EDB CloudNativePG Global Cluster

Release 1.1.1

EnterpriseDB Corporation

Apr 30, 2025

Contents

1 Prerequisite and Release Note **3**
- 1.1 EDB CloudNativePG Global Cluster . . . 3
  - 1.1.1 Before you start . . . 3
  - 1.1.2 Supported Kubernetes distributions . . . 3
  - 1.1.3 Requirements . . . 4
  - 1.1.4 API reference . . . 4
  - 1.1.5 Trademarks . . . 4
- 1.2 EDB Postgres for Kubernetes Release notes . . . 4
  - 1.2.1 EDB Postgres Distributed for Kubernetes 1.1.1 release notes . . . 5
  - 1.2.2 EDB Postgres Distributed for Kubernetes 1.1.0 release notes . . . 5
  - 1.2.3 EDB Postgres Distributed for Kubernetes 1.0.1 release notes . . . 6
  - 1.2.4 EDB Postgres Distributed for Kubernetes 1.0.0 release notes . . . 6

2 Getting Started **9**
- 2.1 Before You Start . . . 9
  - 2.1.1 Kubernetes terminology . . . 9
  - 2.1.2 PGD terminology . . . 10
  - 2.1.3 Cloud terminology . . . 11
  - 2.1.4 What to do next . . . 11
- 2.2 Use cases . . . 11
  - 2.2.1 Case 1: Applications inside Kubernetes . . . 11
  - 2.2.2 Case 2: Applications outside Kubernetes . . . 12
- 2.3 Architecture . . . 12
  - 2.3.1 Relationship with EDB Postgres Distributed . . . 13
  - 2.3.2 Relationship with EDB Postgres for Kubernetes . . . 13
  - 2.3.3 Managing PGD using EDB Postgres Distributed for Kubernetes . . . 13
  - 2.3.4 Deploying PGD on Kubernetes . . . 14
- 2.4 Installation and upgrade . . . 16
  - 2.4.1 Installing the Operator on Kubernetes . . . 16
  - 2.4.2 Details about the deployment . . . 18
  - 2.4.3 Deploy PGD Clusters . . . 19
  - 2.4.4 Upgrade . . . 21
- 2.5 Quick start . . . 23
  - 2.5.1 Part 1 - Set up the local Kubernetes playground . . . 23
  - 2.5.2 Part 2 - Install EDB Postgres Distributed for Kubernetes . . . 24
  - 2.5.3 Part 3 - Deploy a PGD cluster . . . 24

3 Configuration **29**
3.1 Operator configuration . . . 29
3.1.1 EDB Postgres Distributed for Kubernetes Operator configuration . . . 29
3.1.2 EDB Postgres for Kubernetes Operator configuration . . . 32
3.2 Examples of configuration . . . 32
3.2.1 Basics . . . 32
3.2.2 Backup and restore . . . 45
3.2.3 Read node routing . . . 58
3.2.4 LDAP . . . 59
3.2.5 Managed roles . . . 62
3.2.6 TDE . . . 63
3.2.7 Configurations . . . 64

4 Using **73**
4.1 Managing EDB Postgres Distributed (PGD) databases . . . 73
4.1.1 PGD CLI . . . 73
4.1.2 Accessing the database . . . 74
4.2 Backup on object stores . . . 76
4.2.1 Common object stores . . . 76
4.2.2 WAL archive . . . 76
4.2.3 Scheduled backups . . . 77
4.2.4 Retention policies . . . 78
4.2.5 Compression algorithms . . . 79
4.2.6 Tagging of backup objects . . . 79
4.2.7 On-demand backups of a PGD node . . . 79
4.3 Recovery . . . 79
4.3.1 Prerequisites . . . 80
4.3.2 Recovery from an object store . . . 80
4.3.3 PITR from an object store . . . 81
4.3.4 Recovery from an object store specifying a `backupID` . . . 82
4.3.5 Recovery from volumeSnapshot . . . 82
4.3.6 PITR from volumeSnapshot . . . 84
4.3.7 Recovery targets . . . 84
4.3.8 Recovery and create PGD Groups in multiple regions . . . 84
4.4 Security . . . 85
4.4.1 Code . . . 85
4.4.2 Container . . . 85
4.4.3 Cluster . . . 86
4.5 Connectivity . . . 88
4.5.1 Services . . . 88
4.5.2 Domain names resolution . . . 90
4.5.3 TLS configuration . . . 91
4.5.4 Connecting from an application . . . 91
4.6 Certificates . . . 92
4.6.1 Server certificates configuration . . . 92
4.6.2 Client certificates configuration . . . 94
4.6.3 TLS mode . . . 94
4.6.4 Self-signed example . . . 95
4.7 Client TLS/SSL connections . . . 97
4.8 Declarative pausing and resuming . . . 97
4.9 EDB’s private container registries . . . 98
4.9.1 Which repository to choose? . . . 98
4.9.2 How to retrieve the token . . . 98
4.9.3 Trials . . . 99

4.9.4 Operand images . . . 99
4.9.5 Customize Operand images . . . 101
4.10 Predefined labels . . . 103
4.10.1 Predefined annotations . . . 103
4.11 Managed configuration . . . 104
4.12 Monitoring . . . 104
4.12.1 Monitoring Instances . . . 104
4.12.2 Monitoring Operator . . . 105
4.12.3 Monitor on Openshift . . . 105
4.13 PGDGroup parting . . . 105
4.13.1 Deletion and finalizers . . . 105
4.13.2 Time limit . . . 105
4.13.3 Skip finalizer . . . 105
4.13.4 PGDGroup cleanup . . . 105
4.14 SQLMutations . . . 106
4.14.1 SQLMutation types . . . 107
4.15 Red Hat OpenShift . . . 107
4.15.1 Ensuring access to EDB private registry . . . 107
4.15.2 Installing the operator . . . 108
4.15.3 Creating a PGD cluster . . . 110
4.15.4 Channel . . . 118
4.15.5 Dependencies . . . 118
4.16 Transparent data encryption (TDE) . . . 118
4.17 LDAP authentication . . . 120
4.18 API Reference . . . 121
4.18.1 Resource Types . . . 121
4.18.2 CertificateKeystores . . . 121
4.18.3 CertificatePrivateKey . . . 121
4.18.4 CertificateSpec . . . 122
4.18.5 ConditionStatus . . . 123
4.18.6 JKSKeystore . . . 123
4.18.7 KeyUsage . . . 124
4.18.8 LocalObjectReference . . . 124
4.18.9 ObjectReference . . . 125
4.18.10 PKCS12Keystore . . . 125
4.18.11 PrivateKeyAlgorithm . . . 125
4.18.12 PrivateKeyEncoding . . . 126
4.18.13 PrivateKeyRotationPolicy . . . 126
4.18.14 SecretKeySelector . . . 126
4.18.15 X509Subject . . . 126
4.18.16 ClusterImageCatalog . . . 127
4.18.17 ImageCatalog . . . 127
4.18.18 PGDGroup . . . 127
4.18.19 PGDGroupCleanup . . . 127
4.18.20 Backup . . . 128
4.18.21 BackupStatus . . . 128
4.18.22 CNPStatus . . . 128
4.18.23 CatalogImage . . . 129
4.18.24 CertManagerTemplate . . . 129
4.18.25 ClientCertConfiguration . . . 130
4.18.26 ClientPreProvisionedCertificates . . . 130
4.18.27 ClusterStatus . . . 130
4.18.28 CnpBaseConfiguration . . . 131
4.18.29 CnpConfiguration . . . 132

4.18.30 ConnectionString . . . 133
4.18.31 ConnectivityConfiguration . . . 133
4.18.32 ConnectivityStatus . . . 134
4.18.33 DNSConfiguration . . . 134
4.18.34 DiscoveryJobConfig . . . 134
4.18.35 ImageCatalogRef . . . 135
4.18.36 ImageCatalogSpec . . . 135
4.18.37 ImageStatus . . . 135
4.18.38 InheritedMetadata . . . 136
4.18.39 JoinMethod . . . 136
4.18.40 Metadata . . . 136
4.18.41 NodeCertificateStatus . . . 137
4.18.42 NodeKindName . . . 137
4.18.43 NodeSummary . . . 137
4.18.44 NodesExtensionsStatus . . . 138
4.18.45 OTELConfiguration . . . 138
4.18.46 OTELTLSConfiguration . . . 138
4.18.47 OperatorPhase . . . 138
4.18.48 OperatorPhaseCleanup . . . 139
4.18.49 PGDGroupCleanupSpec . . . 139
4.18.50 PGDGroupCleanupStatus . . . 139
4.18.51 PGDGroupSpec . . . 139
4.18.52 PGDGroupStatus . . . 140
4.18.53 PGDNodeGroupEntry . . . 141
4.18.54 PGDNodeGroupSettings . . . 141
4.18.55 PGDProxyConfiguration . . . 142
4.18.56 PGDProxyEntry . . . 142
4.18.57 PGDProxySettings . . . 143
4.18.58 PGDProxyStatus . . . 143
4.18.59 PGDRaftStatus . . . 144
4.18.60 PGDStatus . . . 144
4.18.61 ParentGroupConfiguration . . . 144
4.18.62 PauseStatus . . . 145
4.18.63 PgdConfiguration . . . 145
4.18.64 PhaseType . . . 146
4.18.65 PreProvisionedCertificate . . . 146
4.18.66 ReconciliationStage . . . 146
4.18.67 RecoverabilityPointsByMethod . . . 146
4.18.68 ReplicationCertificateStatus . . . 146
4.18.69 Restore . . . 147
4.18.70 RestoreStatus . . . 147
4.18.71 RootDNSConfiguration . . . 148
4.18.72 SQLMutation . . . 148
4.18.73 SQLMutationType . . . 148
4.18.74 SQLMutations . . . 148
4.18.75 ScheduledBackupSpec . . . 149
4.18.76 ScheduledBackupStatus . . . 149
4.18.77 ServerCertConfiguration . . . 150
4.18.78 ServiceTemplate . . . 150
4.18.79 ServiceUpdateStrategy . . . 150
4.18.80 TLSConfiguration . . . 151
4.18.81 TLSMode . . . 151
4.18.82 VolumeSnapshotRestoreStatus . . . 151
4.18.83 VolumeSnapshotsConfiguration . . . 151

4.19 Supported versions . . . 152
4.19.1 Naming Scheme . . . 152
4.19.2 General Support status . . . 152
4.19.3 Detailed support status . . . 152
4.19.4 Supported versions on the Openshift Platform . . . 153
4.20 Known issues and limitations . . . 153
4.20.1 Postgres major version upgrades . . . 153
4.20.2 Data migration . . . 153
4.20.3 Connectivity with PgBouncer . . . 154
4.20.4 Backup operations . . . 154
4.20.5 Known issues and limitations in EDB Postgres Distributed . . . 154
4.21 Image catalog . . . 155
4.21.1 Example of a namespaced `ImageCatalog` . . . 155
4.21.2 Example of a cluster-wide catalog using `ClusterImageCatalog` resource . . . 155
4.22 Join method . . . 156
4.22.1 The initial group . . . 156
4.22.2 Configure the join method for node join . . . 157
4.22.3 Configure the join method for group join . . . 158

EDB Postgres Distributed for Kubernetes (`PGD4K`) is an operator designed to manage EDB Postgres Distributed (PGD) workloads on Kubernetes, with traffic routed by PGD Proxy.

The main custom resource that the operator provides is called `PGDGroup` .

Architectures can also be deployed across different Kubernetes clusters.

CHAPTER 1

Prerequisite and Release Note

1.1 EDB CloudNativePG Global Cluster

1.1.1 Before you start

EDB Postgres Distributed for Kubernetes provides you with a way to deploy EDB Postgres Distributed in a Kubernetes environment. Therefore, we recommend reading the EDB Postgres Distributed documentation .

To start working with EDB Postgres Distributed for Kubernetes, read the following in the PGD documentation:

- Terminology
- PGD overview
- *Choosing your architecture*
- Choosing a Postgres distribution

For advanced usage and maximum customization, it's also important to be familiar with the EDB Postgres for Kubernetes documentation , as described in *Architecture* .

1.1.2 Supported Kubernetes distributions

EDB Postgres Distributed for Kubernetes is available for:

- Kubernetes version 1.23 or later through a Helm chart
- Red Hat OpenShift version 4.10 or later only through the Red Hat OpenShift certified operator

1.1.3 Requirements

EDB Postgres Distributed for Kubernetes requires that the Kubernetes/OpenShift clusters hosting the distributed PGD cluster were prepared by you to cater for:

- The public key infrastructure (PKI) encompassing all the Kubernetes clusters the PGD global group is spread across. mTLS is required to authenticate and authorize all nodes in the mesh topology and guarantee encrypted communication.
- Networking infrastructure across all Kubernetes clusters involved in the PGD global group to ensure that each node can communicate with each other

EDB Postgres Distributed for Kubernetes also requires Cert Manager 1.10 or later.

See also:

See *Connectivity* for more information.

1.1.4 API reference

For a list of resources provided by EDB Postgres Distributed for Kubernetes, see the *API reference* .

1.1.5 Trademarks

Postgres, PostgreSQL, and the Slonik logo

are trademarks or registered trademarks of the PostgreSQL Community Association of Canada, and used with their permission.

1.2 EDB Postgres for Kubernetes Release notes

The EDB Postgres Distributed for Kubernetes documentation describes the major version of EDB Postgres Distributed for Kubernetes, including minor releases and patches. The release notes provide information on what is new in each release. For new functionality introduced in a minor or patch release, the content also indicates the release that introduced the feature.

Version	Release date
Bootstrap	Mar 18 2025
Bootstrap	Dec 25 2024
Bootstrap	14 Oct 2024
Bootstrap	24 Apr 2024

1.2.1 EDB Postgres Distributed for Kubernetes 1.1.1 release notes

Released: Mar 18 2025

Features

- *New PGD version support:* PGD version 5.7 is now supported.
- *New PG4K operator version support:* PG4K operator v1.25.1 is now supported.

Component	Description
PGD4K	Added support for PGD node physical join to local group.
PGD4K	Added support for PGD node physical join to remote group.
PGD4K	Added ImageCatalog support for PGD operators.
PGD4K	Improved detection to decide whether *writeLead* transfer is needed before patching the *writelead* node.
PGD4K	Ensured 'bdr' can be added only once into PG additional libraries.
PGD4K	Added shared configuration file for pgd-proxy and pgd-cli in proxy pod.

Supported versions

- Kubernetes: 1.32, 1.31, 1.30, and 1.29
- PostgreSQL: 17, 16, 15, 14, and 13
- PG4K: v1.22.8, v1.22.8, v1.25.0, v1.25.1

Default images

- PGD: postgresql-pgd 17.4-5.7.0-1
- PGD Proxy: edb-pgd-proxy 5.7.0-1

1.2.2 EDB Postgres Distributed for Kubernetes 1.1.0 release notes

Released: 25 Dec 2024

Features

Component	Description
PGD4K	Support for New PG4K Release: Starting with version 1.1.0, the PGD4K operator now supports the new PG4K Long-Term Support (LTS) release, version 1.25.
PGD4K	New PGD Version Support: PGD version 5.6.1 is now supported.

continues on next page

Table 3 – continued from previous page

Component	Description
PGD4K	New Operator Architecture Support: Adds support for linux/arm64 and linux/ppc64le architectures.
PGD4K	Configuration support for cnpg-i plugin.
PGD4K	Avoids unnecessary write lead transfer when reconcilePodSpec is used in PG4K cluster.
PGD4K	Scale Down Write Leader Last: The write leader will be the last PGD node to scale down.
PGD4K	Enhanced Restore Process: The restore process has been improved by splitting it into components, and using separate jobs for different types of restore and bootstrap operations.
PGD4K	Webhook Validation enhancement: validity of restore.serverNames will now be checked via the webhook.
PGD4K	Now includes clusterStatus as part of the PGDGroup status for information on the phase of the managed clusters.

1.2.3 EDB Postgres Distributed for Kubernetes 1.0.1 release notes

Released: 14 Oct 2024

Features

Component	Description
PGD4K	Operator base image upgraded to UBI9
PGD4K	Support LDAP authentication configuration in PGDGroup
PGD4K	Support tablespace configuration in PGDGroup
PGD4K	Support projectedVolumeTemplate configuration in PGDGroup
PGD4K	Support read node routing on pgd with creation of a proxy service
PGD4K	Support backup and restore PGDGroup using VolumeSnapshot
PGD4K	Support multiple schedulers in PGDGroup for different backup methods
PGD4K	Support new mutation type which is only executed against write lead node.
PGD4K	Avoid unnecessary write leader transfer when PGDGroup configuration is changed
PGD4K	Handle nodeSelector and toleration changes for paused PGDGroup
PGD4K	Reuse nodeSelector and tolerations in discovery job
PGD4K	Support application database configuration to be applied before join
PGD4K	Support topologySpreadConstraints configuration

1.2.4 EDB Postgres Distributed for Kubernetes 1.0.0 release notes

Released: 24 Apr 2024

This is the first major stable release of EDB Postgres Distributed for Kubernetes, a Kubernetes operator to deploy and manage EDB Postgres Distributed clusters.

Highlights of EDB Postgres Distributed for Kubernetes 1.0.0

The operator implements the `PGDGroup` custom resource in the API group `pgd.k8s.enterprisedb.io`. You can use this resource to create and manage EDB Postgres Distributed clusters inside Kubernetes with capabilities including:

- Deployment of EDB Postgres Distributed clusters with versions 5 and later.
- Additional self-healing capability on top of that of Postgres Distributed, such as recovery and restart of failed PGD nodes.
- Defined services that allow applications to connect to the write leader of each PGD group.

Note: The EDB Postgres Distributed for Kubernetes operator leverages EDB Postgres for Kubernetes (PG4K) and inherits many of that project's capabilities. EDB Postgres Distributed for Kubernetes version 1.0.0 is based, specifically, on release 1.22 of PG4K. See the PG4K release notes for more details.

Features

Component	Description
PGD4K	Deployment of EDB Postgres Distributed clusters with versions 5 and later inside Kubernetes
PGD4K	Self-healing capabilities such as recovery and restart of failed PGD nodes
PGD4K	Defined services that allow applications to connect to the write leader of each PGD group
PGD4K	Implementation of Raft subgroups
PGD4K	TLS connections and client certificate authentication
PGD4K	Continuous backup to an S3-compatible object store

CHAPTER 2

Getting Started

2.1 Before You Start

Before you get started, it's essential that you become familiar with some terminology that's specific to Kubernetes and PGD.

2.1.1 Kubernetes terminology

Node
A *node* is a worker machine in Kubernetes, either virtual or physical, where all services necessary to run pods are managed by the control plane nodes.

Pod
A *pod* is the smallest computing unit that can be deployed in a Kubernetes cluster and is composed of one or more containers that share network and storage.

Service
A *service* is an abstraction that exposes as a network service an application that runs on a group of pods and standardizes important features, such as service discovery across applications, load balancing, and failover.

Secret
A *secret* is an object that's designed to store small amounts of sensitive data, such as passwords, access keys, or tokens, for use within pods.

Storage class
A *storage class* allows an administrator to define the classes of storage in a cluster, including provisioner (such as AWS EBS), reclaim policies, mount options, volume expansion, and so on.

Persistent volume
A *persistent volume* (PV) is a resource in a Kubernetes cluster that represents storage that was either manually provisioned by an administrator or dynamically provisioned by a *storage class* controller. A PV is associated with a pod using a *persistent volume claim*, and its lifecycle is independent of any pod that uses it. Normally, a PV is

a network volume, especially in the public cloud. A *local persistent volume* (LPV) is a persistent volume that exists only on the particular node where the pod that uses it is running.

Persistent volume claim
A *persistent volume claim* (PVC) represents a request for storage, which might include size, access mode, or a particular storage class. Similar to how a pod consumes node resources, a PVC consumes the resources of a PV.

Namespace
A *namespace* is a logical and isolated subset of a Kubernetes cluster and can be seen as a *virtual cluster* within the wider physical cluster. Namespaces allow administrators to create separated environments based on projects, departments, teams, and so on.

RBAC
Role-based access control (RBAC), also known as *role-based security*, is a method used in computer systems security to restrict access to the network and resources of a system to authorized users only. Kubernetes has a native API to control roles at the namespace and cluster level and associate them with specific resources and individuals.

CRD
A *custom resource definition* (CRD) is an extension of the Kubernetes API and allows developers to create new data types and objects, *called custom resources*.

Operator
An *operator* is a Kubernetes software extension that automates those steps that are normally performed by a human operator when managing one or more applications or given services. An operator assists Kubernetes in making sure that the resource' s defined state always matches the observed one.

kubectl
`kubectl` is the command-line tool used to manage a Kubernetes cluster.

EDB Postgres Distributed for Kubernetes requires a Kubernetes version supported by the community. See Supported releases for details.

2.1.2 PGD terminology

For more information, see Terminology in the PGD documentation.

Data node
A PGD database instance.

Failover
The automated process that recognizes a failure in a highly available database cluster and takes action to connect the application to another active database.

Switchover
A planned change in connection between the application and the active database node in a cluster, typically done for maintenance.

Write leader
In always-on architectures, a node is selected as the correct connection endpoint for applications. This node is called the write leader. The write leader is selected by consensus of a quorum of data nodes.

2.1.3 Cloud terminology

Region : A *region* in the cloud is an isolated and independent geographic area organized in *availability zones*. Zones within a region have very little round-trip network latency.

Zone : An *availability zone* in the cloud (also known as a *zone*) is an area in a region where resources can be deployed. Usually, an availability zone corresponds to a data center or an isolated building of the same data center.

2.1.4 What to do next

Now that you are familiar with the terminology, you can *test EDB Postgres Distributed for Kubernetes on your laptop using a local cluster* before deploying the operator in your selected cloud environment.

2.2 Use cases

EDB Postgres Distributed for Kubernetes was designed to work with applications that reside in the same Kubernetes cluster for a full cloud native experience.

However, it might happen that, while the database can be hosted inside a Kubernetes cluster, applications can' t be containerized at the same time and need to run in a traditional environment such as a VM.

The following is a summary of the basic considerations. See the *EDB Postgres for Kubernetes documentation*

for more detail.

2.2.1 Case 1: Applications inside Kubernetes

In a typical situation, the application and the database run in the same namespace inside a Kubernetes cluster.

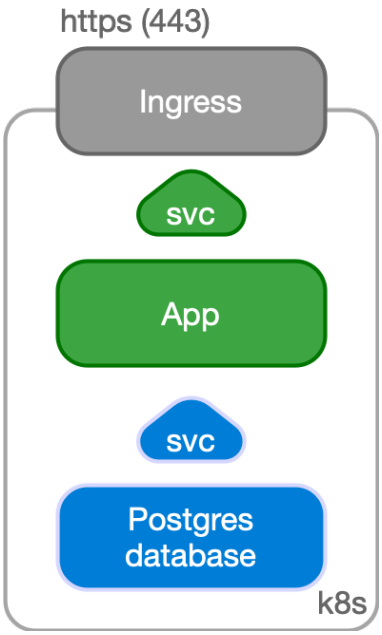

Fig. 1: Application and Database inside Kubernetes

The application, normally stateless, is managed as a standard deployment, with multiple replicas spread over different Kubernetes nodes and internally exposed through a ClusterIP service.

The service is exposed externally to the end user through an Ingress and the provider' s load balancer facility by way of HTTPS.

2.2.2 Case 2: Applications outside Kubernetes

Another possible use case is to manage your PGD database inside Kubernetes while having your applications outside of it, for example, in a virtualized environment. In this case, PGD is represented by an IP address or host name and a TCP port, corresponding to the defined Ingress resource in Kubernetes.

The application can still benefit from a TLS connection to PGD.

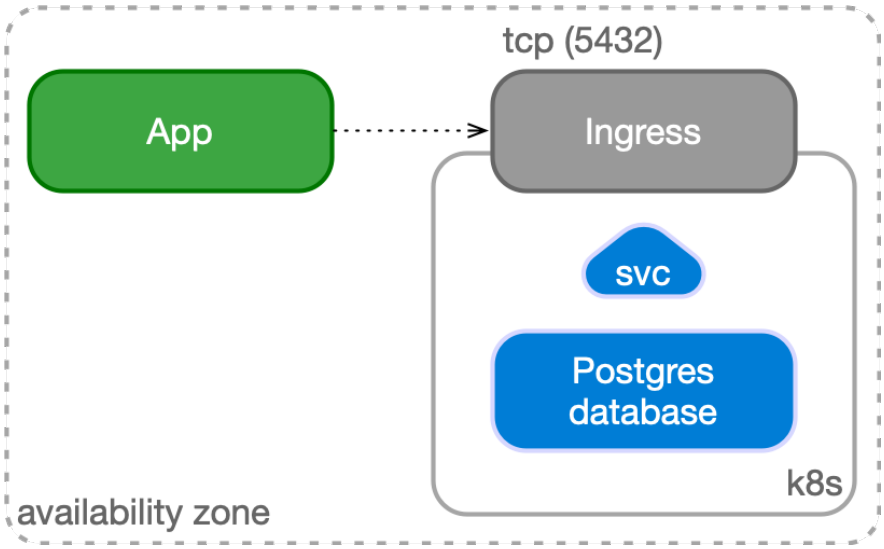

Fig. 2: Application outside Kubernetes

2.3 Architecture

Consider these main architectural aspects when deploying EDB Postgres Distributed in Kubernetes.

EDB Postgres Distributed for Kubernetes is a Kubernetes operator

designed to deploy and manage EDB Postgres Distributed clusters running in private, public, hybrid, or multi-cloud environments.

2.3.1 Relationship with EDB Postgres Distributed

EDB Postgres Distributed (PGD)

is a multi-master implementation of Postgres designed for high performance and availability. PGD generally requires deployment using Trusted Postgres Architect (TPA) , a tool that uses Ansible to provision and deploy PGD clusters.

EDB Postgres Distributed for Kubernetes offers a different way of deploying PGD clusters, leveraging containers and Kubernetes. The advantages are that the resulting architecture:

- Is self-healing and robust.
- Is managed through declarative configuration.
- Takes advantage of the vast and growing Kubernetes ecosystem.

2.3.2 Relationship with EDB Postgres for Kubernetes

A PGD cluster consists of one or more *PGD groups*, each having one or more *PGD nodes*. A PGD node is a Postgres database. EDB Postgres Distributed for Kubernetes internally manages each PGD node using the `Cluster` resource as defined by EDB Postgres for Kubernetes, specifically a cluster with a single instance (that is, no replicas).

You can configure the single PostgreSQL instance created by each `Cluster` in the *.spec.cnp*

of the PGD Group spec.

In EDB Postgres Distributed for Kubernetes, as in EDB Postgres for Kubernetes, the underlying database implementation is responsible for data replication. However, it' s important to note that failover and switchover work differently, entailing Raft election and nominating new write leaders. EDB Postgres for Kubernetes handles only the deployment and healing of data nodes.

2.3.3 Managing PGD using EDB Postgres Distributed for Kubernetes

The EDB Postgres Distributed for Kubernetes operator can manage the complete lifecycle of PGD clusters. As such, in addition to PGD nodes (represented as single-instance `Clusters`), it needs to manage other objects associated with PGD.

PGD relies on the Raft algorithm for distributed consensus to manage node metadata, specifically agreement on a *write leader*. Consensus among data nodes is also required for operations such as generating new global sequences or performing distributed DDL.

These considerations force additional actors in PGD above database nodes.

EDB Postgres Distributed for Kubernetes manages the following:

- Data nodes. A node is a database and is managed by EDB Postgres for Kubernetes, creating a `Cluster` with a single instance.
- Witness nodes

are basic database instances that don' t participate in data replication. Their function is to guarantee that consensus is possible in groups with an even number of data nodes or after network partitions. Witness nodes are also managed using a single-instance `Cluster` resource.

- PGD proxies

act as Postgres proxies with knowledge of the write leader. PGD proxies need information from Raft to route writes to the current write leader.

Proxies and routing

PGD groups assume full mesh connectivity of PGD nodes. Each node must be able to connect to every other node using the appropriate connection string (a `libpq` -style DSN). Write operations don't need to be sent to every node. PGD takes care of replicating data after it's committed to one node.

For performance, we often recommend sending write operations mostly to a single node, the *write leader*. Raft is used to identify which node is the write leader and to hold metadata about the PGD nodes. PGD proxies are used to transparently route writes to write leaders and to quickly pivot to the new write leader in case of switchover or failover.

It's possible to configure *Raft subgroups*, each of which can maintain a separate write leader. In EDB Postgres Distributed for Kubernetes, a PGD group containing a PGD proxy comprises a Raft subgroup.

Two kinds of routing are available with PGD proxies:

- Global routing uses the top-level Raft group and maintains one global write leader.
- Local routing uses subgroups to maintain separate write leaders. Local routing is often used to achieve geographical separation of writes.

In EDB Postgres Distributed for Kubernetes, local routing is used by default, and a configuration option is available to select global routing.

For more information on routing with Raft, see Proxies, Raft, and Raft subgroups in the PGD documentation.

PGD architectures and high availability

EDB proposes several recommended architectures to make good use of PGD's distributed multi-master capabilities and to offer high availability.

The Always On architectures are built from either one group in a single location or two groups in two separate locations. See *Choosing your architecture* in the PGD documentation for more information.

