Optimize NVMe Drive Utilization for NCOS/System Temporary Tablespaces using OpenEBS LVM

**Authors:** [@Labi Adekoya](mailto:Owolabi.Adekoya@ibm.com)**,** [@Aruna de Silva,](mailto:adesilva@ca.ibm.com) **&** [@Hamdi Roumani](mailto:roumani@ca.ibm.com)

# Table of Contents:

1. Introduction to Db2 Native Cloud Object Storage (NCOS)
   1. Why using Non-Volatile Memory Express (NVMe) devices for caching tier is critical
2. Leveraging Fast Storage for Temporary Tablespaces
3. Introduction to OpenEBS Framework
4. Fast Local Storage for Db2: NVMe + LVM Approach
5. Implementing VG/LVM Strategies with OpenEBS
6. Deploying Db2 Warehouse (Db2Wh) with NCOS using OpenEBS on AWS EKS
7. How to Configure VG+LVM for NCOS Use Cases
8. Conclusion

# Executive Summary

One of the recent enhancements to the storage architecture of Db2 Warehouse is the introduction of a local on-disk caching layer, designed to enhance the performance of Cloud Object Storage (COS). In this study, we explored the use of locally attached Non-Volatile Memory Express (NVMe) devices for Db2 temporary tablespaces (TEMPTS) and as a caching tier for Db2 Native Cloud Object Storage (NCOS) feature.

In addition to NVMe devices, we utilized both Block and File Storage solutions-specifically Amazon Elastic Block Store (EBS) and Amazon Elastic File System (EFS)-and integrated [OpenEBS](https://openebs.io/) as the Container Storage Interface (CSI) provider within our AWS Elastic Kubernetes Service (EKS) environment. The growing adoption of locally attached NVMe or solid-state drives (SSDs) has significantly enhanced I/O performance while reducing costs. This trend has led to widespread availability of high-performance compute and fast local storage across cloud platforms and on-premises data centers. These drives deliver superior performance by residing near the compute layer, effectively eliminating many of the bottlenecks associated with traditional storage architectures.

For Db2, the use of locally attached NVMe storage has been adopted in two primary ways: as a high-speed caching layer and as fast temporary storage- or in some cases, both simultaneously. In this blog, we explore various configuration options for leveraging NVMe drives in Db2 to support these use cases. Specifically, we demonstrate how NVMe can be used as a caching tier for the NCOS feature, as well as TEMPTS. To complement these configurations, the Db2 instance was also configured to use both block and file storage for persistent data, ensuring a balanced approach to performance and durability.

To evaluate this setup, we provisioned a 5-node EKS cluster and deployed Db2 Warehouse MPP to investigate strategies for aggregating and managing local NVMe devices using Logical Volume Management (LVM) and Volume Groups (VGs). We examined four distinct approaches for configuring storage for caching and TEMPTS data:

1. Using separate VGs for each storage type.
2. Sharing a single VG across both storage types.
3. Creating VGs from individual NVMe devices.
4. Enabling thin provisioning to optimize space utilization.

Finally, we provided documented step-by-step process for implementing these strategies. A secondary finding of this study is that OpenEBS can be used as the CSI driver for all the storage types required to stand up Db2 engine.

# Leveraging Fast Storage for Db2’s NCOS feature

Db2 Warehouse has introduced a major enhancement that delivers both performance and cost efficiency: support for Cloud Object Storage (COS) as the primary storage medium for column-organized tables. COS offers significantly lower storage costs than traditional block storage, making it ideal for cloud environments. To overcome the higher latency of COS, Db2 Warehouse requires an on-disk caching tier backed by local NVMe drives—that complements the existing in-memory buffer pools, which continue to operate as they always have, caching frequently accessed data in memory.  
  
This two-tier caching architecture allows Db2 to manage larger working sets and deliver fast, consistent query performance, even when data is stored remotely in COS. Under the hood, this was made possible by adopting a Log-Structured Merge (LSM) tree-based storage organization, which is optimized for high-throughput writes and efficient reads from slower storage. Additionally, intelligent data clustering further boosts performance by improving data locality and reducing I/O costs—key reasons why the NCOS architecture delivers up to 4x faster query performance compared to traditional block storage.  
  
For more technical details and deeper dives into this transformation, check out:

* [Why is Db2 Warehouse Faster with Cloud Object Storage?](https://www.idug.org/news/why-is-db2-warehouse-faster-with-cloud-object-storage)
* [Monitoring the New Multi-Tier Storage Layer of Db2 Warehouse](https://www.idug.org/news/monitoring-the-new-multi-tier-storage-layer-of-db2-warehouse)

## Why using NVMe devices for caching tier is critical

Db2 Warehouse 11.5.9 introduced a new [multi-tiered storage architecture](https://www.idug.org/news/why-is-db2-warehouse-faster-with-cloud-object-storage) which includes Cloud Object Storage (COS) and Local NVMe Cache in addition to the existing network-attached storage to mitigate the performance limitations of NCOS in relation to throughput and latency. COS is limited by network bandwidth available to compute nodes and higher latency per request and these limitations posed significant implications on I/O operations. However, with the new layered storage architecture with locally attached NVMe drives, query workload performance, for example, was 4X faster than the traditional network attached storage architecture. The introduction of high-performance NVMe drives as local on-disk caching storage addresses the limitations with COS in the following ways:

1. Local cache allows the persistence of larger working set, leading to significant lower latency cost for I/O operations
2. Cache warm-up improves throughput utilization of COS and maintenance of the working set
3. Fast local cache space improves query performance and speeds up the formation of large blocks for writes and ingest performance

Consequently, using NVMe devices is critical to take advantage of the performance improvements in the new generation of multi-tiered storage architecture of the Db2 warehouse engine.

# Leveraging Fast Storage for Temporary Table Spaces

Similarly, leveraging fast local storage for Db2 System Temporary Tablespaces can lead to significant performance gains. These tablespaces are heavily used for operations like sorting large result sets — for example, during ORDER BY or GROUP BY — that exceed memory thresholds and spill to disk. They are also critical for internal Db2 mechanisms such as table queues, which facilitate data movement between nodes and can similarly require disk spill. Unlike regular user tablespaces, which must reside on durable storage to prevent data loss and minimize recovery reliance on backups, system temp tablespaces are ephemeral by design. Since they’re automatically recreated after a restart and don't persist data long-term, they are ideal candidates for high-speed local storage — such as non-RAID NVMe — where redundancy is not a strict requirement.

# Introduction to OpenEBS Framework

[OpenEBS](https://openebs.io/docs/) is the leading framework for NVMe-based storage deployments. It enables access to fast, highly durable, reliable and scalable container native storage available to stateful workloads by converting available storage on Kubernetes worker nodes to Local or Replicated Persistent Volumes. It uses containers to dynamically provision these volumes and provide data services leveraging Kubernetes as the orchestrating engine. For Local Volumes, OpenEBS creates Kubernetes Persistent Volumes (PVs) with locally attached storage (including NVMe devices) over existing Logical Volume Management (LVM). On the other hand, for Replicated Volumes, OpenEBS creates Persistent Volumes as NVMe accessible targets via TCP connections.

Local Volumes can only be accessed from a single node in a Kubernetes cluster and as such Pods using local volume must be scheduled on the same node where the volume is provisioned. This makes topology management very important for workloads assignment for Persistent Volume Claims (PVCs) scheduling. Node locality and strict topology awareness help reduce latency with low overhead on capacity and performance for distributed workloads using Local Volumes with inherent high availability.

Replicated Volumes, on the other hand, can be accessed from multiple Kubernetes worker nodes since data is synchronously replicated to multiple nodes. This makes Replicated Volumes more suitable for stateful workloads that require durability and performance even with overhead impact.

Based on Db2 requirements-particularly the need for low latency and optimized workload throughput-and its inherent design philosophy, which includes built-in availability, this study implemented Persistent Volumes as Local NVMe Volumes layered over existing LVM. These volumes were used to persist both the caching tier and temporary tablespace data, ensuring high-speed access and reliable storage.

# The case for using LVM over NVMe for fast local storage for Db2

Db2’s NCOS cache and temporary tablespace (TEMPTS) storage demand both high-speed disk I/O and efficient disk management to support scalability, flexibility, and performance. Meeting these requirements calls for a storage solution that combines speed-delivered by NVMe drives-with the flexible volume management capabilities of LVM. This makes the combination of NVMe and LVM a compelling choice for powering Db2’s caching temporary storage layers.

## Logical Volume Management (LVM) Strategies

LVM allows for the aggregation of storage volumes across multiple physical hard disks by creating a layer of abstraction over the physical storage. LVM increases disk I/O by aggregating disk space as logical volumes. These volumes provide flexible and efficient mechanisms for managing disk space including capacity scaling, isolation and reallocation such that conflicting storage use-cases are easier to orchestrate.

One common use case of LVM is the management of storage for different competing applications – making LVM suitable for the allocation of storage for caching tier and tempts in Db2 engine. LVM works by creating Physical Volumes (PVs) from physical disk devices and then using the PVs to create Volume Groups (VGs) from which LVs can then be allocated for different use cases.

