VMware vSAN™ as Persistent Storage for MongoDB in Containers

First Published On: 05-31-2017
Last Updated On: 12-01-2017
# Table of Contents

1. Executive Summary  
   1.1. Business Case  
   1.2. Solution Overview  
   1.3. Solution Benefits  
   1.4. Key Results  
2. Introduction  
   2.1. Purpose  
   2.2. Scope  
   2.3. Audience  
3. Technology Overview  
   3.1. VMware vSphere 6.5  
   3.2. VMware vSAN 6.6  
   3.3. MongoDB 3.4  
   3.4. Kubernetes 1.5  
4. Solution Configuration  
   4.1. Solution Architecture  
   4.2. Hardware Resources  
   4.3. Software Resources  
   4.4. Network Configuration  
   4.5. vSAN as Persistent Storage for a Kubernetes Cluster  
   4.6. VM and Database Configuration  
   4.7. MongoDB Deployment  
5. Test Tools and Settings  
   5.1. YCSB  
   5.2. vSAN Tools-vSAN Performance Service  
6. Solution Validation  
   6.1. MongoDB Performance on vSAN  
   6.2. Failure Validation  
   6.3. Failback and Reconfiguration  
7. Best Practices  
   7.1. Best Practices  
8. Conclusion  
   8.1. Conclusion  
9. About the Author  
   9.1. About the Author
1. Executive Summary

This section covers the business case, solution overview, solution benefits, and key results of vSAN as Persistent Storage for MongoDB in containers solution.
1.1 Business Case

Due to the changing enterprise data landscape unstructured databases, such as NoSQL, are needed in lieu of traditional relational databases for many applications. NoSQL databases provide advantages over relational databases regarding performance, scalability, and suitability for cloud environments. Especially, MongoDB, the popular NoSQL document store database used by thousands of enterprises. More and more customers from across the globe and diverse industries have already adopted MongoDB.

Meanwhile, Docker, a method to package and deploy self-sufficient applications more efficiently, is quickly becoming mainstream. Using Docker Containers for deploying MongoDB instances provides several benefits, such as easy to maintain, scaling up and down on-demand. In addition, using Docker Containers for deploying MongoDB instances is based on globally accessible and shareable images.

Even with these tools, creating, maintaining, upgrading and high availability of multiple containers is not easy. To address this, orchestration frameworks such as Kubernetes is widely used. Kubernetes is an open source system for managing containerized applications across multiple hosts, providing basic mechanisms for deployment, maintenance, and scaling of applications.

Managing storage is a distinct requirement for database applications. When deploying database applications like MongoDB in a Docker Container solution, persistent storage becomes a mandatory requirement. Persistent storage makes sure data is safe and persistent against node failures and cluster failures. Depending on the container manager and cloud platform, the configuration required to enable persistent storage can vary greatly. Kubernetes provides some unique models for managing persistent, cluster-scoped storage for applications requiring persistent data.

When you run databases in a container environment, there are many issues to consider performance, data persistence, and service availability of multiple containers. This solution address all of these.

1.2 Solution Overview

This reference architecture is a showcase of using VMware vSAN™ as Hyper-converged infrastructure for MongoDB in containers on a Kubernetes cluster:

• We illustrate using vSAN as persistent storage for MongoDB in containers.
• We demonstrate the capability of Kubernetes vSphere Cloud Provider on vSAN to provide seamless consolidation to handle Kubernetes node failures without any data loss.
• We show the resiliency of vSAN against host failures.
• We measure the performance of multiple concurrent MongoDB instances.
• We evaluate the performance of a MongoDB with replica set enabled and demonstrate the application-level high availability.

1.3 Solution Benefits

The solution, designing and deploying MongoDB in containers backed by vSAN and vSphere, as a whole, provides the following benefits for customers:

1. Simple deployment—Deployment of the multiple nodes of a Kubernetes cluster can be done in a few minutes by cloning the VM created by a template OS OVA. In addition, users can quickly deploy multiple MongoDB instances using a YAML file, which eliminates the efforts to install binaries and perform configuration of MongoDB instances one by one.
2. Minimal maintenance effort—Orchestrated deployment by Kubernetes can leverage pods that act as a logical host in the Kubernetes cluster to manage the lifecycle of the containers. vSphere Cloud Provider can coordinate with Kubernetes to manage virtual disks automatically.
3. High availability capability—vSAN provides high availability for VMs, reducing the possibility of DU/DL (Data Unavailable/Data Loss)
4. Flexible configurations—The ability to have different configurations for design purposes is possible and still have the performance validated by a standard benchmark tool.
1.4 Key Results

The reference architecture in this paper:

- Demonstrates vSAN as persistent storage for a Kubernetes cluster
- Produces a high availability solution for the storage level, the container orchestrated cluster level and the application level
- Validates predictable performance for various configurations and workloads, and demonstrates performance scalability
2. Introduction

This section provides the purpose, scope, and audience of this document.
2.1 Purpose

This reference architecture verifies MongoDB in containers, orchestrated by Kubernetes cluster using vSAN persistent storage on a vSphere cluster.