2.3.4 Deploying PGD on Kubernetes

EDB Postgres Distributed for Kubernetes leverages Kubernetes to deploy and manage PGD clusters. As such, some adaptations are necessary to translate PGD into the Kubernetes ecosystem.

Images and operands

You can configure PGD to run one of three Postgres distributions. See the PGD documentation

to understand the features of each distribution.

To function in Kubernetes, containers are provided for each Postgres distribution. These are the *operands*. In addition, the operator images are kept in those same repositories.

See EDB private image registries

for details on accessing the images.

Kubernetes architecture

Some of the points of the *PG4K document on Kubernetes architecture*

are reproduced here. See the PG4K documentation for details.

Kubernetes natively provides the possibility to span separate physical locations. These physical locations are also known as data centers, failure zones, or, more frequently, *availability zones*. They are connected to each other by way of redundant, low-latency, private network connectivity.

Being a distributed system, the recommended minimum number of availability zones for a *Kubernetes cluster* is three. This minimum makes the control plane resilient to the failure of a single zone. This means that each data center is active at any time and can run workloads simultaneously.

You can install EDB Postgres Distributed for Kubernetes in a *Single Kubernetes cluster*

or across *Multiple Kubernetes clusters* .

Single Kubernetes cluster

A multi-availability-zone Kubernetes architecture is typical of Kubernetes services managed by cloud providers. Such an architecture enables the EDB Postgres Distributed for Kubernetes and the EDB Postgres for Kubernetes operators to schedule workloads and nodes across availability zones, considering all zones active.

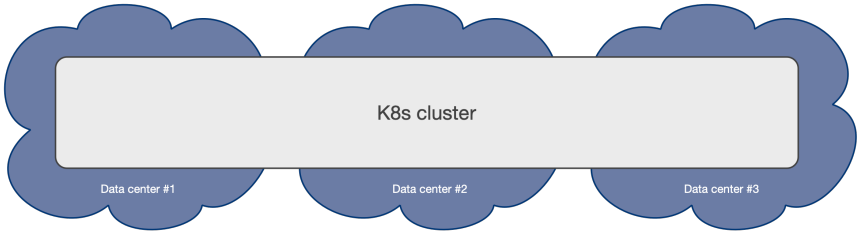

Fig. 3: Kubernetes cluster spanning over 3 independent data centers

PGD clusters can be deployed in a single Kubernetes cluster and take advantage of Kubernetes availability zones to enable high-availability architectures, including the Always On recommended architectures.

You can realize the *Always On Single Location* architecture shown in *Choosing your architecture* in the PGD documentation on a single Kubernetes cluster with three availability zones.

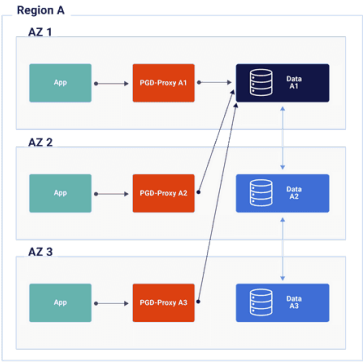

Fig. 4: Always On Single Region

The EDB Postgres Distributed for Kubernetes operator can control the scheduling of pods (that is, which pods go to which data center) using affinity, tolerations, and node selectors, as is the case with EDB Postgres for Kubernetes. Individual scheduling controls are available for proxies as well as nodes.

See the Kubernetes documentation on scheduling , and Scheduling in the EDB Postgres for Kubernetes documentation for more information.

Multiple Kubernetes clusters

PGD clusters can also be deployed in multiple Kubernetes clusters that can reliably communicate with each other.

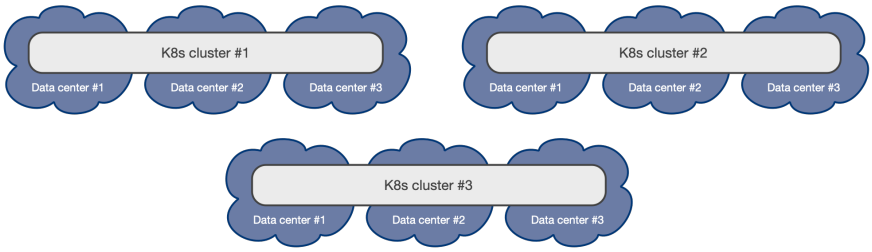

Fig. 5: Multiple Kubernetes clusters

Always On multi-location PGD architectures

can be realized on multiple Kubernetes clusters that meet the connectivity requirements.

For more information, see *Connectivity* .

2.4 Installation and upgrade

See also:

For instructions on how to install Cloud Native PostgreSQL on Red Hat OpenShift Container Platform, please refer to the *Upgrading to 1.0.1 on Red Hat OpenShift*

section.

2.4.1 Installing the Operator on Kubernetes

Obtaining an EDB subscription token

Important: You must obtain an EDB subscription token to install EDB Postgres Distributed for Kubernetes. The token will grant access to the EDB private software repositories.

Installing EDB Postgres Distributed for Kubernetes requires an EDB Repos 2.0 token to gain access to the EDB private software repositories.

You can obtain the token by visiting your EDB Account Profile . You will have to sign in if you are not already logged in.

Your account profile page displays the token next to the label Repos 2.0 Token. By default, the token is obscured. Click the "Show" button (an eye icon) to reveal it.

Your token entitles you to access one of two repositories: `standard` or `enterprise` .

- `standard` - Includes the operators and the EDB Postgres Extended operand images.
- `enterprise` - Includes the operators, and the operand images for EDB Postgres Advanced and EDB Postgres Extended.

Set the relevant value, determined by your subscription, as an environment variable `EDB_SUBSCRIPTION_PLAN` .

```
EDB_SUBSCRIPTION_PLAN=enterprise
```

then set the Repos 2.0 token as an environment variable `EDB_SUBSCRIPTION_TOKEN` .

```
EDB_SUBSCRIPTION_TOKEN=<your-token>
```

Warning: The token is sensitive information. Please ensure that you don't expose it to unauthorized users.

You can now proceed with the installation.

Using the Helm Chart

The operator can be installed using the provided Helm chart .

Directly using the operator manifest

We can deploy the EDB Postgres Distributed for Kubernetes operator directly using the manifest. This manifest will install both `EDB Postgres Distributed for Kubernetes operator` and the latest supported `EDB Postgres for Kubernetes operator` in the same namespace. To deploy the operators using the manifest, follow the steps below:

Install the cert-manager

EDB Postgres Distributed for Kubernetes requires Cert Manager 1.10 or higher. You can follow the installation guide or use the command below to deploy cert-manager.

```
kubectl apply -f \
  https://github.com/cert-manager/cert-manager/releases/download/v1.16.2/cert-manager.
↪yaml
```

Install the EDB pull secret

Before installing EDB Postgres Distributed for Kubernetes, you need to create a *pull secret* for the EDB container registry.

The pull secret needs to be saved in the namespace where the operator will reside (`pgd-operator-system` by default). Create the `pgd-operator-system` namespace using this command:

```
kubectl create namespace pgd-operator-system
```

To create the pull secret, run the following command:

```
kubectl create secret -n pgd-operator-system docker-registry edb-pull-secret \
  --docker-server=docker.enterprisedb.com \
  --docker-username=k8s_${EDB_SUBSCRIPTION_PLAN}_pgd \
  --docker-password=${EDB_SUBSCRIPTION_TOKEN}
```

Install the operator manifest

Now that the pull-secret has been added to the namespace, the operator can be installed like any other resource in Kubernetes, through a YAML manifest applied via `kubectl` .

There are two different manifests available depending on your subscription plan:

- Standard: The latest standard operator manifest .
- Enterprise: The latest enterprise operator manifest .

You can install the manifest for the latest version of the operator by running:

```
kubectl apply --server-side -f \
  https://get.enterprisedb.io/pg4k-pgd/pg4k-pgd-${EDB_SUBSCRIPTION_PLAN}-1.1.1.yaml
```

You can check the operator deployment with the following command:

```
kubectl get deployment -n pgd-operator-system pgd-operator-controller-manager
```

Note: As EDB Postgres Distributed for Kubernetes internally manages each PGD node using the *Cluster* resource defined by EDB Postgres for Kubernetes, we also need to have the EDB Postgres for Kubernetes operator installed as a dependency. The manifest used above contains a well-tested version of EDB Postgres for Kubernetes operator, which will be installed into the same namespace as the EDB Postgres Distributed for Kubernetes operator.

2.4.2 Details about the deployment

In Kubernetes, the operator is by default installed in the `pgd-operator-system` namespace as a Kubernetes `Deployment` . The name of this deployment depends on the installation method. When installed through the manifest, by default, it is named `pgd-operator-controller-manager` . When installed via Helm, by default, the deployment name is derived from the helm release name, appended with the suffix `-edb-postgres-distributed-for-kubernetes` (i.e., `<name>-edb-postgres-distributed-for-kubernetes`).

Note: With Helm you can customize the name of the deployment via the *fullnameOverride* field in the *"values.yaml"* file .

You can get more information using the `describe` command in `kubectl` :

```
$ kubectl get deployments -n pgd-operator-system
```

```
kubectl describe deploy \
  -n pgd-operator-system \
  <deployment-name>
```

(continues on next page)

(continued from previous page)

```
NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
<deployment-name>    1/1     1            1           18m
```

As with any Deployment, it sits on top of a ReplicaSet and supports rolling upgrades. The default configuration of the EDB Postgres Distributed for Kubernetes operator is a Deployment of a single replica, which is suitable for most installations. If the node where the pod is running is not reachable anymore, the pod will be rescheduled on another node.

If you require high availability at the operator level, it is possible to specify multiple replicas in the Deployment configuration, given that the operator supports leader election. In addition, you can take advantage of taints and tolerations to make sure that the operator does not run on the same nodes where the actual PostgreSQL clusters are running (this might even include the control plane for self-managed Kubernetes installations).

See also:

You can change the default behavior of the operator by overriding some default options. For more information, please refer to the *Operator configuration* section.

2.4.3 Deploy PGD Clusters

Be sure to create a cert issuer before you start deploying PGD clusters. The Helm chart prompts you to do this, but in case you miss it, you can, for example, run:

```
kubectl apply -f \
  https://raw.githubusercontent.com/EnterpriseDB/edb-postgres-for-kubernetes-charts/main/
↪hack/samples/issuer-selfsigned.yaml
```

With the operators and a self-signed cert issuer deployed, you can start creating PGD clusters. See the *Quick start* for an example.

Default Operand images

By default, each operator release binds a default version and flavor for PGD and PGD Proxy images. If the image names are not specified in the `spec.pgd.imageName` and `spec.pgdproxy.imageName` fields of the pgdgroup YAML file, the default images will be used.

Default images can be overwritten using the `pgd-operator-controller-manager-config` operator configuration map. For more details, see the section on *EDB Postgres Distributed for Kubernetes Operator configuration* .

We can find what images PGD cluster is using by checking the PGDGroup status:

```
kubectl get pgdgroup <pgdgroup name> -o yaml | yq ".status.image"
```

Specifying operand images

Here is an example of a PGD Cluster using explicit image names:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: group-example-customized
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  imageName: docker.enterprisedb.com/k8s_enterprise_pgd/edb-postgres-extended-pgd:17.4-5.
↪7.0-1
  imagePullSecrets:
  - name: registry-pullsecret
  pgd:
    parentGroup:
      name: world
      create: true
  cnp:
    storage:
      size: 1Gi
  pgdProxy:
    imageName: docker.enterprisedb.com/k8s_enterprise_pgd/edb-pgd-proxy:5.7.1-1
```

Specifying operand images using ImageCatalog

Since the release of version 1.1.1, the PGD4K-PGD operator supports using `ImageCatalog` to specify operand images.

Different `ImageCatalogs` are available based on your subscription plan and PGD versions for each PostgreSQL flavor. Note that the image included in the `ImageCatalog` are ubi-9 based.

- EDB Postgres Advanced PGD: https://get.enterprisedb.io/pgd-k8s-image-catalogs/epas-k8s-<EDB_SUBSCRIPTION_PLAN>-pgd-<PGD_VERSION>.yaml
- EDB Postgres Extended PGD: https://get.enterprisedb.io/pgd-k8s-image-catalogs/pgextended-k8s-<EDB_SUBSCRIPTION_PLAN>-pgd-<PGD_VERSION>.yaml
- Postgres Community PGD: https://get.enterprisedb.io/pgd-k8s-image-catalogs/postgresql-k8s-<EDB_SUBSCRIPTION_PLAN>-pgd-<PGD_VERSION>.yaml

We can create an `Imagecatalog` in the PGD Cluster namespace and reference it in your YAML file.

Let' s create an EDB Postgres Extended PGD 5.7 `ImageCatalog` :

```
kubectl create -f \
        https://get.enterprisedb.io/pgd-k8s-image-catalogs/epas-k8s-enterprise-pgd-5.7.
↪yaml
```

The following example shows how to use the EDB Postgres Extended PGD 5.7 `ImageCatalog` to specify a PostgreSQL major version of 17 for the PGD operand image. The PGD proxy image is automatically sourced from the `ImageCatalog` .

```yaml
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: group-example-catalog
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  imageCatalogRef:
    apiGroup: pgd.k8s.enterprisedb.io
    kind: ImageCatalog
    major: 17
    name: epas-enterprise-pgd-5.7
...
```

2.4.4 Upgrade

Important: Please carefully read the *EDB Postgres Distributed for Kubernetes 1.0.0 release notes*

before performing an upgrade, as some versions might require extra steps.

The EDB Postgres Distributed for Kubernetes (PGD4K) operator relies on the EDB Postgres for Kubernetes (PG4K) operator to manage clusters. To upgrade the EDB PGD4K operator, the PG4K operator must also be upgraded to a supported version. It is recommended to keep the EDB PG4K operator on the Long Term Support (LTS)

version, as this is the tested version compatible with the PGD4K operator.

To upgrade the EDB Postgres Distributed (PGD) for Kubernetes operator, follow these steps:

1. Upgrade the EDB Postgres for Kubernetes operator as a dependency
2. Upgrade the PGD4K controller and the related Kubernetes resources

Note that when you upgrade the EDB Postgres for Kubernetes operator, the instance manager on each PGD node will also be upgraded. For more details, please refer to the *EDB Postgres for Kubernetes upgrades documentation* .

Unless differently stated in the release notes, those steps are normally done by applying the manifest of the newer version for plain Kubernetes installations.

Compatibility among versions

EDB Postgres Distributed for Kubernetes (PGD4K) follows semantic versioning. Every release of the operator within the same API version is compatible with the previous one. The current API version is v1, corresponding to versions 1.x.y of the operator.

The `minor` version of PGD4K operator is tracking a PG4K LTS release change. For example:

- PGD4K operator 1.0.x is fully tested against PG4K LTS 1.22.
- PGD4K operator 1.1.x is fully tested against PG4K LTS 1.25.

A PGD4K operator release has the same support scope as the PG4K LTS release it is tracking.

In addition to new features, new versions of the operator contain bug fixes and stability enhancements. Each version is released in order to maintain the most secure and stable Postgres environment. Because of this, **we strongly encourage users to upgrade to the latest version of the operator**.

The *EDB Postgres Distributed for Kubernetes 1.0.0 release notes* page contains a detailed list of the changes introduced in every released version of EDB Postgres Distributed for Kubernetes, and it must be read before upgrading to a newer version of the software.

Most versions are directly upgradable, and in that case, simply applying the newer manifest for plain Kubernetes installations will complete the upgrade.

When versions are not directly upgradable, the old version (of both PGD4K and PG4K) needs to be removed before installing the new one. This won't affect user data, only the operator itself.

Upgrading to 1.0.1 on Red Hat OpenShift

On the openshift platform, starting from version 1.0.1, the EDB Postgres Distributed for Kubernetes operator is required to reference the LTS releases of the PG4K operator. For example, PGD4K v1.0.1 specifically references the PG4K LTS version 1.22.x (any patch release of the 1.22 LTS branch is valid).

To upgrade PGD4K operator, ensure that the PG4K operator is upgraded to a supported version first. Only then can the PGD4K operator be upgraded.

Important: As PGD4K v1.0.0 does not require referencing an LTS release of the PG4K operator, it may have been installed with any PG4K version. If the installed PG4K version is less than 1.22, you can upgrade to PG4K version 1.22 by changing the subscription channel. For more information, see *upgrading the operator* . However, if the installed PG4K version is greater than 1.22, you will need to reinstall the operator from the stable-1.22 channel to upgrade PGD4K to v1.0.1.

Server-side apply of manifests

To ensure compatibility with Kubernetes 1.29 and upcoming versions, EDB Postgres Distributed for Kubernetes now mandates the utilization of Server-side apply

when deploying the operator manifest.

While employing this installation method poses no challenges for new deployments, updating existing operator manifests using the `--server-side` option may result in errors like the example below:

```
Apply failed with 1 conflict: conflict with "kubectl-client-side-apply" using..
```

If such errors arise, they can be resolved by explicitly specifying the `--force-conflicts` option to enforce conflict resolution:

```
kubectl apply --server-side --force-conflicts -f <OPERATOR_MANIFEST>
```

From then on, `kube-apiserver` will be automatically acknowledged as a recognized manager for the CRDs, eliminating the need for any further manual intervention on this matter.

Upgrading operands

The PGD Cluster supports in-place upgrades of the operand image' s minor version, though the PostgreSQL service will be temporarily unavailable during the upgrade.

Checking Current PGD and Proxy Versions

Before upgrading, you can check the current PGD and PGD proxy versions with the following command:

```
kubectl get pgdgroup <pgdgroup name> -o yaml | yq ".status.image"
```

Upgrade Procedure

- Using Default or Customized Image Name: To upgrade the operand to a new minor version, replace the imageName in the `spec.imageName` and `spec.pgdproxy.imageName` sections of the PGD group YAML file with the new imageName. The images on each node will be upgraded sequentially and restarted accordingly.
- Using Image Catalog: If the PGD Cluster manages image versions using an `ImageCatalog` , simply upgrade the image version specified in the referenced `ImageCatalog` . The PGD Cluster will automatically apply the new image version.

2.5 Quick start

You can test an EDB Postgres Distributed (PGD) cluster on your laptop or computer using EDB Postgres Distributed for Kubernetes on a single local Kubernetes cluster built with Kind .

> **Warning:** These instructions are only for demonstration, testing, and practice purposes and must not be used in production.

This quick start shows you how to start an EDB Postgres Distributed cluster on your local Kubernetes installation so you can experiment with it.

Important: To connect to the Kubernetes cluster, make sure that you have *kubectl* installed on your machine. See the Kubernetes documentation on *installing `kubectl* <https://kubernetes.io/docs/tasks/tools/install-kubectl/>`_ .

2.5.1 Part 1 - Set up the local Kubernetes playground

Install Kind, a tool for running local Kubernetes clusters using Docker container nodes. (Kind stands for Kubernetes IN Docker.) If you already have access to a Kubernetes cluster, you can skip to Part 2.

Install Kind on your environment following the instructions in Kind Quick Start . Then, create a Kubernetes cluster:

```
kind create cluster --name pgd
```

2.5.2 Part 2 - Install EDB Postgres Distributed for Kubernetes

After you have a Kubernetes installation up and running on your laptop, you can install EDB Postgres Distributed for Kubernetes.

See *Installation and upgrade* for details.

2.5.3 Part 3 - Deploy a PGD cluster

As with any other deployment in Kubernetes, to deploy a PGD cluster you need to apply a configuration file that defines your desired `PGDGroup` resources that make up a PGD cluster.

Some sample files are included in the EDB Postgres Distributed for Kubernetes repository. The manifest contains the definition of a PGD cluster with two data groups and a global witness node spread across three regions. Each data group consists of two data nodes and a local witness node.

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-a
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
      create: true
    discovery:
      - host: region-a-group
      - host: region-b-group
      - host: region-c-group
  cnp:
    storage:
      size: 1Gi
  connectivity:
    dns:
      additional:
        - domain: alternate.domain
        - domain: my.domain
          hostSuffix: -dc1
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
```

(continues on next page)

(continued from previous page)

```
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
- --
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-b
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    discovery:
      - host: region-a-group
      - host: region-b-group
      - host: region-c-group
  cnp:
    storage:
      size: 1Gi
  connectivity:
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
- --
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-c
spec:
  instances: 0
  proxyInstances: 0
```

(continues on next page)

(continued from previous page)

```
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    discovery:
      - host: region-a-group
      - host: region-b-group
      - host: region-c-group
  cnp:
    storage:
      size: 1Gi
  connectivity:
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
```

You can deploy the `flexible-3-regions` example by saving it first and running:

```
kubectl apply -f flexible_3regions.yaml
```

You can check that the pods are being created using the `get pods` command:

```
kubectl get pods
```

The pods are being created as part of PGD nodes. As described in *Architecture* , they' re implemented on top of EDB Postgres for Kubernetes clusters.

You can list the clusters then, which shows the PGD nodes:

```
$ kubectl get clusters
NAME         AGE     INSTANCES   READY   STATUS                     PRIMARY
region-a-1   2m50s   1           1       Cluster in healthy state   region-a-1-1
region-a-2   118s    1           1       Cluster in healthy state   region-a-2-1
region-a-3   91s     1           1       Cluster in healthy state   region-a-3-1
…
…
```

Ultimately, the PGD nodes are created as part of the PGD groups that make up your PGD cluster.

```
$ kubectl get pgdgroups
NAME       DATA INSTANCES   WITNESS INSTANCES   PHASE                AGE
region-a   2                1                   PGDGroup - Healthy   4m50s
region-b   2                1                   PGDGroup - Healthy   4m50s
region-c   0                1                   PGDGroup - Healthy   4m50s
```

Notice how the region-c group is only a witness node.

CHAPTER 3

Configuration

3.1 Operator configuration

3.1.1 EDB Postgres Distributed for Kubernetes Operator configuration

The operator for EDB Postgres Distributed for Kubernetes (PGD4K) is installed from a standard deployment manifest and follows the convention over configuration paradigm. While this is fine in most cases, there are some scenarios where you want to change the default behavior, such as:

- defining a different default image for PGD or PGD proxy
- defining an additional pull secret for operator and operand images

The behavior of the operator can be customized through a `ConfigMap` /`Secret` that is located in the same namespace of the operator deployment and with `pgd-operator-controller-manager-config` as the name.

Important: Any change to the config's *ConfigMap* /*Secret* will not be automatically detected by the operator, - and as such, it needs to be reloaded (see below). Moreover, changes only apply to the resources created after the configuration is reloaded.

Important: The operator first processes the ConfigMap values and then the Secret' s, in this order. As a result, if a parameter is defined in both places, the one in the Secret will be used.

Available options

The operator looks for the following environment variables to be defined in the `ConfigMap` /`Secret` :

Name	Description
PGD_IMAGE_NAME	PGD operand image name to overwrite the default one in manifest.
PGD_PROXY_IMAGE_NAI	PGD proxy operand image name to overwrite the default one in manifest.
PULL_SECRET_NAME	Name of an additional pull secret to be defined in the operator's namespace and to be used to download images

When you specify an additional pull secret name using the `PULL_SECRET_NAME` parameter, the operator will use that secret to create a pull secret for every created PGD group. That secret will be named `<group-name>-pull` .

The namespace where the operator looks for the `PULL_SECRET_NAME` secret is where you installed the operator. If the operator is not able to find that secret, it will ignore the configuration parameter.

Defining an operator config map

The example below customizes the behavior of the operator, by defining an additional image pull secret.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: pgd-operator-controller-manager-config
  namespace: pgd-operator-system
data:
  PULL_SECRET_NAME: additional-pull-secret
```

Defining an operator secret

The example below customizes the behavior of the operator, by defining the default operand image names.

```
apiVersion: v1
kind: Secret
metadata:
  name: pgd-operator-controller-manager-config
  namespace: pgd-operator-system
type: Opaque
stringData:
  PGD_IMAGE_NAME: docker.enterprisedb.com/k8s_enterprise_pgd/postgresql-pgd:17.4-5.7.0-1
  PGD_PROXY_IMAGE_NAME: docker.enterprisedb.com/k8s_enterprise_pgd/edb-pgd-proxy:5.7.0-1
```

Restarting the operator to reload configs

For the change to be effective, you need to recreate the operator pods to reload the config map. If you have installed the operator on Kubernetes using the manifest you can do that by issuing:

```
kubectl rollout restart deployment \
    -n pgd-operator-system \
    pgd-operator-controller-manager
```

Otherwise, If you have installed the operator using OLM, or you are running on Openshift, run the following command specifying the namespace the operator is installed in:

```
kubectl delete pods -n [NAMESPACE_NAME_HERE] \
  -l app.kubernetes.io/name=pgd-operator
```

> **Warning:** Customizations will be applied only to *PGD groups* resources created after the reload of the operator deployment.

pprof HTTP Server

The operator can expose a PPROF HTTP server with the following endpoints on `localhost:6060` :

- `/debug/pprof/` . Responds to a request for "/debug/pprof/" with an HTML page listing the available profiles
- `/debug/pprof/cmdline` . Responds with the running program's command line, with arguments separated by NULL bytes.
- `/debug/pprof/profile` . Responds with the pprof-formatted cpu profile. Profiling lasts for duration specified in seconds GET parameter, or for 30 seconds if not specified.
- `/debug/pprof/symbol` . Looks up the program counters listed in the request, responding with a table mapping program counters to function names.
- `/debug/pprof/trace` . Responds with the execution trace in binary form. Tracing lasts for duration specified in seconds GET parameter, or for 1 second if not specified.

To enable the operator you need to edit the operator deployment add the flag `--pprof-server=true` .

You can do this by executing these commands:

```
kubectl edit deployment -n pgd-operator-system pgd-operator-controller-manager
```

Then on the edit page scroll down the container args and add `--pprof-server=true` , as in this example:

```
containers:
- args:
  - controller
  - --enable-leader-election
  - --config-map-name=pgd-operator-controller-manager-config
  - --secret-name=pgd-operator-controller-manager-config
  - --log-level=info
  - --pprof-server=true # relevant line
  command:
  - /manager
```

Save the changes; the deployment now will execute a roll-out, and the new pod will have the PPROF server enabled.

3.1.2 EDB Postgres for Kubernetes Operator configuration

The EDB Postgres Distributed for Kubernetes (PGD4K) operator manages each PGD node using the `Cluster` resource defined by the EDB Postgres for Kubernetes (PG4K) operator. Therefore, the configuration for the PG4K operator remains applicable. Please refer to the operator configuration

for more details.

3.2 Examples of configuration

Important: The available examples are for demonstration and experimentation purposes only.

These examples are configuration files for setting up your EDB Postgres Distributed cluster in a Kubernetes environment.

3.2.1 Basics

Flexible 3 regions

:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-a
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
      create: true
    discovery:
      - host: region-a-group
      - host: region-b-group
      - host: region-c-group
  cnp:
    storage:
      size: 1Gi
  connectivity:
    dns:
      additional:
        - domain: alternate.domain
        - domain: my.domain
          hostSuffix: -dc1
    tls:
```

(continues on next page)

(continued from previous page)

```
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
- --
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-b
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    discovery:
      - host: region-a-group
      - host: region-b-group
      - host: region-c-group
  cnp:
    storage:
      size: 1Gi
  connectivity:
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
```

(continues on next page)

(continued from previous page)

```
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
- --
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-c
spec:
  instances: 0
  proxyInstances: 0
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    discovery:
      - host: region-a-group
      - host: region-b-group
      - host: region-c-group
  cnp:
    storage:
      size: 1Gi
  connectivity:
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
```

: provides a PGD cluster with two data groups and a global witness node spread across three regions, where each data group consists of two data nodes and a local witness node.