Leveraging LVM for caching tier and tempts tablespaces in Db2, this study validated the following volume strategies:

* Separate VG per storage
* Single VG (from one or more PVs) for both cache and tempts
* Multiple partitioned VGs from a single PV
* Thin-provisioned VGs

### Separate VGs per storage

Here, we create separate PVs for caching tier and tempts data and then create corresponding VGs from which LVs are allocated. Following the creation of the VGs, StorageClasses (SCs) are created with the VG names referenced as volume parameters as well as allowed topologies key/value labels. Here’s a typical example of the SC created for caching tier data:

kubectl apply -f - <<EOF

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: openebs-db2ucaching-lvm

allowVolumeExpansion: true

provisioner: local.csi.openebs.io

parameters:

storage: "lvm"

volgroup: "db2ucaching\_vg"

volumeBindingMode: WaitForFirstConsumer

allowedTopologies:

- matchLabelExpressions:

- key: openebs.io/lvmvg-caching

values:

- db2ucaching\_vg

EOF

The name of the volume group created is **db2ucaching\_vg** and the same name is specified in the **allowedTopologies** parameter value. The same key-value specified as **allowedTopologies** is then used to label the worker nodes on which the db2 workloads would run. For example, nodes are labeled as shown below:

kubectl label node <node\_name> openebs.io/lvmvg-caching/db2ucaching\_vg

Detailed deployment details are provided in the NCOS use cases section.

### Single VG for caching and tempts storage

In this case, a single VG is created from one or more PVs (the PVs are created on all the participating worker nodes) and that VG is then used to provision logical volumes for caching tier and tempts storage. This essentially means that both caching and tempts storage are sharing the same underlying VG for persisting data. Logical volume provisioning is handled with corresponding SCs like that defined for the Separate VGs deployment strategy. The size of each LV will depend on the PVC request size, i.e. capacity.

### Partitioned VGs from a Single PV

For partitioned VGs, we use a single NVMe disk and partitioned it to create corresponding PVs and VGs for the partitioned disks. Subsequently, we use the VGs as the storage layer from which local volumes are created for caching tier and tempts data in the db2 engine. This allows us to define the exact capacity split between caching tier and tempts compared to Single VG strategy.

### Local PV-LVM Thin Provisioning

With any of the previously mentioned provisioning strategies, we can create VGs and provision PVs with **thinProvision** parameter set to “yes” in the defined SCs. With thin provisioning enabled, OpenEBS CSI driver creates thin volume by picking appropriate node based on scheduling attributes including topology information (specified in the SC and in the plugin), matching VG name, available capacity as well as node labels. Thin provisioned volume will use disk storage only on demand for efficient space management. Here is an example of a SC with thinProvision set to yes:

kubectl apply -f - <<EOF

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: openebs-db2ucaching-lvm

allowVolumeExpansion: true

provisioner: local.csi.openebs.io

parameters:

storage: "lvm"

**thinProvision: "yes"**

volgroup: "db2ucaching\_vg"

volumeBindingMode: WaitForFirstConsumer

allowedTopologies:

- matchLabelExpressions:

- key: openebs.io/lvmvg-caching

values:

- db2ucaching\_vg

EOF

## Db2U NCOS – OpenEBS Data Engine Architecture Use cases

Leveraging OpenEBS data engine architecture, here we discuss the different LVM use cases we considered in this study in relation to OpenEBS data layers. The Db2U use case for OpenEBS Data Engine architecture shows only 2 OpenEBS engine layers for brevity as depicted in Figure 1 namely:

1. Volume Access Layer
2. Storage Layer

This is because the functionalities of the other two layers - *Volume Services Layer* and *Volume Data Layer* – are inherently handled by the device aggregation substrate (LVM in this case) in the Storage layer. The Volume Services Layer is responsible for providing logical volume as device or directory path targets. Db2U use case does not rely on LVM replication for performance reasons and, hence the Volume Data Layer usage is limited to creating a volume replica on top of the storage layer as a logical volume.

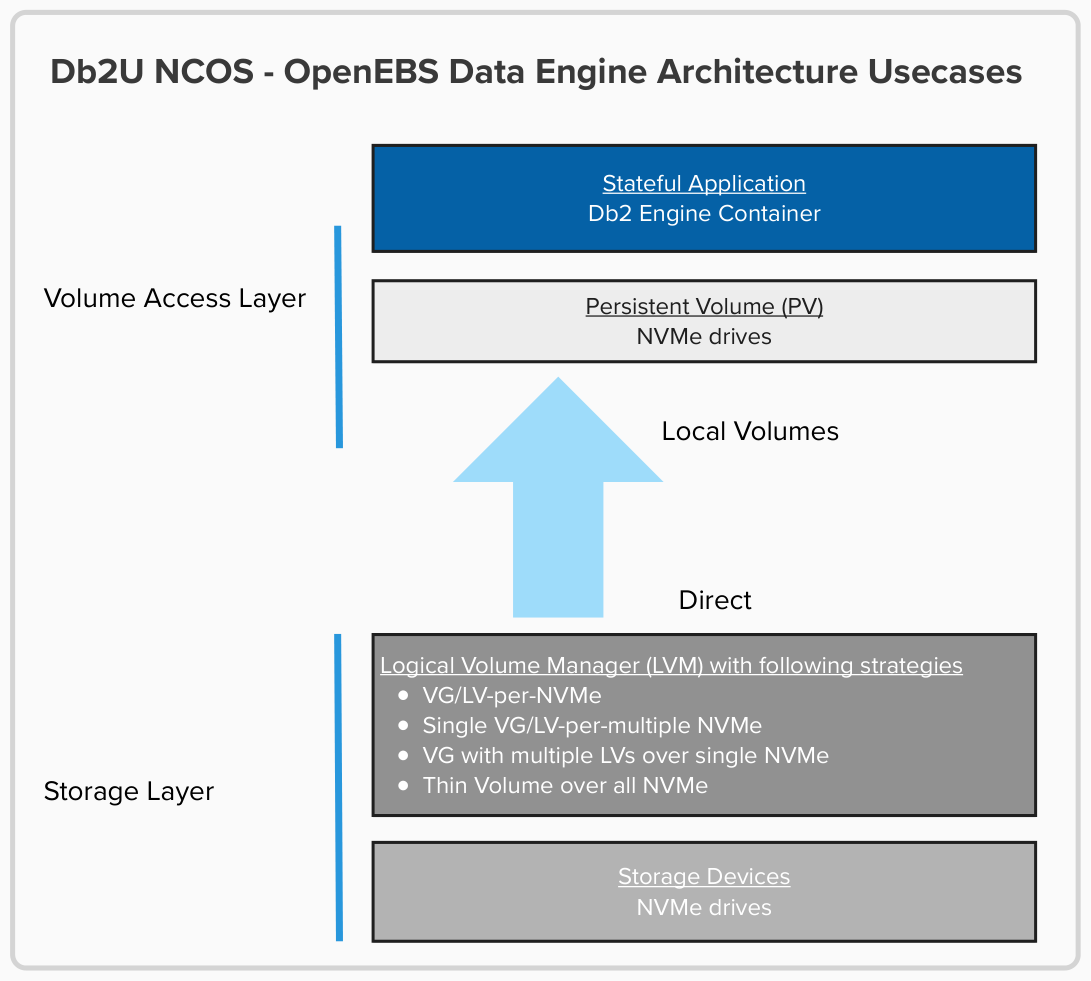


Fig 1: Db2U NCOS – Data Engine Architecture

### Db2U NCOS use case – OpenEBS Volume Access Layer

As shown above, this layer is responsible for creating, attaching and mounting local volumes and making the volumes available to Db2 Engine Container as Persistent Volume Claim (PVC) object. The Local Volumes (LV) are created via the corresponding Volume Group (VG) and Physical Volume (PV). The details for creating the LVs are defined in the Kubernetes Persistent Volume (PV) spec object including the SC type. A typical example of what the dynamically generated PV spec looks like is shown as below:

spec:

accessModes:

- ReadWriteOnce

capacity:

storage: 4Gi

claimRef:

apiVersion: v1

kind: PersistentVolumeClaim

name: csi-lvmpv

namespace: default

resourceVersion: "3767"

uid: c2289f5f-0050-431b-864a-fdb1000f181b

csi:

driver: local.csi.openebs.io

fsType: ext4

volumeAttributes:

openebs.io/cas-type: localpv-lvm

openebs.io/format-options: ""

openebs.io/volgroup: db2ucaching\_vg

storage.kubernetes.io/csiProvisionerIdentity: 1752587396891-7420-local.csi.openebs.io

volumeHandle: pvc-c2289f5f-0050-431b-864a-fdb1000f181b

nodeAffinity:

required:

nodeSelectorTerms:

- matchExpressions:

- key: openebs.io/lvmvg-caching

operator: In

values:

- db2ucaching\_vg

persistentVolumeReclaimPolicy: Delete

storageClassName: openebs-db2ucaching-lvm

volumeMode: Filesystem

In the spec above, the following CSI attributes are worth to reviewing:

* CSI Driver/Provider used (driver: local.csi.openebs.io): is set to OpenEBS driver
* File system type (fsType: ext4): applies when volumeMode: Filesystem, allowing the PV to be mounted into the container environment
* Container Attached Storage (CAS) type label (openebs.io/cas-type: localpv-lvm): ensures that the Persistent Volume is provisioned using local Persistent Volume via LVM
* The Volume Group (openebs.io/volgroup: db2ucaching\_vg): used by LVM to create Logical Volumes, in this case from db2ucaching\_vg VG
* The Node Affinity (nodeAffinity.nodeSelectorTerms): ensures the workload is scheduled to the expected nodes where the VG can be accessed

In the context of Db2U, the PV (with spec like that shown above and the PVC) required by the db2 engine is created dynamically by defining the storage spec in the Db2uInstance Custom Resource (CR) including the SC name and the storage size. Here’s an example of the storage definition for cachingtier in the db2u CR:

storage:

- name: cachingtier

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 850Gi

storageClassName: openebs-db2ucaching-lvm

type: template

With the SC specified for the storage, the corresponding PV & PVC are created dynamically by the OpenEBS CSI driver as defined in the SC.