2.2 Scope

The reference architecture covers the following scenarios:

- Various configurations to support running a MongoDB in containers on vSAN
- Kubernetes vSphere Cloud Provider to support persistent storage
- Resiliency against node failure and host failure
- Performance validation of various configurations

2.3 Audience

This reference architecture is intended for MongoDB and other NoSQL database administrators and storage architects involved in planning, designing, or administering container based micro services on vSAN.
3. Technology Overview

This section provides an overview of the technologies used in this solution:

- VMware vSphere 6.5
- VMware vSAN 6.6
- MongoDB 3.4
- Kubernetes 1.5
3.1 VMware vSphere 6.5

VMware vSphere® 6.5 is the next-generation infrastructure for next-generation applications. It provides a powerful, flexible, and secure foundation for business agility that accelerates the digital transformation to cloud computing and promotes success in the digital economy.

vSphere 6.5 supports both existing and next-generation applications through its:

- Simplified customer experience for automation and management at scale
- Comprehensive built-in security for protecting data, infrastructure, and access
- Universal application platform for running any application anywhere

With vSphere 6.5, customers can run, manage, connect, and secure their applications in a common operating environment, across clouds and devices.

3.2 VMware vSAN 6.6

vSAN 6.6 focuses on enabling customers to modernize their infrastructure by enhancing three key areas of today’s IT needs: faster performance, higher security, and lower cost.

The industry’s first native HCI encryption solution and a highly available control plane is delivered in vSAN 6.6 to help customers evolve without risk and without sacrificing flash storage efficiencies. Operational costs are reduced with 1-click firmware and driver updates. vSAN 6.6 significant enhancements enable customers to scale to tomorrow’s IT demands. See VMware vSAN 6.6 Technical Overview for details.

The performance increase benefits this solution. vSAN has a performance advantage thanks to its native, vSphere architecture. vSAN 6.6 introduces further optimizations to deliver up to 50% higher flash performance, enabling over 150K IOPS per host. This means you can run both traditional enterprise workloads more efficiently and with greater consolidation while also having the confidence to deploy new workloads like big data. Specifically, some of the performance enhancements include reduced overhead of checksum; improved dedupe and compress; destaging optimizations; object and management improvements.

3.3 MongoDB 3.4

MongoDB is a document-oriented database. The data structure is composed of field and value pairs. MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents.

The advantages of using documents are:

- Documents (such as objects) correspond to native data types in many programming languages.
- Embedded documents and arrays reduce the need for expensive joins.
- Dynamic schema supports fluent polymorphism.

For more key features of MongoDB, refer to Introduction to MongoDB.

MongoDB 3.4 is the latest release of the leading database for modern applications, a culmination of native database features and enhancements included in this version. For more information about this version, refer to mongodb-3.4.

Replica Set

A replica set in MongoDB is a group of processes that maintain the same data set. Replica sets provide redundancy and high availability and are the basis for all production deployments.
**Sharding**

*Sharding* in MongoDB is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations.

### 3.4 Kubernetes 1.5

Kubernetes is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts, providing a container-centric infrastructure.

With Kubernetes, users can quickly and efficiently respond to customers’ needs to:

- Deploy applications quickly and predictably.
- Scale applications on the fly.
- Seamlessly roll out new features.
- Optimize use of hardware by using only the resources you need.
- Describe persistent storage and allow customers to discover and request persistent storages on-demand.

Kubernetes 1.5 is a production workload ready version, which can run stateful applications in containers with the eventual goal of running all applications on Kubernetes.
4. Solution Configuration

This section introduces the resources and configurations for the solution including an architecture diagram, hardware and software resources and other relevant configurations.
4.1 Solution Architecture

The solution architecture is depicted in Figure 1. We created a 4-node vSphere and vSAN cluster for the Kubernetes cluster and then deployed MongoDB services with sharding enabled on replica sets on the nodes.

We deployed eight Kubernetes workers (nodes) along with one Kubernetes master using Kubernetes-Anywhere in the vSAN and vSphere cluster, with each host having two workers. All workers are active and have identical configurations. Four nodes will be the working nodes after our MongoDB shard services are deployed and the other four will be standby workers. VMware vSphere vMotion® is supported and required to make sure every host have one working node residing on it to avoid the possible compute resource contention on one host.

We used dynamic provisioning to provision the persistent storage for MongoDB services. The disks are independent and persistent even when the MongoDB services are deleted.

We enabled Journal in the MongoDB configuration so that any database update can be acknowledged after the write is flushed to the disk.

![Figure 1. Solution Architecture](image)

4.2 Hardware Resources

We used direct-attached SSDs on VMware ESXi™ servers to provide a vSAN Datastore. Each ESXi server has two disk groups each consisting of one cache-tier SSD and four capacity-tier SSDs.
Each ESXi Server in the vSAN Cluster has the following configuration as shown in Table 1.

Table 1. Hardware Resources per ESXi server

<table>
<thead>
<tr>
<th>PROPERTY</th>
<th>SPECIFICATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>Dell PowerEdge R630</td>
</tr>
<tr>
<td>CPU</td>
<td>2 sockets, 24 cores each of 2.6GHz with hyper-threading enabled</td>
</tr>
<tr>
<td>RAM</td>
<td>256GB DDR4 RDIMM</td>
</tr>
<tr>
<td>Network adapter</td>
<td>2 x Intel 10 Gigabit X540-AT2, + I350 1Gb Ethernet</td>
</tr>
<tr>
<td>Storage adapter</td>
<td>2 x 12Gbps SAS PCI-Express (Dell PowerEdge RAID H730 mini)</td>
</tr>
</tbody>
</table>
| Disks          | SSD: 2 x 400GB drive as cache SSD  
SSD: 4 x 400GB drive as capacity SSD (ATA Intel SSD SC2BX40) |

4.3 Software Resources

Table 2 shows the software resources used in this solution.

