae4   50Gi       RWO             nvmf    3m57s
server-data-cluster1-dc1-default-sts-2      Bound    pvc-32910ab4-9064-
4dbe-bf76-eee8c5bec03f   50Gi       RWO             nvmf    3m56s
```

Check the readiness of the Cassandra DataCenter by running the following command:

```
$ kubectl -n cass-operator get cassdc/dc1 -o "jsonpath=
{.status.cassandraOperatorProgress}"
```

If output returns as Ready, then you can use the Cassandra DB for database operations.

You can also verify the health of each instance of the DataStax Cassandra Datacenter by the following command.

```
$ kubectl -n cass-operator exec -it -c cassandra cluster1-dc1-default-
sts-0 -- nodetool status

Datacenter: dc1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens       Owns (effective)  Host ID
Rack
UN  10.64.0.5   84.42 KiB  1            67.3%             84774e3d-77da-
4baa-9997-72648191b8ab  default
UN  10.64.1.6   70.21 KiB  1            68.6%             68117f18-b453-
4f44-a102-27d94d144f41  default
UN  10.64.2.8   70.34 KiB  1            64.1%             bae395b4-3884-
4665-bb02-c95fc3d0a656  default
```

Accessing Cassandra Database

Let’s do some sample Database operations. First, we can take one of the application pods and exec into it. For that, you have to use a username and password to authenticate with the database. To get the username and password, get the information from the secret.

```
$ kubectl get secret -o yaml cluster1-superuser -n cass-operator
```

This will return the information of Username and Password. The username and password will be in base64 encoded format, and it should be decoded first before using for authentication.

In our example, the following is a snippet of secret information.

```
data:
  password:
VldZVjltOUstalI3cTB3X1JEb2lsREEySTVyRkFOcFJrcFpQTmVEa0VvNmhlSEE3a2F3bEh
n
  username: Y2x1c3RlcjEtc3VwZXJ1c2Vy
```

The decoded format can be obtained by using the following command.

```
Username:
$ echo 'Y2x1c3RlcjEtc3VwZXJ1c2Vy' | base64 -d
cluster1-superuser

Password:

$ echo
'UDdhb1JySGRvWTVySVZiN0RYbndNZENuUjQwQkNSVk90dm92TkF5S0VEOEN1U0Zrc2JVemJr
' | base64 -d
VWYV9m9K-jR7q0w_RDoilDA2I5rFANpRkpZPNeDkEo6heHA7kawlHg
```

Using the above information, login to database using the following command:

```
$ kubectl exec -n cass-operator -i -t -c cassandra cluster1-dc1-
default-sts-0 -- /opt/cassandra/bin/cqlsh -u cluster1-superuser -p
VWYV9m9K-jR7q0w_RDoilDA2I5rFANpRkpZPNeDkEo6heHA7kawlHg

Connected to cluster1 at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.6 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
cluster1-superuser@cqlsh>
```

Now, let's create a Keyspace and add a table with some entries into it.

Creating a Keyspace

```
$ CREATE KEYSPACE IF NOT EXISTS cycling WITH replication = { 'class' :
'NetworkTopologyStrategy', 'dc1' : '3' };
```

Creating Data Objects

```
$ use cycling;
$ CREATE TABLE IF NOT EXISTS cycling.cyclist_semi_pro (
   id int,
   firstname text,
   lastname text,
   age int,
   affiliation text,
   country text,
   registration date,
   PRIMARY KEY (id));
```

Inserting and Querying Data

```
$ INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age,
affiliation, country, registration) VALUES (5, 'Irene', 'Cantona', 24,
'Como Velocità', 'ITA', '2012-07-22');

$ INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age,
affiliation, country, registration) VALUES (10, 'Agnes', 'Cavani', 22,
'Chamonix Hauteurs', 'FRA', '2020-01-02');

$ INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age,
affiliation, country, registration) VALUES (16, 'Jenny', 'Hamler', 28,
'CU Alums Crankworkz', 'USA', '2012-07-22');
```

Get the list of details from the database:

```
$ SELECT * FROM cycling.cyclist_semi_pro;
 id | affiliation           | age | country | firstname | lastname  |
registration
----+-----------------------+-----+---------+-----------+-----------+--
------------
  5 |         Como Velocità |  24 |     ITA |     Irene |   Cantona |
2012-07-22
 10 |     Chamonix Hauteurs |  22 |     FRA |     Agnes |    Cavani |
2020-01-02
 16 |   CU Alums Crankworkz |  28 |     USA |     Jenny |    Hamler |
2012-07-22
(3 rows)
$ exit
```

DataStax stress benchmark

In this section, we benchmark the performance of DataStax Cassandra running on Kubera Propel and OpenEBS Mayastor. In order to do so, we run the following cassandra-stress workload:

```
./cassandra-stress write n=3000000 -rate threads=50 -mode native cql3
user=cluster1-superuser password=REMOVED
```

DataStax benchmark results

These numbers are extremely high and are provided by a **Mayastor** device.

```
Results:
Op rate                   :   24,258 op/s  [WRITE: 24,258 op/s]
Partition rate            :   24,258 pk/s  [WRITE: 24,258 pk/s]
Row rate                  :   24,258 row/s [WRITE: 24,258 row/s]
Latency mean              :    2.0 ms [WRITE: 2.0 ms]
Latency median            :    1.5 ms [WRITE: 1.5 ms]
Latency 95th percentile   :    4.2 ms [WRITE: 4.2 ms]
Latency 99th percentile   :   11.2 ms [WRITE: 11.2 ms]
Latency 99.9th percentile :  111.9 ms [WRITE: 111.9 ms]
Latency max               :  355.5 ms [WRITE: 355.5 ms]
Total partitions          :  3,000,000 [WRITE: 3,000,000]
Total errors              :          0 [WRITE: 0]
Total GC count            : 0
Total GC memory           : 0.000 KiB
Total GC time             :    0.0 seconds
Avg GC time               :    NaN ms
StdDev GC time            :    0.0 ms
Total operation time      : 00:02:03
```

CONCLUSION

The Container Attached Storage pattern has proliferated & is becoming a de facto standard means of running stateful workloads on Kubernetes. The CNCF project OpenEBS has a variety of underlying storage engines that can be deployed to provide the desired capabilities to individual workloads.

Mayastor is the first Container Attached Storage engine that was developed with the tremendous performance capabilities of systems such as Intel's Optane & the NVMe protocol itself in mind. Written in Rust and open source, Mayastor is fast becoming the preferred choice for workloads that need the ease of use and portability of CAS plus the performance otherwise only delivered by the much harder to manage and protect direct access to high performing disk and cloud volumes.

In this benchmarking solution guide, we first demonstrated that we could achieve our goal of staying under a 10% performance overhead.

With two replicas, considering that for writes, we need to double the work, we remain within the bounds of our 10% goal. The read performance increases significantly and does fully double. Further optimization is possible, including configurations that would reduce the time spent on CPU.

Recognizing the importance of determining how underlying storage performance translates into the performance of important workloads, we then examined the performance of Datastax Cassandra on the same systems. In order to do so, we utilized the Datastax Cassandra stress benchmark to run 3 million writes on an example database. Our results showed extremely low latency performance - in the range of 1.5 to 2.0 ms and high row and partition rates.

In conclusion -- this examination has shown that the Container Attached Storage project OpenEBS Mayastor -- as provisioned and operated by the MayaData software Kubera Propel -- can deliver outstanding performance at the storage layer. Additionally, we demonstrated excellent performance from Datastax Cassandra running on this same solution.

www.mayadata.io