### Db2U NCOS use case – OpenEBS Storage Layer

The Storage layer is the primary layer in the OpenEBS data plane architecture. It comprises of the block devices that are attached to the node either locally via NVMe or remotely. The Volume Access layer consumes the storage as a device, a device pool or a filesystem directory.

In this study, we used the locally attached NVMe devices on the cluster nodes as block devices. The NVMe devices are then consumed as aggregated LVM Volume Group pools as shown in Fig 1 on page 8. We used several LVM strategies to aggregate the NVMe devices as discussed in the Logical Volume Management (LVM) Strategies section. For example, say there’s a node with the following NVMe drives:  
  
lsblk | grep nvme

nvme1n1 259:0 0 3.4T 0 disk

nvme2n1 259:4 0 3.4T 0 disk

The corresponding Physical Volume for NVMe drive nvme1n1 will have the following characteristics for caching tier data:

--- Physical volume ---

PV Name /dev/nvme1n1

VG Name db2ucaching\_vg

PV Size 1.00 TiB / not usable 4.00 MiB

Allocatable yes

PE Size 4.00 MiB

Total PE 262143

Free PE 261119

Allocated PE 1024

PV UUID PdmqM1-5Vby-4S9h-xdLG-6z7j-ZpKN-P1556k

And the corresponding LVM pool with the VG name db2ucaching\_vg would look like the following:

--- Logical volume ---

LV Path /dev/db2ucaching\_vg/pvc-c2289f5f-0050-431b-864a-fdb1000f181b

LV Name pvc-c2289f5f-0050-431b-864a-fdb1000f181b

VG Name db2ucaching\_vg

LV UUID v8VzKN-0BTW-Y6Qh-356U-7XEV-Ktbd-HNWFUm

LV Write Access read/write

LV Creation host, time openebs-lvm-localpv-node-qwdx4, 2025-07-15 10:13:01 -0400

LV Status available

**# open 0**

LV Size 4.00 GiB

Current LE 1024

Segments 1

Allocation inherit

Read ahead sectors auto

-currently set to 256

Block device 252:0

Thereafter, the VG db2ucaching\_vg, for example is used to define the SC for cachingtier data as the volgroup parameter, as shown in the SC example on page 7.

# Implementing VG/LVM Strategies with OpenEBS

For this study, the VG/LVM strategies were implemented on Amazon Web Services (AWS) Elastic Kubernetes Service (EKS). We deployed Db2 Warehouse MPP on a 5-node EKS cluster with 2 Node Pool Types: Db2wh MPP node pool and OpenEBS LPV and System Pods node pool. Three nodes were dedicated for Db2wh MPP and the remaining 2 nodes for OpenEBS LPV and System pods. The specification for the node pools is described as follows:

Db2Wh Node Pool:

* Each node is of **r6id.12xlarge** instance type with 48vCPUs, 256GiB, and 2x1425 GB NVMe
* Hosted Db2wh pods including other control plane agents

OpenEBS and System Pods Node Pool:

* Node in the pool is of **i41.4xlarge** instance type with 16vCPU, 128GiB and 2x937GB NVMe
* Hosted OpenEBS components including other control plane agents

With the infrastructure established, we then proceeded to first deploy the OpenEBS components including implementing the VG/LVM strategies before Db2wh deployment.

## OpenEBS Deployment

We used OpenEBS as the CSI driver and as such we deployed it by following the documented [instructions](https://openebs.io/docs/quickstart-guide/installation) but with replicated storage disabled since we are only interested in local storage deployment:

helm install openebs --namespace openebs openebs/openebs --set lvm-localpv.lvmNode.kubeletDir=/var/lib/kubelet --set engines.replicated.mayastor.enabled=false --create-namespace

Subsequently, we validated that the deployment was successful with an output like below:  
  
$ k get po -n openebs | grep lvm-localpv

NAME READY STATUS RESTARTS AGE

openebs-localpv-provisioner-78cbfb94c8-4sprc 1/1 Running 0 123m

openebs-lvm-localpv-controller-6f5f57f77f-tdtjb 5/5 Running 0 123m

openebs-lvm-localpv-node-2b5lx 2/2 Running 0 80m

openebs-lvm-localpv-node-cmzqr 2/2 Running 0 80m

openebs-lvm-localpv-node-d2t9t 2/2 Running 0 80m

openebs-lvm-localpv-node-tstxm 2/2 Running 0 80m

openebs-lvm-localpv-node-zzp27 2/2 Running 0 80m

## Implementing VG/LVM Strategies

Here we briefly discuss how we implement the VG/LVM strategies used for persisting caching tier and temp tablespace (tempts) data. As mentioned earlier, we explored the following four strategies for aggregating NVMe devices as Volume Groups (VGs) and/or configuring them to provision volumes that support the storage layer in the Db2 Warehouse architecture.

1. Separate VG per storage
2. Single VG for both NCOS cache and tempts
3. Multiple partitioned VGs from a single PV
4. Thin-provisioned VGs

At a high level, the creation of VGs is a common step across all the strategies discussed. The differences lie in the number of VGs created and how they are configured. The VG creation process follows this sequence:

1. Identify the NVMe drives present in the participating cluster nodes. Here we check the NVMe drives on the nodes by listing them as follows:  
     
   lsblk | grep nvme

nvme1n1 259:0 0 3.4T 0 disk

nvme0n1 259:1 0 100G 0 disk

├─nvme0n1p1 259:2 0 100G 0 part /

└─nvme0n1p128 259:3 0 1M 0 part

nvme2n1 259:4 0 3.4T 0 disk

1. Create Physical Volumes (PV) for the drive(s) for respective storage type. After identifying the drives, we then create PVs for the drives. Using the sample output in 1 above, we create two VGs for both storage (Strategy 2) thus:  
     
     
   pvcreate /dev/nvme1n1

pvcreate /dev/nvme2n1

1. Create corresponding VGs for the created PVs. With the PVs in place, we create the corresponding VGs for the storage following the example output in 2:  
     
   vgcreate db2ucaching\_vg /dev/nvme1n1

vgcreate db2usystemp\_vg /dev/nvme2n1

Here is an example of creating distinct VGs for caching tier (db2ucaching\_vg) and tempts (db2usystemp\_vg).

1. Configure StorageClass(es) to use the created VG(s) for provisioning Persistent Volume(s) that can be claimed by Persistent Volume Claim(s).   
   To make the VGs available for volume provisioning, we configured SCs with the VGs as shown in the following output:

**db2ucaching**  
kubectl apply -f - <<EOF

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: openebs-db2ucaching-lvm

allowVolumeExpansion: true

provisioner: local.csi.openebs.io

parameters:

storage: "lvm"

volgroup: "db2ucaching\_vg"

volumeBindingMode: WaitForFirstConsumer

allowedTopologies:

- matchLabelExpressions:

- key: openebs.io/lvmvg-caching

values:

- db2ucaching\_vg

EOF  
  
**db2usystemp**

kubectl apply -f - <<EOF

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: openebs-db2usystemp-lvm

allowVolumeExpansion: true

provisioner: local.csi.openebs.io

parameters:

storage: "lvm"

volgroup: "db2usystemp\_vg"

volumeBindingMode: WaitForFirstConsumer

allowedTopologies:

- matchLabelExpressions:

- key: openebs.io/lvmvg-temp

values:

- db2usystemp\_vg

EOF

# Deploying Db2 Warehouse (Db2Wh) with NCOS using OpenEBS on AWS EKS

With the LVM strategies in place, we now discuss the entire architecture for deploying Db2Wh with NCOS using OpenEBS on AWS EKS with emphasis on the different storage types required by Db2 engine. This section will also show how the different storage types are defined in the Db2uInstance Custom Resource (CR) object as well as how to use the CR to deploy Db2Wh accordingly.