Flexible 3 regions using physical join

:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-a
```

(continues on next page)

(continued from previous page)

```
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
      create: true
  cnp:
    joinMethod: physical
    storage:
      size: 1Gi
  connectivity:
    dns:
      additional:
        - domain: alternate.domain
        - domain: my.domain
          hostSuffix: -dc1
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
- --
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-b
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    # The first node in region-b will join to one of the node defined in discovery list␣
↪using physical join
    groupJoinMethod: physical
    # list the data node of region a to physical Join
```

(continues on next page)

(continued from previous page)

```
    discovery:
      - host: region-a-3-node
      - host: region-a-1-node
      - host: region-a-2-node
  cnp:
    joinMethod: physical
    storage:
      size: 1Gi
  connectivity:
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
- --
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-c
spec:
  instances: 0
  proxyInstances: 0
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    discovery:
      - host: region-a-group
      - host: region-b-group
      - host: region-c-group
  cnp:
    storage:
      size: 1Gi
  connectivity:
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
```

(continues on next page)

(continued from previous page)

```
        spec:
          issuerRef:
            name: client-ca-issuer
            kind: Issuer
            group: cert-manager.io
    serverCert:
      caCertSecret: server-ca-key-pair
      certManager:
        spec:
          issuerRef:
            name: server-ca-issuer
            kind: Issuer
            group: cert-manager.io
```

: provides a PGD cluster with two data groups and a global witness node just like `Flexible 3 regions` . The difference is the group join in this sample is using physical join. Group region-a is the first group to startup (`spec.pgd.parentGroup.create=true`), nodes in region-a join with each other using physical join. Group region-b define its `spec.pgd.groupJoinMethod=physical` , which means the first node in region-b will start and physical join a remote node, the join target is defined in `spec.pgd.discovery` section of group region-b, The rest node in region-b will join with each using physical join as `spec.cnp.joinMethod=physical` . region-c is a witness group which will always use logical join.

Flexible 3 regions with pre-provisioned client secrets

:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-a
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
      create: true
    discovery:
      - host: region-a-group
      - host: region-b-group
      - host: region-c-group
  cnp:
    storage:
      size: 1Gi
  connectivity:
    dns:
      additional:
        - domain: my.domain
          hostSuffix: -dc1
    tls:
      mode: verify-ca
      clientCert:
```

(continues on next page)

(continued from previous page)

```
      # if the pre-provisioned secrets is set, the caCertSecret will accept a CA secrets,
 which contains ca.crt or ca.key optionally
        caCertSecret: client-ca-secrets
        preProvisioned:
          streamingReplica:
            secretRef: replication-secrets-region-a
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
- --
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-b
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    discovery:
      - host: region-a-group
      - host: region-b-group
      - host: region-c-group
  cnp:
    storage:
      size: 1Gi
  connectivity:
    dns:
      additional:
        - domain: my.domain
          hostSuffix: -dc1
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-secrets
        preProvisioned:
          streamingReplica:
            secretRef: replication-secrets-region-b
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
```

(continues on next page)

(continued from previous page)

```
              group: cert-manager.io
- --
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-c
spec:
  instances: 0
  proxyInstances: 0
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    discovery:
      - host: region-a-group
      - host: region-b-group
      - host: region-c-group
  cnp:
    storage:
      size: 1Gi
  connectivity:
    dns:
      additional:
        - domain: my.domain
          hostSuffix: -dc1
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-secrets
        preProvisioned:
          streamingReplica:
            secretRef: replication-secrets-region-c
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
- --

###  This is the pre-provisioned client certificate which is signed by certificate in␣
↪client-ca-secrets

apiVersion: v1
data:
  tls.crt:␣
↪LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNYekNDQWdTZ0F3SUJBZ0lSQUw0N2V5eXEydzNEN0gwM2RPaE9iaFF3Q2dZSU
  tls.key:␣
↪LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlFcFFJQkFBS0NBUUVBNGRHejZsYktsUUFXa0J1K2pwRjgrM0t5RFBQZn
kind: Secret
```

(continues on next page)

(continued from previous page)

```
metadata:
  name: replication-secrets-region-a
type: Opaque
- --

###  This is the pre-provisioned client certificate which is signed by certificate in␣
↪client-ca-secrets

apiVersion: v1
data:
  tls.crt:␣
↪LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNYekNDQWdTZ0F3SUJBZ0lSQUw0N2V5eXEydzNEN0gwM2RPaE9iaFF3Q2dZSU
  tls.key:␣
↪LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlFcFFJQkFBS0NBUUVBNGRHejZsYktsUUFXa0J1K2pwRjgrM0t5RFBQZn
kind: Secret
metadata:
  name: replication-secrets-region-b
type: kubernetes.io/tls
- --

###  This is the pre-provisioned client certificate which is signed by certificate in␣
↪client-ca-secrets

apiVersion: v1
data:
  tls.crt:␣
↪LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNYekNDQWdTZ0F3SUJBZ0lSQUw0N2V5eXEydzNEN0gwM2RPaE9iaFF3Q2dZSU
  tls.key:␣
↪LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlFcFFJQkFBS0NBUUVBNGRHejZsYktsUUFXa0J1K2pwRjgrM0t5RFBQZn
kind: Secret
metadata:
  name: replication-secrets-region-c
type: kubernetes.io/tls
- --

###  This is the client CA certificate, which will be used to sign other client␣
↪certificate

apiVersion: v1
data:
  ca.crt:␣
↪LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUJoakNDQVN1Z0F3SUJBZ0lSQU9ZYmNIcmxuSVgzcU55bi8wWkNSem93Q2dZSU
kind: Secret
metadata:
  name: client-ca-secrets
type: Opaque
```

: provides a PGD cluster with two data groups and a global witness node just like `Flexible 3 regions` . In each PGD group, the server TLS certificate is managed by the operator and cert-manager. The client replication certificate is pre-provisioned. Here are the steps to set up this sample:

1. Establish the self-signed issuer and certificate in target namespace. Since we are using a pre-provisioned client replication certificate, we can delete the client secrets, certificate, and issuer created by `issuer-selfsigned`.

yaml .

```
kubectl -n <namespace> apply -f issuer-selfsigned.yaml
kubectl -n <namespace> delete secrets/client-ca-key-pair \
    certificate/client-ca \
    issuer/client-ca-issuer
```

2. Create the PGD group. The pre-provisioned client certificate and its CA certificate are included in the `flexible_3regions_previsioned_secrets.yaml` file as well.

```
kubectl -n <namespace> apply -f flexible_3regions_previsioned_secrets.yaml
```

Flexible 3 regions in different namespaces

:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-a
  namespace: region-a
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
      create: true
    discovery:
    # need list the full discovery name for the group
      - host: region-a-group.region-a.svc.cluster.local
      - host: region-b-group.region-b.svc.cluster.local
      - host: region-c-group.region-c.svc.cluster.local
  cnp:
    storage:
      size: 1Gi
  connectivity:
    dns:
      # we need to configure the domain for the group so it could be resolved
      domain: region-a.svc.cluster.local
      additional:
        - domain: my.domain
          hostSuffix: -dc1
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
```

(continues on next page)

(continued from previous page)

```
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
- --
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-b
  namespace: region-b
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    discovery:
    # need list the full discovery name for the group
      - host: region-a-group.region-a.svc.cluster.local
      - host: region-b-group.region-b.svc.cluster.local
      - host: region-c-group.region-c.svc.cluster.local
  cnp:
    storage:
      size: 1Gi
  connectivity:
    dns:
    # we need to configure the domain for the group so it could be resolved
      domain: region-b.svc.cluster.local
      additional:
        - domain: my.domain
          hostSuffix: -dc1
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
```

(continues on next page)

(continued from previous page)

```
              kind: Issuer
              group: cert-manager.io
- --
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-c
  namespace: region-c
spec:
  instances: 0
  proxyInstances: 0
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    discovery:
    # need list the full discovery name for the group
      - host: region-a-group.region-a.svc.cluster.local
      - host: region-b-group.region-b.svc.cluster.local
      - host: region-c-group.region-c.svc.cluster.local
  cnp:
    storage:
      size: 1Gi
  connectivity:
    dns:
    # we need to configure the domain for the group so it could be resolved
      domain: region-c.svc.cluster.local
      additional:
        - domain: my.domain
          hostSuffix: -dc1
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
```

: provides a sample that builds upon the `Flexible 3 regions` example. Each region is assigned a different namespace. The `discovery` and `connectivity` section are changed in accordance with the namespaces. To setup this sample, we need to pre-create the `client-ca-key-pair` and `server-ca-key-pair` secrets in each namespace utilizing

private keys generated with the ECDSA algorithm. This ensures that the server TLS certificates and client replication certificates generated by `cert-manager` are signed from the same private key.

- Create the namespaces `region-a` , `region-b` and `region-c` .
- Create the CA secrets using

```
##  Those two secrets are used to store the ECDSA private key for the client and server␣
↪CA pairs used by certificate

##  in self-signed issuer. The client-ca-key-pair and server-ca-key-pair is finally used␣
↪in sample flexible_3regions_3ns.yaml

##  This is to make sure in different namespaces, we have same CA certificate

apiVersion: v1
data:
tls.key:␣
↪LS0tLS1CRUdJTiBFQyBQUklWQVRFIEtFWS0tLS0tCk1IY0NBUUVFSUNOZW1KbHplK3VPbFR1UHltWHZLWHN0aFlUSkRRSkpIQzJsQ
kind: Secret
metadata:
name: client-ca-key-pair
type: Opaque

- --

apiVersion: v1
data:
tls.key:␣
↪LS0tLS1CRUdJTiBFQyBQUklWQVRFIEtFWS0tLS0tCk1IY0NBUUVFSUZqVXc5b2hQNVl2blo0RkRML1ZQV092ODhRUGpRb2xHYXFCL
kind: Secret
metadata:
name: server-ca-key-pair
type: Opaque
```

in each namespace for the certificates.

```
kubectl -n region-a apply -f issuer-ecdsa-key.yaml
kubectl -n region-b apply -f issuer-ecdsa-key.yaml
kubectl -n region-c apply -f issuer-ecdsa-key.yaml
```

- Create the self-signed issuer and corresponding certificate in each namespace. The CA secrets `client-ca-key-pair` and `server-ca-key-pair` will be refreshed with `ca.crt` .

```
kubectl -n region-a apply -f issuer-selfsigned.yaml
kubectl -n region-b apply -f issuer-selfsigned.yaml
kubectl -n region-c apply -f issuer-selfsigned.yaml
```

- Create the PGD group

```
kubectl apply -f flexible_3regions_3ns.yaml
```

parted group cleanup

:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroupCleanup
metadata:
  name: region-b-cleanup
spec:
  executor: region-a
  target: region-b
```

: provides a sample of using PGDGroupCleanup to clean up the metadata of `region-b` . The cleanup is run from `region-a` . All nodes belonging to `region-b` need to be in `PARTED` status before running this CR.

3.2.2 Backup and restore

sample group with two schedulers :

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: group-example-with-2schedulers
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
      create: true
  cnp:
    storage:
      storageClass: csi-hostpath-sc
      size: 1Gi
  connectivity:
    dns:
      additional:
        - domain: alternate.domain
        - domain: my.domain
          hostSuffix: -dc1
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
```

(continues on next page)

(continued from previous page)

```
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
  backup:
    schedulers:
  # volumeSnapshot scheduler backup
      - method: volumeSnapshot
        immediate: true
        schedule: "0 0 8 * * *"
  # barmanObjectStore scheduler backup
      - method: barmanObjectStore
        immediate: true
        schedule: "0 0 9 * * *"
    configuration:
  # backup configuration for volumeSnapshot
      volumeSnapshot:
        className: csi-hostpath-snapclass
  # backup configuration for barmanObjectStore
      barmanObjectStore:
        destinationPath: s3://backups/
        endpointURL: https://minio-service.minio:9000
        s3Credentials:
          accessKeyId:
            name: minio-secrets
            key: ACCESS_KEY_ID
          secretAccessKey:
            name: minio-secrets
            key: ACCESS_SECRET_KEY
        wal:
          compression: gzip
          maxParallel: 8
        data:
          jobs: 4
          immediateCheckpoint: true
```

: provides a PGD group sample with two scheduled backups configured: one using volumeSnapshot, the other using barmanObjectStore.

sample group with backup and restore using ``barmanObjectStore``

:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: group-example-with-barman-backup
spec:
  instances: 3
  proxyInstances: 2
  witnessInstances: 0
  pgd:
    parentGroup:
      name: world
```

(continues on next page)

(continued from previous page)

```
    create: true
  cnp:
    storage:
      size: 1Gi
    walStorage:
      size: 2Gi
  connectivity:
    dns:
      additional:
        - domain: alternate.domain
        - domain: my.domain
          hostSuffix: -dc1
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
  backup:
    schedulers:
      - method: barmanObjectStore
        immediate: true
        schedule: "0 0 9 * * *"
    configuration:
      barmanObjectStore:
        destinationPath: s3://backups/
        endpointURL: http://minio-service.minio:9000
        s3Credentials:
          accessKeyId:
            name: minio-secrets
            key: ACCESS_KEY_ID
          secretAccessKey:
            name: minio-secrets
            key: ACCESS_SECRET_KEY
        wal:
          compression: gzip
          maxParallel: 8
        data:
          jobs: 4
          immediateCheckpoint: true
```

: provides a backup sample for a three-node PGD group. This setup includes scheduled backups and continuous WAL archiving to barmanObjectStore.

:

```
#  group-example-with-barman-restore-a is configured to full restore from

#  backups in barmanObjectStore, will restore from one of servername

#  group-example-with-barman-bak-1, group-example-with-barman-bak-2 or group-example-
↪with-barman-bak-3

apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
name: group-example-with-barman-restore-a
spec:
instances: 3
proxyInstances: 2
witnessInstances: 0
pgd:
parentGroup:
name: world
create: true
cnp:
storage:
size: 1Gi
walStorage:
size: 2Gi
connectivity:
dns:
additional: - domain: alternate.domain - domain: my.domain
hostSuffix: -dc1
tls:
mode: verify-ca
clientCert:
caCertSecret: client-ca-key-pair
certManager:
spec:
issuerRef:
name: client-ca-issuer
kind: Issuer
group: cert-manager.io
serverCert:
caCertSecret: server-ca-key-pair
certManager:
spec:
issuerRef:
name: server-ca-issuer
kind: Issuer
group: cert-manager.io
restore:
```

(continues on next page)

(continued from previous page)

```
#  List all the serverNames, so pgdgroup will find a more suitable server to restore

    serverNames:
      - group-example-with-barman-backup-1
      - group-example-with-barman-backup-2
      - group-example-with-barman-backup-3
    barmanObjectStore:
      destinationPath: s3://backups/
      endpointURL: http://minio-service.minio:9000
      s3Credentials:
        accessKeyId:
          name: minio-secrets
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: minio-secrets
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
        maxParallel: 8
      data:
        jobs: 4
        immediateCheckpoint: true

#  The restored pgdgroup also configured with backup, so that it can be used for future␣
↪restore

#  the backup here use a different serverName

backup:
schedulers: - method: barmanObjectStore
immediate: true
schedule: "0 0 9 \* \* \*"
configuration:
barmanObjectStore:
destinationPath: s3://backups/
endpointURL: http://minio-service.minio:9000
s3Credentials:
accessKeyId:
name: minio-secrets
key: ACCESS_KEY_ID
secretAccessKey:
name: minio-secrets
key: ACCESS_SECRET_KEY
wal:
compression: gzip
maxParallel: 8
data:
jobs: 4
immediateCheckpoint: true

- --
```

(continues on next page)

(continued from previous page)

```
#  group-example-with-barman-restore-b is a three-node data group, it is created from

#  scratch and configured to join group-example-with-barman-restore-a and group-example-
↪with-barman-restore-c

apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
name: group-example-with-barman-restore-b
spec:
instances: 3
proxyInstances: 2
witnessInstances: 0
pgd:
parentGroup:
name: world
discovery: - host: group-example-with-barman-restore-a-group - host: group-example-with-
↪barman-restore-b-group - host: group-example-with-barman-restore-c-group
cnp:
storage:
size: 1Gi
walStorage:
size: 2Gi
connectivity:
dns:
additional: - domain: alternate.domain - domain: my.domain
hostSuffix: -dc1
tls:
mode: verify-ca
clientCert:
caCertSecret: client-ca-key-pair
certManager:
spec:
issuerRef:
name: client-ca-issuer
kind: Issuer
group: cert-manager.io
serverCert:
caCertSecret: server-ca-key-pair
certManager:
spec:
issuerRef:
name: server-ca-issuer
kind: Issuer
group: cert-manager.io

#  The restored pgdgroup also configured with backup, so that it can be used for future␣
↪restore

#  the backup here use a different serverName

backup:
```

(continues on next page)

(continued from previous page)

```
schedulers: - method: barmanObjectStore
immediate: true
schedule: "0 0 9 \* \* \*"
configuration:
barmanObjectStore:
destinationPath: s3://backups/
endpointURL: http://minio-service.minio:9000
s3Credentials:
accessKeyId:
name: minio-secrets
key: ACCESS_KEY_ID
secretAccessKey:
name: minio-secrets
key: ACCESS_SECRET_KEY
wal:
compression: gzip
maxParallel: 8
data:
jobs: 4
immediateCheckpoint: true

- --

#  group-example-with-barman-restore-c is a witness group, it will not have any instances

#  it is configured to join group-example-with-barman-restore-a or group-example-with-
↪barman-restore-b

apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
name: group-example-with-barman-restore-c
spec:
instances: 0
proxyInstances: 0
witnessInstances: 1
pgd:
parentGroup:
name: world
discovery: - host: group-example-with-barman-restore-a-group - host: group-example-with-
↪barman-restore-b-group - host: group-example-with-barman-restore-c-group
cnp:
storage:
size: 1Gi
walStorage:
size: 2Gi
connectivity:
dns:
additional: - domain: alternate.domain - domain: my.domain
hostSuffix: -dc1
tls:
mode: verify-ca
```

(continues on next page)

(continued from previous page)

```
clientCert:
caCertSecret: client-ca-key-pair
certManager:
spec:
issuerRef:
name: client-ca-issuer
kind: Issuer
group: cert-manager.io
serverCert:
caCertSecret: server-ca-key-pair
certManager:
spec:
issuerRef:
name: server-ca-issuer
kind: Issuer
group: cert-manager.io
```

: offers a restore sample for three regions PGD groups. In the first region, group `group-example-with-barman-restore-a` is restored from backups, and parent group `world` is created in this group. In the second and third region, group `group-example-with-barman-restore-b` and `group-example-with-barman-restore-c` are created from scratch, and respectively join the restored group.

sample group with backup and restore using ``volumeSnapshot`` :

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
name: group-example-with-vs-backup
spec:
instances: 2
proxyInstances: 2
witnessInstances: 1
pgd:
parentGroup:
name: world
create: true
cnp:
storage:
storageClass: csi-hostpath-sc
size: 1Gi
walStorage:
storageClass: csi-hostpath-sc
size: 2Gi
tablespaces: - name: ts1
storage:
storageClass: csi-hostpath-sc
size: 1Gi - name: ts2
storage:
storageClass: csi-hostpath-sc
size: 1Gi
connectivity:
dns:
```

(continues on next page)

(continued from previous page)

```
additional: - domain: alternate.domain - domain: my.domain
hostSuffix: -dc1
tls:
mode: verify-ca
clientCert:
caCertSecret: client-ca-key-pair
certManager:
spec:
issuerRef:
name: client-ca-issuer
kind: Issuer
group: cert-manager.io
serverCert:
caCertSecret: server-ca-key-pair
certManager:
spec:
issuerRef:
name: server-ca-issuer
kind: Issuer
group: cert-manager.io
backup:
schedulers: - method: volumeSnapshot
immediate: true
schedule: "0 0 8 \* \* \*"
configuration:
volumeSnapshot:
className: csi-hostpath-snapclass
barmanObjectStore:
destinationPath: s3://backups/
endpointURL: http://minio-service.minio:9000
s3Credentials:
accessKeyId:
name: minio-secrets
key: ACCESS_KEY_ID
secretAccessKey:
name: minio-secrets
key: ACCESS_SECRET_KEY
wal:
compression: gzip
maxParallel: 8
data:
jobs: 4
immediateCheckpoint: true
```

: specifies a three-node PGD group configured with scheduled volume snapshot backup and continuous WAL archiving to barmanObjectStore.

:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
```

(continues on next page)

(continued from previous page)

```
name: group-example-with-vs-restore
spec:
instances: 2
proxyInstances: 2
witnessInstances: 1
pgd:
parentGroup:
name: world
create: true
cnp:
storage:
storageClass: csi-hostpath-sc
size: 1Gi
walStorage:
storageClass: csi-hostpath-sc
size: 2Gi
tablespaces: - name: ts1
storage:
storageClass: csi-hostpath-sc
size: 1Gi - name: ts2
storage:
storageClass: csi-hostpath-sc
size: 1Gi
connectivity:
dns:
additional: - domain: alternate.domain - domain: my.domain
hostSuffix: -dc1
tls:
mode: verify-ca
clientCert:
caCertSecret: client-ca-key-pair
certManager:
spec:
issuerRef:
name: client-ca-issuer
kind: Issuer
group: cert-manager.io
serverCert:
caCertSecret: server-ca-key-pair
certManager:
spec:
issuerRef:
name: server-ca-issuer
kind: Issuer
group: cert-manager.io
restore:
volumeSnapshots:
selector:
matchLabels: ## we select all the backups from the group-example-with-vs-backup group
"k8s.pgd.enterprisedb.io/group": group-example-with-vs-backup
barmanObjectStore:
destinationPath: s3://backups/
```

(continues on next page)

(continued from previous page)

```
endpointURL: http://minio-service.minio:9000
s3Credentials:
accessKeyId:
name: minio-secrets
key: ACCESS_KEY_ID
secretAccessKey:
name: minio-secrets
key: ACCESS_SECRET_KEY
wal:
compression: gzip
maxParallel: 8
data:
jobs: 4
immediateCheckpoint: true

##  This is the backup configuration for the restored pgdgroup

backup:
configuration:
volumeSnapshot:
className: csi-hostpath-snapclass
barmanObjectStore:
destinationPath: s3://backups/
endpointURL: http://minio-service.minio:9000
s3Credentials:
accessKeyId:
name: minio-secrets
key: ACCESS_KEY_ID
secretAccessKey:
name: minio-secrets
key: ACCESS_SECRET_KEY
wal:
compression: gzip
maxParallel: 8
data:
jobs: 4
immediateCheckpoint: true
```

: defines full restore from volume snapshot backup.

:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: group-example-with-vs-pitr
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
```

(continues on next page)

(continued from previous page)

```
    name: world
    create: true
  cnp:
    storage:
      storageClass: csi-hostpath-sc
      size: 1Gi
    walStorage:
      storageClass: csi-hostpath-sc
      size: 2Gi
    tablespaces:
      - name: ts1
        storage:
          storageClass: csi-hostpath-sc
          size: 1Gi
      - name: ts2
        storage:
          storageClass: csi-hostpath-sc
          size: 1Gi
  connectivity:
    dns:
      additional:
        - domain: alternate.domain
        - domain: my.domain
          hostSuffix: -dc1
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
  restore:
    recoveryTarget:
      targetTime: "2024-09-10 00:20:12.213757"
    volumeSnapshots:
      selector:
        matchLabels:
          ## we select all the backups from the group-example-with-vs-backup group
          "k8s.pgd.enterprisedb.io/group": group-example-with-vs-backup
    barmanObjectStore:
      destinationPath: s3://backups/
```

(continues on next page)

(continued from previous page)

```
    endpointURL: http://minio-service.minio:9000
    s3Credentials:
      accessKeyId:
        name: minio-secrets
        key: ACCESS_KEY_ID
      secretAccessKey:
        name: minio-secrets
        key: ACCESS_SECRET_KEY
    wal:
      compression: gzip
      maxParallel: 8
    data:
      jobs: 4
      immediateCheckpoint: true
## This is the backup configuration for the restored pgdgroup
backup:
  configuration:
    volumeSnapshot:
      className: csi-hostpath-snapclass
    barmanObjectStore:
      destinationPath: s3://backups/
      endpointURL: http://minio-service.minio:9000
      s3Credentials:
        accessKeyId:
          name: minio-secrets
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: minio-secrets
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
        maxParallel: 8
      data:
        jobs: 4
        immediateCheckpoint: true
```

: defines restore from volume snapshot backup followed by point-in-time recovery.

Note: The *volumeSnapshot* sample above utilizes the *csi-hostpath-sc* storage class. Please verify that your storage class supports volume snapshots. For more details, refer to *Backup on volume snapshots* .

3.2.3 Read node routing

sample group with read node routing enabled :

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
name: group-example-with-readnode-routing
spec:
instances: 2
proxyInstances: 2
witnessInstances: 1
pgd:
proxySettings:
enableReadNodeRouting: true
parentGroup:
name: world
create: true
cnp:
storage:
size: 1Gi
connectivity:
proxyReadServiceTemplate:
metadata:
annotations:
annotationName: "annotationValue"
dns:
additional: - domain: alternate.domain - domain: my.domain
hostSuffix: -dc1
tls:
mode: verify-ca
clientCert:
caCertSecret: client-ca-key-pair
certManager:
spec:
issuerRef:
name: client-ca-issuer
kind: Issuer
group: cert-manager.io
serverCert:
caCertSecret: server-ca-key-pair
certManager:
spec:
issuerRef:
name: server-ca-issuer
kind: Issuer
group: cert-manager.io
```

: offers a sample with read node routing enabled.

3.2.4 LDAP

sample group use ldap with bind and search :

```
apiVersion: v1
data:
password: YWRtaW4=
kind: Secret
metadata:
name: ldap-bind-password
type: Opaque

- --

apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
name: group-example-with-ldap-bind-search
spec:
instances: 2
proxyInstances: 2
witnessInstances: 1
pgd:
parentGroup:
name: world
create: true
cnp:
postgresql:
ldap:
server: ldap-service.ldap.svc.cluster.local
bindSearchAuth:
baseDN: "dc=example,dc=org"
bindDN: "cn=admin,dc=example,dc=org"
searchAttribute: "uid"
bindPassword:
name: ldap-bind-password
key: password
storage:
size: 1Gi
connectivity:
dns:
additional: - domain: alternate.domain - domain: my.domain
hostSuffix: -dc1
tls:
mode: verify-ca
clientCert:
caCertSecret: client-ca-key-pair
certManager:
spec:
issuerRef:
name: client-ca-issuer
kind: Issuer
group: cert-manager.io
```

(continues on next page)

(continued from previous page)

```
serverCert:
caCertSecret: server-ca-key-pair
certManager:
spec:
issuerRef:
name: server-ca-issuer
kind: Issuer
group: cert-manager.io
backup:
schedulers: - method: barmanObjectStore
immediate: true
schedule: "0 0 9 \* \* \*"
configuration:
barmanObjectStore:
destinationPath: s3://backups/
endpointURL: https://minio-service.minio:9000
s3Credentials:
accessKeyId:
name: minio-secrets
key: ACCESS_KEY_ID
secretAccessKey:
name: minio-secrets
key: ACCESS_SECRET_KEY
wal:
compression: gzip
maxParallel: 8
data:
jobs: 4
immediateCheckpoint: true
```

: provides an LDAP sample of PGD group configured to use `bind` and `search` for authentication.

sample group use ldap with simple bind

:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
name: group-example-with-ldap-simple-bind
spec:
instances: 2
proxyInstances: 2
witnessInstances: 1
pgd:
parentGroup:
name: world
create: true
cnp:
postgresql:
ldap:
server: ldap-service.ldap.svc.cluster.local
bindAsAuth:
```

(continues on next page)

(continued from previous page)

```
prefix: "uid="
suffix: ",dc=example,dc=org"
storage:
size: 1Gi
connectivity:
dns:
additional: - domain: alternate.domain - domain: my.domain
hostSuffix: -dc1
tls:
mode: verify-ca
clientCert:
caCertSecret: client-ca-key-pair
certManager:
spec:
issuerRef:
name: client-ca-issuer
kind: Issuer
group: cert-manager.io
serverCert:
caCertSecret: server-ca-key-pair
certManager:
spec:
issuerRef:
name: server-ca-issuer
kind: Issuer
group: cert-manager.io
backup:
schedulers: - method: barmanObjectStore
immediate: true
schedule: "0 0 9 \* \* \*"
configuration:
barmanObjectStore:
destinationPath: s3://backups/
endpointURL: https://minio-service.minio:9000
s3Credentials:
accessKeyId:
name: minio-secrets
key: ACCESS_KEY_ID
secretAccessKey:
name: minio-secrets
key: ACCESS_SECRET_KEY
wal:
compression: gzip
maxParallel: 8
data:
jobs: 4
immediateCheckpoint: true
```

: provides an LDAP sample of PGD group to use simple bind for authentication.

3.2.5 Managed roles

sample group with managed roles

:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
name: group-example-with-managed
spec:
instances: 2
proxyInstances: 2
witnessInstances: 1
pgd:
parentGroup:
name: world
create: true
cnp:
storage:
size: 1Gi
managed:
roles: - name: foo
comment: Foo
ensure: present
connectionLimit: 100
login: true
superuser: true
createdb: true
createrole: true
replication: true
inherit: true - name: bar
ensure: present
inRoles: - foo
login: true
services:
disabledDefaultServices: - ro - r
connectivity:
dns:
additional: - domain: alternate.domain - domain: my.domain
hostSuffix: -dc1
tls:
mode: verify-ca
clientCert:
caCertSecret: client-ca-key-pair
certManager:
spec:
issuerRef:
name: client-ca-issuer
kind: Issuer
group: cert-manager.io
serverCert:
caCertSecret: server-ca-key-pair
```

(continues on next page)

(continued from previous page)

```
certManager:
spec:
issuerRef:
name: server-ca-issuer
kind: Issuer
group: cert-manager.io
```

: provides a PGD group sample with managed roles and managed services.

3.2.6 TDE

sample group using tde

:

```
- --

apiVersion: v1
kind: Secret
metadata:
name: tde-key
data:
key: bG9zcG9sbGl0b3NkaWNlbnBpb3Bpb3Bpb2N1YW5kb3RpZW5lbmhhbWJyZWN1YW5kb3RpZW5lbmZyaW8=

- --

apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
name: group-example-with-tde
spec:
instances: 2
proxyInstances: 2
witnessInstances: 1
imageName: docker.enterprisedb.com/k8s_enterprise_pgd/edb-postgres-advanced-pgd:16.4.1-5.
↪5.1
pgdProxy:
imageName: docker.enterprisedb.com/k8s_enterprise_pgd/edb-pgd-proxy:5.5.0
pgd:
parentGroup:
name: world
create: true
cnp:
postgresql:
epas:
tde:
enabled: true
secretKeyRef:
name: tde-key
key: key
storage:
```

(continues on next page)

(continued from previous page)

```
size: 1Gi
connectivity:
dns:
additional: - domain: alternate.domain - domain: my.domain
hostSuffix: -dc1
tls:
mode: verify-ca
clientCert:
caCertSecret: client-ca-key-pair
certManager:
spec:
issuerRef:
name: client-ca-issuer
kind: Issuer
group: cert-manager.io
serverCert:
caCertSecret: server-ca-key-pair
certManager:
spec:
issuerRef:
name: server-ca-issuer
kind: Issuer
group: cert-manager.io
```

: provides a PGD group sample with TDE enabled. Since TDE requires PostgreSQL to be a specific flavor and version, the YAML file includes configurations for both the PGD and PGD proxy images.