Table 2. Software Resources

<table>
<thead>
<tr>
<th>SOFTWARE</th>
<th>VERSION</th>
<th>PURPOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMware vCenter Server® and ESXi</td>
<td>6.5.0d (vSAN 6.6 is included)</td>
<td>ESXi Cluster to host virtual machines and provide vSAN Cluster. VMware vCenter Server provides a centralized platform for managing VMware vSphere environments</td>
</tr>
<tr>
<td>VMware vSAN</td>
<td>6.6</td>
<td>Software-defined storage solution for hyper-converged infrastructure</td>
</tr>
<tr>
<td>Project Photon OS™</td>
<td>1.0</td>
<td>Photon OS is a minimal Linux container host, optimized to run on VMware platforms</td>
</tr>
<tr>
<td>Kubernetes</td>
<td>1.5.3</td>
<td>Kubernetes is a production-ready, open source platform designed with Google’s accumulated experience in container orchestration</td>
</tr>
<tr>
<td>MongoDB</td>
<td>3.4</td>
<td>MongoDB is an open source database that uses a document-oriented data model. MongoDB is built on an architecture of collections and documents. Documents comprise sets of key-value pairs and are the basic unit of data in MongoDB.</td>
</tr>
<tr>
<td>Yahoo Cloud Serving Benchmark (YCSB)</td>
<td>0.12.0</td>
<td>YSCB is a framework and common set of workloads for evaluating the performance of different “key-value” and “cloud” serving stores.</td>
</tr>
</tbody>
</table>
4.4 Network Configuration

We created a vSphere Distributed Switch™ to act as a single virtual switch across all associated hosts in the data cluster.

The vSphere Distributed Switch uses two 10GbE adapters for the teaming and failover. A port group defines properties regarding security, traffic shaping, and NIC teaming. To isolate vSAN, VM (node) and vMotion traffic, we used the default port group settings except for the uplink failover order. We assigned one dedicated NIC as the active link and assigned another NIC as the standby link. For vSAN and vMotion, the uplink order is reversed. See Table 3.

Table 3. Network Configuration

<table>
<thead>
<tr>
<th>DISTRIBUTED PORT GROUP</th>
<th>ACTIVE UPLINK</th>
<th>STANDBY UPLINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMware vSAN</td>
<td>Uplink2</td>
<td>Uplink1</td>
</tr>
<tr>
<td>VM and vSphere vMotion</td>
<td>Uplink1</td>
<td>Uplink2</td>
</tr>
</tbody>
</table>

vSAN and vMotion are on separate VLANs.

4.5 vSAN as Persistent Storage for a Kubernetes Cluster

Managing storage is a definite requirement for database applications. The persistent storage subsystem of Kubernetes provides a group of APIs for managing persistent, cluster-scoped storage for applications requiring long-lived data. Using the static provision mode, administrators describe available storage with access modes and storage capacity; users request storage by describing their requirements for resources and mount capabilities. Furthermore, Kubernetes provides the unique dynamic provisioning feature, which allows storage volumes to be created on demand. Dynamic provisioning eliminates the need for cluster administrators to pre-provision storage and automatically provisions storage according to users’ requests.

The persistent storage feature of Kubernetes is highly dependent on the cloud environment. To support the persistent storage requirement for running Kubernetes applications inside the vSphere environment, VMware developed vSphere Cloud Provider and its corresponding volume plugin:

Cloud Provider is a module in Kubernetes, which provides interfaces to manage nodes, volumes, and networking routes in corresponding cloud environments. The vSphere Cloud Provider is responsible for converting the requests of Kubernetes Volume Plugins into vCenter/vSphere tasks. For now, the vSphere Cloud Provider mainly supports functionalities related to volumes/storages. For example, when users need dynamically provisioned storage, vSphere Cloud Provider communicates with vCenter Server to create new disks from vSAN. Similarly, attach/detach/delete volume operations are executed according to the requests of vSphere Cloud Provider. Only the Cloud Provider on the Master nodes have privileges to communicate with vCenter Server. Except for vSphere Cloud Provider, VMware also contributed Photon Cloud Provider, which provides similar functionality for Kubernetes applications running inside Photon Controller environment.

Volume Plugin is the middle layer, which provides all the interfaces for volume related operations, such as the mount/unmount/provision/deletion of the volumes. It forwards operations that need communication with vCenter Servers to vSphere Cloud Provider. It also handles vSphere specific configurations for persistent storage. It is one part of Kubernetes core services, which connects cloud-specific volume operations with Cloud Providers. As a result, it is running on every Worker node.

The vSphere Cloud Provider together with vSphere Volume Plugin can reduce the disk management complexity and risks for any Kubernetes applications inside a vSphere cluster to utilize vSAN’s benefits.
How to Create Persistent Storage for Containers

This section describes the steps to create persistent storage for containers to be consumed by MongoDB services on vSAN. After these steps are completed, Cloud Provider will create the virtual disks (volumes in Kubernetes) and mount them to the Kubernetes nodes automatically. The virtual disks are created with the vSAN default policy; however, you can modify the vSAN policy according to your requirements, such as changing the number of failures to tolerate (FTT) to zero.