## Db2 Warehouse with NCOS using OpenEBS on AWS EKS Architecture

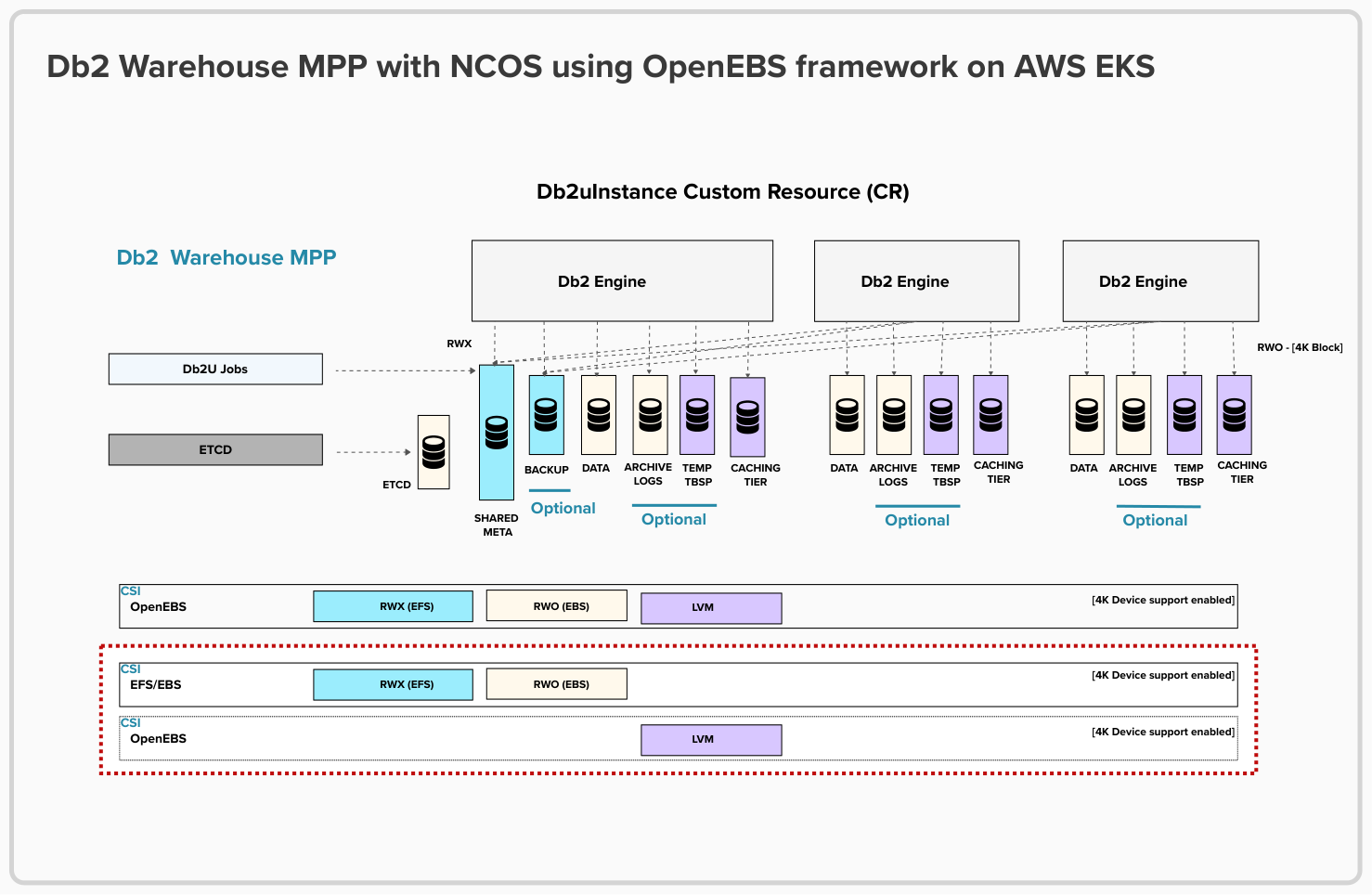
Db2Wh MPP comprises of many functional components including the Db2 engine but for the purpose of this study, the architectural discussion will focus briefly on the storage layer vis-a-vis Native Cloud Object Storage on AWS EKS. Db2 engine core is designed to support the following storage types named to reflect their functionality:

* *Archive Logs* – Db2 archive logs are stored here
* *Backup –* landing area for Db2u backup to disk
* *Caching Tier* – used as a fast local cache for NCOS feature
* *Data* – used as storage for Db2 database paths
* *ETCD* – ETCD is used by Db2U build-in HA service for state information
* *Shared Meta* – Db2 instance home and other key shared attributes are stored here
* *Temp Tablespace* -- Db2 System Temporary tablespace storage

As shown in Figure 2 on page 15, storage types including archive logs, backup and temp tablespace are optional but highly recommended while data, shared meta and caching tier are required. In the context of NCOS on AWS EKS, storage types requiring volume access mode of ReadWriteMany (RWX) including backup and shared meta use Amazon Elastic File System (EFS) as the Container Storage Interface (CSI) while those requiring volume access mode of ReadWriteOnce (RWO) including data, etcd and archive logs use Amazon Elastic Block Storage (EBS) as the CSI.

On the other hand, OpenEBS (with LVM-provided VGs) acts as the CSI for caching tier and temp tablespace storage types in resonance with this study. It is important to mention, however, that OpenEBS can be used as the CSI for all the storage types and not just for caching tier and temp tablespace as depicted in Fig 2.

Subsequently, the engine can then be deployed by defining the specifications for all the storage types in the Db2uInstance Custom Resource (CR). The deployment of Db2Wh with the CR is the focus of the following section.

 Fig 2: Db2 Warehouse MPP with NCOS on AWS EKS

## Deploying Db2 Warehouse MPP with Db2uInstance Custom Resource

Now that we are clear on the NCOS definition for Db2Wh on EKS and on how OpenEBS will provide the LVM aggregation for local on-disk caching tier and system temp tablespace data storage, we deploy Db2Wh with the Db2uInstance CR. Db2uInstance is a Kubernetes Custom Resource (CR) object that provides a declarative way of deploying the different components required for standing up the Db2 engine with a [YAML](https://www.ibm.com/think/topics/yaml#:~:text=YAML%20is%20a%20versatile%2C%20human,humans%20and%20interpretable%20by%20machines.) configuration file. Following is an example of a YAML definition we used for creating a Db2uInstance CR object:

apiVersion: db2u.databases.ibm.com/v1

kind: Db2uInstance

metadata:

name: nvme-db2whmpp

namespace: db2u

spec:

version: s12.1.2.0

nodes: 4

advOpts:

cosProvider: "aws"

enableCos: "true"

podTemplate:

db2u:

resource:

db2u:

limits:

cpu: "12"

memory: "112Gi"

environment:

dbType: db2wh

databases:

- name: BLUDB

partitionConfig:

total: 24

volumePerPartition: true

authentication:

ldap:

enabled: true

license:

accept: true

storage:

- name: meta

spec:

accessModes:

- ReadWriteMany

resources:

requests:

storage: 500Gi

storageClassName: efs-terraform-sc

type: create

- name: data

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 200Gi

storageClassName: ebs-terraform-sc

type: template

- name: backup

spec:

accessModes:

- ReadWriteMany

resources:

requests:

storage: 200Gi

storageClassName: efs-terraform-sc

type: create

- name: tempts

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 300Gi

storageClassName: openebs-db2usystemp-lvm

type: template

- name: etcd

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 10Gi

storageClassName: ebs-terraform-sc

type: template

- name: archivelogs

spec:

accessModes:

- ReadWriteMany

resources:

requests:

storage: 500Gi

storageClassName: efs-terraform-sc

type: create

- name: cachingtier

type: template

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 850Gi

storageClassName: openebs-db2ucaching-lvm

A careful look at the CR defined above reveals a storage CSI specification that combines the EFS, EBS and OpenEBS as indicated with the doted rectangular shape in Fig 2 on page 15. A successful deployment shows all the Db2 Pods in ready and running states as shown in the sample output below:

NAME READY STATUS RESTARTS AGE

c-nvme-db2whmpp-db2u-0 1/1 Running 0 25h

c-nvme-db2whmpp-db2u-1 1/1 Running 0 25h

c-nvme-db2whmpp-db2u-2 1/1 Running 0 25h

c-nvme-db2whmpp-etcd-0 1/1 Running 0 25h

c-nvme-db2whmpp-etcd-1 1/1 Running 0 25h

c-nvme-db2whmpp-etcd-2 1/1 Running 0 25h

c-nvme-db2whmpp-ldap-988d687b5-nkg8f 1/1 Running 0 25h

c-nvme-db2whmpp-restore-morph-jmzlw 0/1 Completed 0 25h

c-nvme-db2whmpp-tools-864d49886-76lwd 1/1 Running 0 25h

db2u-day2-ops-controller-manager-6dff47f8f8-q8k98 1/1 Running 0 27h

db2u-operator-manager-649fcd9b77-xnhf6 1/1 Running 0 27h

A detailed step by step guide of the different deployment scenarios considered in this study is the focus of the next section.

# How to configure VG+LVM for Db2’s NCOS Use Cases

In this section, we present a step-by-step guide to replicating various VG/LVM strategies tailored to different Db2 NCOS use cases. We begin by demonstrating how to create separate VGs for caching tier and temporary tablespaces (tempts). Next, we show how a single VG can be configured to support both storage types. We then explore the approach of partitioning VGs from a single Physical Volume (PV), followed by a walkthrough of enabling thin provisioning to optimize space utilization. Each strategy is designed to offer flexibility depending on specific workload and node requirements.