3.2.7 Configurations

sample group with mutations :

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
name: group-example-with-mutations
spec:
instances: 2
proxyInstances: 2
witnessInstances: 1
pgd:
parentGroup:
name: world
create: true
mutations: - isApplied: - |
WITH node_route_config AS (
SELECT node_name, route_dsn
FROM bdr.node_routing_config_summary
)
SELECT NOT EXISTS(SELECT \* FROM node_route_config WHERE regexp_count(route_dsn,␣
↪application_name=) >= 1);
exec: - |
```

(continues on next page)

(continued from previous page)

```
WITH node_route_config AS (
SELECT node_name,
regexp_replace(route_dsn, concat( application_name=, node_name), ) as route_dsn
FROM bdr.node_routing_config_summary
)
SELECT bdr.alter_node_option(node_name, route_dsn, route_dsn)
FROM node_route_config;
cnp:
storage:
size: 1Gi
connectivity:
dns:
additional: - domain: alternate.domain - domain: my.domain
hostSuffix: -dc1
tls:
mode: verify-ca
clientCert:
caCertSecret: client-ca-key-pair
certManager:
spec:
issuerRef:
name: client-ca-issuer
kind: Issuer
group: cert-manager.io
serverCert:
caCertSecret: server-ca-key-pair
certManager:
spec:
issuerRef:
name: server-ca-issuer
kind: Issuer
group: cert-manager.io
```

: offers a PGD group sample with `always` mutation configured.

sample group with service template configured :

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
name: group-example-service-template
spec:
instances: 3
proxyInstances: 2
pgd:
parentGroup:
create: true
name: world
proxySettings:
enableReadNodeRouting: true
inheritedMetadata:
labels:
```

(continues on next page)

(continued from previous page)

```
region: zeta
annotations:
region.domain.com: region-domain
cnp:
storage:
size: 1Gi
connectivity:
dns:
additional: - domain: alternate.domain - domain: my.domain
hostSuffix: -dc1
tls:
mode: verify-ca
clientCert:
caCertSecret: client-ca-key-pair
certManager:
spec:
issuerRef:
name: client-ca-issuer
kind: Issuer
group: cert-manager.io
serverCert:
caCertSecret: server-ca-key-pair
certManager:
spec:
issuerRef:
name: server-ca-issuer
kind: Issuer
group: cert-manager.io
groupServiceTemplate:
metadata:
labels:
application: test-app
annotations:
test.domain.com: domain-value
nodeServiceTemplate:
metadata:
labels:
application: test-app
annotations:
test.domain.com: domain-value
proxyReadServiceTemplate:
metadata:
annotations:
annotationName: "annotationValue"
proxyServiceTemplate:
metadata:
labels:
application: test-app
annotations:
test.domain.com: domain-value
```

: offers a sample with `groupSeviceTemplate` , `nodeServiceTemplate` `proxyServiceTemplate` and

`proxyReadServiceTemplate` configured.

sample group with operand image customized :

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
name: group-example-customized
spec:
instances: 2
proxyInstances: 2
witnessInstances: 1
imageName: docker.enterprisedb.com/k8s_enterprise_pgd/edb-postgres-extended-pgd:17.4-5.7.
↪0-1
imagePullSecrets:


- name: registry-pullsecret
  pgd:
  parentGroup:
  name: world
  create: true
  cnp:
  storage:
  size: 1Gi
  pgdProxy:
  imageName: docker.enterprisedb.com/k8s_enterprise_pgd/edb-pgd-proxy:5.7.0-1
  connectivity:
  dns:
  additional: - domain: alternate.domain - domain: my.domain
  hostSuffix: -dc1
  tls:
  mode: verify-ca
  clientCert:
  caCertSecret: client-ca-key-pair
  certManager:
  spec:
  issuerRef:
  name: client-ca-issuer
  kind: Issuer
  group: cert-manager.io
  serverCert:
  caCertSecret: server-ca-key-pair
  certManager:
  spec:
  issuerRef:
  name: server-ca-issuer
  kind: Issuer
  group: cert-manager.io
```

: offers a PGD group sample with pgd and proxy image name customized.

sample group with operand image customized using ImageCatalog :

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: ImageCatalog
metadata:
name: postgresql-pgd-catalog
spec:
pgdImages:


-  image: docker.enterprisedb.com/k8s_standard_pgd/postgresql-pgd:15.10-5.6.1-1
  major: 15

-  image: docker.enterprisedb.com/k8s_standard_pgd/postgresql-pgd:16.6-5.6.1-1
  major: 16
  proxyImage: docker.enterprisedb.com/k8s_standard_pgd/edb-pgd-proxy:5.6.0-ubi9

- --

apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
name: group-example-catalog
spec:
instances: 2
proxyInstances: 2
witnessInstances: 1
imageCatalogRef:
apiGroup: pgd.k8s.enterprisedb.io
kind: ImageCatalog
major: 16
name: postgresql-pgd-catalog
imagePullSecrets:


-  name: registry-pullsecret
  pgd:
  parentGroup:
  name: world
  create: true
  cnp:
  storage:
  size: 1Gi
  connectivity:
  dns:
  additional: - domain: alternate.domain - domain: my.domain
  hostSuffix: -dc1
  tls:
  mode: verify-ca
  clientCert:
  caCertSecret: client-ca-key-pair
  certManager:
  spec:
  issuerRef:
```

(continues on next page)

(continued from previous page)

```
  name: client-ca-issuer
  kind: Issuer
  group: cert-manager.io
  serverCert:
  caCertSecret: server-ca-key-pair
  certManager:
  spec:
  issuerRef:
  name: server-ca-issuer
  kind: Issuer
  group: cert-manager.io
```

: offers a PGD group sample with pgd and proxy image customized using ImageCatalog.

sample group with scale up using physical join

:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
name: group-example-with-physical-local-join
spec:
instances: 2
proxyInstances: 2
witnessInstances: 1
pgd:
parentGroup:
name: world
create: true
cnp:
joinMethod: physical
storage:
size: 1Gi
connectivity:
dns:
additional: - domain: alternate.domain - domain: my.domain
hostSuffix: -dc1
tls:
mode: verify-ca
clientCert:
caCertSecret: client-ca-key-pair
certManager:
spec:
issuerRef:
name: client-ca-issuer
kind: Issuer
group: cert-manager.io
serverCert:
caCertSecret: server-ca-key-pair
certManager:
spec:
issuerRef:
```

(continues on next page)

(continued from previous page)

```
name: server-ca-issuer
kind: Issuer
group: cert-manager.io
```

: offers a PGD group sample with physical join configured.

For a list of available options, see the *API reference* .

Note: The PGD group sample above requires cert-manager with a self-signed issuer. You can use

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
name: selfsigned-issuer
spec:
selfSigned: {}

- --

apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
name: server-ca
spec:
isCA: true
commonName: my-selfsigned-server-ca
secretName: server-ca-key-pair
privateKey:
algorithm: ECDSA
size: 256
issuerRef:
name: selfsigned-issuer
kind: Issuer
group: cert-manager.io

- --

apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
name: server-ca-issuer
spec:
ca:
secretName: server-ca-key-pair

- --

apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
name: client-ca
```

(continues on next page)

(continued from previous page)

```
spec:
isCA: true
commonName: my-selfsigned-client-ca
secretName: client-ca-key-pair
privateKey:
algorithm: ECDSA
size: 256
issuerRef:
name: selfsigned-issuer
kind: Issuer
group: cert-manager.io

- --

apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
name: client-ca-issuer
spec:
ca:
secretName: client-ca-key-pair
```

to create the self-signed issuer before setting up the PGD group.

CHAPTER 4

Using

4.1 Managing EDB Postgres Distributed (PGD) databases

As described in the *architecture document* , EDB Postgres Distributed for Kubernetes is an operator created to deploy PGD databases. By leveraging the Kubernetes ecosystem, it can offer self-healing and declarative control. The operator is also responsible of the backup and restore operations. See *Backup and restore* .

However, many of the operations and control of PGD clusters aren't managed by the operator. The pods created by EDB Postgres Distributed for Kubernetes come with the PGD CLI installed. You can use this tool, for example, to execute a switchover.

4.1.1 PGD CLI

> **Warning:** Don't use the PGD CLI to create and delete resources. For example, avoid the *create-proxy* and *delete-proxy* commands. Provisioning of resources is under the control of the operator, and manual creation and deletion isn't supported.

As an example, execute a switchover command.

We recommend that you use the PGD CLI from proxy pods. To find them, get a pod listing for your cluster:

```
kubectl get pods -n my-namespace

NAME                    READY   STATUS    RESTARTS      AGE
location-a-1-1          1/1     Running   0             2h
location-a-2-1          1/1     Running   0             2h
location-a-3-1          1/1     Running   0             2h
location-a-proxy-0      1/1     Running   0             2h
location-a-proxy-1      1/1     Running   0             2h
```

The proxy nodes have `proxy` in the name. Choose one, and get a command prompt in it:

```
kubectl exec -n my-namespace -ti location-a-proxy-0 -- bash
```

You now have a bash session open with the proxy pod. The `pgd` command is available:

```
pgd

Available Commands:
  check-health       Checks the health of the EDB Postgres Distributed cluster.
  <- snipped ->
  switchover         Switches over to new write leader.
  <- snipped ->
```

You can easily move your way through getting the information needed for the switchover:

```
pgd switchover --help

  $ pgd switchover --group-name group_a --node-name bdr-a1
  switchover is complete
```

```
pgd show-groups -f $PGD_CONFIG_FILE

Group      Group ID   Type   Parent Group Location Raft Routing Write Leader
- ----      --------   ----   ------------ -------- ---- ------- ------------
world      3239291720 global                        true true    location-a-2
location-a 2135079751 data   world                   true true    location-a-1
```

```
pgd show-nodes -f $PGD_CONFIG_FILE
Node         Node ID    Group      Type Current State Target State Status Seq ID
- ---         -------    -----      ---- ------------- ------------ ------ ------
location-a-1 3165289849 location-a data ACTIVE        ACTIVE       Up     1
location-a-2 3266498453 location-a data ACTIVE        ACTIVE       Up     2
location-a-3 1403922770 location-a data ACTIVE        ACTIVE       Up     3
```

4.1.2 Accessing the database

In *Use cases* is a discussion on using the database within the Kubernetes cluster versus from outside. In *Connectivity* , you can find a discussion on services, which is relevant for accessing the database from applications.

However you implement your system, your applications must use the proxy service to connect to reap the benefits of PGD and of the increased self-healing capabilities added by the EDB Postgres Distributed for Kubernetes operator.

Important: As per the EDB Postgres for Kubernetes defaults, data nodes are created with a database called *app* and owned by a user named *app* , in contrast to the *bdrdb* database described in the EDB Postgres Distributed documentation. You can configure these values in the *cnp* section of the manifest. For reference, see Bootstrap in the EDB Postgres for Kubernetes documentation.

You might, however, want access to your PGD data nodes for administrative tasks, using the psql CLI.

You can get a pod listing for your PGD cluster and `kubectl exec` into a data node:

```
kubectl exec -n my-namespace -ti location-a-1-1 -- psql
```

In the familiar territory of psql, remember that the default created database is named app (see previous warning).

```
postgres=# \c app
You are now connected to database "app" as user "postgres".
app=# \x
Expanded display is on.
app=# select * from bdr.node_summary;
- [ RECORD 1 ]----------------------------------------
node_name              | location-a-1
node_group_name        | location-a
interface_connstr      | host=location-a-1-node user=streaming_replica sslmode=verify-ca␣
↪port=5432 sslkey=/controller/certificates/streaming_replica.key sslcert=/controller/
↪certificates/streaming_replica.crt sslrootcert=/controller/certificates/server-ca.crt␣
↪application_name=location-a-1 dbname=app
peer_state_name        | ACTIVE
peer_target_state_name | ACTIVE

<- snipped ->
```

For your applications, use the non-privileged role (app by default).

You need the user credentials, which are stored in a Kubernetes secret:

```
kubectl get secrets

NAME                      TYPE                       DATA   AGE
<- snipped ->
location-a-app            kubernetes.io/basic-auth   2      2h
```

This secret contains the username and password needed for the Postgres DSN, encoded in base64:

```
kubectl get secrets location-a-app -o yaml

apiVersion: v1
data:
  password: <base64-encoded-password>
  username: <base64-encoded-username>
kind: Secret
metadata:
  creationTimestamp: <timestamp>
  labels:

<- snipped ->
```

4.2 Backup on object stores

EDB Postgres Distributed for Kubernetes supports *online/hot backup* of PGD clusters through physical backup and WAL archiving on an object store. This means that the database is always up (no downtime required) and that point-in-time recovery (PITR) is available.

4.2.1 Common object stores

Multiple object stores are supported, such as AWS S3, Microsoft Azure Blob Storage, Google Cloud Storage, MinIO Gateway, or any S3-compatible provider. Given that EDB Postgres Distributed for Kubernetes configures the connection with object stores by relying on EDB Postgres for Kubernetes, see the EDB Postgres for Kubernetes common object stores for backups

documentation for more information.

Important: The EDB Postgres for Kubernetes documentation's Cloud Provider configuration section is available at *spec.backup.barmanObjectStore* . In EDB Postgres Distributed for Kubernetes examples, the object store section is at a different path: *spec.backup.configuration.barmanObjectStore* .

4.2.2 WAL archive

WAL archiving is the process that sends WAL files to the object storage, and it' s essential to execute online/hot backups or PITR. In EDB Postgres Distributed for Kubernetes, each PGD node is set up to archive WAL files in the object store independently.

The WAL archive is defined in the PGD Group `spec.backup.configuration.barmanObjectStore` stanza, and is enabled as soon as a destination path and cloud credentials are set. You can choose to compress WAL files before they' re uploaded and you can encrypt them. You can also enable parallel WAL archiving:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
[...]
spec:
  backup:
    configuration:
      barmanObjectStore:
        [...]
        wal:
          compression: gzip
          encryption: AES256
          maxParallel: 8
```

For more information, see the *EDB Postgres for Kubernetes WAL archiving* documentation.

4.2.3 Scheduled backups

Scheduled backups are the recommended way to configure your backup strategy in EDB Postgres Distributed for Kubernetes. When the PGD group `spec.backup.configuration.barmanObjectStore` and `.spec.backup.schedulers[].schedule` stanza is configured, the operator selects one of the PGD data nodes as the elected backup node in which it creates a `Scheduled Backup` resource.

The `.spec.backup.schedulers[].method` field allows you to define the scheduled backup method. Two backup methods are supported:

- `volumeSnapshot`
- `barmanObjectStore` (the default)

You can define more than one scheduler, but each method can be used by only one scheduler. That is, two schedulers aren't allowed to use the same method.

For object store backups, with the default `barmanObjectStore` method, use the stanza `spec.backup.configuration.barmanObjectStore` to define the object store information for both backup and WAL archiving. For more information, see Backup on object stores in the EDB Postgres for Kubernetes documentation.

To perform volumeSnapshot backups, you can select the `volumeSnapshot` method. Use the stanza `spec.backup.configuration.barmanObjectStore.volumeSnapshot` to define the volumeSnapshot configuration. For more information, see *Backup on volume snapshots* in the EDB Postgres for Kubernetes documentation.

This example shows how to use the `volumeSnapshot` method for backup. WAL archiving is still done onto the Barman object store.

```yaml
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
[...]
spec:
  backup:
    configuration:
      volumeSnapshot:
        className: csi-hostpath-snapclass
      barmanObjectStore:
        destinationPath: "<destination path here>"
        s3Credentials:
          accessKeyId:
            name: backup-storage-creds
            key: ID
          secretAccessKey:
            name: backup-storage-creds
            key: KEY
        wal:
          compression: gzip
          encryption: AES256
          maxParallel: 8
    schedulers:
    - method: volumeSnapshot
      schedule: "0 0 0 * * *"
      backupOwnerReference: self
      suspend: false
      immediate: true
```

For a comparison of these two backup methods, see *Object stores or volume snapshots* in the EDB Postgres for Kubernetes documentation.

The `.spec.backup.schedulers[].schedule` field allows you to define a cron schedule, expressed in the *Go `cron* package format <https://pkg.go.dev/github.com/robfig/cron#hdr-CRON_Expression_Format>`_ :

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
[...]
spec:
  backup:
    schedulers:
    - method: barmanObjectStore
      schedule: "0 0 0 * * *"
      backupOwnerReference: self
      suspend: false
      immediate: true
```

If necessary, you can suspend scheduled backups by setting `.spec.backup.schedulers[].suspend` to `true` . This setting prevents new backups from being scheduled.

If you want to execute a backup as soon as the `ScheduledBackup` resource is created, set `.spec.backup.schedulers[].immediate` to `true` .

`.spec.backupOwnerReference` indicates the `ownerReference` to use in the created backup resources. The options are:

- **none** — Doesn' t set an owner reference for created backup objects.
- **self** — Sets the `ScheduledBackup` object as owner of the backup.
- **cluster** — Sets the cluster as owner of the backup.

Warning: The *.spec.backup.cron* field is deprecated. Use *.spec.backup.schedulers* instead. While you can still use *.spec.backup.cron* , you can't use it at the same time as *.spec.backup.schedulers* .

Note: The EDB Postgres for Kubernetes *ScheduledBackup* object contains the *cluster* option to specify the cluster to back up. This option currently isn't supported by EDB Postgres Distributed for Kubernetes and is ignored if specified.

If an elected backup node is deleted, the operator transparently elects a new backup node and reconciles the `ScheduledBackup` resource accordingly.

4.2.4 Retention policies

EDB Postgres Distributed for Kubernetes can manage the automated deletion of backup files from the backup object store using retention policies based on the recovery window. This process also takes care of removing unused WAL files and WALs associated with backups that are scheduled for deletion.

You can define your backups with a retention policy of 30 days:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
[...]
```

(continues on next page)

(continued from previous page)

```
spec:
  backup:
    configuration:
      retentionPolicy: "30d"
```

For more information, see the *EDB Postgres for Kubernetes retention policies* in the EDB Postgres for Kubernetes documentation.

Important: Currently, the retention policy is applied only for the elected *Backup Node* backups and WAL files. Given that each other PGD node also archives its own WALs independently, it's your responsibility to manage the lifecycle of those WAL files, for example by leveraging the object storage data retention policy. Also, if you have an object storage data retention policy set up on every PGD node directory, make sure it's not overlapping or interfering with the retention policy managed by the operator.

4.2.5 Compression algorithms

Backups and WAL files are uncompressed by default. However, multiple compression algorithms are supported. For more information, see the *EDB Postgres for Kubernetes compression algorithms* documentation.

4.2.6 Tagging of backup objects

It's possible to specify tags as key-value pairs for the backup objects, namely base backups, WAL files, and history files. For more information, see the EDB Postgres for Kubernetes documentation about tagging of backup objects .

4.2.7 On-demand backups of a PGD node

A PGD node is represented as single-instance EDB Postgres for Kubernetes `Cluster` object. As such, if you need to, it's possible to request an on-demand backup of a specific PGD node by creating a EDB Postgres for Kubernetes `Backup` resource. To do that, see *EDB Postgres for Kubernetes on-demand backups* in the EDB Postgres for Kubernetes documentation.

Hint: You can retrieve the list of EDB Postgres for Kubernetes clusters that make up your PGD group by running *kubectl get cluster -l k8s.pgd.enterprisedb.io/group=my-pgd-group -n my-namespace* .

4.3 Recovery

In EDB Postgres Distributed for Kubernetes, recovery is available as a way to bootstrap a new PGD group starting from an available physical backup of a PGD node. Recovery can't be performed in place on an existing PGD group.

EDB Postgres Distributed for Kubernetes also supports point-in-time recovery (PITR), which allows you to restore a PGD group up to any point in time, from the first available backup in your catalog to the last archived WAL. Having a WAL archive is mandatory for PITR.

4.3.1 Prerequisites

Before recovering from a backup:

- Make sure that the PostgreSQL configuration (`.spec.cnp.postgresql.parameters`) of the recovered cluster is compatible with the original one from a physical replication standpoint.
- When recovering in a newly created namespace, first set up a cert-manager CA issuer before deploying the recovered PGD group.

For more information, see *EDB Postgres for Kubernetes recovery - Additional considerations* in the EDB Postgres for Kubernetes documentation.

4.3.2 Recovery from an object store

You can recover from a PGD node backup created by Barman Cloud and stored on supported object storage.

For example, given a PGD group named `pgdgroup-example` with three instances with backups available, your object storage will contain a directory for each node:

`pgdgroup-example-1`, `pgdgroup-example-2`, `pgdgroup-example-3`

This example defines a full recovery from the object store. The operator transparently selects the latest backup among the defined `serverNames` and replays up to the last available WAL.

```yaml
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: pgdgroup-restore
spec:
  [...]
  restore:
    serverNames:
      - pgdgroup-backup-1
      - pgdgroup-backup-2
      - pgdgroup-backup-3
    barmanObjectStore:
      destinationPath: "<destination path here>"
      s3Credentials:
        accessKeyId:
          name: backup-storage-creds
          key: ID
        secretAccessKey:
          name: backup-storage-creds
          key: KEY
      wal:
        compression: gzip
        encryption: AES256
        maxParallel: 8
```

Important: Make sure to correctly configure the WAL section according to the source cluster. In the example, since the *pgdgroup-example* PGD group uses *compression* and *encryption*, make sure to set the proper parameters also in the PGD group that's being created by the *restore*.

Note: The example takes advantage of the parallel WAL restore feature, dedicating up to eight jobs to concurrently fetch the required WAL files from the archive. This feature can appreciably reduce the recovery time. Make sure that you plan ahead for this scenario and tune the value of this parameter for your environment. It makes a difference when you need it.

4.3.3 PITR from an object store

Instead of replaying all the WALs up to the latest one, after extracting a base backup, you can ask PostgreSQL to stop replaying WALs at any point in time. PostgreSQL uses this technique to achieve PITR. (The presence of a WAL archive is mandatory.)

This example defines a time-base target for the recovery:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: pgdgroup-restore
spec:
  [...]
  restore:
    recoveryTarget:
      targetTime: "2023-08-11 11:14:21.00000+02"
    serverNames:
      - pgdgroup-backup-1
      - pgdgroup-backup-2
      - pgdgroup-backup-3
    barmanObjectStore:
      destinationPath: "<destination path here>"
      s3Credentials:
        accessKeyId:
          name: backup-storage-creds
          key: ID
        secretAccessKey:
          name: backup-storage-creds
          key: KEY
      wal:
        compression: gzip
        encryption: AES256
        maxParallel: 8
```

Important: PITR requires you to specify a *targetTime* recovery target by using the options described in *Recovery targets* . When you use *targetTime* or *targetLSN* , the operator selects the closest backup that was completed before that target. Otherwise, it selects the last available backup in chronological order between the specified *serverNames* .

4.3.4 Recovery from an object store specifying a `backupID`

The `.spec.restore.recoveryTarget.backupID` option allows you to specify a base backup from which to start the recovery process. By default, this value is empty. If you assign a value to it, the operator uses that backup as the base for the recovery. The value must be in the form of a Barman backup ID.

This example recovers a new PGD group from a specific backupID of the `pgdgroup-backup-1` PGD node:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: pgdgroup-restore
spec:
  [...]
  restore:
    recoveryTarget:
      backupID: 20230824T133000
    serverNames:
      - pgdgroup-backup-1
    barmanObjectStore:
      destinationPath: "<destination path here>"
      s3Credentials:
        accessKeyId:
          name: backup-storage-creds
          key: ID
        secretAccessKey:
          name: backup-storage-creds
          key: KEY
      wal:
        compression: gzip
        encryption: AES256
        maxParallel: 8
```

Important: When you specify a *backupID* , make sure to list only the related PGD node in the *serverNames* option, and avoid listing the other ones.

Note: Defining a specific *backupID* is especially needed when using one of the following recovery targets: *targetName* , *targetXID* , and *targetImmediate* . In such cases, it's important to specify *backupID* , unless the last available backup in the catalog is okay.

4.3.5 Recovery from volumeSnapshot

You can also recover a PGDgroup from a volumeSnapshot backup. Stanza `spec.restore.volumeSnapshots` is used to define the criteria for volumeSnapshots restore candidates. The operator transparently selects the latest volumeSnapshot among the candidates.

The operator requires the following annotations/labels in the volumeSnapshot. These annotations/labels are automatically added if volumeSnapshots are taken by the operator.

Annotations:

- `k8s.enterprisedb.io/backupEndTime` is used to compare and select the latest snapshot.
- `k8s.enterprisedb.io/pvcRole` represents the pvcRole of the volumeSnapshot. Supported roles include PG_WAL and PG_DATA.

Labels:

- `k8s.enterprisedb.io/cluster` indicates the node where the volumeSnapshot is taken, crucial for fetching the serverName in the object store for WAL replaying.
- `k8s.enterprisedb.io/backupName` is the backup name of the volumeSnapshot. Used to group volumeSnapshots when more volumes are defined in the backup.
- `k8s.enterprisedb.io/tablespaceName` represents the tablespace name of the volumeSnapshot when the volumeSnapshot role is `PG_TABLESPACE` .

This example shows a full recovery from volumeSnapshots. After the volumeSnapshot recovery, WAL replaying for full recovery will target server `pgdgroup-backup-vs-1` .

```yaml
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: pgdgroup-restore
spec:
  [...]
  restore:
    volumeSnapshots:
      selector:
        matchLabels:
          "k8s.enterprisedb.io/cluster": pgdgroup-backup-vs-1
          "k8s.pgd.enterprisedb.io/group": pgdgroup-backup-vs
    barmanObjectStore:
      destinationPath: "<destination path here>"
      s3Credentials:
        accessKeyId:
          name: backup-storage-creds
          key: ID
        secretAccessKey:
          name: backup-storage-creds
          key: KEY
      wal:
        compression: gzip
        encryption: AES256
        maxParallel: 8
```

For more information, see *Recovery from volumeSnapshot objects* in the EDB Postgres for Kubernetes documentation.

4.3.6 PITR from volumeSnapshot

You can instruct PostgreSQL to halt the replay of write-ahead logs (WALs) at any specific moment during volumeSnapshot recovery. This is the same capability as when recovering from an object store.

This example shows setting a time-based target for recovery using volume snapshots:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: pgdgroup-restore
spec:
  [...]
  restore:
    recoveryTarget:
      targetTime: "2023-08-11 11:14:21.00000+02"
    volumeSnapshots:
      selector:
        matchLabels:
          "k8s.enterprisedb.io/cluster": pgdgroup-backup-vs-1
          "k8s.pgd.enterprisedb.io/group": pgdgroup-backup-vs
    barmanObjectStore:
      destinationPath: "<destination path here>"
      s3Credentials:
        accessKeyId:
          name: backup-storage-creds
          key: ID
        secretAccessKey:
          name: backup-storage-creds
          key: KEY
      wal:
        compression: gzip
        encryption: AES256
        maxParallel: 8
```

4.3.7 Recovery targets

Beyond PITR are other recovery target criteria you can use. For more information on all the available recovery targets, see *EDB Postgres for Kubernetes recovery targets* in the EDB Postgres for Kubernetes documentation.

4.3.8 Recovery and create PGD Groups in multiple regions

In order to recover from a backup and create multiple PGD Groups joined across regions, we can only restore the first group (where `spec.pgd.parentGroup.create` is set to true) from the backup. The other groups need to be set up from scratch, and then joined to the first group. This follows recommendations when using `bdr.join_node_group` :

> We recommend that the newly joining database be empty except for the BDR extension.

Refer to the PGD documentation for more detail.

4.4 Security

Security for EDB Postgres Distributed for Kubernetes is analyzed at three layers: code, container, and cluster.

> **Warning:** In addition to security practices described here, you must perform regular InfoSec duties on your Kubernetes cluster. Familiarize yourself with *Overview of Cloud Native Security*

in the Kubernetes documentation.

See also:

See *The 4C's Security Model in Kubernetes*

blog article for a better understanding and context of the approach EDB takes with security in EDB Postgres Distributed for Kubernetes.

4.4.1 Code

Source code of EDB Postgres Distributed for Kubernetes is systematically scanned for static analysis purposes, including security problems. EDB uses a popular open-source linter for Go called GolangCI-Lint directly in the CI/CD pipeline. GolangCI-Lint can run several linters on the same source code.

One of these is Golang Security Checker , or `gosec` . `gosec` is a linter that scans the abstract syntactic tree of the source against a set of rules aimed at discovering well-known vulnerabilities, threats, and weaknesses hidden in the code. These threads include hard-coded credentials, integer overflows, SQL injections, and others.