Step 1 Define StorageClass

A StorageClass provides a mechanism for the administrators to describe the "classes" of storage they offer. Different classes map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators. The YAML format in Figure 3 defines a "platinum" level StorageClass.

```yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
  name: platinum
provisioner: kubernetes.io/vsphere-volume
parameters:
  diskformat: thin
```

Figure 3: Platinum StorageClass Definition in YAML Format
Note: Although all volumes are created on the same vSAN datastore, you can adjust the policy according to actual storage capability requirement by modifying the vSAN policy in vCenter Server. The vSAN policy attributes in StorageClass definition can be specified in the next version of Kubernetes. See https://blogs.vmware.com/virtualblocks/2017/04/17/storage-policy-based-management-containers-orchestrated-kubernetes/ for more details.

Step 2 Claim Persistent Volume

A PersistentVolumeClaim (PVC) is a request for storage by a user. Claims can request specific size and access modes (for example, can be mounted once read/write or many times read-only). The YAML format in Figure 4 claims a 128GB volume with read and write capability.

```yaml
apiVersion: v1
custom: PersistentVolumeClaim
metadata:
name: pvc128gb
annotations:
volume.beta.kubernetes.io/storage-class: "platinum"
spec:
accessModes:
- ReadWriteOnce
resources:
requests:
storage: 128Gi
```

Figure 4: Claiming a 128GM volume in YAML Format

Step 3 Specify the Volume to be Mounted for the Consumption by the Containers

The YAML format in Figure 5 specifies a MongoDB 3.4 image to use the volume from Step 2 and mount it to path /data/db.

```yaml
spec:
containers:
- image: mongo:3.4
  name: mongo-ps
  ports:
  - name: mongo-ps
    containerPort: 27017
    hostPort: 27017
  volumeMounts:
  - name: pvc-128gb
    mountPath: /data/db
    volumes:
    - name: pvc-128gb
      persistentVolumeClaim:
        claimName: pvc128gb
```

Figure 5: Mount a Volume in YAML Format

4.6 VM and Database Configuration

The Kubernetes node VM was created by using Kubernetes-Anywhere with Photon OS 1.0 operating system installed and configured with 8 vCPU and 32GB memory.
We choose a 200GB database to test because 200GB is a medium sized database that stores 200 million documents for two reasons. First, a 200GB database is considered a good starting point to evaluate the NoSQL capability. Secondly, a 200GB database cannot be fully cached or buffered by the memory of the virtual machine, whose total memory size is 4 x 32GB or 128GB. That means almost half of the working set will need to be fetched from the storage subsystem or the vSAN datastore in the solution design. This mimics a real production environment.

The virtual disk layout design is:

- 2 x 250GB for data and journal
- 1 x 20GB for configuration server
- 1 x 20GB for arbiter of the container for MongoDB services

The disks were created and mounted by Kubernetes vSphere Cloud Provider with Paravirtual SCSI controller (PVSCSI) automatically when using YAML file to create persistent storage and persistent storage claim. PVSCSI is more efficient in the number of host compute cycles that are required to process the same number of IOPS. This can improve the performance of virtual machine with a storage IO-intensive workload.

### 4.7 MongoDB Deployment

We designed a MongoDB sharding on Kubernetes with a replica set to use the compute resource of vSphere and vSAN. Shard members and data replicas are distributed evenly on available nodes.

Storage was created and provisioned from vSAN for containers for the MongoDB service by using dynamic provisioning in YAML files. Storage volumes were claimed as persistent ones to preserve the data on the volumes. All mongo servers are combined into one Kubernetes pod per node.

In Kubernetes, as each pod gets one IP address assigned, each service within a pod must have a distinct port. As the mongos are the services by which you access your shard from other applications, the standard MongoDB port 27017 is assigned to them. Table 4 lists port assignments.

<table>
<thead>
<tr>
<th>SERVICE</th>
<th>PORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mongos</td>
<td>27017</td>
</tr>
<tr>
<td>Configuration Server</td>
<td>27018</td>
</tr>
<tr>
<td>Arbiter</td>
<td>27019</td>
</tr>
<tr>
<td>Replication Server (Primary)</td>
<td>27020</td>
</tr>
<tr>
<td>Replication Server (Secondary)</td>
<td>27021</td>
</tr>
</tbody>
</table>

For startup parameters, we enabled the following:

- **shardsvr**: to configure the instance as a shard in a sharded cluster
- **journal**: to make sure the MongoDB instance provides a view of the data on the disk and allow MongoDB to recover from the last checkpoint.