## Prerequisites for all use cases

Before proceeding to implementing any of the use cases discussed below, ensure that the following steps are taken to install helm (if it is not running already) and, setup and install OpenEBS.

1. Install [helm](https://helm.sh/docs/intro/install/)
2. Setup [OpenEBS](https://openebs.io/docs/quickstart-guide/installation#installation-via-helm) helm repo
3. Validate that OpenEBS installation is successful by checking that OpenEBS is running and in ready state
4. Check that Db2 operator is installed and is in ready state. If not already installed, refer to the [official documentation](https://www.ibm.com/docs/en/db2-warehouse?topic=warehouse-installing-db2-operator) on how to install Db2 operator. A validation output will be like the following output:  
     
   kubectl get clusterserviceversion

NAME DISPLAY VERSION REPLACES PHASE

db2u-operator.v120102.0.0 IBM Db2 120102.0.0 db2u-operator.v120101.0.0 Succeeded

## Use Case 1: Using distinct NVMe devices

Here we show how distinct NVMe devices can be used for caching tier and tempts on EKS.

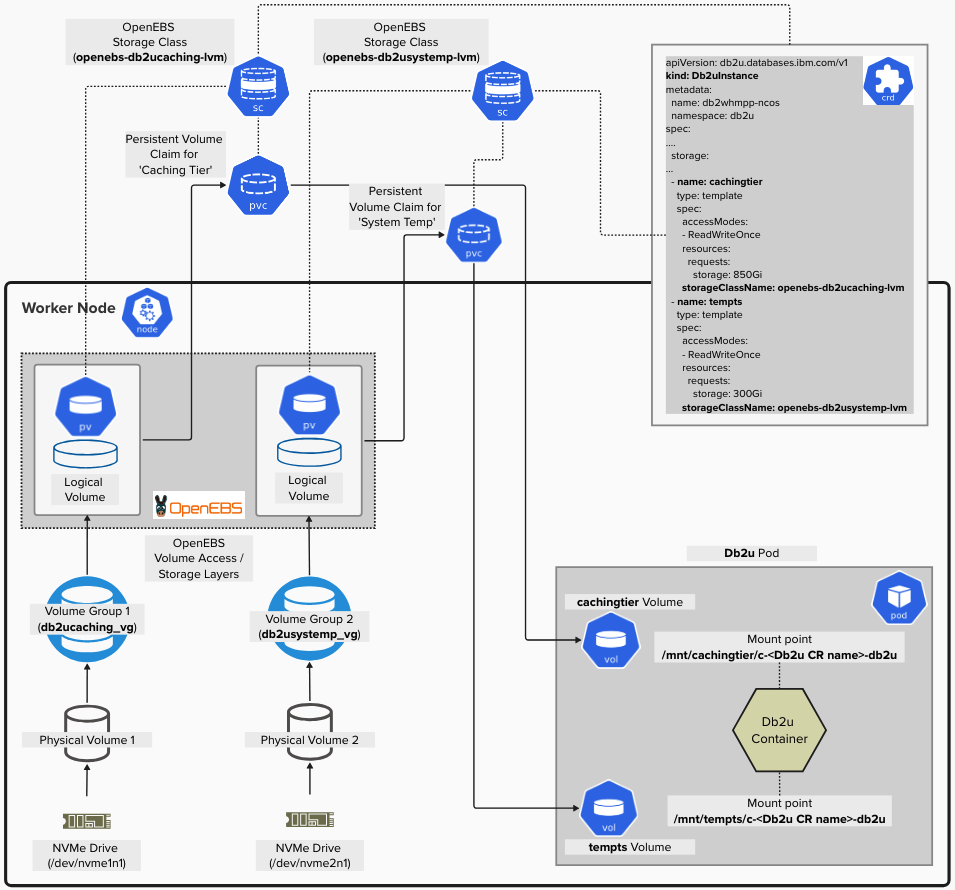


Fig 3: Using distinct NVMe devices with separate Volume Groups

1. Check NVMe devices are available on the cluster nodes. As stated previously, this study is based on a 5-cluster node EKS set up and each node has two NVMe devices respectively. A sample output is shown thus:  
     
   lsblk | grep nvme

nvme1n1 259:0 0 3.4T 0 disk

nvme0n1 259:1 0 100G 0 disk

├─nvme0n1p1 259:2 0 100G 0 part /

└─nvme0n1p128 259:3 0 1M 0 part

nvme2n1 259:4 0 3.4T 0 disk

1. From the output in step 1, nvme1n1 and nvme2n1 devices are available. So, create Physical Volumes (PV) for them – one for caching tier and the other for tempts:  
     
   pvcreate /dev/nvme1n1

pvcreate /dev/nvme2n1

1. Next, create Volume Groups (VGs) for the PVs created previously:  
     
   vgcreate db2ucaching\_vg /dev/nvme1n1

vgcreate db2usystemp\_vg /dev/nvme2n1

1. Create corresponding SCs with the VGs specified in parameters and in allowedTopologies:  
     
   db2ucaching:  
     
   kubectl apply -f - <<EOF

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: openebs-db2ucaching-lvm

allowVolumeExpansion: true

provisioner: local.csi.openebs.io

**parameters:**

storage: "lvm"

**volgroup: "db2ucaching\_vg"**

volumeBindingMode: WaitForFirstConsumer

**allowedTopologies:**

**- matchLabelExpressions:**

**- key: openebs.io/lvmvg-caching**

**values:**

**- db2ucaching\_vg**

EOF  
  
db2usystemp  
  
kubectl apply -f - <<EOF

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: openebs-db2usystemp-lvm

allowVolumeExpansion: true

provisioner: local.csi.openebs.io

**parameters:**

storage: "lvm"

**volgroup: "db2usystemp\_vg"**

volumeBindingMode: WaitForFirstConsumer

**allowedTopologies:**

**- matchLabelExpressions:**

**- key: openebs.io/lvmvg-temp**

**values:**

**- db2usystemp\_vg**

EOF

1. Label nodes with the same key-value defined for the allowedTopologies in the SCs:

kubectl label node ip-10-0-108-141.us-east-2.compute.internal **openebs.io/lvmvg-caching=db2ucaching\_vg**

kubectl label node ip-10-0-108-141.us-east-2.compute.internal **openebs.io/lvmvg-temp=db2usystemp\_vg**  
  
  
Kubernetes only allows one value per label, so different label keys must be used for the VGs. And the labels must be applied on all relevant nodes.

1. The node labels are not read by default by OpenEBS LVM plugin. The plugin needs to be updated with the label keys. Therefore, edit the openebs-lvm-localpv-node DaemonSet by updating the ALLOWED\_TOPOLOGIES environment variable to include the label keys and wait for the DaemonSet to rollout completely on all the nodes.  
     
   kubectl edit ds openebs-lvm-localpv-node -n openebs  
     
   And replace the environment variable ALLOWED\_TOPOLOGIES with the following in the openebs-lvm-plugin container spec.  
     
   Replace this:

- name: ALLOWED\_TOPOLOGIES

value: kubernetes.io/hostname  
  
With the keys defined for allowedTopologies in the SCs. For example:  
  
- name: ALLOWED\_TOPOLOGIES

value: kubernetes.io/hostname,**openebs.io/lvmvg-caching,openebs.io/lvmvg-temp**

And then wait for the pods to rollout on all nodes

1. Confirm that the labels have been picked up by OpenEBS LVM plugin  
     
   kubectl logs -n openebs -l app=openebs-lvm-node -c openebs-lvm-plugin | grep accessible\_topology  
     
   A log entry such as that below confirms that the node labels are now updated  
     
   I0529 18:42:38.559328 1 grpc.go:81] GRPC response: {"accessible\_topology":{"segments":{"kubernetes.io/hostname":"ip-10-0-108-141.us-east-2.compute.internal","openebs.io/lvmvg-caching":"db2ucaching\_vg","openebs.io/lvmvg-temp":"db2usystemp\_vg","openebs.io/nodename":"ip-10-0-108-141.us-east-2.compute.internal"}},"node\_id":"ip-10-0-108-141.us-east-2.compute.internal"}
2. Deploy Db2Wh CR with the storage resources updated accordingly. In this study, we used the YAML definition previously mentioned on pages 16 & 17. The application PersistentVolumeClaims (PVCs) should bind successfully to the corresponding PersistentVolumes (PVs), and the pods start successfully. Example output shown below:  
     
   PVCs  
   kubectl get pvc | grep openebs

cachingtier-c-nvme-db2whmpp-db2u-0 Bound pvc-351c8546-8104-42a2-bf96-65ce6ad07952 850Gi RWO openebs-db2ucaching-lvm 25h

cachingtier-c-nvme-db2whmpp-db2u-1 Bound pvc-e6c71dd3-4b13-4d04-9b5e-ca9297325b46 850Gi RWO openebs-db2ucaching-lvm 25h

cachingtier-c-nvme-db2whmpp-db2u-2 Bound pvc-a6607c5e-45a2-4b77-925d-370c123571ec 850Gi RWO openebs-db2ucaching-lvm 25h

tempts-c-nvme-db2whmpp-db2u-0 Bound pvc-5a973e80-ffe8-47fe-8786-521f827971c0 300Gi RWO openebs-db2usystemp-lvm 25h

tempts-c-nvme-db2whmpp-db2u-1 Bound pvc-bc15cff7-095f-410d-965c-3aec7bf68b68 300Gi RWO openebs-db2usystemp-lvm 25h

tempts-c-nvme-db2whmpp-db2u-2 Bound pvc-bc62b0ea-0f58-4fb0-b485-d945042b6dbd 300Gi RWO openebs-db2usystemp-lvm 25h