Important: A failure in the static code analysis phase of the CI/CD pipeline is a blocker for the entire delivery of EDB Postgres Distributed for Kubernetes, meaning that each commit is validated against all the linters defined by GolangCI-Lint.

4.4.2 Container

Every container image that' s part of EDB Postgres Distributed for Kubernetes is built by way of CI/CD pipelines following every commit. Such images include not only those of the operator but also of the operands, specifically every supported PostgreSQL version. In the pipelines, images are scanned with:

- `Dockle <https://github.com/goodwithtech/dockle>`_ for best practices in terms of the container build process
- `Clair <https://github.com/quay/clair>`_ for vulnerabilities found in both the underlying operating system and libraries and applications that they run

---

Important: All operand images are rebuilt once a day by our pipelines in case of security updates at the base image and package level, providing patch level updates for the container images that EDB distributes.

The following guidelines and frameworks were taken into account for container-level security:

- The Container Image Creation and Deployment Guide , developed by the Defense Information Systems Agency (DISA) of the United States Department of Defense (DoD)

- The *CIS Benchmark for Docker* , developed by the Center for Internet Security (CIS)

See also:

See the *Security and Containers in EDB Postgres Distributed for Kubernetes*

blog article for more information about the approach that EDB takes on security at the container level in EDB Postgres Distributed for Kubernetes.

4.4.3 Cluster

Security at the cluster level takes into account all Kubernetes components that form both the control plane and the nodes as well as the applications that run in the cluster, including PostgreSQL.

Role-based access control (RBAC)

The operator interacts with the Kubernetes API server with a dedicated service account called pgd-operator-controller-manager. In Kubernetes this account is installed by default in the `pgd-operator-system` namespace. A cluster role binds between this service account and the pgd-operator-controller-manager cluster role that defines the set of rules, resources, and verbs granted to the operator.

RedHat OpenShift directly manages the operator RBAC entities by way of *OperatorLifecycleManager (OLM)* . OLM allows you to grant permissions only where they' re required, implementing the principle of least privilege.

Important: These permissions are exclusively reserved for the operator's service account to interact with the Kubernetes API server. They aren't directly accessible by the users of the operator that interact only with *PGDGroup* and *PGDGroupCleanup* resources.

The following are some examples and, most importantly, the reasons why EDB Postgres Distributed for Kubernetes requires full or partial management of standard Kubernetes namespaced resources.

`jobs` : The operator needs to handle jobs to manage different `PGDGroup` phases.

`poddisruptionbudgets` : The operator uses pod disruption budgets to make sure enough PGD nodes are kept active during maintenance operations.

`pods` : The operator needs to manage PGD nodes as a `Cluster` resource.

`secrets` : Unless you provide certificates and passwords to your data nodes, the operator adopts the "convention over configuration" paradigm by self-provisioning random-generated passwords and TLS certificates and by storing them in secrets.

`serviceaccounts` : The operator needs to create a service account to enable the `PGDGroup` recovery job to retrieve the backup objects from the object store where they reside.

`services` : The operator needs to control network access to the PGD cluster from applications and properly manage failover/switchover operations in an automated way.

`statefulsets` : The operator needs to manage PGD proxies.

`validatingwebhookconfigurations` and `mutatingwebhookconfigurations` : The operator injects its self-signed webhook CA into both webhook configurations, which are needed to validate and mutate all the resources it manages. For more details, see the Kubernetes documentation .

To see all the permissions required by the operator, you can run `kubectl describe clusterrole pgd-operator-manager-role` .

EDB Postgres Distributed for Kubernetes internally manages the PGD nodes using the `Cluster` resource as defined by EDB Postgres for Kubernetes. See the *EDB Postgres for Kubernetes documentation*

for the list of permissions used by the EDB Postgres for Kubernetes operator service account.

Calls to the API server made by the instance manager

The instance manager, which is the entry point of the operand container, needs to make some calls to the Kubernetes API server to ensure that the status of some resources is correctly updated and to access the config maps and secrets that are associated with that Postgres cluster. Such calls are performed through a dedicated `ServiceAccount` created by the operator that shares the same PostgreSQL `Cluster` resource name.

Important: The operand can access only a specific and limited subset of resources through the API server. A service account is the recommended way to access the API server from within a pod. See the Kubernetes documentation for details.

See the *EDB Postgres for Kubernetes documentation*

for more information on the instance manager.

Pod security policies

A *pod security policy*

is the Kubernetes way to define security rules and specifications that a pod needs to meet to run in a cluster. For InfoSec reasons, every Kubernetes platform must implement them.

EDB Postgres Distributed for Kubernetes doesn't require privileged mode for containers execution. The PostgreSQL containers run as the postgres system user. No component requires running as root.

Likewise, volumes access doesn't require privileged mode or root privileges. Proper permissions must be assigned by the Kubernetes platform or administrators. The PostgreSQL containers run with a read-only root filesystem, that is, no writable layer.

The operator explicitly sets the required security contexts.

On Red Hat OpenShift, Cloud Native PostgreSQL runs in the `restricted` security context constraint, the most restrictive one. The goal is to limit the execution of a pod to a namespace allocated UID and SELinux context.

See also:

For more information on security context constraints (SCC) in OpenShift, see the *Managing SCC in OpenShift*

article.

Exposed ports

EDB Postgres Distributed for Kubernetes exposes ports at operator, instance manager, and operand levels, as shown in the table.

System	Port number	Exposing	Name	Certificates	Authentication
operator	9443	webhook server	*webhook-server*	TLS	Yes
operator	8080	metrics	*metrics*	no TLS	No

continues on next page

Table 1 – continued from previous page

System	Port number	Exposing	Name	Certificates	Authentication
instance manager	9187	metrics	*metrics*	no TLS	No
instance manager	8000	status	*status*	no TLS	No
operand	5432	PostgreSQL instance	*postgresql*	optional TLS	Yes

PGD

The current implementation of EDB Postgres Distributed for Kubernetes creates passwords for the postgres superuser and the database owner.

As far as encryption of passwords is concerned, EDB Postgres Distributed for Kubernetes follows the default behavior of PostgreSQL: starting with PostgreSQL 14, `password_encryption` is by default set to `scram-sha-256` . On earlier versions, it' s set to `md5` .

Important: See Connection DSNs and SSL

in the PGD documentation for details.

EDB Postgres Distributed for Kubernetes uses the `postgres` role to maintain PGD nodes. Due to this, it is not allowed to disable `postgres` user access by setting `enableSuperuserAccess` to `false` in the `cnp` section of the spec.

Storage

EDB Postgres Distributed for Kubernetes delegates encryption at rest to the underlying storage class. For data protection in production environments, we highly recommend that you choose a storage class that supports encryption at rest.

4.5 Connectivity

Information about secure network communications in a PGD cluster includes:

- *Services*
- *Domain names resolution* using fully qualified domain names (FQDN)
- *TLS configuration*
- *Connecting from an application*

4.5.1 Services

Resources in a PGD cluster are accessible through Kubernetes services. Every PGD group manages several of them, namely:

- One service per node, used for internal communications (*node service*).
- A *group service* to reach any node in the group, used primarily by EDB Postgres Distributed for Kubernetes to discover a new group in the cluster.
- A *proxy service* to enable applications to reach the write leader of the group transparently using PGD Proxy.

- A *proxy-r service* to enable applications to reach the *read nodes* of the group, transparently using PGD Proxy. This service is disabled by default and controlled by the `.spec.proxySettings.enableReadNodeRouting` setting.

Note: The term _read **nodes**_ refers to nodes that are not designated as the write leader. In the PGD group, each node is capable of handling writes. The Write leader is the elected node designated to process writes through the *proxy service* . Meanwhile, _read **nodes**_ reached by the *proxy-r service* are also writable, but it is the user's responsibility to ensure that write SQL commands are not sent to the *proxy-r service* . For more details, please refer to Read-only routing with PGD Proxy

For an example that uses these services, see *Connecting from an application* .

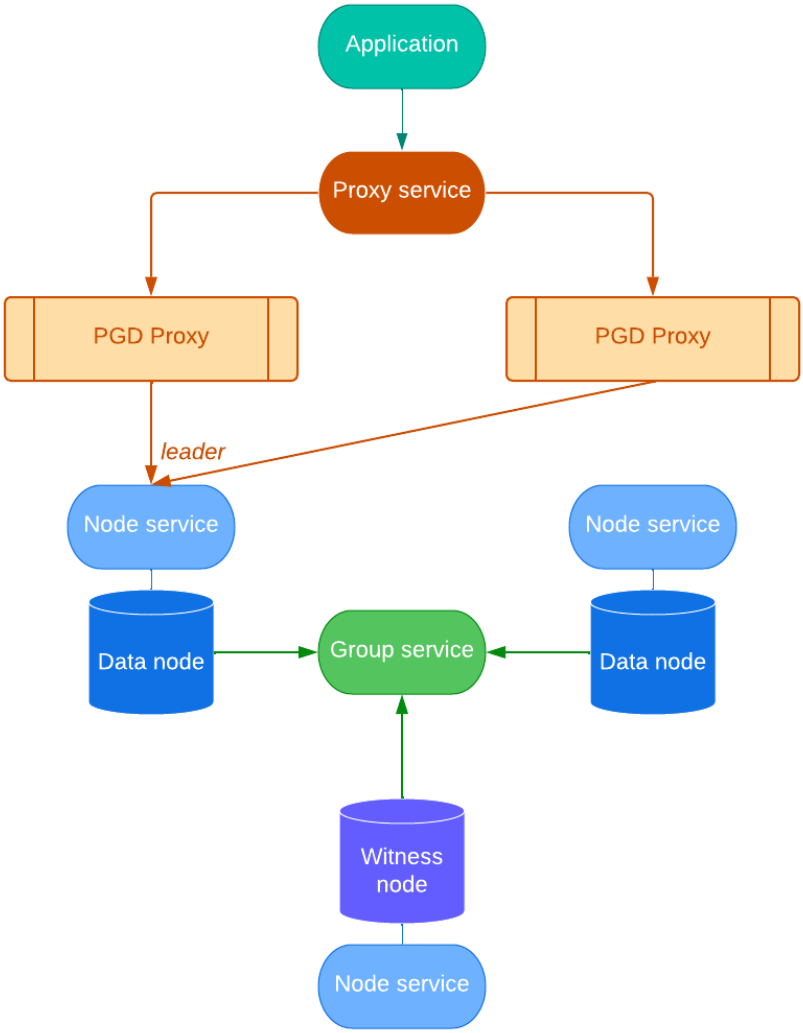

Fig. 1: Basic architecture of an EDB Postgres Distributed for Kubernetes PGD group

Each service is generated from a customizable template in the `.spec.connectivity` section of the manifest.

All services must be reachable using their FQDN from all the PGD nodes in all the Kubernetes clusters. See *Domain names resolution* .

EDB Postgres Distributed for Kubernetes provides a service templating framework that gives you the availability to easily customize services at the following levels:

- Node Service Template : Each PGD node is reachable using a service that can be configured in the `.spec.connectivity.nodeServiceTemplate` section.
- Group Service Template : Each PGD group has a group service that' s a single entry point for the whole group and that can be configured in the `.spec.connectivity.groupServiceTemplate` section.
- Proxy Service Template : Each PGD group has a proxy service to reach the group write leader through the PGD proxy and can be configured in the `.spec.connectivity.proxyServiceTemplate` section. This is the entry-point service for the applications.
- Proxy Read Service Template : Each PGD group has a proxy service to reach the group read nodes through the PGD proxy, can be enabled by `.spec.proxySettings.enableReadNodeRouting` , and can be configured in the `.spec.connectivity.proxyReadServiceTemplate` section. This is the entry-point service for the applications.

You can use templates to create a LoadBalancer service or to add arbitrary annotations and labels to a service to integrate with other components available in the Kubernetes system (that is, to create external DNS names or tweak the generated load balancer).

Here is an example of proxy service template, which create a LoadBalancer proxy service.

```
spec:
  connectivity:
    ...
    proxyServiceTemplate:
      metadata:
        annotations:
        ...
      spec:
        loadBalancerSourceRanges:
        - 0.0.0.0/0
        ports:
        - name: postgres
          port: 5432
          protocol: TCP
          targetPort: 5432
        type: LoadBalancer
      updateStrategy: patch
```

4.5.2 Domain names resolution

EDB Postgres Distributed for Kubernetes ensures that all resources in a PGD group have a FQDN, and by convention uses the PGD group name as a prefix for all of them.

As a result, it expects you to define the domain name of the PGD group. This can be done through the `.spec.connectivity.dns` section, which controls how the FQDN for the resources are generated with two fields:

- `domain` — Domain name for all the objects in the PGD group to use (mandatory).
- `hostSuffix` — Suffix to add to each service in the PGD group (optional).

4.5.3 TLS configuration

EDB Postgres Distributed for Kubernetes requires that resources in a PGD cluster communicate over a secure connection. It relies on PostgreSQL' s native support for SSL connections

to encrypt client/server communications using TLS protocols for increased security.

Currently, EDB Postgres Distributed for Kubernetes requires that cert-manager is installed. Cert-manager was chosen as the tool to provision dynamic certificates given that it' s widely recognized as the standard in a Kubernetes environment.

The `spec.connectivity.tls` section describes how the communication between the nodes happens. Please refers to the *certificates section* for more details on how TLS is configured.

4.5.4 Connecting from an application

Connecting to a PGD Group from an application running inside the same Kubernetes cluster or from outside the cluster is a simple procedure. In both cases, you connect to the proxy service of the PGD Group as the `app` user. The proxy service is a LoadBalancer service that routes the connection to the write leader or read nodes of the PGD Group, depending on which proxy service it' s connecting to.

Connecting from inside the cluster

When connecting from inside the cluster, you can use the proxy service name to connect to the PGD group. The proxy service name is composed of the PGD group name plus `-proxy` , and the optional host suffix defined in the `.spec.connectivity.dns` section of the `PGDGroup` custom resource.

For example, if the PGD group name is `my-group` , and the host suffix is `.my-domain.com` , the proxy service name is `my-group-proxy.my-domain.com` .

Before connecting, you need to get the password for the app user from the app user secret. The app user name is defined by `spec.pgd.ownerName` , and the app user secret is defined by `spec.pgd.ownerCredentialsSecret`

You can get the username and password from the secret using the following commands:

```
kubectl get secret <app user secret> -o jsonpath={.data.username} | base64 --decode
kubectl get secret <app user secret> -o jsonpath={.data.password} | base64 --decode
```

With this, you have all the pieces for a connection string to the PGD group:

```
postgresql://<app-user>:<app-password>@<proxy-service-name>:5432/<database>
```

Or, for a `psql` invocation:

```
psql -U <app-user> -h <proxy-service-name> <database>
```

Where `app-user` and `app-password` are the values you got from the secret, and `database` is the name of the database you want to connect to, defined by `spec.pgd.databaseName` . (The default is `app` for the app user.)

Connecting from outside the Kubernetes cluster

When connecting from outside the Kubernetes cluster, in the general case, the Ingress resource or a load balancer is necessary. Check your cloud provider or local installation for more information about their behavior in your environment.

Ingresses and load balancers require a pod selector to forward connection to the PGD proxies. When configuring them, we suggest using the following labels:

- `k8s.pgd.enterprisedb.io/group` — Set the PGD group name.
- `k8s.pgd.enterprisedb.io/workloadType` — Set to `pgd-proxy` .

If using Kind or other solutions for local development, the easiest way to access the PGD group from outside is to use port forwarding to the proxy service. You can use the following command to forward port 5432 on your local machine to the proxy service:

```
kubectl port-forward svc/my-group.my-domain.com 5432:5432
```

Where `my-group.my-domain.com` is the proxy service name from the previous example.

4.6 Certificates

EDB Postgres Distributed for Kubernetes was designed to natively support TLS certificates. To set up an PGD cluster, each PGD node requires:

- Server certificates configuration.
- Client certificates configuration.

Note: You can find all the secrets used by each PGD node and the expiry dates in the cluster (PGD node) status.

4.6.1 Server certificates configuration

The server certificate configuration is handled in the `spec.connectivity.tls.serverCert` section of the PGDGroup custom resource. This configuration requires a server CA secret and a cert-manager template to generate the TLS certificates needed for the underlying Postgres instance to terminate TLS connections.

The following assumptions must be met for this section to function correctly:

1. Cert-manager must be installed.
2. An issuer specified `spec.connectivity.tls.serverCert.certManager.issuerRef` is available for the domain specified in `spec.connectivity.dns.domain` and any additional domains listed in `spec.connectivity.tls.serverCert.certManager.spec.dnsNames` .
3. A server CA secret containing the public certificate of the CA used by the issuer must be created.

Note: The server CA secret specified by *spec.connectivity.tls.clientCert.serverCA* will be referenced as *serverCASecret* in the underlying CNP nodes. The public part, *ca.crt* , validates the server certificate and is included as *sslrootcert* into client connection strings. The private part, *ca.key* , is used to automatically sign the server SSL certificate, if a self-signed certificate is employed.

Note: The server TLS secret generated by PGD group will be specified as the value of *serverTLSSecret* in the underlying CNP nodes. For more information, refer to server certificates .

DNS names

The operator will add the following `altDnsNames` to the server TLS certificate:

```
${nodeName}${hostSuffix}.${domain}
${groupName}${hostSuffix}.${domain}
${proxyName}${hostSuffix}.${domain}
${nodeName}${additionalHostSuffix}.${additionalDomain}
${groupName}${additionalHostSuffix}.${additionalDomain}
${proxyName}${additionalHostSuffix}.${additionalDomain}
```

Users are responsible for including any necessary names in `spec.connectivity.tls.serverCert.certManager.spec.dnsNames` , based on their underlying networking architecture (e.g. any load balancers to access the nodes).

For example, consider a PGD Group configured as follows:

```yaml
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-a
spec:
  instances: 3
  ...
  connectivity:
    dns:
      # we need to configure the domain for the group so it could be resolved
      domain: enterprisedb.network
      additional:
        - domain: alternate.network
          hostSuffix: -dc1
```

All of the following Subject Alternative Names (SANs) will be added to the server TLS certificate:

```
region-a-1-node.domain.enterprisedb.network
region-a-group.domain.enterprisedb.network
region-a-proxy.domain.enterprisedb.network
region-a-1-node-dc1.alternate.network
region-a-group-dc1.alternate.network
region-a-proxy-dc1.alternate.network
```

4.6.2 Client certificates configuration

The client certificates configuration is managed under the `spec.connectivity.tls.clientCert` section of the PGDGroup custom resource. This configuration requires a client CA secret and a client cert-manager template to generate the client streaming replication certificate for the `streaming_replica` Postgres user.

The following assumptions must be met for this section to function properly:

1. Cert-manager must be installed.
2. An issuer specified in `spec.connectivity.tls.clientCert.certManager.issuerRef` is available; this issuer will be used to sign a certificate with the common name `streaming_replica` .
3. A client CA secret must be present. It contains the public certificate of the CA used by the issuer.

The operator will use the configuration under `spec.connectivity.tls.clientCert.certManager` to create a certificate request for the `streaming_replica` Postgres user. The resulting certificate will be used to secure communication between the nodes.

Note: The client CA secret specified by *spec.connectivity.tls.clientCert.clientCA* will be used as the value of *clientCASecret* in the underlying CNP nodes. The public part, *ca.crt* , will be provided as *ssl_ca_file* to all the instances, allowing them to verify client certificates they have signed. The private part, *ca.key* , is optional and can be used to sign client certificate generated by the kubectl cnp plugin .

Client pre-provisioned replication certificate

Alternatively, you can specify a secret containing the pre-provisioned client certificate for the `streaming_replica` user using the `spec.connectivity.tls.clientCert.preProvisioned.streamingReplica.secretRef` option. In this case, the certificate lifecycle is managed entirely by a third party (manually or automatically), by simply updating the content of the secret.

Note: Regardless of how the client streaming replication certificate is provided, it will be used as the value of *replicationTLSSecret* in the underlying CNP nodes. For more information, refer to the section on Client certificate .

4.6.3 TLS mode

You can configure the TLS mode, which determines how the server certificates are verified during communication between nodes, using `spec.connectivity.tls.mode` . Its default value is `verify-ca` . Note that the TLS mode cannot be changed once the PGD Group is set up. The `mode` accepts the following values, as documented in SSL Support in the PostgreSQL documentation:

- `verify-full`
- `verify-ca`
- `required`

4.6.4 Self-signed example

This example demonstrates how to use cert-manager to setup a self-signed CA and generated required certificates.

First, we need to generate the server and client CA certificates. We will create two self-signed issuers, `server-ca-issuer` and `client-ca-issuer` , in the target namespace.

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: server-ca-issuer
spec:
  ca:
    secretName: server-ca-key-pair
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: client-ca-issuer
spec:
  ca:
    secretName: client-ca-key-pair
```

With the following Certificate object, we can generate the private key and a signed certificate from the issuer. The private key and signed certificate will be stored in the secrets named `server-ca-key-pair` and `client-ca-key-pair` .

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: server-ca
spec:
  isCA: true
  commonName: my-selfsigned-server-ca
  secretName: server-ca-key-pair
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: client-ca
spec:
```

(continues on next page)

(continued from previous page)

```
  isCA: true
  commonName: my-selfsigned-client-ca
  secretName: client-ca-key-pair
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
```

We can now configure the PGD group. The `server-ca-key-pair` and `client-ca-key-pair` will be used as the server CA secret and client CA secret, respectively. The cert-manager template can be set up to reference the corresponding issuer to automatically generate the server TLS certificate and client replication certificate.

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-a
spec:
  ...
  connectivity:
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
```

For more information about how certificate works, see the *EDB Postgres for Kubernetes documentation* .

4.7 Client TLS/SSL connections

See also:

See *Certificates*

for more details on how EDB Postgres Distributed for Kubernetes supports TLS certificates.

The EDB Postgres Distributed for Kubernetes operator was designed to work with TLS/SSL for both encryption in transit and authentication on server and client sides. PGD nodes are created as cluster resources using the EDB Postgres for Kubernetes operator. This includes deploying a certification authority (CA) to create and sign TLS client certificates.

See the *EDB Postgres for Kubernetes documentation*

for more information on issuers and certificates.

4.8 Declarative pausing and resuming

The *declarative pausing and resuming* feature enables saving CPU power by removing the database pods while keeping the database PVCs.

Declarative pausing and resuming leverages the hibernation functionality available for EDB Postgres for Kubernetes. For additional depth and an explanation of how hibernation works, see the Postgres for Kubernetes documentation on declarative hibernation .

Request pause by adding the `k8s.pgd.enterprisedb.io/pause` annotation in the desired PGD group.

For example:

```
kubectl annotate pgdgroup region-a k8s.pgd.enterprisedb.io/pause=on
```

After a few seconds, the requested PGD group will be in paused state, with all the database pods removed:

```
kubectl get pgdgroups

NAME       DATA INSTANCES   WITNESS INSTANCES   PHASE                    AGE
region-a   2                1                   PGDGroup - Paused        25m
region-b   2                1                   PGDGroup - Healthy       25m
region-c   0                1                   PGDGroup - Healthy       25m
```

To resume a paused PGD group, set the annotation to `off` . Remember to add the `--overwrite` flag:

```
kubectl annotate pgdgroup region-a k8s.pgd.enterprisedb.io/pause=off --overwrite
```

In a few seconds, you should see the nodes start resuming, and the pods to be re-created.

```
kubectl get pgdgroups

NAME       DATA INSTANCES   WITNESS INSTANCES   PHASE                    AGE
region-a   2                1                   Pause - resume nodes        1m
region-b   2                1                   PGDGroup - Healthy       25m
region-c   0                1                   PGDGroup - Healthy       25m
```

There are some requirements before the pause annotation can put the PGD group on Pause. Ideally, the PGD Group should be in Healthy state. Alternatively, if all the data nodes in the PGD Group are healthy at the individual level, Pause can also be initiated.

4.9 EDB' s private container registries

The images for the EDB Postgres Distributed for Kubernetes and EDB Postgres for Kubernetes operators, as well as various operands, are kept in private container image registries under `docker.enterprisedb.com` .

Important: Access to the private registries requires an account with EDB and is reserved for EDB customers with a valid *subscription plan* . Credentials are run through your EDB account. For trials, see *Trials* .

4.9.1 Which repository to choose?

EDB Postgres Distributed for Kubernetes is available as part of the Extreme High Availability Add-On on top of either the EDB Enterprise Plan or EDB Standard Plan.

Depending on your subscription plan, EDB Postgres Distributed for Kubernetes is in one of the following repositories.

Plan	Repository
EDB Standard Plan (standard)	*k8s_standard_pgd*
EDB EnterpriseDB Plan (enterprise)	*k8s_enterprise_pgd*

The name of the repository shall be used as the Username when you try to login to the EDB container registry, for example through `docker login` or a kubernetes.io/dockerconfigjson .

Important: Each repository contains all the images you can access with your plan. You don't need to connect to different repositories to access different images, such as operator or operand images.

4.9.2 How to retrieve the token

In the repositories page in EDB , you' ll find an *EDB Repos 2.0* section where a `Repo Token` is shown, obscured. The same token is also available in your Account profile , labeled as `Repos 2.0 token` .

Next to the token you' ll find a button to copy the token, and an eye icon in case you want to view the content of the token as clear text. The token shall be used as the *Password* when you try to access the EDB container registry.

Example with `docker login`

You can log in using Docker from your terminal. We suggest that you copy the repo token using **Copy Token**. The `docker` command prompts you for a username and a password.

The username is the repo you' re trying to access, and the password is the token you just copied:

```
$ docker login docker.enterprisedb.com
Username: k8s_enterprise_pgd
Password:
Login Succeeded
```

4.9.3 Trials

If you're a trialist or a preview user, use `k8s_enterprise_pgd` as the name of the repository, and follow the instructions in *How to retrieve the token* for the token.

4.9.4 Operand images

Operand distributions

EDB Postgres Distributed for Kubernetes is an operator that supports running EDB Postgres Distributed (PGD) version 5 on three PostgreSQL distributions:

- PostgreSQL
- EDB Postgres Advanced Server (EPAS)
- EDB Postgres Extended (PGE)

Important: See Choosing a Postgres distribution

in the PGD documentation for details and a comparison of PGD on the different supported PostgreSQL distributions.

Due to the immutable application container adoption in EDB operators, the operator expects the container images to include all the binaries required to run the requested version of PGD on top of the required distribution and version of Postgres.

These images follow the requirements and the conventions described in Container image requirements

in the EDB Postgres for Kubernetes documentation, adding the `bdr5` extension.

The table shows the image name prefix for each Postgres distribution.

Postgres distribution	Versions	Image name	Repositories
PostgreSQL	13 - 17	postgresql-pgd	*k8s_standard_pgd*, *k8s_enterprise_pgd*
EDB Postgres Extended	13 - 17	edb-postgres-extended-pgd	*k8s_standard_pgd*, *k8s_enterprise_pgd*
EDB Postgres Advanced	13 - 17	edb-postgres-advanced-pgd	*k8s_enterprise_pgd*

Identifying image names

You can select a specific operand and proxy image version that's appropriate for your Postgres distribution.

Operand image name

The operand image name is composed of the Postgres distribution, EDB Postgres Distributed key word, their version numbers, and the build version.

Postgres type: The first part of the image name is the Postgres distribution you're using. The table shows how Postgres types correspond to this part of the image name.

Postgres distribution	Image name
EDB Postgres Advanced Server	*edb-postgres-advanced*
EDB Postgres Extended Server	*edb-postgres-extended*
PostgreSQL	*postgresql*

EDB Postgres Distributed: For all environment configurations, `pgd` is the next part of the image name.

Version numbers: The versions of the Postgres distribution and PGD extension, separated by a dash, are next in the image name.

Build version: The value reflects the current build, for example, 1. EDB creates new builds to include security patches.

These identifiers, together with their versions, form the operand image name. The table shows some examples.

Postgres version	EDB Postgres Distributed version	Build	Operand image name
EDB Postgres Advanced 16.6.0	PGD 5.6.1	1	edb-postgres-advanced-pgd:16.6.0-5.6.1-1
EDB Postgres Extended 15.10	PGD 5.6.1	1	edb-postgres-extended-pgd:15.10-5.6.1-1
PostgreSQL 16.6	PGD 5.6.1	1	postgresql-pgd:16.6-5.6.1-1

For an overview of Postgres compatibility, see Platform compatibility .

Proxy image name

The proxy image name and version derive from EDB Postgres Distributed. See the EDB Postgres Distributed release notes

for an overview of PGD versions and corresponding PGD proxy versions. The table shows an example.

Proxy version	Proxy image name
PGD Proxy 5.6.0	edb-pgd-proxy:5.6.0

4.9.5 Customize Operand images

You can customize the default operand images for the EDB Postgres Distributed for Kubernetes operator, and these changes will apply to all future PGDGroup creations. Additionally, you have the option to customize the operand for each individual PGDGroup deployment.

Customize Default Operand Images in Helm Chart Deployment

Use the following Helm chart command to deploy the EDB Postgres Distributed for Kubernetes operator with customized operand and proxy names. Please note that all operand and operator images will be pulled from the same private repository defined by `global.repository` . For more information, visit the Helm chart page.