An example of deployment for one node of a MongoDB cluster in YAML file is shown in Figure 6 below.
apiVersion: v1
kind: Service
metadata:
  name: mongodb-node01
labels:
  app: mongodb-node01
  role: mongoshard
  tier: backend
spec:
  selector:
    app: mongodb-shard-node01
    role: mongoshard
    tier: backend
  ports:
  - name: arb03-node01
    port: 27019
    protocol: TCP
  - name: cfg01-node01
    port: 27018
    protocol: TCP
  - name: mgs01-node01
    port: 27017
    protocol: TCP
  - name: rsp01-node01
    port: 27020
    protocol: TCP
  - name: rss04-node01
    port: 27021
    protocol: TCP

---

apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: mongodb-shard-node01
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: mongodb-shard-node01
        role: mongoshard
        tier: backend
    spec:
      containers:
      - name: arb03-node01
        image: mongo:3.4
        args:
        - "--replSet"
        - rs03
        - "--port"
        - "27019"
        - "--shardsvr"
        - "--journal"
        ports:
          - name: arb03-node01
            containerPort: 27019
            volumeMounts:
              - name: vsphere-storage-db-rsa03
                mountPath: /data/db
      - name: rss04-node01
        image: mongo:3.4
        args:
        - "--replSet"
        - rs04
        - "--port"
        - "27021"
        - "--shardsvr"
        - "--journal"
Figure 6. Deployment of a node in YAML format

After the deployment, four pods were created and hosted by the active workers in the Kubernetes cluster. Each pod, one per host, had five containers. The services exposed to the external clients used the same port number shown in Table 4.

MongoDB Configurations

We proposed three configurations. Each configuration has different replication settings and vSAN storage policies. We intended to compare the performance with the same client and workload setting.

To explain the replica set distribution across nodes, we use the RS0x row to represent a replica set of MongoDB. The RSP0x represents the primary of the replica set, and the RSS0x represents the secondary of the replica set. The ARB0x represents the arbiter, which does not have the data stored on the role. Each replica set spreads on one or three nodes, and it acts as one shard in the sharding cluster.

After the data is loaded by YCSB, the total data size is approximately 217GB for the 200 million documents. Each shard has 25 percent data or approximately 54GB data. All three configurations used the same data distribution.

Configuration 1

This configuration is high performance oriented. The replica set has no secondary and arbiter configured. For the vSAN policy setting, we use the default number of FTT (=1) so that the database can be protected by vSAN. Under this configuration, to ensure database integrity and accessibility, MongoDB node corruption is not allowed, and one physical host failure is permitted without losing database integrity and accessibility. See Figure 6 for the detailed data distribution and the configuration example of the disk of Configuration 1.

<table>
<thead>
<tr>
<th>RS01</th>
<th>Node1</th>
<th>Node2</th>
<th>Node3</th>
<th>Node4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSP01 Data Size: 54GB</td>
<td>RSP02 Data Size: 54GB</td>
<td>RSP03 Data Size: 54GB</td>
<td>RSP04 Data Size: 54GB</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7. MongoDB Configuration 1 Layout

Configuration 2

This configuration is high protection oriented. Unlike configuration 1, for every replica set, we have one secondary and one arbiter configured. We use the default number of FTT (=1) so that the database can also be protected by vSAN. This configuration will provide double protections (replica set and vSAN) since every data has four copies. Under this configuration, MongoDB node corruption is allowed, and one physical failure is permitted without losing database integrity and accessibility. See Figure 7 for the detailed data distribution and the configuration example of the disk of Configuration 2.
This configuration aims at application-level protection only without tradeoff of the mirroring write of vSAN and the double sized space consumption. For every replica set, we have one secondary, and one arbiter configured, similar to Configuration 2. However, we change the default number of FTT = 1 to FTT = 0 so that the mirroring write to the write buffer and double space consumption are eliminated. Under this configuration, MongoDB node corruption on one node is allowed, but no physical host failure is permitted because there is possible data loss. See Figure 8 for the detailed data distribution and the configuration example of the disk of Configuration 3.
5. Test Tools and Settings

This section describes the test tools and settings used in this solution.
5.1 YCSB

YCSB is a popular Java open-source specification, and program suite developed at Yahoo! to compare the relative performance of various NoSQL databases. Its workloads are used in various comparative studies of NoSQL databases. We use YCSB to test the performance of MongoDB running as container service.

We used YCSB workload A and B as summarized below:

- Workload A (update heavy workload): 50/50% mix of reads/writes
- Workload B (read mostly workload): 95/5% mix of reads/writes

How to Run Load

Choose a record count (Property ‘recordcount’) to insert into the database that is close to the number of operations you intend to run on it. Choose an appropriate number of threads to generate the desired workload.

To prepare and run YCSB on test client, use the following steps:

1. Download the latest release of YCSB:
   ```
curl -O --location https://github.com/brianfrankcooper/YCSB/releases/download/0.12.0/ycsb-0.12.0.tar.gz
tar xfvz ycsb-0.12.0.tar.gz
cd ycsb-0.12.0
```
2. Set up a database to benchmark. There is a README file under each binding directory.
3. Run the YCSB command (take workload A as an example).
   ```
./bin/ycsb.sh load basic -P workloads/workloada
./bin/ycsb.sh run basic -P workloads/workloada
```

Running Parameters

The following examples show how to run the actual workloads on MongoDB.

```
./bin/ycsb run MongoDB -s -P workloads/workloadb -p operationcount=50000000 -threads 64 -p MongoDB.url="MongoDB://ip_address_of_node:27017/test?w=1" -p readallfields=true -p requestdistribution=zipfian
```

- **thread**: we ran the test four times by changing the thread setting with 8, 16, 32, and 64.
- **readallfields**: set to true to read all fields.
- **requestdistribution**: which distribution should be used to select the records to operate on – uniform, zipfian or the latest distribution (default: uniform). In our test, we choose zipfian as the distribution parameter. This setting means items in the database are selected according to the popularity irrespective of insertion time, which represents social media applications where popular users have many connections, regardless of the duration of their membership.