PVs  
kubectl get pv | grep openebs

pvc-351c8546-8104-42a2-bf96-65ce6ad07952 850Gi RWO Delete Bound db2u/cachingtier-c-nvme-db2whmpp-db2u-0 openebs-db2ucaching-lvm 25h

pvc-5a973e80-ffe8-47fe-8786-521f827971c0 300Gi RWO Delete Bound db2u/tempts-c-nvme-db2whmpp-db2u-0 openebs-db2usystemp-lvm 25h

pvc-a6607c5e-45a2-4b77-925d-370c123571ec 850Gi RWO Delete Bound db2u/cachingtier-c-nvme-db2whmpp-db2u-2 openebs-db2ucaching-lvm 25h

pvc-bc15cff7-095f-410d-965c-3aec7bf68b68 300Gi RWO Delete Bound db2u/tempts-c-nvme-db2whmpp-db2u-1 openebs-db2usystemp-lvm 25h

pvc-bc62b0ea-0f58-4fb0-b485-d945042b6dbd 300Gi RWO Delete Bound db2u/tempts-c-nvme-db2whmpp-db2u-2 openebs-db2usystemp-lvm 25h

pvc-e6c71dd3-4b13-4d04-9b5e-ca9297325b46 850Gi RWO Delete Bound db2u/cachingtier-c-nvme-db2whmpp-db2u-1 openebs-db2ucaching-lvm 25h

Db2 Pods

kubectl get po

NAME READY STATUS RESTARTS AGE

c-nvme-db2whmpp-db2u-0 1/1 Running 0 25h

c-nvme-db2whmpp-db2u-1 1/1 Running 0 25h

c-nvme-db2whmpp-db2u-2 1/1 Running 0 25h

c-nvme-db2whmpp-etcd-0 1/1 Running 0 25h

c-nvme-db2whmpp-etcd-1 1/1 Running 0 25h

c-nvme-db2whmpp-etcd-2 1/1 Running 0 25h

c-nvme-db2whmpp-ldap-988d687b5-nkg8f 1/1 Running 0 25h

c-nvme-db2whmpp-restore-morph-jmzlw 0/1 Completed 0 25h

c-nvme-db2whmpp-tools-864d49886-76lwd 1/1 Running 0 25h

db2u-day2-ops-controller-manager-6dff47f8f8-q8k98 1/1 Running 0 27h

db2u-operator-manager-649fcd9b77-xnhf6 1/1 Running 0 27h

From the output of the PVC/PV, we can see that the volumes are provisioned with OpenEBS via the SCs defined previously, and that confirms that distinct VGs are used for caching tier and tempts data.

## Use Case 2: Using single Volume Group with one or more NVMe devices

The only difference between this use case and use case 1 above is that this shares a single Volume Group with one or more NVMe devices for caching tier and tempts data.

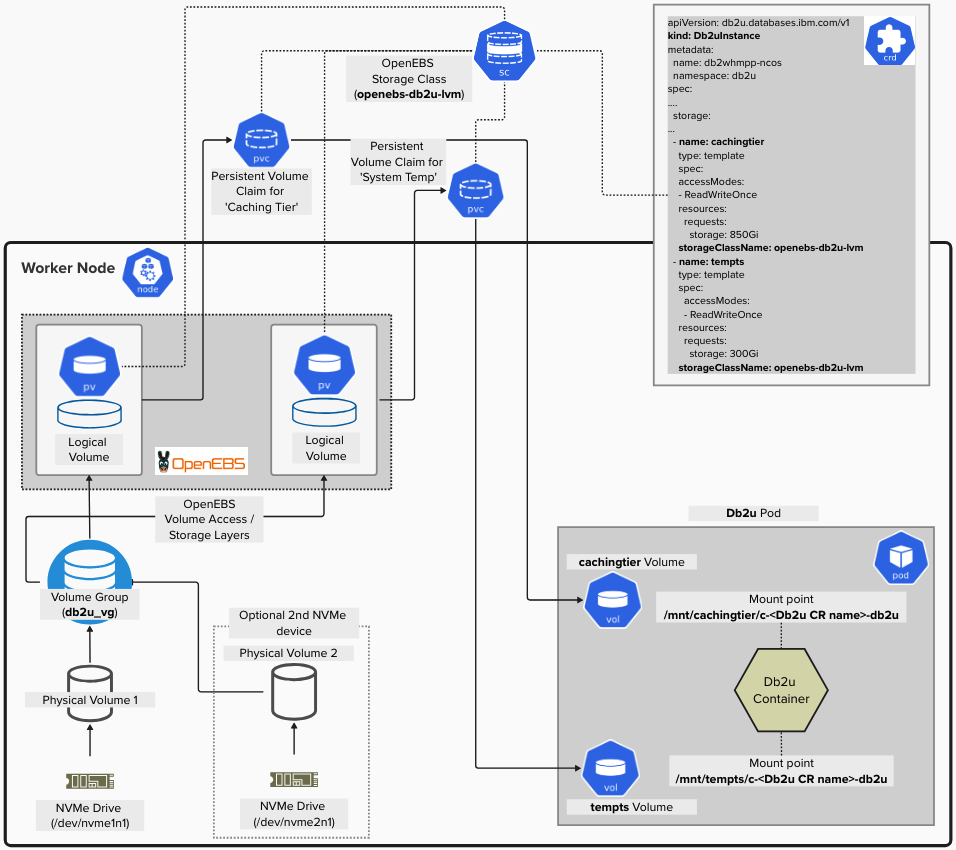


Fig 4: Using a single Volume Group with one or more NVMe devices

Here are the steps involved:

1. Check the NVMe devices available and choose one of them (if more than one). For example, using similar output as in use case 1:  
     
   lsblk | grep nvme

nvme1n1 259:0 0 3.4T 0 disk

nvme0n1 259:1 0 100G 0 disk

├─nvme0n1p1 259:2 0 100G 0 part /

└─nvme0n1p128 259:3 0 1M 0 part

nvme2n1 259:4 0 3.4T 0 disk

1. Here, since we have two devices per node, we used one (nvme1n1) of them to create a Physical Volume. For example:  
     
   pvcreate /dev/nvme1n1

If you want to consolidate the secondary NVMe device (nvme2n1) create Physical Volume for it as well

pvcreate /dev/nvme2n1

1. Create a VG for the PV(s) created in 2:  
     
   vgcreate db2u\_vg /dev/nvme1n1  
     
   If you are consolidating both Physical Volumes/NVMe devices into single storage pool, issue vgcreate command with both PVs

vgcreate db2u\_vg /dev/nvme1n1 /dev/nvme2n1

1. Create SC from the VG created previously:  
     
   db2u (we create a single SC here)  
   kubectl apply -f - <<EOF

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: openebs-db2u-lvm

allowVolumeExpansion: true

provisioner: local.csi.openebs.io

**parameters:**

storage: "lvm"

**volgroup: "db2u\_vg"**

volumeBindingMode: WaitForFirstConsumer

**allowedTopologies:**

**- matchLabelExpressions:**

**- key: openebs.io/lvmvg-db2u**

**values:**

**- db2u\_vg**

EOF

The key thing to note in the SC defined above is that the storage types will be sharing the same VG as configured in the SC.

1. Label nodes with the same key-value defined in the SCs   
     
   kubectl label node ip-10-0-108-141.us-east-2.compute.internal **openebs.io/lvmvg-db2u=db2u\_vg**
2. Refer to steps 6 and 7 of use case 1 to ensure that the node label is read by OpenEBS LVM plugin
3. Deploy Db2Wh CR with the storage configuration updated for caching tier and tempts. For this study, we used the same YAML defined in use case 1 but with the storage for tier and tempts updated to reference the SC created previously:  
     
   ...

- name: tempts

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 300Gi

**storageClassName: openebs-db2u-lvm**

type: template

- name: cachingtier

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 850Gi

**storageClassName: openebs-db2u-lvm**

type: template

...