```
helm upgrade --dependency-update \
  --install edb-pg4k-pgd \
  --namespace pgd-operator-system \
  --create-namespace \
  edb/edb-postgres-distributed-for-kubernetes \
  --set global.repository=docker.enterprisedb.com/k8s_${EDB_SUBSCRIPTION_PLAN}_pgd \
  --set global.pgdImageName=${EDB_PGD_IMAGE_NAME} \
  --set global.proxyImageName=${EDB_PROXY_IMAGE_NAME} \
  --set image.imageCredentials.username=k8s_${EDB_SUBSCRIPTION_PLAN}_pgd \
  --set image.imageCredentials.password=${EDB_SUBSCRIPTION_TOKEN}
```

Note: With the above installation, the default operand and proxy image are configured in the operator's ConfigMap *pgd-operator-controller-manager-config* . For more information, please refer to the *Operator configuration* .

Here' s an example of the customization for:

- Using `enterprise` subscription plan and pull images from the `k8s_enterprise_pgd` repository.
- Use EDB Postgres Advanced Server 15.10.0 as the Postgres option.
- Use PGD 5.6.1 as the Postgres Distributed version.
- Use 5.6.0 as the PGD Proxy version.

```
helm upgrade --dependency-update \
    --install edb-pg4k-pgd \
    --namespace pgd-operator-system \
    --create-namespace \
    edb/edb-postgres-distributed-for-kubernetes \
    --set global.repository=docker.enterprisedb.com/k8s_enterprise_pgd \
    --set global.pgdImageName=edb-postgres-advanced-pgd:15.10.0-5.6.1-1 \
    --set global.proxyImageName=edb-pgd-proxy:5.6.0 \
    --set image.imageCredentials.username=k8s_enterprise_pgd \
    --set image.imageCredentials.password=<my_registry_token>
```

Update Default Operand Images for an Existing Operator

You can modify the default operand images by updating the `pgd-operator-controller-manager-config` ConfigMap or Secret. For further details, please refer to the *Operator configuration*

documentation.

Customize Operand Images for Single PGDGroup Deployment

You can also customize operand and proxy image for a single PGDGroup deployment. In this case, it' s necessary to create the image pull secrets in the target namespace. If you' ve already installed the EDB Postgres Distributed for Kubernetes operator from the private registry, you should have set up an image pull secret.

```
kubectl create secret docker-registry registry-pullsecret \
  -n <CLUSTER-NAMESPACE> --docker-server=docker.enterprisedb.com \
  --docker-username=<REGISTRY-NAME> \
  --docker-password=<TOKEN>
```

As mentioned earlier, the `docker-username` corresponds to your registry name, such as `k8s_standard_pgd` or `k8s_enterprise_pgd` , while the `docker-password` is the token obtained from the *How to retrieve the token* .

After creating your pull secret, ensure you set the `imagePullSecrets` field in the PGD group manifest, along with the `imageName` . The manifest below will create a PGD group running PG Extended from the `k8s_enterprise_pgd` repository.

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: group-example-pge
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  imageName: docker.enterprisedb.com/k8s_enterprise_pgd/edb-postgres-extended-pgd:15.10-
↪5.6.1-1
  imagePullSecrets:
  - name: registry-pullsecret
  pgdProxy:
    imageName: docker.enterprisedb.com/k8s_enterprise_pgd/edb-pgd-proxy:5.6.0
  pgd:
    parentGroup:
      name: world
      create: true
  cnp:
    storage:
      size: 1Gi
```

4.10 Predefined labels

These predefined labels are managed by the EDB Postgres Distributed for Kubernetes operator.

`k8s.pgd.enterprisedb.io/certificateType` : Indicates the type of the certificates. `replication` indicates a certificate to be used to authenticate the replication client. `server` indicates a certificate to be used for server authentication.

`k8s.pgd.enterprisedb.io/group` : Name of the PGDGroup that the resource belongs to. Added to cluster or instance resources.

`k8s.pgd.enterprisedb.io/isWitnessService` : Indicates a service is for a witness node.

`k8s.pgd.enterprisedb.io/type` : Type of the resource added to cluster or instance resources, usually `node` .

`k8s.pgd.enterprisedb.io/workloadType` : Indicates the workload type of the resource added to cluster or instance resources. `pgd-node-data` indicates data node; `pgd-node-witness` a witness node; `pgd-proxy` for PGD Proxy node; `proxy-svc` for PGD Proxy service; `group-svc` for PGD group service to communicate with any node in the PGDGroup; `node-svc` is a service created from the CNP service template; `scheduled-backup` is added to `scheduledBackup` resources; `bootstrap-cross-location-pgd-group` is added to the pod that creates a cross-location PGD group; `pgd-node-restore` is added to the pod that starts the node restore process.

4.10.1 Predefined annotations

`k8s.pgd.enterprisedb.io/dirtyMetadata` : Set in CNP cluster that have been generated from a backup and need to have their metadata cleaned up before creating the PGD node. This is written by the restore job.

`k8s.pgd.enterprisedb.io/hash` : To determine if any updates are required for the corresponding resources, holds the hash of the certain part of PGDGroup spec that' s used in entities like `Cluster` , `ScheduledBackup` , `StatefulSet` , and `Service (node, group and proxy service)` .

`k8s.pgd.enterprisedb.io/latestCleanupExecuted` : Set in the PGDGroup to indicate that the cleanup was executed.

`k8s.pgd.enterprisedb.io/node` : Contains the name of the node for which a certain certificate was generated. Added to the certificate resources.

`k8s.pgd.enterprisedb.io/nodeRestartHash` : Stores the hash of the CNP configuration in PGDGroup. A restart is needed when the configuration is changed.

`k8s.pgd.enterprisedb.io/noFinalizers` : Set in the PGDGroup with value `true` to skip the finalizer execution. For internal use only.

`k8s.pgd.enterprisedb.io/pause` : Set in the PGDGroup to pause a PGDGroup.

`k8s.pgd.enterprisedb.io/recoverabilityPointsByMethod` : Set in the PGDGroup to store the CNP cluster' s first recoverability points by method in a tamper-proof place.

`k8s.pgd.enterprisedb.io/seedingServer` : Set in the PGDGroup to indicate to the operator which server to restore. This is written by the restore job.

`k8s.pgd.enterprisedb.io/seedingSnapshots` : Set in the PGDGroup to indicate to the operator which snapshots to restore. This is written by the restore job.

4.11 Managed configuration

The PGD operator allows configuring the `managed` section of a PG4K cluster. The `spec.cnp.managed` stanza is used for configuring the supported managed roles and services within the cluster.

In this example, a PGDgroup is defined with default `read only` and `read` services disabled. Additionally, it is configured to have a managed role named `foo` with the specified properties set up in postgres.

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: group-example-with-managed
spec:
  [...]
  cnp:
    [...]
    managed:
      roles:
        - name: foo
          comment: Foo
          ensure: present
          connectionLimit: 100
          login: true
          superuser: true
          createdb: true
          createrole: true
          replication: true
      services:
        disabledDefaultServices:
          - ro
          - r
```

For more information about managed roles, see Database role management in the EDB Postgres for Kubernetes documentation.

4.12 Monitoring

4.12.1 Monitoring Instances

By default, the PG4K operator provides an HTTP/HTTPS metrics exporter for Prometheus

and also comes with a *predefined set of metrics* as well as a highly configurable and customizable *user defined metrics* . The PGD operator offers a central location for monitoring settings in the `spec.cnp.monitoring` section, which will be applied to all nodes in the PGD group. For more details on configuring monitoring, please refer to the *monitoring section* .

4.12.2 Monitoring Operator

Like PG4K operator, the PGD operator also internally exposes Prometheus metrics via HTTP on port 8080, named `metrics` .

Currently, the PGD operator exposes default kubebuilder metrics. See kubebuilder documentation for more details.

4.12.3 Monitor on Openshift

Please refer to *Monitoring on OpenShift* section for details.

4.13 PGDGroup parting

4.13.1 Deletion and finalizers

When deleting a PGD Group, the operator will start parting every node in the group first. It will connect to an active instance and part every node in the target group. Once a node is parted, it will not participate in replication and consensus operations. To make sure the node is correctly parted before being deleted, the operator uses the `k8s.pgd.enterprisedb.io/partNodes` finalizer. Please refer to the kubernetes document on finalizers for context.

Note: If a namespace holding a PGD Group is deleted directly, we can't ensure the deleting and parting sequence is carried out correctly. Before deleting a namespace, it is recommended to delete all the contained PGD groups.

4.13.2 Time limit

When parting a node, the operator needs to connect to an active instance to execute the `bdr.part_node` function. To avoid this operation hanging, a time limit for the finalizer is used; by default, it is 300 seconds. After the time limit expires, the finalizer will be removed, and the node will be deleted anyway, potentially leaving stale metadata in the global PGD catalog. This time limit can be configured through `spec.failingFinalizerTimeLimitSeconds` , which is specified in seconds.

4.13.3 Skip finalizer

For testing purposes only, the operator also provides an annotation to skip the finalizer: `k8s.pgd.enterprisedb.io/noFinalizers` . When this annotation is added to a PGDGroup, the finalizer will be skipped when the PGDGroup is being deleted, and the nodes will not be parted from the PGD cluster.

4.13.4 PGDGroup cleanup

Cleanup parted node

Once the PGDGroup is deleted, its metadata will remain in the catalog in `PARTED` state in the `bdr.node_summary` table. The PGD4K operator defines a CRD named `PGDGroupCleanup` to help clean up the `PARTED` PGDGroup.

In the example below, the `PGDGroupCleanup` executes locally from `region-a` , and will clean up all of region-b, with the pre-requisite that all the nodes must be in the `PARTED` state.

```yaml
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroupCleanup
metadata:
  name: region-b-cleanup
spec:
  executor: region-a
  target: region-b
```

Please note that if the target group (`region-b` in the example) contains nodes not in a `PARTED` state, the Group Cleanup will stop in phase `PGDGroupCleanup - Target PGDGroup is not parted, waiting for it to be parted before executing PGDGroupCleanup` . In cases of extreme need, we can add the `force` option.

> **Warning:** Using *force* can leave the PGD cluster in an inconsistent state. Use it only to recover from failures in which you can't part the group nodes any other way.

```yaml
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroupCleanup
metadata:
  name: region-b-cleanup
spec:
  force: true
  executor: region-a
  target: region-b
```

4.14 SQLMutations

SQLMutations consist of a list of SQL queries to execute on the application database via the superuser role after a PGD node joins the PGDgroup. Each SQLMutation includes an `isApplied` list of queries and an `exec` list of queries. The `isApplied` SQL queries are used to check if the mutation was already applied. If any of the `isApplied` queries return false, the `exec` list of SQL queries is added to the execution queue.

Here' s a sample of SQLMutations:

```yaml
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
[...]
spec:
  pgd:
    mutations:
      - isApplied:
          - |
            WITH node_route_config AS (
              SELECT node_name, route_dsn
              FROM bdr.node_routing_config_summary
            )
           SELECT NOT EXISTS(SELECT * FROM node_route_config WHERE regexp_count(route_
↪dsn, application_name=) >= 1);
        exec:
          - |
```

(continues on next page)

(continued from previous page)

```
            WITH node_route_config AS (
              SELECT node_name,
                     regexp_replace(route_dsn, concat( application_name=,  node_name), )␣
↪as route_dsn
              FROM bdr.node_routing_config_summary
            )
            SELECT bdr.alter_node_option(node_name, route_dsn, route_dsn)
            FROM node_route_config;
```

4.14.1 SQLMutation types

The operator offers three types of SQLMutations, which you specify with `spec.pgd.mutations[].type` . The default is `always` .

- `beforeSubgroupRaft`
- `always`
- `writeLeader`

The `beforeSubgroupRaft` and `always` mutations are evaluated in every reconcile loop. The difference between the two mutations lies in their execution phase:

- For `always` mutations, they' re run in each reconcile loop without any restrictions on the PGDgroup.
- `beforeSubgroupRaft` mutations are executed only if the PGDgroup has defined data nodes and PGD proxies, and specifically before the subgroup Raft becomes ready.

Both `beforeSubgroupRaft` and `always` mutations can run on any PGD node in the PGDgroup, including witness nodes. Therefore, don' t use them for making data changes to the application database, as witness nodes don' t contain application database data.

The `writeLeader` mutation is triggered and executed after the write leader is elected. The `exec` operations are carried out exclusively on the write leader node.

4.15 Red Hat OpenShift

EDB Postgres Distributed for Kubernetes is a certified operator that can be installed on OpenShift using a web interface.

4.15.1 Ensuring access to EDB private registry

Important: You need access to the private EDB repository where both the operator and operand images are stored. Access requires a valid EDB subscription plan . See Accessing EDB private image registries for details.

The OpenShift install uses pull secrets to access the operand and operator images, which are held in a private repository.

Once you have credentials to the private repo, you need to create two pull secrets in the `openshift-operators` namespace:

- `pgd-operator-pull-secret` for the EDB Postgres Distributed for Kubernetes operator images
- `postgresql-operator-pull-secret` for the EDB Postgres for Kubernetes operator images

You can create each secret using the `oc create` command:

```
oc create secret docker-registry pgd-operator-pull-secret \
  -n openshift-operators --docker-server=docker.enterprisedb.com \
  --docker-username="@@REPOSITORY@@" \
  --docker-password="@@TOKEN@@"

oc create secret docker-registry postgresql-operator-pull-secret \
  -n openshift-operators --docker-server=docker.enterprisedb.com \
  --docker-username="@@REPOSITORY@@" \
  --docker-password="@@TOKEN@@"
```

Where:

- `@@REPOSITORY@@` is the name of the repository, as explained in *Which repository to choose?* .
- `@@TOKEN@@` is the repository token for your EDB account, as explained in *How to retrieve the token* .

4.15.2 Installing the operator

The EDB Postgres Distributed for Kubernetes operator can be found in the Red Hat OperatorHub directly from your OpenShift dashboard.

1. From the hamburger menu, select **Operators > OperatorHub**.
2. In the web console, use the search box to filter the listing. For example, enter `EDB` or `pgd` :
3. Read the information about the operator and select **Install**.
4. In the Operator Installation page, select:

- The installation mode. *Cluster-wide installation* is currently the only mode.
- The update channel (currently **preview**).
- The approval strategy, following the availability on the marketplace of a new release of the operator, certified by Red Hat: - **Automatic**: OLM upgrades the running operator with the new version. - **Manual**: OpenShift waits for human intervention by requiring an approval in the **Installed Operators** section.

Cluster-wide installation

With cluster-wide installation, you' re asking OpenShift to install the operator in the default `openshift-operators` namespace and to make it available to all the projects in the cluster. This is the default and normally recommended approach to install EDB Postgres Distributed for Kubernetes.

From the web console, for **Installation mode**, select **All namespaces on the cluster (default)**.

On installation, the operator is visible in all namespaces. In case there were problems during installation, check the logs in any pods in the `openshift-operators` project on the **Workloads > Pods** page as you would with any other OpenShift operator.

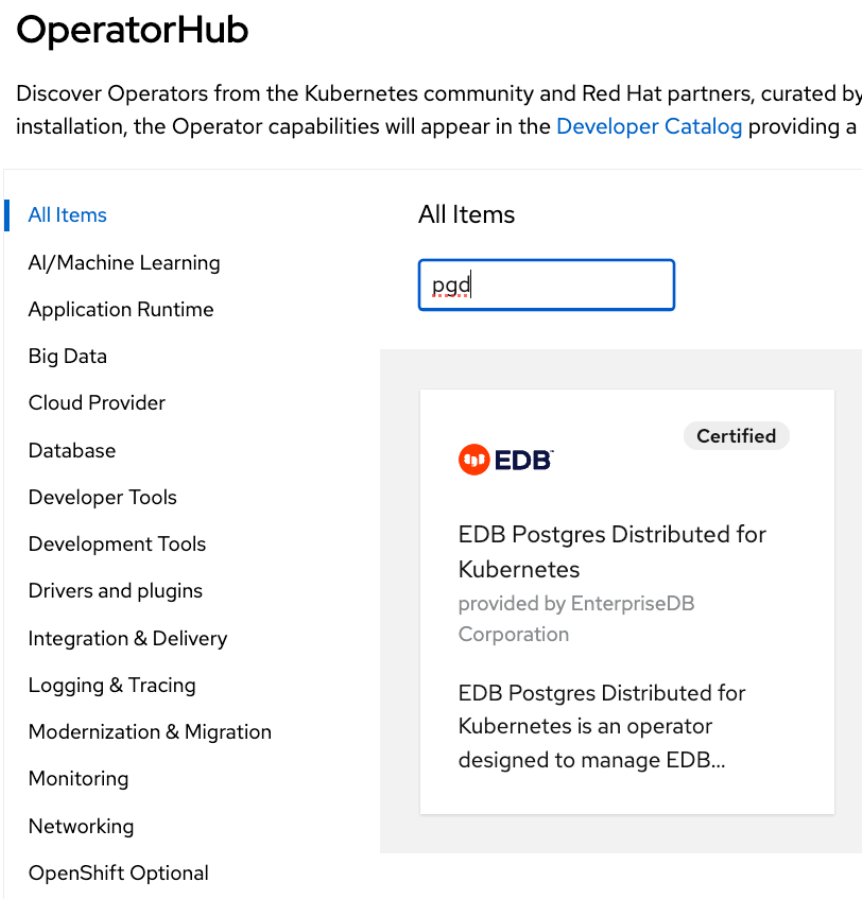

Fig. 2: Install OperatorHub

4.15.3 Creating a PGD cluster

After the installation by OpenShift, the operator deployment is in the `openshift-operators` namespace. Notice the cert-manager operator was also installed, as was the EDB Postgres for Kubernetes operator (`postgresql-operator-controller-manager`).

```
$ oc get deployments -n openshift-operators
NAME                                          READY   UP-TO-DATE   AVAILABLE   AGE
cert-manager-operator                         1/1     1            1           11m
pgd-operator-controller-manager               1/1     1            1           11m
postgresql-operator-controller-manager-1-20-0 1/1     1            1           23h
…
```

After checking that the `pgd-operator-controller-manager` deployment is READY, you can start creating PGD clusters. The EDB Postgres Distributed for Kubernetes repository contains some useful sample files.

You must deploy your PGD clusters on a dedicated namespace/project. The default namespace is reserved.

First, then, create a new namespace, and deploy a self-signed certificate `Issuer <Examples of configuration>`

in it:

```
oc create ns my-namespace
oc apply -n my-namespace -f \
  https://raw.githubusercontent.com/EnterpriseDB/edb-postgres-for-kubernetes-charts/main/
↪hack/samples/issuer-selfsigned.yaml
```

Using PGD in a single OpenShift cluster in a single region

Now you can deploy a PGD cluster, for example a flexible 3-region, which contains two data groups and a witness group. You can find the YAML manifest in the file.

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-a
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
      create: true
    discovery:
      - host: region-a-group
      - host: region-b-group
      - host: region-c-group
  cnp:
    storage:
      size: 1Gi
  connectivity:
    dns:
      additional:
```

(continues on next page)

(continued from previous page)

```
        - domain: alternate.domain
        - domain: my.domain
          hostSuffix: -dc1
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
- --
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-b
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    discovery:
      - host: region-a-group
      - host: region-b-group
      - host: region-c-group
  cnp:
    storage:
      size: 1Gi
  connectivity:
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
```

(continues on next page)

(continued from previous page)

```
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
- --
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-c
spec:
  instances: 0
  proxyInstances: 0
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    discovery:
      - host: region-a-group
      - host: region-b-group
      - host: region-c-group
  cnp:
    storage:
      size: 1Gi
  connectivity:
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
```

```
oc apply -f flexible_3regions.yaml -n my-namespace
```

Your PGD groups start to come up:

```
$ oc get pgdgroups -n my-namespace
NAME       DATA INSTANCES   WITNESS INSTANCES   PHASE                PHASE DETAILS   AGE
```

(continues on next page)

(continued from previous page)

```
region-a   2                 1                    PGDGroup - Healthy                    23m
region-b   2                 1                    PGDGroup - Healthy                    23m
region-c   0                 1                    PGDGroup - Healthy                    23m
```

Using PGD in multiple OpenShift clusters in multiple regions

To deploy PGD in multiple OpenShift clusters in multiple regions, you must first establish a way for the PGD groups to communicate with each other. The recommended way of achieving this with multiple OpenShift clusters is to use *Submariner* . Configuring the connectivity is outside the scope of this documentation. However, once you' ve established connectivity between the OpenShift clusters, you can deploy PGD groups synced with one another.

> **Warning:** This example assumes you're deploying three PGD groups, one in each OpenShift cluster, and that you established connectivity between the OpenShift clusters using Submariner.

Similar to the *Using PGD in a single OpenShift cluster in a single region* , this example creates two data PGD groups and one witness group. In contrast to that example, each group lives in a different OpenShift cluster.

In addition to basic connectivity between the OpenShift clusters, you need to ensure that each OpenShift cluster contains a certificate authority that' s trusted by the other OpenShift clusters. This condition is required for the PGD groups to communicate with each other.

The OpenShift clusters can all use the same certificate authority, or each cluster can have its own certificate authority. Either way, you need to ensure that each OpenShift cluster' s certificates trust the other OpenShift clusters' certificate authorities.

This example uses a self-signed certificate that has a single certificate authority used for all certificates on all the OpenShift clusters.

The example refers to the OpenShift clusters as `OpenShift Cluster A` , `OpenShift Cluster B` , and `OpenShift Cluster C` . In OpenShift, an installation of the EDB Postgres Distributed for Kubernetes operator from OperatorHub includes an installation of the cert-manager operator. We recommend creating and managing certificates with cert-manager.

1. Create a namespace to hold `OpenShift Cluster A` , and in it also create the needed objects for a self-signed certificate. Assuming that the PGD operator and the cert-manager are installed, you create a self-signed certificate `Issuer <Examples of configuration>`

in that namespace.

```
oc create ns pgd-group
oc apply -n pgd-group -f \
  https://raw.githubusercontent.com/EnterpriseDB/edb-postgres-for-kubernetes-charts/main/
↪hack/samples/issuer-selfsigned.yaml
```

1. After a few moments, cert-manager creates the issuers and certificates. There are also now two secrets in the `pgd-group` namespace: `server-ca-key-pair` and `client-ca-key-pair` . These secrets contain the certificates and private keys for the server and client certificate authorities. You need to copy these secrets to the other OpenShift clusters before applying the `issuer-selfsigned.yaml` manifest. You can use the `oc get secret` command to get the contents of the secrets:

```
oc get secret server-ca-key-pair -n pgd-group -o yaml > server-ca-key-pair.yaml
oc get secret client-ca-key-pair -n pgd-group -o yaml > client-ca-key-pair.yaml
```

1. After removing the content specific to `OpenShift Cluster A` from these secrets (such as uid, resourceVersion, and timestamp), you can switch context to `OpenShift Cluster B` . Then create the namespace, create the secrets in it, and only then apply the `issuer-selfsigned.yaml` file:

```
oc create ns pgd-group
oc apply -n pgd-group -f server-ca-key-pair.yaml
oc apply -n pgd-group -f client-ca-key-pair.yaml
oc apply -n pgd-group -f \
  https://raw.githubusercontent.com/EnterpriseDB/edb-postgres-for-kubernetes-charts/main/
↪hack/samples/issuer-selfsigned.yaml
```

1. You can switch context to `OpenShift Cluster C` and repeat the same process followed for Cluster B:

```
oc create ns pgd-group
oc apply -n pgd-group -f server-ca-key-pair.yaml
oc apply -n pgd-group -f client-ca-key-pair.yaml
oc apply -n pgd-group -f \
  https://raw.githubusercontent.com/EnterpriseDB/edb-postgres-for-kubernetes-charts/main/
↪hack/samples/issuer-selfsigned.yaml
```

1. On `OpenShift Cluster A` , you can create your first PGD group, called `region-a` . The YAML manifest for the PGD group is:

```yaml
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-a
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
      create: true
    discovery:
      - host: region-a-group.pgd-group.svc.clusterset.local
      - host: region-b-group.pgd-group.svc.clusterset.local
      - host: region-c-group.pgd-group.svc.clusterset.local
  cnp:
    storage:
      size: 1Gi
  connectivity:
    dns:
      domain: "pgd-group.svc.clusterset.local"
      additional:
        - domain: alternate.domain
        - domain: my.domain
          hostSuffix: -dc1
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
```

(continues on next page)

(continued from previous page)

```
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
```

Important: The format of the hostnames in the *discovery* section differs from the single-cluster example. That's because Submariner is being used to connect the OpenShift clusters, and Submariner uses the *<service>.<ns>.svc.clusterset.local* domain to route traffic between the OpenShift clusters. *region-a-group* is the name of the service to be created for the PGD group named *region-a* .

1. Apply the `region-a` PGD group YAML:

```
oc apply -f region-a.yaml -n pgd-group
```

1. You can now switch context to `OpenShift Cluster B` and create the second PGD group. The YAML for the PGD group in Cluster B is as follows. The only difference is the `metadata.name` .

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-b
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    discovery:
      - host: region-a-group.pgd-group.svc.clusterset.local
      - host: region-b-group.pgd-group.svc.clusterset.local
      - host: region-c-group.pgd-group.svc.clusterset.local
  cnp:
    storage:
      size: 1Gi
  connectivity:
    dns:
      domain: "pgd-group.svc.clusterset.local"
    tls:
      mode: verify-ca
      clientCert:
```

(continues on next page)

(continued from previous page)

```
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
```

1. Apply the `region-b` PGD group YAML:

```
oc apply -f region-b.yaml -n pgd-group
```

1. You can switch context to `OpenShift Cluster C` and create the third PGD group. The YAML for the PGD group is:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-c
spec:
  instances: 0
  proxyInstances: 0
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    discovery:
      - host: region-a-group.pgd-group.svc.clusterset.local
      - host: region-b-group.pgd-group.svc.clusterset.local
      - host: region-c-group.pgd-group.svc.clusterset.local
  cnp:
    storage:
      size: 1Gi
  connectivity:
    dns:
      domain: "pgd-group.svc.clusterset.local"
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
```

(continues on next page)

(continued from previous page)

```
            kind: Issuer
            group: cert-manager.io
    serverCert:
      caCertSecret: server-ca-key-pair
      certManager:
        spec:
          issuerRef:
            name: server-ca-issuer
            kind: Issuer
            group: cert-manager.io
```

1. Apply the `region-c` PGD group YAML:

```
oc apply -f region-c.yaml -n pgd-group
```

Now you can switch context back to `OpenShift Cluster A` and check the status of the PGD group there:

```
oc get pgdgroup region-a -n pgd-group
```

The PGD group is in the phase `PGD - Waiting for node discovery` .

After creating the PGD groups in each OpenShift cluster, which in turn creates the services for each node, you need to expose the services to the other OpenShift clusters. You can do this in various ways.

If you' re using Submariner, you can do it using the subctl

command. Run the `subctl export service` command for each service in the `pgd-group` namespace that has a `-group` or `-node` suffix. You can do this by running the following bash `for` loop on each cluster:

```
for service in $(oc get svc -n pgd-group --no-headers -o custom-columns="NAME:.metadata.
↪name" | grep -E (-group|-node)$); do
  subctl export service $service -n pgd-group
done
```

After a few minutes, the status shows that the PGD group is healthy. Once each PGD group is healthy, you can write to the `app` database in either of the two data nodes: `region-a` or `region-b` . The data is replicated to the other data node.

4.15.4 Channel

The EDB Postgres Distributed for Kubernetes operator is available in a single OLM channel

named `stable` since v1.0.0. All the releases will be available in this channel.

4.15.5 Dependencies

The EDB Postgres Distributed for Kubernetes operator (PGD4K) on OpenShift has the following dependencies:

- EDB Postgres for Kubernetes operator (PG4K): In release v1.0.1 and later, PGD4K is constrained to use an LTS release of PG4K.
- cert-manager: In release v1.1.1 and later, both cert-manager Operator for Red Hat OpenShift and cert-manager Operator for Community are supported.

For more details about the supported versions for each release, see *Supported versions* . Be aware that can you install or upgrade the PGD4K operator only when those dependencies are met.

4.16 Transparent data encryption (TDE)

Important: TDE is available *only* for operands that support it: EDB Postgres Advanced Server versions 15 and newer and EDB Postgres Extended versions 15 and newer.

Transparent data encryption, or TDE, is a technology used by several database vendors to encrypt data at rest, that is, database files on disk. However, TDE doesn' t encrypt data in use.

TDE is included in EDB Postgres Advanced Server or EDB Postgres Extended, starting with version 15, and is supported by EDB Postgres Distributed for Kubernetes.

Important: Before you proceed, please take some time to familiarize with the *TDE feature in the EPAS documentation* .

With TDE activated, both WAL files and files for tables are encrypted. Data encryption/decryption is entirely transparent to the user, as it' s managed by the database without requiring any application changes or updated client drivers.

The support for TDE on EDB Postgres Distributed for Kubernetes relies on the implementation from EDB Postgres for Kubernetes (PG4K). See *the PG4K documentation*

for the full context.

You can use TDE with a passphrase stored in a Kubernetes secret, which is used to encrypt the EDB Postgres Advanced Server binary key.

See also:

See *the EDB Postgres Advanced Server documentation*

for details on the this encryption key.