Besides the parameters above and connection string to MongoDB router (mongos), we used the default parameters of YCSB.

Durability Settings for the Database

In these tests, we define durability as the write having been successfully written to persistent storage (disk). MongoDB uses a write-ahead log (WAL) to record operations against the database. Writes are first committed to the WAL in memory, and then the WAL is written to disk to ensure durability. We tested two kinds of durability.

- **Balanced**—Client’s write acknowledged when written to WAL in RAM, WAL written to disk more frequently than data files. This approach allows the database to optimize writing the data files
to disk, while still ensuring durability. The window of possible data loss is a function of how frequently the WAL is flushed to disk. Journal is flushed to disk every 100MB, which means we may lose up to 100MB of data.

- Optimized Durability—The client waits for WAL or data to be written to disk. MongoDB flushes journal to disk, and there is no possible of any data loss.

**Write Concern**

Write concern describes the level of acknowledgment requested from MongoDB for write operations to a standalone mongod, daemon process for MongoDB, or to replica sets or to a sharded clusters. In sharded clusters, mongos instances will pass the write concern on to the shards. The write concern-parameters used in the test client (YCSB) are as follows:

- **w**: 1

Requests acknowledgment that the write operation has propagated to the standalone MongoDB or the primary in a replica set. w: 1 is the default write concern for MongoDB.

- **w**: majority

Requests acknowledgment that the write operations have propagated to the majority of voting nodes, including the primary. After the write operation returns with a w: "majority" acknowledgment to the client, the client can read the result of that write with a "majority" readConcern.

- **j**: true

Requests acknowledgment that the mongod instances, as specified in the w: <value>, have written to the on-disk journal. j: true does not by itself guarantee that the write will not be rolled back due to replica set primary failover.

For Balanced durability, we use w:1 to make sure write operation is acknowledged in the RAM, and we use w: majority to make sure write operation is acknowledged in the majority vote (primary and secondary in our designed replica set). In this setting, the worst-case data loss in case of server crash is up to 100MB.

For Optimized Durability, we use w:1 or w: majority plus j: true to make sure every write is hardened in the journal, so there is no loss data in the case of a server crash.

**5.2 vSAN Tools—vSAN Performance Service**

*vSAN performance service* is used to monitor the performance of the vSAN environment, using the web client. The performance service collects and analyzes performance statistics and displays the data in a graphical format. You can use the performance charts to manage your workload and determine the root cause of problems.
6. Solution Validation

In this section, we present the test methodologies and results used to validate this solution.
6.1 MongoDB Performance on vSAN

Test Overview

MongoDB performance test used MongoDB v3.4 running on Photon OS guest VMs or Kubernetes nodes, which were stressed by YCSB workload A and workload B. The four active nodes of the Kubernetes cluster were on the 4-node all-flash vSAN cluster with each host having one node.

We disabled checksum in the applied storage policy to the virtual disks for the database. Also, we changed the storage policy to FTT=0 for RAID0 test scenario or in the Configuration 3 from the default setting (FTT=1).

The 200GB MongoDB database was split into four shards, with each node having one shard. Every shard has configured shard. Every shard has configured mongos as routing service to process queries from clients.shard. Every shard has configured shard. Every shard has configured mongos as routing service to process queries from clients.

We set the threads of the YCSB with 8, 16, 32 and 64 on every test client to increase the workloads gradually. We reckon that the operation latency that is less than 10ms is ideal. Therefore, we summarized the test reports until 10ms-operation latency was reached.

Note: This performance validation does not aim at maximizing the MongoDB performance of each configuration on vSAN because the performance, such as operation per second and operation latency, is compute resources (CPU, Memory) and vSAN cache tier medium capability highly dependent. Nevertheless, we demonstrate the performance differences between the ideal response time of different configurations with different I/O pattern and durability. We also demonstrate the vSAN and vSphere cluster has no I/O subsystem bottleneck with these workloads.

Number of Test Clients

We did experiments to determine the optimized number of clients for the performance tests.

We configured multiple MongoDB routers. Each Kubernetes node had one router. This made it possible to query or update all data from one client or all four clients. To find out the optimized client number, we verified the performance differences by using a different number of clients to access all records (200 Million records in the 200GB database). One client will access the complete set of records, and two, three or four clients will access partial records of the whole data set. We did this by evenly distributing the records. For example, a 2-client test will use 100 million as the test records on each client to accomplish the total record number although the same record might be accessed more than once under Zipfian distribution mode of YCSB. We ran these tests using Configuration 2, Workload B with Balance durability, and the stress thread of YCSB was set to 64.

Figure 9 shows the performance results from one client up to four clients with same operation records.
From the results, we found that four clients could achieve the best performance of the cumulative ops/sec. As for the latency, four test clients kept the 99th read latency less than 10ms (around 8ms), although the update latency increased up to 6ms. Based on this result, we decided to use four clients for all the performance tests.

**Test Results**

For each of the durability configurations, we tested the 200GB database for the YCSB workload A and B. The results are detailed in this section.

**Balanced**

Most applications are best served by this balanced configuration where the throughput is good, while potential data loss is minimized. We measured the performance of Configuration 1 and Configuration 2. Figure 10 and Figure 11 show the average operations per second of the four nodes, and the totality of the four nodes; and the 99th latency of read and update of the operations.