Examples of sample data output:

PVCs

kubectl get pvc | grep openebs

cachingtier-c-nvme-db2whmpp-db2u-0 Bound pvc-351c8546-8104-42a2-bf96-65ce6ad07952 850Gi RWO openebs-db2u-lvm 17h

cachingtier-c-nvme-db2whmpp-db2u-1 Bound pvc-e6c71dd3-4b13-4d04-9b5e-ca9297325b46 850Gi RWO openebs-db2u-lvm 17h

cachingtier-c-nvme-db2whmpp-db2u-2 Bound pvc-a6607c5e-45a2-4b77-925d-370c123571ec 850Gi RWO openebs-db2u-lvm 17h

tempts-c-nvme-db2whmpp-db2u-0 Bound pvc-5a973e80-ffe8-47fe-8786-521f827971c0 300Gi RWO openebs-db2u-lvm 17h

tempts-c-nvme-db2whmpp-db2u-1 Bound pvc-bc15cff7-095f-410d-965c-3aec7bf68b68 300Gi RWO openebs-db2u-lvm 17h

tempts-c-nvme-db2whmpp-db2u-2 Bound pvc-bc62b0ea-0f58-4fb0-b485-d945042b6dbd 300Gi RWO openebs-db2u-lvm 17h

PVs

kubectl get pv | grep openebs

pvc-351c8546-8104-42a2-bf96-65ce6ad07952 850Gi RWO Delete Bound db2u/cachingtier-c-nvme-db2whmpp-db2u-0 openebs-db2u-lvm 17h

pvc-5a973e80-ffe8-47fe-8786-521f827971c0 300Gi RWO Delete Bound db2u/tempts-c-nvme-db2whmpp-db2u-0 openebs-db2u-lvm 17h

pvc-a6607c5e-45a2-4b77-925d-370c123571ec 850Gi RWO Delete Bound db2u/cachingtier-c-nvme-db2whmpp-db2u-2 openebs-db2u-lvm 17h

pvc-bc15cff7-095f-410d-965c-3aec7bf68b68 300Gi RWO Delete Bound db2u/tempts-c-nvme-db2whmpp-db2u-1 openebs-db2u-lvm 17h

pvc-bc62b0ea-0f58-4fb0-b485-d945042b6dbd 300Gi RWO Delete Bound db2u/tempts-c-nvme-db2whmpp-db2u-2 openebs-db2u-lvm 17h

pvc-e6c71dd3-4b13-4d04-9b5e-ca9297325b46 850Gi RWO Delete Bound db2u/cachingtier-c-nvme-db2whmpp-db2u-1 openebs-db2u-lvm 17h

Db2 Pods

kubectl get po

NAME READY STATUS RESTARTS AGE

c-nvme-db2whmpp-db2u-0 1/1 Running 0 17h

c-nvme-db2whmpp-db2u-1 1/1 Running 0 17h

c-nvme-db2whmpp-db2u-2 1/1 Running 0 17h

c-nvme-db2whmpp-etcd-0 1/1 Running 0 17h

c-nvme-db2whmpp-etcd-1 1/1 Running 0 17h

c-nvme-db2whmpp-etcd-2 1/1 Running 0 17h

c-nvme-db2whmpp-ldap-966d917b5-nkg8f 1/1 Running 0 17h

c-nvme-db2whmpp-restore-morph-jmzlw 0/1 Completed 0 17h

c-nvme-db2whmpp-tools-998d57889-72kpq 1/1 Running 0 17h

db2u-day2-ops-controller-manager-6dff47f8f8-q8k98 1/1 Running 0 17h

db2u-operator-manager-649fcd9b77-xnhf6 1/1 Running 0 17h

## Use Case 3: Using Partitioned VGs from a single NVMe device

This use case involves partitioning a single NVMe device into two and then creating VGs over those partitions. It is like the use case 1 except for the device partitioning steps. So, we provide steps to partition the device and then refer to the steps in the use case 1 to complete the deployment process.

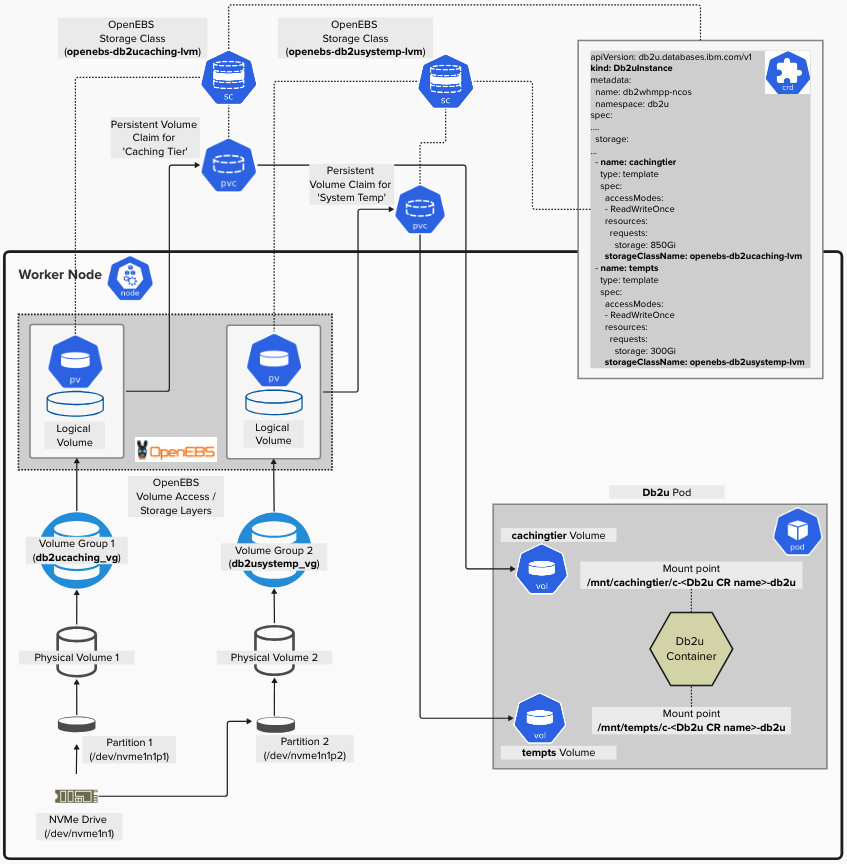


Fig 5: Using Partitioned Volume Groups from a single NVMe device

Here are the steps to partition a device:

1. Take one of the devices, say /dev/nvme1n1, and partition it into two thus:  
     
   parted /dev/nvme1n1

Then inside the prompt, enter the following:  
  
mklabel gpt

mkpart primary 0% 50%

mkpart primary 50% 100%

print

The above example partitions the disk into two equal sizes. You can divide the partition based on your storage use-case, such as 30% to NCOS caching tier vs 70% to system temps.

1. Verify that the device has been partitioned. For a successful partitioning, the output should be like that below:  
     
   lsblk | grep nvme

nvme1n1 259:0 0 1.7T 0 disk

├─nvme1n1p1 259:6 0 884.8G 0 part

└─nvme1n1p2 259:7 0 884.8G 0 part  
  
ls -la /dev/nvme1n1\*

brw-rw---- 1 root disk 259, 0 Jun 3 14:33 /dev/nvme1n1

brw-rw---- 1 root disk 259, 6 Jun 3 15:35 /dev/nvme1n1p1

brw-rw---- 1 root disk 259, 7 Jun 3 15:35 /dev/nvme1n1p2  
  
The output shows that the device has been partitioned into 2 parts of ~885G size each.

1. Create PVs with the partitions  
     
   pvcreate /dev/nvme1n1p1

pvcreate /dev/nvme1n1p2

1. Create corresponding VGs with the PVs  
     
   vgcreate db2ucaching\_vg /dev/nvme1n1p1

vgcreate db2usystemp\_vg /dev/nvme1n1p2  
  
And confirm that the creation succeeds; output should be like that below:  
  
PVs:

PV VG Fmt Attr PSize PFree

/dev/nvme1n1p1 db2ucaching\_vg lvm2 a-- 884.75g 884.75g

/dev/nvme1n1p2 db2usystemp\_vg lvm2 a-- 884.75g 884.75g  
  
VGs:

VG #PV #LV #SN Attr VSize VFree

db2ucaching\_vg 1 0 0 wz--n- 884.75g 884.75g

db2usystemp\_vg 1 0 0 wz--n- 884.75g 884.75g

Note that these steps (1-4) must be completed on all the participating nodes.

1. Follow steps 4-8 in use case 1 to complete the deployment processes.

## Use Case 4: Enabling Thin Provisioning

Thin provisioning helps with efficient space management by allocating space only when it is needed (or requested) by applications for data storage rather than reserving the space upfront. Any of the previously mentioned use cases can be configured for thin provisioning; the only change that is required is enabling thin provisioning in the SCs.

A common pattern is to use Thin provisioning to consolidate all available (local) NVMe devices into a single Storage Pool via one Volume Group (similar to use case 2) and then use a *Thin Pool* to provision storage for the application pods. This approach offers flexibility to add more storage devices to the Thin-provisioned Volume Group as needed, increasing the available capacity of the Thin Pool – all transparent to the application pods mounting Persistent Volumes (I.E., Thin Volumes) from that Volume Group.