TDE on EDB Postgres Distributed for Kubernetes relies on the PG4K implementation. Activating TDE on a cluster uses the `epas` section of the manifest, which is in the `cnp` section used for PG4K-level directives such as storage. Use the `tde` stanza to enable TDE, and set the name of the Kubernetes secret holding the TDE encryption key.

The following YAML portion contains both a secret holding a passphrase (base-64 encoded), and the `epas` section activating TDE with the passphrase.

```
- --
apiVersion: v1
kind: Secret
metadata:
  name: tde-key
data:
  key: bG9zcG9sbGl0b3NkaWNlbnBpb3Bpb3Bpb2N1YW5kb3RpZW5lbmhhbWJyZWN1YW5kb3RpZW5lbmZyaW8=

- --
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
[···]
spec:
  instances: 3
[···]
  cnp:
    postgresql:
      epas:
        tde:
          enabled: true
          secretKeyRef:
            name: tde-key
            key: key
    storage:
      size: 1Gi
```

Again, see *the PG4K documentation*

for additional depth, including how to create the encryption secret and additional ways of using TDE.

As shown in the *TDE feature documentation* , the information is encrypted at rest.

For example, open a psql terminal into one of your data nodes.

```
kubectl exec -ti <DATA-NODE> -- psql app
```

Create a new table including a text column:

```
create table foo(bar int, baz varchar);
insert into foo(bar, baz) values (1, hello), (2, goodbye);
```

Verify the location where the newly defined table is stored on disk:

```
select pg_relation_filepath(foo);
 pg_relation_filepath
- ----------------------
 base/16385/16387
```

You can open a terminal on the same data node:

```
kubectl exec -ti <DATA-NODE> -- bash
```

There, you can verify the file was encrypted:

```
cd $PGDATA/base/16385
hexdump -C 16387 | grep hello
hexdump -C 16387 | grep goodbye
```

4.17 LDAP authentication

EDB Postgres Distributed for Kubernetes supports LDAP authentication. LDAP configuration on EDB Postgres Distributed for Kubernetes relies on the implementation from EDB Postgres for Kubernetes (PG4K). See the *PG4K documentation*

for the full context.

Important: Before you proceed, familiarize yourself with the *LDAP authentication feature in the Postgres documentation* .

With LDAP support, only the user authentication is sent to LDAP, so the user must already exist in the postgres database.

This example shows an LDAP configuration using `simple bind` mode in PGDGroup. The Postgres server uses `prefix + username + suffix` and password to bind the LDAP server to achieve the authentication.

```
spec:
  [...]
  cnp:
    postgresql:
      ldap:
        server: ldap-service.ldap.svc.cluster.local
        bindAsAuth:
          prefix: "uid="
          suffix: ",dc=example,dc=org"
```

This example shows configuring LDAP using `search+bind` mode in PGDGroup. In this mode, the Postgres instance is first bound to the LDAP using `bindDN` with its password stored in the secret `bindPassword` . Then Postgres tries to perform a search under `baseDN` to find a username that matches the item specified by `searchAttribute` . If a match is found, Postgres finally verifies the entry and the password against the LDAP server.

```
spec:
  [...]
  cnp:
    postgresql:
    ldap:
        server: ldap-service.ldap.svc.cluster.local
        bindSearchAuth:
        baseDN: "dc=example,dc=org"
        bindDN: "cn=admin,dc=example,dc=org"
        searchAttribute: "uid"
        bindPassword:
            name: ldap-bind-password
            key: password
```

4.18 API Reference

Package v1beta1 contains API Schema definitions for the pgd v1beta1 API group

4.18.1 Resource Types

- Example of a cluster-wide catalog using `ClusterImageCatalog resource <Example of a cluster-wide catalog using *ClusterImageCatalog* resource>`
- *Specifying operand images using ImageCatalog*
- *PGDGroup parting*
- *PGDGroupCleanup*

4.18.2 CertificateKeystores

Appears in:

- *CertificateSpec*

CertificateKeystores configures additional keystore output formats to be created in the Certificate' s output Secret.

Field	Description
*jks _JKSKeystore_*	JKS configures options for storing a JKS keystore in the *spec.secretName* Secret resource.
*pkcs12 _PKCS12Keystore_*	PKCS12 configures options for storing a PKCS12 keystore in the *spec.secretName* Secret resource.

4.18.3 CertificatePrivateKey

Appears in:

- *CertificateSpec*

CertificatePrivateKey contains configuration options for private keys used by the Certificate controller. This allows control of how private keys are rotated.

Field	Description
*rotationPolicy _PrivateKeyRotationPolicy_*	RotationPolicy controls how private keys should be regenerated when a re-issuance is being processed. If set to Never, a private key will only be generated if one does not already exist in the target{" "} *spec.secretName*. If one does exists but it does not have the correct algorithm or size, a warning will be raised to await user intervention. If set to Always, a private key matching the specified requirements will be generated whenever a re-issuance occurs. Default is 'Never' for backward compatibility.

continues on next page

Table 8 – continued from previous page

Field	Description
encoding _PrivateKeyEncoding_	The private key cryptography standards (PKCS) encoding for this certificate's private key to be encoded in. If provided, allowed values are *PKCS1* and *PKCS8* standing for PKCS#1 and PKCS#8, respectively. Defaults to *PKCS1* if not specified.
algorithm _PrivateKeyAlgorithm_	Algorithm is the private key algorithm of the corresponding private key for this certificate. If provided, allowed values are either{" "} *RSA*,`Ed25519` or *ECDSA* If *algorithm* is specified and *size* is not provided, key size of 256 will be used for *ECDSA* key algorithm and key size of 2048 will be used for *RSA* key algorithm. key size is ignored when using the *Ed25519* key algorithm.
size _int_	Size is the key bit size of the corresponding private key for this certificate. If *algorithm* is set to *RSA*, valid values are *2048*, *4096* or{" "} *8192*, and will default to *2048* if not specified. If *algorithm* is set to *ECDSA*, valid values are *256*, *384* or{" "} *521*, and will default to *256* if not specified. If *algorithm* is set to *Ed25519*, Size is ignored. No other values are allowed.

4.18.4 CertificateSpec

Appears in:

- *CertManagerTemplate*

CertificateSpec defines the desired state of Certificate. A valid Certificate requires at least one of a CommonName, DNSName, or URISAN to be valid.

Field	Description
subject _X509Subject_	Full X509 name specification (https://golang.org/pkg/crypto/x509/pkix/#Name).
commonName _string_	CommonName is a common name to be used on the Certificate. The CommonName should have a length of 64 characters or fewer to avoid generating invalid CSRs. This value is ignored by TLS clients when any subject alt name is set. This is x509 behaviour: https://tools.ietf.org/html/rfc6125#section-6.4.4
duration _Duration_	The requested 'duration' (i.e. lifetime) of the Certificate. This option may be ignored/overridden by some issuer types. If unset this defaults to 90 days. Certificate will be renewed either 2/3 through its duration or *renewBefore* period before its expiry, whichever is later. Minimum accepted duration is 1 hour. Value must be in units accepted by Go time.ParseDuration https://golang.org/pkg/time/#ParseDuration
renewBefore _Duration_	How long before the currently issued certificate's expiry cert-manager should renew the certificate. The default is 2/3 of the issued certificate's duration. Minimum accepted value is 5 minutes. Value must be in units accepted by Go time.ParseDuration https://golang.org/pkg/time/#ParseDuration
dnsNames _[]string_	DNSNames is a list of DNS subjectAltNames to be set on the Certificate.
ipAddresses _[]string_	IPAddresses is a list of IP address subjectAltNames to be set on the Certificate.
uris _[]string_	URIs is a list of URI subjectAltNames to be set on the Certificate.

continues on next page

Table 9 – continued from previous page

Field	Description
emailAddresses _[]string_	EmailAddresses is a list of email subjectAltNames to be set on the Certificate.
secretName **[Required]** _string_	SecretName is the name of the secret resource that will be automatically created and managed by this Certificate resource. It will be populated with a private key and certificate, signed by the denoted issuer. IMPORTANT: this field was required in the original cert-manager API declaration
keystores *_CertificateKeystores_*	Keystores configures additional keystore output formats stored in the *secretName* Secret resource.
issuerRef **[Required]** *_ObjectReference_*	IssuerRef is a reference to the issuer for this certificate. If the{" "} *kind* field is not set, or set to *Issuer*, an Issuer resource with the given name in the same namespace as the Certificate will be used. If the *kind* field is set to{" "} *ClusterIssuer*, a ClusterIssuer with the provided name will be used. The *name* field in this stanza is required at all times.
isCA _bool_	IsCA will mark this Certificate as valid for certificate signing. This will automatically add the *cert sign* usage to the list of{" "} *usages*.
usages [_[]KeyUsage_](KeyUsage)	Usages is the set of x509 usages that are requested for the certificate. Defaults to *digital signature* and{" "} *key encipherment* if not specified.
privateKey *_CertificatePrivateKey_*	Options to control private keys used for the Certificate.
encodeUsagesInRequest _bool_	EncodeUsagesInRequest controls whether key usages should be present in the CertificateRequest
revisionHistoryLimit _int32_	revisionHistoryLimit is the maximum number of CertificateRequest revisions that are maintained in the Certificate's history. Each revision represents a single *CertificateRequest* created by this Certificate, either when it was created, renewed, or Spec was changed. Revisions will be removed by oldest first if the number of revisions exceeds this number. If set, revisionHistoryLimit must be a value of *1* or greater. If unset (*nil*), revisions will not be garbage collected. Default value is{" "} *nil*.

4.18.5 ConditionStatus

(Alias of `string`)

ConditionStatus represents a condition’ s status.

4.18.6 JKSKeystore

Appears in:

- *CertificateKeystores*

JKSKeystore configures options for storing a JKS keystore in the{” “} `spec.secretName` Secret resource.

Field	Description
create **[Required]** _bool_	Create enables JKS keystore creation for the Certificate. If true, a file named *keystore.jks* will be created in the target Secret resource, encrypted using the password stored in *passwordSecretRef*. The keystore file will only be updated upon re-issuance. A file named *truststore.jks* will also be created in the target Secret resource, encrypted using the password stored in *passwordSecretRef* containing the issuing Certificate Authority
passwordSecretRef **[Required]** *_SecretKeySelector_*	PasswordSecretRef is a reference to a key in a Secret resource containing the password used to encrypt the JKS keystore.

4.18.7 KeyUsage

(Alias of `string`)

Appears in:

- *CertificateSpec*

KeyUsage specifies valid usage contexts for keys. See: https://tools.ietf.org/html/rfc5280#section-4.2.1.3

`https://tools.ietf.org/html/rfc5280#section-4.2.1.12`

Valid KeyUsage values are as follows: "signing", "digital signature", "content commitment", "key encipherment", "key agreement", "data encipherment", "cert sign", "crl sign", "encipher only", "decipher only", "any", "server auth", "client auth", "code signing", "email protection", "s/mime", "ipsec end system", "ipsec tunnel", "ipsec user", "timestamping", "ocsp signing", "microsoft sgc", "netscape sgc"

4.18.8 LocalObjectReference

Appears in:

- *SecretKeySelector*

LocalObjectReference is a reference to an object in the same namespace as the referent. If the referent is a cluster-scoped resource (e.g. a ClusterIssuer), the reference instead refers to the resource with the given name in the configured 'cluster resource namespace', which is set as a flag on the controller component (and defaults to the namespace that cert-manager runs in).

Field	Description
name **[Required]** _string_	Name of the resource being referred to. More info: https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#names

4.18.9 ObjectReference

Appears in:

- *CertificateSpec*

ObjectReference is a reference to an object with a given name, kind and group.

Field	Description
name **[Required]** _string_	Name of the resource being referred to.
group _string_	Group of the resource being referred to.

4.18.10 PKCS12Keystore

Appears in:

- *CertificateKeystores*

PKCS12Keystore configures options for storing a PKCS12 keystore in the `spec.secretName` Secret resource.

Field	Description
create **[Required]** _bool_	Create enables PKCS12 keystore creation for the Certificate. If true, a file named *keystore.p12* will be created in the target Secret resource, encrypted using the password stored in *passwordSecretRef*. The keystore file will only be updated upon re-issuance. A file named *truststore.p12* will also be created in the target Secret resource, encrypted using the password stored in *passwordSecretRef* containing the issuing Certificate Authority
passwordSecretRef **[Required]** *_SecretKeySelector_*	PasswordSecretRef is a reference to a key in a Secret resource containing the password used to encrypt the PKCS12 keystore.

4.18.11 PrivateKeyAlgorithm

(Alias of `string`)

Appears in:

- *CertificatePrivateKey*

PrivateKeyAlgorithm represent a private key algorithm

4.18.12 PrivateKeyEncoding

(Alias of `string`)

Appears in:

- *CertificatePrivateKey*

PrivateKeyEncoding represent a private key encoding

4.18.13 PrivateKeyRotationPolicy

(Alias of `string`)

Appears in:

- *CertificatePrivateKey*

PrivateKeyRotationPolicy denotes how private keys should be generated or sourced when a Certificate is being issued.

4.18.14 SecretKeySelector

Appears in:

- *JKSKeystore*

SecretKeySelector is a reference to a specific 'key' within a Secret resource. In some instances, `key` is a required field.

Field	Description
LocalObjectReference _*LocalObjectReference*_	(Members of *LocalObjectReference* are embedded into this type.) The name of the Secret resource being referred to.
key _string_	The key of the entry in the Secret resource's *data* field to be used. Some instances of this field may be defaulted, in others it may be required.

4.18.15 X509Subject

Appears in:

- *CertificateSpec*

X509Subject Full X509 name specification

Field	Description
organizations _[]string_	Organizations to be used on the Certificate.
countries _[]string_	Countries to be used on the Certificate.
organizationalUnits _[]string_	Organizational Units to be used on the Certificate.
localities _[]string_	Cities to be used on the Certificate.
provinces _[]string_	State/Provinces to be used on the Certificate.
streetAddresses _[]string_	Street addresses to be used on the Certificate.
postalCodes _[]string_	Postal codes to be used on the Certificate.
serialNumber _string_	Serial number to be used on the Certificate.

4.18.16 ClusterImageCatalog

ClusterImageCatalog is the Schema for the clusterimagecatalogs API

Field	Description
apiVersion **[Required]** string	*pgd.k8s.enterprisedb.io/v1beta1*
kind **[Required]** string	*ClusterImageCatalog*
spec **[Required]** *_ImageCatalogSpec_*	Specification of the desired behavior of the ClusterImageCatalog. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

4.18.17 ImageCatalog

ImageCatalog is the Schema for the imagecatalogs API

Field	Description
apiVersion **[Required]** string	*pgd.k8s.enterprisedb.io/v1beta1*
kind **[Required]** string	*ImageCatalog*
spec **[Required]** *_ImageCatalogSpec_*	Specification of the desired behavior of the ImageCatalog. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

4.18.18 PGDGroup

PGDGroup is the Schema for the pgdgroups API

Field	Description
apiVersion **[Required]** string	*pgd.k8s.enterprisedb.io/v1beta1*
kind **[Required]** string	*PGDGroup*
spec **[Required]** *_PGDGroupSpec_*	No description provided.
status **[Required]** *_PGDGroupStatus_*	No description provided.

4.18.19 PGDGroupCleanup

PGDGroupCleanup is the Schema for the pgdgroupcleanups API

Field	Description
apiVersion **[Required]** string	*pgd.k8s.enterprisedb.io/v1beta1*
kind **[Required]** string	*PGDGroupCleanup*
spec **[Required]** *_PGDGroupCleanupSpec_*	No description provided.
status **[Required]** *_PGDGroupCleanupStatus_*	No description provided.

4.18.20 Backup

Appears in:

- *PGDGroupSpec*

Backup configures the backup of cnp-pgd nodes

Field	Description
configuration **[Required]** *Bootstrap*	The CNP configuration to be used for backup. ServerName value is reserved by the operator.
cron *_ScheduledBackupSpec_*	The scheduled backup for the data. This field is deprecated and will be removed in future versions.
schedulers **[Required]** [_[]ScheduledBackupSpec_](ScheduledBackupSpec)	Define schedulers for the backup. Each scheduler has a different backup method. Only one of either *backup.cron* or{" "} *backup.schedulers* can be defined.

4.18.21 BackupStatus

Appears in:

- *PGDGroupStatus*

BackupStatus contains the current status of the pgd backup

Field	Description
clusterName **[Required]** _string_	ClusterName the elected cluster to take the backup, the backup could be scheduled with different methods, but only one cluster will take the backup
scheduledBackupName **[Required]** _string_	ScheduledBackupName is the name of the scheduled backup. This field is deprecated, will be migrated to ScheduledBackupStatus
scheduledBackupHash **[Required]** _string_	ScheduledBackupHash is the hash of the scheduled backup configuration. This field is deprecated, will be migrated to ScheduledBackupStatus
scheduledBackups **[Required]** [_[]ScheduledBackupStatus_](ScheduledBackupStatus)	ScheduledBackupStatus contains the status of all scheduled backups

4.18.22 CNPStatus

Appears in:

- *PGDGroupStatus*

CNPStatus contains any relevant status for the operator about CNP

Field	Description
dataInstances **[Required]** _int32_	No description provided.
witnessInstances **[Required]** _int32_	No description provided.

continues on next page

Table 22 – continued from previous page

Field	Description
clusterStatus **[Required]** [_[]ClusterStatus_](ClusterStatus)	ClusterStatus contains the list of the status of the CNP clusters
firstRecoverabilityPointsByMethod **[Required]** [_map[string]RecoverabilityPointsByMeth	The recoverability points by method, keyed per CNP clusterName nolint: lll
firstRecoverabilityPoints **[Required]** _map[string]string_	The recoverability points, keyed per CNP clusterName, as a date in RFC3339 format
superUserSecretIsPresent **[Required]** _bool_	No description provided.
applicationUserSecretIsPresent **[Required]** _bool_	No description provided.
podDisruptionBudgetIsPresent **[Required]** _bool_	No description provided.

4.18.23 CatalogImage

Appears in:

- *ImageCatalogSpec*

CatalogImage defines the image and major version

Field	Description
image **[Required]** _string_	The image reference
major **[Required]** _int_	The PostgreSQL major version of the image. Must be unique within the catalog.

4.18.24 CertManagerTemplate

Appears in:

- *ClientCertConfiguration*

CertManagerTemplate contains the data to generate a certificate request

Field	Description
spec **[Required]** *_CertificateSpec_*	The Certificate object specification
metadata **[Required]** *_Metadata_*	The label and annotations metadata

4.18.25 ClientCertConfiguration

Appears in:

- *OTELTLSConfiguration*

ClientCertConfiguration contains the information to generate the certificate for the streaming_replica user

Field	Description
caCertSecret **[Required]** _string_	CACertSecret is the secret of the CA to be injected into the CloudNativePG ClientCASecret configuration
certManager **[Required]** *_CertManagerTemplate_*	The cert-manager template used to generate the certificates
preProvisioned **[Required]** *_ClientPreProvisionedCertificates_*	PreProvisioned contains how to fetch the pre-generated client certificates

4.18.26 ClientPreProvisionedCertificates

Appears in:

- *ClientCertConfiguration*

ClientPreProvisionedCertificates instruct how to fetch the pre-generated client certificates

Field	Description
streamingReplica **[Required]** *_PreProvisionedCertificate_*	StreamingReplica the pre-generated certificate for 'streaming_replica' user

4.18.27 ClusterStatus

Appears in:

- *CNPStatus*

ClusterStatus contains the current status of the CNP cluster

Field	Description
name **[Required]** _string_	Name is the name of the CNP cluster
phase **[Required]** _string_	Phase is the current phase of the CNP cluster

4.18.28 CnpBaseConfiguration

Appears in:

- *CnpConfiguration*

CnpBaseConfiguration contains the configuration parameters that can be applied to both CNP Witness and Data nodes

Field	Description
startDelay **[Required]** _int32_	The time in seconds that is allowed for a PostgreSQL instance to successfully start up (default 3600)
stopDelay **[Required]** _int32_	The time in seconds that is allowed for a PostgreSQL instance node to gracefully shutdown (default 180)
smartShutdownTimeout _int32_	The time in seconds that controls the window of time reserved for the smart shutdown of Postgres to complete. Make sure you reserve enough time for the operator to request a fast shutdown of Postgres (that is:{" "} *stopDelay- smartShutdownTimeout*).
storage **[Required]** _StorageConfiguration_	Configuration of the storage of the instances
walStorage **[Required]** _StorageConfiguration_	Configuration of the WAL storage for the instances
clusterMaxStartDelay **[Required]** _int32_	The time in seconds that is allowed for a PostgreSQL instance to successfully start up (default 300)
affinity _AffinityConfiguration_	Affinity/Anti-affinity rules for Pods
resources _ResourceRequirements_	Resources requirements of every generated Pod. Please refer to https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ for more information.
postgresql _PostgresConfiguration_	Configuration of the PostgreSQL server
monitoring **[Required]** *Bootstrap*	The configuration of the monitoring infrastructure of this cluster
logLevel **[Required]** _string_	The instances' log level, one of the following values: error, warning, info (default), debug, trace
serviceAccountTemplate **[Required]** _ServiceAccountTemplate_	The service account template to be passed to CNP
otel **[Required]** *_OTELConfiguration_*	OpenTelemetry Configuration
postInitSQL _[]string_	List of SQL queries to be executed as a superuser immediately after a node has been created- to be used with extreme care (by default empty)
postInitTemplateSQL _[]string_	List of SQL queries to be executed as a superuser in the{" "} *template1* after a node has been created- to be used with extreme care (by default empty)
postInitApplicationSQL _[]string_	List of SQL queries to be executed as a superuser in the application database right after the cluster has been created- to be used with extreme care as any data created in application database before logical join will cause the join to fail (by default empty)
postInitApplicationSQLRefs _SQLRefs_	List of references to ConfigMaps or Secrets containing SQL files to be executed as a superuser in the application database right after the cluster has been created. The references are processed in a specific order: first, all Secrets are processed, followed by all ConfigMaps. Within each group, the processing order follows the sequence specified in their respective arrays.- to be used with extreme care (by default empty)

continues on next page

Table 28 – continued from previous page

Field	Description
postInitTemplateSQLRefs _SQLRefs_	List of references to ConfigMaps or Secrets containing SQL files to be executed as a superuser in the *template1* database right after the cluster has been created. The references are processed in a specific order: first, all Secrets are processed, followed by all ConfigMaps. Within each group, the processing order follows the sequence specified in their respective arrays.- to be used with extreme care (by default empty)
postInitSQLRefs _SQLRefs_	List of references to ConfigMaps or Secrets containing SQL files to be executed as a superuser in the *postgres* database right after the cluster has been created. The references are processed in a specific order: first, all Secrets are processed, followed by all ConfigMaps. Within each group, the processing order follows the sequence specified in their respective arrays.- to be used with extreme care (by default empty)
seccompProfile **[Required]** _SeccompProfile_	The SeccompProfile applied to every Pod and Container. Defaults to:{" "} *RuntimeDefault*
metadata **[Required]** *_InheritedMetadata_*	Metadata applied exclusively to the generated Cluster resources. Useful for applying AppArmor profiles.
managed **[Required]** *Bootstrap*	The configuration that is used by the portions of PostgreSQL that are managed by the CNP instance manager
projectedVolumeTemplate _ProjectedVolumeSource_	Template to be used to define projected volumes, projected volumes will be mounted under */projected* base folder
tablespaces [_[]TablespaceConfiguration_](https://www.enterprisedb.com/docs/postgres_for_kubernetes/latest/pg4k.v1/#tablespaceconfiguration)	The tablespaces configuration
topologySpreadConstraints [_[]TopologySpreadConstraint_](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.28/#topologyspreadconstraint-v1-core)	TopologySpreadConstraints specifies how to spread matching pods among the given topology. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/
plugins [_[]PluginConfiguration_](https://www.enterprisedb.com/docs/postgres_for_kubernetes/latest/pg4k.v1/#pluginconfiguration)	The plugins configuration, containing any plugin to be loaded with the corresponding configuration

4.18.29 CnpConfiguration

Appears in:

- *PGDGroupSpec*

CnpConfiguration contains the configurations of the data nodes that will be injected into the resulting clusters composing the PGD group

Field	Description
CnpBaseConfiguration *_CnpBaseConfiguration_*	(Members of *CnpBaseConfiguration* are embedded into this type.) No description provided.

continues on next page

Table 29 – continued from previous page

Field	Description
joinMethod **[Required]** *_JoinMethod_*	JoinMethod specifies the method data nodes will use to join the local group. PGD supports two ways of joining a local group: logical and physical. In logical join, the node will join the group by using{" "} *bdr.join_node_group*, synchronizing data from scratch. In physical join the node will join the group by using{" "} *bdr_init_physical*, synchronizing data from a physical backup.
enableSuperuserAccess _bool_	When this option is enabled, the CNP operator will create or use the secret defined in the SuperuserSecret to allow superuser (postgres) access to the database. When this option is disabled on a running Group, the operator will ignore the content of the secret and set the password of the *postgres* user to *NULL*. Enabled by default.
superuserSecret _LocalObjectReference_	The secret containing the superuser password. A new secret will be created with a randomly generated password if not defined. This field is only allowed in the CNP Instances configuration. A Witness Node will always use the same SuperuserSecret as the other instances.

4.18.30 ConnectionString

(Alias of `map[string]string`)

Appears in:

- *PgdConfiguration*

ConnectionString represent the parameters to connect to a PostgreSQL cluster

4.18.31 ConnectivityConfiguration

Appears in:

- *PGDGroupSpec*

ConnectivityConfiguration describes how to generate the services and certificates for the PGDGroup

Field	Description
dns **[Required]** *_RootDNSConfiguration_*	Describes how the FQDN for the resources should be generated
tls **[Required]** *_TLSConfiguration_*	The configuration of the TLS infrastructure
nodeServiceTemplate **[Required]** *_ServiceTemplate_*	Instructs how to generate the service for each node
groupServiceTemplate **[Required]** *_ServiceTemplate_*	Instructs how to generate the service for the PGDGroup
proxyServiceTemplate **[Required]** *_ServiceTemplate_*	Instructs how to generate the service pointing to the PGD Proxy for write leader node routing
proxyReadServiceTemplate **[Required]** *_ServiceTemplate_*	Instructs how to generate the service pointing to the PGD Proxy for read nodes routing

4.18.32 ConnectivityStatus

Appears in:

- *PGDGroupStatus*

ConnectivityStatus contains any relevant status for the operator about Connectivity

Field	Description
replicationTLSCertificate **[Required]** _ReplicationCertificateStatus_	ReplicationTLSCertificate is the name of the replication TLS certificate, if we have it
nodeTLSCertificates **[Required]** [_[]NodeCertificateStatus_](NodeCertificateStatus)	NodeTLSCertificates are the names of the certificates that have been created for the PGD nodes
unusedCertificates **[Required]** _[]string_	UnusedCertificates are the names of the certificates that we don't use anymore for the PGD nodes
nodesWithoutCertificates **[Required]** _[]string_	NodesWithoutCertificates are the names of the nodes which have not a server certificate
nodesNeedingServiceReconciliation **[Required]** _[]string_	NodesNeedingServiceReconciliation are the names of the nodes which have not a server certificate
configurationHash **[Required]** _string_	ConfigurationHash is the hash code of the connectivity configuration, used to check if we had a change in the configuration or not

4.18.33 DNSConfiguration

Appears in:

- *RootDNSConfiguration*

DNSConfiguration describes how the FQDN for the resources should be generated

Field	Description
domain **[Required]** _string_	Contains the domain name of by all services in the PGDGroup. It is responsibility of the user to ensure that the value specified here matches with the rendered nodeServiceTemplate and groupServiceTemplate
hostSuffix **[Required]** _string_	Contains an optional suffix to add to all the service names in the PGDGroup. The meaning of this setting it to allow the user to easily mark all the services created in a location for routing purpose (i.e., add a generic rule to CoreDNS to rewrite some service suffixes as local)

4.18.34 DiscoveryJobConfig

Appears in:

- *PgdConfiguration*

DiscoveryJobConfig contains a series of fields that configure the discovery job

Field	Description
delay **[Required]** _int_	Delay amount of time to sleep between retries, measured in seconds
retries **[Required]** _int_	Retries how many times the operation should be retried
timeout **[Required]** _int_	Timeout amount of time given to the operation to succeed, measured in seconds

4.18.35 ImageCatalogRef

Appears in:

- *PGDGroupSpec*

ImageCatalogRef defines the referenced ImageCatalog and the referenced major Postgres version

Field	Description
TypedLocalObjectReference _TypedLocalObjectReference_	(Members of *TypedLocalObjectReference* are embedded into this type.) No description provided.
major **[Required]** _int_	The major version of PostgreSQL we want to use from the ImageCatalog

4.18.36 ImageCatalogSpec

Appears in:

- Example of a cluster-wide catalog using `ClusterImageCatalog resource <Example of a cluster-wide catalog using *ClusterImageCatalog* resource>`

ImageCatalogSpec defines the desired ImageCatalog

Field	Description
pgdImages **[Required]** [_[]CatalogImage_](CatalogImage)	List of CatalogImages available in the catalog
proxyImage **[Required]** _string_	The proxy image available in the catalog

4.18.37 ImageStatus

Appears in:

- *PGDGroupStatus*

ImageStatus defines the current used images

Field	Description
pgd **[Required]** _string_	PGD is the current used pgd image
proxy **[Required]** _string_	PGDProxy is the current used pgd proxy image

4.18.38 InheritedMetadata

Appears in:

- *CnpBaseConfiguration*

InheritedMetadata contains metadata to be inherited by all resources related to a Cluster

Field	Description
labels **[Required]** _map[string]string_	No description provided.
annotations **[Required]** _map[string]string_	No description provided.