We did not run a performance test on Configuration 3 because the possibility of data loss or data unavailability increases greatly without storage high availability protection and data durability if the virtual disks are from the same host. The performance validation of Configuration 3 is done in the Optimized Durability section below.

**Configuration 1 Test Results**

![Figure 11. Configuration 1 Workload A Test Results](image1)

In Configuration 1, with a configuration balanced for throughput and durability, the update heavy workload (50/50) in these tests demonstrates that MongoDB provided a near-linear increment of the average ops/sec and cumulative ops/sec until 32 threads. The 99th latency read and update with 32 threads had the maximum latency of 6 milliseconds (ms) for an update and 4.5 ms for a read.

![Figure 12. Configuration 1 Workload B Test Results](image2)
With a configuration balanced for throughput and durability, the read heavy workload (95% reads) shows MongoDB providing a near-linear increment of the average ops/sec and cumulative ops/sec until 64 threads. The 99th latency read and update with 64 threads had the maximum latency of 5.6 ms for an update and 7.7 ms for a read.

We monitored that the average vSAN backend latency for both read and write and it was less than 1ms and the IOPS ranged from 800 to 11,000 via vSAN performance service.

**Configuration 2 Test Results**

In Configuration 2, the 50/50 workload demonstrates that MongoDB also provided a near-linear increment of the average ops/sec and cumulative ops/sec till 32 threads. The 99th latency read and update with 32 threads had the maximum latency of 6.5 milliseconds (ms) for an update and 5.3 ms for a read.

We monitored that the average vSAN backend latency for both read and write was less than 1.7ms, and the IOPS ranged from 8,000 to 16,000 via vSAN performance service.

**Comparison between Configuration 1 and Configuration 2**

When comparing the two configurations, we found that for workload A, the average and cumulative ops/sec decreased less than 10 percent (ranging from 7.8 to 9.2 percent), and read and update latency increased less than 20 percent (ranging from 9.1 percent to 18.7 percent) from Configuration 1 to Configuration 2. For workload B, the average and cumulative ops/sec decreased less than 7 percent (ranging from 4 to 6.2 percent), and read and update latency increased less than 10 percent (ranging from 6.2 percent to 9.9 percent).
The comparison results demonstrated that replication has a performance impact in general; however, it is impaired less when running the read intensive workloads like workload B.

Optimized Durability

When data loss is more sensitive, this durability needs to be considered. Under this durability, writes are acknowledged after they are written to disk. There is no possible data loss. This configuration is suitable for applications where durability is more important than performance. We measured and compared the performance of Configuration 1, Configuration 2, and Configuration 3 with a fixed thread in test tool YCSB. We set 10 ms as an ideal latency for the workload. With this threshold, we tested 8 threads on workload A and 16 threads on workload B, starting from Configuration 1 as the comparison baseline, because the larger thread number will incur the latency above 10 ms.

**Workload A with 8 Threads Test Results**

![Workload A with 8 Threads Test Results](image)

With a configuration optimized for durability, the update heavy workload (50/50) in these tests demonstrates that the Configuration 1 provided the best performance with the highest average (3,505) and cumulative ops/sec (14,012). The 99th latency read and update in Configuration 1 had the lowest value for read (1.1 ms) and update (9.9 ms) operations.

We monitored that the average vSAN backend latency for both read and write was less than 1ms and the IOPS ranged from 10,000 to 17,000 via vSAN performance service on all three configurations.

**Workload B with 16 Threads Test Results**

![Workload B with 16 Threads Test Results](image)

With a configuration optimized for durability, the read heavy workload (95% reads) shows Configuration 1 also providing the best performance with the highest average (17,090) and cumulative (68,362) ops/sec. And the lowest latency for read (1.9 ms) and update (7.4 ms) operations.

We monitored that the average vSAN backend latency for both read and write was less than 1ms and the IOPS ranged from 200 to 9,500 via vSAN performance service on all three configurations.
Comparison between Configuration 1, Configuration 2, and Configuration 3

When comparing Configuration 2 with Configuration 1 for workload A, the average and cumulative ops/sec decreased around 13.5 percent, but the read and update latency increased about 18 and 31 percent respectively. The average and cumulative ops/sec of Configuration 3 decreased around 29 percent but the downgrading of read and update latency was less than 10 percent and 13 percent respectively.

When comparing Configuration 2 with Configuration 1 for the read-intensive workload B, the average and cumulative ops/sec decreased around 8.1 percent, and the read and update latency increased about 9.2 and 19.4 percent respectively. When comparing Configuration 2 with Configuration 3 for the read-intensive workload B, the average and cumulative ops/sec decreased around 14.3 percent, and the read and update latency increased about 2.8 and 62.9 percent respectively.

The comparison demonstrated that the application-level high protection mode had a higher cost of ops/sec and latency comparing with vSAN level protection.

6.2 Failure Validation

This section tested and recorded the failovers of the primary in a MongoDB replica set, node shutdown in the Kubernetes cluster, and one ESXi host powering off.

Primary Failure in Replica Set

When the primary is unavailable unexpectedly such as node shutting down, the secondary of MongoDB replica set will change to primary in a few seconds. This failure does not impact the data integrity during the failure time.