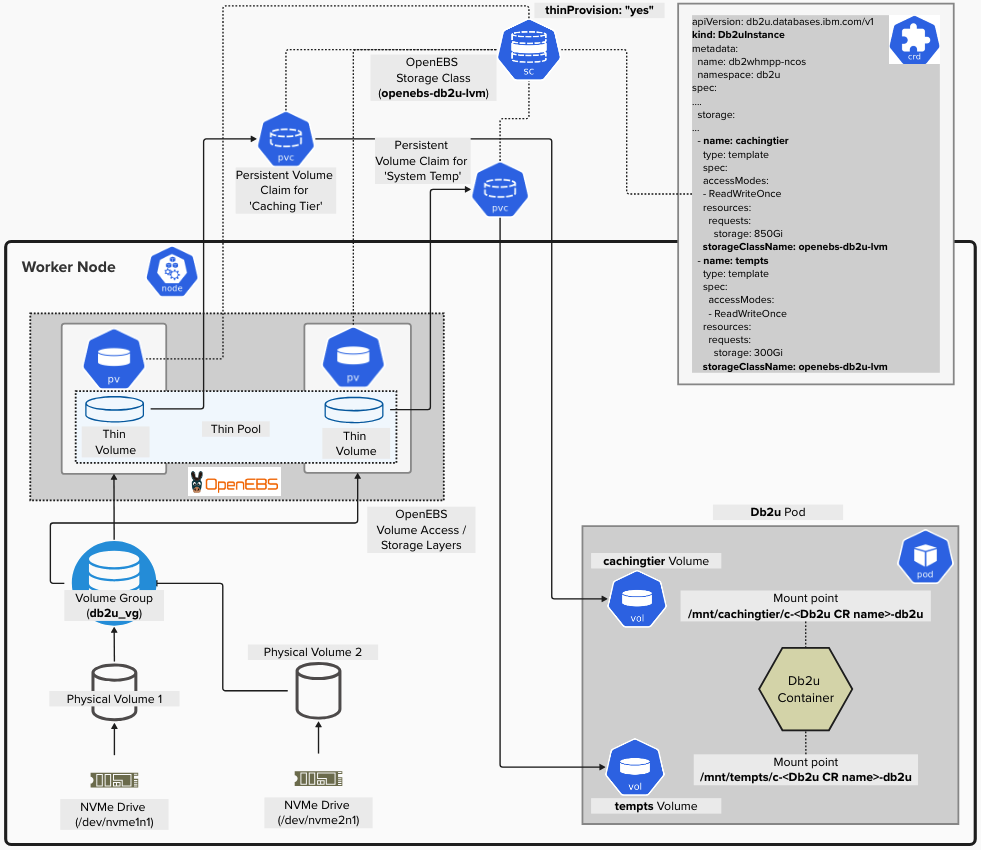


Fig 6: Using Thin Provisioning with multiple NVMe devices

Therefore, to enable thin provisioning for any use case, the SC needs to be configured, for example, as shown below:

SC with thin provision enabled  
  
kubectl apply -f - <<EOF

allowVolumeExpansion: true

allowedTopologies:

- matchLabelExpressions:

- key: openebs.io/lvmvg-caching

values:

- db2ucaching\_vg

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: openebs-db2ucaching-lvm

**parameters:**

storage: lvm

**thinProvision: "yes"**

volgroup: db2ucaching\_vg

provisioner: local.csi.openebs.io

reclaimPolicy: Delete

volumeBindingMode: WaitForFirstConsumer

EOF

The change in the SC shown above is the setting thinProvision: "yes". Any of the previous VG/LVM strategy use cases can use this setting in their respective Storage Classes and enable thin provisioning.

Upon successful deployment and checking the nodes, you will observe that *thinpool volumes* are created for the PVCs as expected like an output shown below:

lsblk

NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT

nvme1n1 259:0 0 1.7T 0 disk

├─**nvme1n1p1**  259:6 0 884.8G 0 part

├─db2ucaching\_vg-db2ucaching\_vg\_thinpool\_tmeta 253:1 0 52M 0 lvm

└─db2ucaching\_vg-db2ucaching\_vg\_thinpool-tpool 253:4 0 50G 0 lvm

├─db2ucaching\_vg-db2ucaching\_vg\_thinpool 253:5 0 50G 1 lvm

└─db2ucaching\_vg-pvc--ef09375a--72e1--4728--bdb4--1511c5298765 253:6 0 50G 0 lvm /var/lib/kubelet/pods/29df74c9-4ec2-4609-9065-3ee3e1dc2dc7/volumes/kubernetes.iocsi/pvc-ef09375a-72e1-4728-bdb4-1511c5298765/mount

└─db2ucaching\_vg-db2ucaching\_vg\_thinpool\_tdata 253:2 0 50G 0 lvm

└─db2ucaching\_vg-db2ucaching\_vg\_thinpool-tpool 253:4 0 50G 0 lvm

├─db2ucaching\_vg-db2ucaching\_vg\_thinpool 253:5 0 50G 1 lvm

└─db2ucaching\_vg-pvc--ef09375a--72e1--4728--bdb4--1511c5298765 253:6 0 50G 0 lvm /var/lib/kubelet/pods/29df74c9-4ec2-4609-9065-3ee3e1dc2dc7/volumes/kubernetes.iocsi/pvc-ef09375a-72e1-4728-bdb4-1511c5298765/mount

└─**nvme1n1p2**  259:7 0 884.8G 0 part

├─db2usystemp\_vg-db2usystemp\_vg\_thinpool\_tmeta 253:0 0 52M 0 lvm

└─db2usystemp\_vg-db2usystemp\_vg\_thinpool-tpool 253:7 0 50G 0 lvm

├─db2usystemp\_vg-db2usystemp\_vg\_thinpool 253:8 0 50G 1 lvm

└─db2usystemp\_vg-pvc--2e74d144--ed47--4c20--af19--2eb564924c5c 253:9 0 50G 0 lvm /var/lib/kubelet/pods/29df74c9-4ec2-4609-9065-3ee3e1dc2dc7/volumes/kubernetes.iocsi/pvc-2e74d144-ed47-4c20-af19-2eb564924c5c/mount

└─db2usystemp\_vg-db2usystemp\_vg\_thinpool\_tdata 253:3 0 50G 0 lvm

└─db2usystemp\_vg-db2usystemp\_vg\_thinpool-tpool 253:7 0 50G 0 lvm

├─db2usystemp\_vg-db2usystemp\_vg\_thinpool 253:8 0 50G 1 lvm

└─db2usystemp\_vg-pvc--2e74d144--ed47--4c20--af19--2eb564924c5c 253:9 0 50G 0 lvm /var/lib/kubelet/pods/29df74c9-4ec2-4609-9065-3ee3e1dc2dc7/volumes/kubernetes.iocsi/pvc-2e74d144-ed47-4c20-af19-2eb564924c5c/mount

# Conclusion

In this study, we evaluated the use of NVMe devices to optimize both space utilization and performance for Db2 Warehouse storage types, including NCOS caching tier and temporary tablespaces (tempts). We explored several strategies for aggregating NVMe devices on Amazon EKS and provided practical guidance on how each strategy can be implemented based on specific deployment scenarios.

Selecting the optimal VG/LVM configuration strategy for your Db2 Warehouse deployment will largely depend on the available node types – such as whether nodes have multiple NVMe devices – and your specific performance priorities. For instance, if your workload places greater emphasis on system temporary space performance than on NCOS caching, that priority should guide your configuration strategy. With that in mind, the general rule of thumb is as follows:

* When two or more NVMe devices are available per worker node  
  Use Case 2: Creating a single Volume Group across multiple NVMe devices enables efficient utilization of all available drives without compromising performance. OpenEBS automatically allocates Logical Volumes (LVMs) based on the capacity defined in each storage section of the Custom Resource (CR). Additionally, thin provisioning can be applied to maximize space efficiency by allocating storage dynamically rather than upfront.
* When only one NVMe device is available per worker node  
  Use Case 3: Partitioning the NVMe device into multiple Volume Groups provides fine-grained control over how storage is allocated between Db2’s NCOS caching tier and system temporary spaces. This approach is especially useful when workload characteristics demand prioritization – for example, if performance impact from join spills to system temps is a greater concern than access latency to objects on Cloud Object Storage.

About the Authors

Aruna De Silva is the architect for Db2/Db2 Warehouse containerized offerings on IBM Cloud Pack for Data, OpenShift and Kubernetes. He has nearly 20 years of database technology experience and is based off IBM Toronto software laboratory.

Since 2015, he has been actively involved with modernizing Db2 bringing *Db2 Warehouse – Common Container*, the first containerized Db2 solution out into production in 2016. Since 2019, he has been primarily focused on bringing the success of Db2 Warehouse into cloud native platforms such as OpenShift and Kubernetes while embracing micro service architecture and deployment patterns. He can be contacted at [adesilva@ca.ibm.com](mailto:adesilva@ca.ibm.com).

Hamdi Roumani is a senior manager of the Db2 COS team, overseeing Db2's native cloud object storage support and the columnar storage engine. With extensive experience at IBM, Hamdi began his career on the availability team as a developer, contributing to numerous enhancements for Db2's backup, restore, and write-ahead logging facilities. He then transitioned to the newly formed IBM Cloud IAAS team, where he focused on building the next-generation platform layer, specifically the storage layer (object and block storage) and the monitoring framework used for billing. Recently, Hamdi returned to Db2 to work on the next-generation warehouse engine. He can be reached at [roumani@ca.ibm.com](mailto:roumani@ca.ibm.com)

Labi Adekoya is a Reliability Engineer working on containerized Db2 Warehouse offerings. With over 12 years of experience, he focuses on building and demystifying reliable distributed systems. He can be reached at [owolabi.adekoya@ibm.com](mailto:owolabi.adekoya@ibm.com).