Node VM Failure

Node VM corruption will cause Kubernetes to recreate a new pod to run the containers. vSphere Cloud Provider will mount the disk to a live node and unmount disk from the dead node automatically. The validation description is as follows:

1. Shutdown one Kubernetes node VM. This will cause Kubernetes to remove the node VM from the Kubernetes cluster.
2. The Kubernetes cluster will recreate the pod on an idle node in the original cluster after the simulated node failure. Kubernetes vSphere Cloud Provider will:
   - Mount the disks from the shutdown node VM to the idle node.
   - Unmount the disks from the powered off node VM.
3. Fix the issue of the node VM (if any) and power it on. Kubernetes will add the node back to the original cluster and it will be available for new pod creation.

Physical Host Failure

Powering off one of the ESXi hosts will cause the vSphere Availability to restart the node on one of the running ESXi servers. The node in Kubernetes cluster will temporarily change to UNKNOWN. After less than two minutes, the node will be available in the cluster. No pod recreation is required.

Recovery Time Comparison

Table 5 compares the recovery time of different components of the solution.

Table 5. Recovery Time Comparison
<table>
<thead>
<tr>
<th>FAILURE COMPONENTS</th>
<th>RECOVERY TIME</th>
<th>RESULTS AND BEHAVIOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary fails in a replica set</td>
<td>&lt;15 seconds</td>
<td>MongoDB role fails over. The secondary is unavailable until the previous primary available.</td>
</tr>
<tr>
<td>Kubernetes node shutdown or corruption</td>
<td>&lt; 1 minute</td>
<td>The pod is recreated, and Kubernetes vSphere Cloud Provider automatically unmounts/mounts the disk to the standby node. The standby node changes to an active or working node.</td>
</tr>
<tr>
<td>ESXi host failure or powered off</td>
<td>&lt;2 minutes</td>
<td>vSphere Availability restarted the Kubernetes node on another ESXi host; no node recreation required.</td>
</tr>
</tbody>
</table>

6.3 Failback and Reconfiguration

We suggest performing failback after the previous primary is available again, either the node comes back or the failed ESXi can join the vSphere and vSAN cluster to utilize the compute and storage resource.

After the failover of the replica set, use the following steps to restore the primary role in a replica set.

**Change Back the Primary Role**

1. In the mongo shell, use rs.conf() to retrieve the replica set configuration and assign it to a variable. For example:
   ```
   cfg = rs.conf()
   ```
2. Change each member’s members[n].priority value, as configured in the member’s array:
   ```
   cfg.members[0].priority = 2
   cfg.members[1].priority = 0.5
   ```
3. Use rs.reconfig() to apply the new configuration:
   ```
   rs.reconfig(cfg)
   ```

After a few seconds, the primary will be restored to the original role.

**Reapply the vSAN Storage Policy**

1. After the shard failed to the new node, make sure the storage policy on the new node is the same as the previous node.
2. If the policy is different, go to vCenter
7. Best Practices

This section provides the recommended best practices for this solution.
7.1 Best Practices

When configuring MongoDB in containers in a Kubernetes cluster, see the following best practices:

- vSphere: Follow [Getting Started on vSphere](#) to deploy Kubernetes cluster on vSphere 6.5.
- Kubernetes cluster scale: Create 2 x node number so that every working node has a standby node in case the working node fails, allowing the backup node to host the pods as soon as possible.
- vSAN Storage Policy-Based Management (SPBM): Disable checksum of vSAN if 10% latency increase cannot be tolerated. The reason is when comparing performance with checksum enabled/disabled, the performance difference (ops/sec or latency) is around ten percent for an intensive update workload like YCSB workload A.
8. Conclusion

This section provides a summary of this reference architecture and validates vSAN as persistent storage for MongoDB in containers.
8.1 Conclusion

vSphere Cloud Provider enables access to vSphere managed storage (vSAN, VMFS, NFS) for applications deployed in Kubernetes. To simplify deployment further, for applications with replicas, consider using StatefulSets to set up all claims for every replica of a container. vSphere Cloud Provider manages persistent volumes and supports dynamic volume provisioning. It interacts with vCenter to support various operations such as creation and deletion of volumes, attaching and detaching volumes to application pods and nodes.

VMware vSAN combined with Kubernetes and MongoDB replication provides the best-fit protection from an application, node and physical host levels.

From a scalability perspective, consider multi-server deployments to address high availability and scale-out for one database. With this design methodology, we can design multiple hosts to support very large database running on one vSphere and vSAN cluster to achieve near-linear performance increments.

From MongoDB performance and availability perspective, VMware vSAN provides a better choice for container based MongoDB services compared to application level replication. For best performance design purposes, we recommend using sharding without secondary replica set and set FTT=1. For best protection design purposes, we recommend using sharding with secondary and arbiter while setting FTT=1.
9. About the Author

This section provides a brief background on the authors of this solution guide.
9.1 About the Author

Tony Wu, Senior Solution Architect in the Product Enablement team of the Storage and Availability Business Unit, wrote the original version of this paper.

Miao Luo, Senior Member of Technical Staff in Cloud Native Application Storage team of the Storage and Availability Business Unit, contributed part of this paper.

Catherine Xu, Senior Technical Writer in the Product Enablement team of the Storage and Availability Business Unit, edited this paper to ensure that the contents conform to the VMware writing style.

Ellen Herrick, Technical Writer in the Product Enablement team of the Storage and Availability Business Unit, edited this paper to ensure that the contents conform to the VMware writing style.