4.18.39 JoinMethod

(Alias of `string`)

Appears in:

- *CnpConfiguration*

JoinMethod represents one of the supported methods of joining a node

4.18.40 Metadata

Appears in:

- *CertManagerTemplate*

Metadata is a structure similar to the metav1.ObjectMeta, but still parseable by controller-gen to create a suitable CRD for the user.

Field	Description
labels _map[string]string_	Map of string keys and values that can be used to organize and categorize (scope and select) objects. May match selectors of replication controllers and services. More info: http://kubernetes.io/docs/user-guide/labels
annotations _map[string]string_	Annotations is an unstructured key value map stored with a resource that may be set by external tools to store and retrieve arbitrary metadata. They are not queryable and should be preserved when modifying objects. More info: http://kubernetes.io/docs/user-guide/annotations

4.18.41 NodeCertificateStatus

Appears in:

- *ConnectivityStatus*

NodeCertificateStatus encapsulate the status of the server certificate of a CNP node

Field	Description
ReplicationCertificateStatus *_ReplicationCertificateStatus_*	(Members of *ReplicationCertificateStatus* are embedded into this type.) No description provided.
nodeName **[Required]** _string_	NodeName is the name of the CNP cluster using this certificate

4.18.42 NodeKindName

(Alias of `string`)

Appears in:

- *NodeSummary*

NodeKindName is a type containing the potential values of node_kind_name from bdr.node_summary

4.18.43 NodeSummary

Appears in:

- *PGDGroupStatus*

NodeSummary shows relevant info from bdr.node_summary

Field	Description
node_name **[Required]** _string_	Name of the node
node_group_name **[Required]** _string_	NodeGroupName is the name of the joined group
peer_state_name **[Required]** _string_	Consistent state of the node in human-readable form
peer_target_state_name **[Required]** _string_	State which the node is trying to reach (during join or promotion)
node_kind_name **[Required]** *_NodeKindName_*	The kind of node: witness or data

4.18.44 NodesExtensionsStatus

(Alias of `[]github.com/EnterpriseDB/pg4k-pgd/api/v1beta1.NodeExtensionStatus`)

NodesExtensionsStatus contains a list of NodeExtensionStatus entries

4.18.45 OTELConfiguration

Appears in:

- *CnpBaseConfiguration*

OTELConfiguration is the configuration for external openTelemetry

Field	Description
metricsURL **[Required]** _string_	The OpenTelemetry HTTP endpoint URL to accept metrics data
traceURL **[Required]** _string_	The OpenTelemetry HTTP endpoint URL to accept trace data
traceEnable **[Required]** _bool_	Whether to push trace data to OpenTelemetry traceUrl
tls **[Required]** *_OTELTLSConfiguration_*	TLSConfiguration provides the TLS certificate configuration when MetricsURL and TraceURL are using HTTPS

4.18.46 OTELTLSConfiguration

Appears in:

- *OTELConfiguration*

OTELTLSConfiguration contains the certificate configuration for TLS connections to openTelemetry

Field	Description
caBundleSecretRef **[Required]** _SecretKeySelector_	CABundleSecretRef is a reference to a secret field containing the CA bundle to verify the openTelemetry server certificate
clientCertSecret **[Required]** _LocalObjectReference_	ClientCertSecret is the name of the secret containing the client certificate used to connect to openTelemetry. It must contain both the standard "tls.crt" and "tls.key" files, encoded in PEM format.

4.18.47 OperatorPhase

(Alias of `string`)

Appears in:

- *PGDGroupStatus*

OperatorPhase it represents a phase of the PGDGroup controller

4.18.48 OperatorPhaseCleanup

(Alias of `string`)

Appears in:

- *PGDGroupCleanupStatus*

OperatorPhaseCleanup it represents a phase of the PGDGroupCleanup controller

4.18.49 PGDGroupCleanupSpec

Appears in:

- *PGDGroupCleanup*

PGDGroupCleanupSpec defines the desired state of PGDGroupCleanup

Field	Description
executor **[Required]** _string_	No description provided.
target **[Required]** _string_	No description provided.
force **[Required]** _bool_	Force will force the removal of the PGDGroup even if the target PGDGroup nodes are not parted

4.18.50 PGDGroupCleanupStatus

Appears in:

- *PGDGroupCleanup*

PGDGroupCleanupStatus defines the observed state of PGDGroupCleanup

Field	Description
phase **[Required]** *_OperatorPhaseCleanup_*	No description provided.

4.18.51 PGDGroupSpec

Appears in:

- *PGDGroup parting*

PGDGroupSpec defines the desired state of PGDGroup

Field	Description
imageName **[Required]** _string_	Name of the container image, supporting both tags (*<image>:<tag>*) and digests for deterministic and repeatable deployments (*<image>:<tag>@sha256:<digestValue>*)
imageCatalogRef *_ImageCatalogRef_*	Defines the referenced ImageCatalog and the referenced major Postgres version

continues on next page

Table 45 – continued from previous page

Field	Description
imagePullPolicy _PullPolicy_	Image pull policy. One of *Always*, *Never* or{" "} *IfNotPresent*. If not defined, it defaults to{" "} *IfNotPresent*. Cannot be updated. More info: https://kubernetes.io/docs/concepts/containerimages#updating-images
imagePullSecrets **[Required]** [_[]LocalObjectReference_](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.28/#localobjectreference-v1-core)	The list of pull secrets to be used to pull operator and or the operand images
inheritedMetadata **[Required]** *_InheritedMetadata_*	Metadata that will be inherited by all objects related to the pgdGroup
instances **[Required]** _int32_	Number of instances required in the cluster
proxyInstances **[Required]** _int32_	Number of proxy instances required in the cluster
witnessInstances **[Required]** _int32_	Number of witness instances required in the cluster
backup **[Required]** *_Backup_*	The configuration to be used for backups in the CNP instances.
restore **[Required]** *_Restore_*	The configuration to restore this PGD group from an Object Store service
cnp **[Required]** *_CnpConfiguration_*	Instances configuration that will be injected into the CNP clusters that compose the PGD Group
witness **[Required]** *_CnpBaseConfiguration_*	WitnessInstances configuration that will be injected into the WitnessInstances CNP clusters If not defined, it will default to the Instances configuration
pgd **[Required]** *_PgdConfiguration_*	Pgd contains instructions to bootstrap this cluster
pgdProxy **[Required]** *_PGDProxyConfiguration_*	PGDProxy contains instructions to configure PGD Proxy
connectivity **[Required]** *_ConnectivityConfiguration_*	Configures the connectivity of the PGDGroup, like services and certificates that will be used.
failingFinalizerTimeLimitSeconds **[Required]** _int32_	The amount of seconds that the operator will wait in case of a failing finalizer. A finalizer is considered failing when the operator cannot reach any nodes of the PGDGroup

4.18.52 PGDGroupStatus

Appears in:

- *PGDGroup parting*

PGDGroupStatus defines the observed state of PGDGroup

Field	Description
latestGeneratedNode **[Required]** _int32_	ID of the latest generated node (used to avoid node name clashing)
phase **[Required]** *_OperatorPhase_*	The initialization phase of this cluster
phaseDetails **[Required]** _string_	The details of the current phase
phaseTroubleshootHints **[Required]** _string_	PhaseTroubleshootHints general troubleshooting indications for the given phase
phaseType **[Required]** *_PhaseType_*	PhaseType describes the phase category.
conditions **[Required]** [_[]Condition_](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.28/#condition-v1-meta)	Conditions for PGDGroup object

continues on next page

Table 46 – continued from previous page

Field	Description
nodes **[Required]** [_[]NodeSummary_](NodeSummary)	The list of summaries for the nodes in the group
backup **[Required]** *_BackupStatus_*	The node that is taking backups of this PGDGroup
restore **[Required]** *_RestoreStatus_*	The status of the restore process
PGD **[Required]** *_PGDStatus_*	Last known status of PGD
CNP **[Required]** *_CNPStatus_*	Last known status of CNP
PGDProxy **[Required]** *_PGDProxyStatus_*	Last known status of PGDProxy
connectivity **[Required]** *_ConnectivityStatus_*	Last known status of Connectivity
pause **[Required]** *_PauseStatus_*	Last known status of Pause
image **[Required]** *_ImageStatus_*	Last known status of used image

4.18.53 PGDNodeGroupEntry

Appears in:

- *PGDStatus*

PGDNodeGroupEntry shows information about the node groups available in the PGD configuration

Field	Description
name **[Required]** _string_	Name is the name of the node group
enableProxyRouting **[Required]** _bool_	EnableProxyRouting is true is the node group allows running PGD Proxies
enableRaft **[Required]** _bool_	EnableRaft is true if the node group has a subgroup raft instance
routeWriterMaxLag **[Required]** _int64_	RouteWriterMaxLag Maximum lag in bytes of the new write candidate to be selected as write leader, if no candidate passes this, there will be no writer selected automatically
routeReaderMaxLag **[Required]** _int64_	RouteReaderMaxLag Maximum lag in bytes for node to be considered viable read-only node
routeWriterWaitFlush **[Required]** _bool_	RouteWriterWaitFlush Whether to wait for replication queue flush before switching to new leader when using{" "} *bdr.routing_leadership_transfer()*

4.18.54 PGDNodeGroupSettings

Appears in:

- *PgdConfiguration*

PGDNodeGroupSettings contains the settings of the PGD Group

Field	Description
routeWriterMaxLag **[Required]** _int64_	RouteWriterMaxLag Maximum lag in bytes of the new write candidate to be selected as write leader, if no candidate passes this, there will be no writer selected automatically Defaults to -1

continues on next page

Table 48 – continued from previous page

Field	Description
routeReaderMaxLag **[Required]** _int64_	RouteReaderMaxLag Maximum lag in bytes for node to be considered viable read-only node Defaults to -1
routeWriterWaitFlush **[Required]** _bool_	RouteWriterWaitFlush Whether to wait for replication queue flush before switching to new leader when using{" "} *bdr.routing_leadership_transfer()* Defaults to false

4.18.55 PGDProxyConfiguration

Appears in:

- *PGDGroupSpec*

PGDProxyConfiguration defines the configuration of PGD Proxy

Field	Description
imageName **[Required]** _string_	Name of the PGDProxy container image
logLevel **[Required]** _string_	The PGD Proxy log level, one of the following values: error, warning, info (default), debug, trace
logEncoder **[Required]** _string_	The format of the log output
proxyAffinity **[Required]** _Affinity_	ProxyAffinity/Anti-affinity rules for pods
proxyNodeSelector **[Required]** _map[string]string_	ProxyNodeSelector rules for pods
proxyTolerations **[Required]** [_[]Toleration_](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.28/#toleration-v1-core)	ProxyTolerations rules for pods
proxyResources _ResourceRequirements_	Defines the resources assigned to the proxy. If not defined uses defaults requests and limits values.

4.18.56 PGDProxyEntry

Appears in:

- *PGDStatus*

PGDProxyEntry shows information about the proxies available in the PGD configuration

Field	Description
name **[Required]** _string_	Name is the name of the proxy
fallbackGroupNames **[Required]** _[]string_	FallbackGroupNames are the names of the fallback groups configured for this proxy
parentGroupName **[Required]** _string_	ParentGroupName is the parent PGD group of this proxy
maxClientConn **[Required]** _int_	MaxClientConn maximum number of connections the proxy will accept
maxServerConn **[Required]** _int_	MaxServerConn maximum number of connections the proxy will make to the Postgres node
serverConnTimeout **[Required]** _int64_	ServerConnTimeout connection timeout for server connections in seconds

continues on next page

Table 50 – continued from previous page

Field	Description
serverConnKeepalive **[Required]** _int64_	ServerConnKeepalive keepalive interval for server connections in seconds
fallbackGroupTimeout **[Required]** _int64_	FallbackGroupTimeout the interval after which the routing falls back to one of the fallback_groups
consensusGracePeriod **[Required]** _int64_	ConsensusGracePeriod the duration in seconds for which proxy continues to route even upon loss of a Raft leader.
readListenPort **[Required]** _int_	ReadListenPort is the port where the proxy will listen and route queries to read nodes

4.18.57 PGDProxySettings

Appears in:

- *PgdConfiguration*

PGDProxySettings contains the settings of the proxy

Field	Description
fallbackGroups **[Required]** _[]string_	FallbackGroups is the list of groups the proxy should forward connection to when all the data nodes of this PGD group are not available
maxClientConn **[Required]** _int_	MaxClientConn maximum number of connections the proxy will accept. Defaults to 32767
maxServerConn **[Required]** _int_	MaxServerConn maximum number of connections the proxy will make to the Postgres node. Defaults to 32767
serverConnTimeout **[Required]** _int64_	ServerConnTimeout connection timeout for server connections in seconds. Defaults to 2
serverConnKeepalive **[Required]** _int64_	ServerConnKeepalive keepalive interval for server connections in seconds. Defaults to 10
fallbackGroupTimeout **[Required]** _int64_	FallbackGroupTimeout the interval after which the routing falls back to one of the fallback_groups. Defaults to 60
consensusGracePeriod **[Required]** _int64_	ConsensusGracePeriod the duration in seconds for which proxy continues to route even upon loss of a Raft leader. If set to 0s, proxy stops routing immediately. Defaults to 6
enableReadNodeRouting **[Required]** _bool_	EnableReadNodeRouting is the switch to control whether the proxy will route queries to read nodes through read_listen_port or not. By default it is false

4.18.58 PGDProxyStatus

Appears in:

- *PGDGroupStatus*

PGDProxyStatus any relevant status for the operator about PGDProxy

Field	Description
proxyInstances **[Required]** _int32_	No description provided.

continues on next page

Table 52 – continued from previous page

Field	Description
writeLead **[Required]** _string_	WriteLead is a reserved field for the operator, is not intended for external usage. Will be removed in future versions
proxyHash **[Required]** _string_	ProxyHash contains the hash we use to detect if we need to reconcile the proxies

4.18.59 PGDRaftStatus

(Alias of `string`)

Appears in:

- *PGDStatus*

PGDRaftStatus indicates a known status of the PGDRaft

4.18.60 PGDStatus

Appears in:

- *PGDGroupStatus*

PGDStatus any relevant status for the operator about PGD

Field	Description
raftConsensusLastChangedStatus **[Required]** *_PGDRaftStatus_*	RaftConsensusLastChangedStatus indicates the latest reported status from bdr.monitor_group_raft
raftConsensusLastChangedMessage **[Required]** _string_	RaftConsensusLastChangedMessage indicates the latest reported message from bdr.monitor_group_raft
raftConsensusLastChangedTimestamp **[Required]** _string_	RaftConsensusLastChangedTimestamp indicates when the status and message were first reported
registeredProxies **[Required]** [_[]PGDProxyEntry_](PGDProxyEntry)	RegisteredProxies is the status of the registered proxies
nodeGroup **[Required]** *_PGDNodeGroupEntry_*	NodeGroup is the status of the node group associated with the PGDGroup

4.18.61 ParentGroupConfiguration

Appears in:

- *PgdConfiguration*

ParentGroupConfiguration contains the topology configuration of PGD

Field	Description
name **[Required]** _string_	Name of the parent group
create **[Required]** _bool_	Create is true when the operator should create the parent group if it doesn't exist

4.18.62 PauseStatus

Appears in:

- *PGDGroupStatus*

PauseStatus contains the information of group hibernating

Field	Description
active **[Required]** _bool_	Active indicates the PGDGroup is either: _ in process of pausing _ already paused * in process of resuming
instances **[Required]** _int32_	Instances is the number of paused PGD instances
lastStartedTime **[Required]** _Time_	LastStartedTime is the last time the PGDGroup started pausing
lastCompletedTime **[Required]** _Time_	LastCompletedTime is last time the PGDGroup completed pausing
lastResumeStartedTime **[Required]** _Time_	LastResumeStartedTime is the last time the PGDGroup started resuming
lastResumeCompletedTime **[Required]** _Time_	LastCompletedTime is last time the PGDGroup completed resuming

4.18.63 PgdConfiguration

Appears in:

- *PGDGroupSpec*

PgdConfiguration is the configuration of the PGD group structure

Field	Description
parentGroup **[Required]** *_ParentGroupConfiguration_*	ParentGroup configures the topology of the PGD group
groupJoinMethod *_JoinMethod_*	GroupJoinMethod defines the method in case of cross region join
discovery **[Required]** [_[]ConnectionString_](ConnectionString)	The parameters we will use to connect to a node belonging to the parent PGD group. Even if provided, the following parameters will be overridden with default values: *application_name*, *sslmode*,{" "} *dbname* and *user*. The following parameters should not be provided nor used, as they are not even overridden with defaults:*sslkey*, *sslcert*,{" "} *sslrootcert*
discoveryJob **[Required]** *_DiscoveryJobConfig_*	DiscoveryJob the configuration of the PGD Discovery job
databaseName **[Required]** _string_	Name of the database used by the application. Default:{" "} *app*.
ownerName **[Required]** _string_	Name of the owner of the database in the instance to be used by applications. Defaults to the value of the *database* key.
ownerCredentialsSecret **[Required]** _LocalObjectReference_	Name of the secret containing the initial credentials for the owner of the user database. If empty a new secret will be created from scratch
proxySettings **[Required]** *_PGDProxySettings_*	Configuration for the proxy
nodeGroupSettings **[Required]** *_PGDNodeGroupSettings_*	Configuration for the PGD Group
globalRouting **[Required]** _bool_	GlobalRouting is true when global routing is enabled, and in this case the proxies will be created in the parent group
mutations **[Required]** *_SQLMutations_*	List of SQL mutations to apply to the node group

4.18.64 PhaseType

(Alias of `string`)

Appears in:

- *PGDGroupStatus*

PhaseType describes the type of the OperatorPhase

4.18.65 PreProvisionedCertificate

Appears in:

- *ClientPreProvisionedCertificates*

PreProvisionedCertificate contains the data needed to supply a pre-generated certificate

Field	Description
secretRef **[Required]** _string_	SecretRef a name pointing to a secret that contains a tls.crt and tls.key

4.18.66 ReconciliationStage

(Alias of `string`)

Appears in:

ReconciliationStage describes in which stage the reconciliation process is at

4.18.67 RecoverabilityPointsByMethod

(Alias of `map[github.com/EnterpriseDB/cloud-native-postgres/api/v1.BackupMethod]k8s.io/apimachinery/pkg/apis/meta/v1.Time`)

Appears in:

- *CNPStatus*

RecoverabilityPointsByMethod contains the first recoverability points for a given backup method

4.18.68 ReplicationCertificateStatus

Appears in:

- *ConnectivityStatus*

ReplicationCertificateStatus encapsulate the certificate status

Field	Description
name **[Required]** _string_	Name is the name of the certificate
hash **[Required]** _string_	Hash is the hash of the configuration for which it has been generated
isReady **[Required]** _bool_	Ready is true when the certificate is ready

continues on next page

Table 58 – continued from previous page

Field	Description
preProvisioned **[Required]** _bool_	PreProvisioned is true if the certificate is preProvisioned

4.18.69 Restore

Appears in:

- *PGDGroupSpec*

Restore configures the restore of a PGD group from an object store

Field	Description
volumeSnapshots *_VolumeSnapshotsConfiguration_*	The configuration for volumeSnapshot restore
barmanObjectStore **[Required]** _BarmanObjectStoreConfiguration_	The configuration for the barman-cloud tool suite
recoveryTarget **[Required]** *Bootstrap*	By default, the recovery process applies all the available WAL files in the archive (full recovery). However, you can also end the recovery as soon as a consistent state is reached or recover to a point-in-time (PITR) by specifying a *RecoveryTarget* object, as expected by PostgreSQL (i.e., timestamp, transaction Id, LSN, ...). More info: https://www.postgresql.org/docs/current/runtime-config-wal.html#RUNTIME-CONFIG-WAL-RECOVERY-TARGET
serverNames **[Required]** _[]string_	The list of server names to be used as a recovery origin. One of these servers will be elected as the seeding one when evaluating the recovery target, this option is only used when restore from barmanObjectStore.

4.18.70 RestoreStatus

Appears in:

- *PGDGroupStatus*

RestoreStatus contains the current status of the restore process

Field	Description
serverName **[Required]** _string_	The name of the server to be restored
VolumeSnapshots **[Required]** [_[]VolumeSnapshotRestoreStatus_](VolumeSnapshotRestoreStatus)	selected volumeSnapshots to restore

4.18.71 RootDNSConfiguration

Appears in:

- *ConnectivityConfiguration*

RootDNSConfiguration describes how the FQDN for the resources should be generated

Field	Description
*DNSConfiguration _DNSConfiguration_*	(Members of *DNSConfiguration* are embedded into this type.) No description provided.
additional **[Required]** [_[]DNSConfiguration_](DNSConfiguration)	AdditionalDNSConfigurations adds more possible FQDNs for the resources

4.18.72 SQLMutation

SQLMutation is a series of SQL statements to apply atomically

Field	Description
isApplied **[Required]** _[]string_	List of boolean-returning SQL queries. If any of them returns false the mutation will be applied
exec **[Required]** _[]string_	List of SQL queries to be executed to apply this mutation
*type _SQLMutationType_*	Type determines when the SQLMutation occurs. 'always': reconcile the mutation at each reconciliation cycle 'beforeSubgroupRaft': are executed only before the subgroupRaft is enabled If not specified, the Type defaults to 'always'.

4.18.73 SQLMutationType

(Alias of `string`)

Appears in:

- *SQLMutations*

SQLMutationType a supported type of SQL Mutation

4.18.74 SQLMutations

(Alias of `[]github.com/EnterpriseDB/pg4k-pgd/api/v1beta1.SQLMutation`)

Appears in:

- *PgdConfiguration*

SQLMutations A list of SQLMutation

4.18.75 ScheduledBackupSpec

Appears in:

- *Backup and restore*

ScheduledBackupSpec defines the desired state of ScheduledBackup

Field	Description
suspend **[Required]** _bool_	If this backup is suspended or not
immediate **[Required]** _bool_	If the first backup has to be immediately start after creation or not
schedule **[Required]** _string_	The schedule does not follow the same format used in Kubernetes CronJobs as it includes an additional second specifier, see https://pkg.go.dev/github.com/robfig/cron#hdr-CRON_Expression_Format
backupOwnerReference **[Required]** _string_	Indicates which ownerReference should be put inside the created backup resources. _ none: no owner reference for created backup objects (same behavior as before the field was introduced) _ self: sets the Scheduled backup object as owner of the backup * cluster: set the cluster as owner of the backup
target **[Required]** *Bootstrap*	The policy to decide which instance should perform this backup. If empty, it defaults to *cluster.spec.backup.target*. Available options are empty string, *primary* and{" "} *prefer-standby*.`primary` to have backups run always on primary instances, *prefer-standby* to have backups run preferably on the most updated standby, if available.
method *Bootstrap*	The backup method to be used, possible options are{" "} *barmanObjectStore* and *volumeSnapshot*. Defaults to:{" "} *barmanObjectStore*.
online _bool_	Whether the default type of backup with volume snapshots is online/hot (*true*, default) or offline/cold (*false*) Overrides the default setting specified in the cluster field '.spec.backup.volumeSnapshot.online'
onlineConfiguration _OnlineConfiguration_	Configuration parameters to control the online/hot backup with volume snapshots Overrides the default settings specified in the cluster '.backup.volumeSnapshot.onlineConfiguration' stanza

4.18.76 ScheduledBackupStatus

Appears in:

- *BackupStatus*

ScheduledBackupStatus contains the status of the scheduled backup

Field	Description
method **[Required]** *Bootstrap*	Method is the backup method used to take the backup
name **[Required]** _string_	Name is the name of the scheduled backup
hash **[Required]** _string_	Hash is the hash of the scheduled backup configuration

4.18.77 ServerCertConfiguration

Appears in:

- *OTELTLSConfiguration*

ServerCertConfiguration contains the information to generate the certificates for the nodes

Field	Description
caCertSecret **[Required]** _string_	CACertSecret is the secret of the CA to be injected into the CloudNativePG ServerCASecret configuration
certManager **[Required]** *_CertManagerTemplate_*	The cert-manager template used to generate the certificates

4.18.78 ServiceTemplate

Appears in:

- *ConnectivityConfiguration*

ServiceTemplate is a structure that allows the user to set a template for the Service generation.

Field	Description
metadata *_Metadata_*	Standard object's metadata. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata
spec _ServiceSpec_	Specification of the desired behavior of the service. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status
updateStrategy *_ServiceUpdateStrategy_*	UpdateStrategy indicates how to update the services generated by this template.

4.18.79 ServiceUpdateStrategy

(Alias of `string`)

Appears in:

- *ServiceTemplate*

ServiceUpdateStrategy defines the type for updating LoadBalancers. Allowed values are "patch" and "replace".

4.18.80 TLSConfiguration

Appears in:

- *ConnectivityConfiguration*

TLSConfiguration is the configuration of the TLS infrastructure used by PGD to connect to the nodes

Field	Description
mode **[Required]** *_TLSMode_*	No description provided.
serverCert **[Required]** *_ServerCertConfiguration_*	The configuration for the server certificates
clientCert **[Required]** *_ClientCertConfiguration_*	The configuration for the client certificates

4.18.81 TLSMode

(Alias of `string`)

Appears in:

- *OTELTLSConfiguration*

TLSMode describes which mode should be used for the node to node communications

4.18.82 VolumeSnapshotRestoreStatus

Appears in:

- *RestoreStatus*

VolumeSnapshotRestoreStatus the volumeSnapshot to restore

Field	Description
snapshotName **[Required]** _string_	SnapshotName is the snapshot name to restore
pvcRole **[Required]** _string_	PVCRole is the pvcRole snapshot to restore
tablespaceName **[Required]** _string_	TablespaceName is the tablespace name the snapshot belongs to, if the pvcRole is *PG_TABLESPACE*

4.18.83 VolumeSnapshotsConfiguration

Appears in:

- *Backup and restore*

VolumeSnapshotsConfiguration contains the configuration for the volumeSnapshots restore

Field	Description
selector **[Required]** _LabelSelector_	Label selector used to select the volumeSnapshot to restore

4.19 Supported versions

- This page lists the status for currently supported releases of EDB Postgres Distributed for Kubernetes*.

4.19.1 Naming Scheme

The EDB Postgres Distributed for Kubernetes (PGD4K) releases follows Semantic Versioning 2.0.0 and is structured as follows:

```
<major>.<minor>.<patch>
```

- `<major>` maps to the supported PGD version. Version 1.x corresponds to PGD version 5.x.
- `<minor>` is incremented for each supported PG4K Long-Term Support release dependency change.
- `<patch>` indexes the patches for the current `<minor>` release, representing small changes relative to the `<minor>` release.

Git tags for versions are prefixed with `v` .

4.19.2 General Support status

EDB Postgres Distributed for Kubernetes version 1.x is used to manage Postgres Distributed version 5.x. The general support status is as follows:

Version	Currently Supported	Release Date	Supported Kubernetes Versions	Supported PG4K Versions	Supported Postgres versions	Supported Postgres Distributed versions
1.x	Yes	Apr 24, 2024	v1.28 - v1.32	v1.22, v1.25	v13 - v17	v5.5.1 - v5.7.0

The Postgres (operand) versions are limited to those supported by EDB Postgres Distributed (PGD).

Important: Please be aware that this page is informative only. The Platform Compatibility page from the EDB website contains the official list of supported software and Kubernetes distributions.

4.19.3 Detailed support status

The EDB Postgres Distributed for Kubernetes (PGD4K) operator uses the EDB Postgres for Kubernetes (PG4K) operator to manage each PGD node as a PG4K Cluster. Each minor release reaches its End of Life in alignment with the corresponding PG4K LTS release. For details on the support scope of each PG4K LTS release, please refer to the platform compatibility

Version	Currently Supported	Release Date	Supported Kubernetes Versions	Supported PG4K Versions	Supported Postgres versions	Supported Postgres Distributed versions
1.1.x	Yes	Dec 25, 2024	1.29, 1.30, 1.31, 1.32	v1.22.8, v1.25.x	v13 - v17	v5.5.1 - v5.7.0
1.0.x	Yes	Apr 24, 2024	1.28, 1.29, 1.30, 1.31	v1.22.x	v13 - v17	v5.5.1 - v5.6.0

4.19.4 Supported versions on the Openshift Platform

Version	Supported OpenShift Versions	Supported PG4K Versions	Supported Cert-manager versions
1.1.1	v4.12 - v4.18	v1.22.8, v1.22.9, v1.25.x	>v1.10.0
1.1.0	v4.12 - v4.17	v1.22.8, v1.25.x	>v1.10.0
1.0.1	v4.12 - v4.16	v1.22.x	>v1.10.0
1.0.0	v4.12 - v4.15	>v1.18.0	>v1.10.0

Note: EDB Postgres Distributed for Kubernetes (PGD4K) version 1.0.0 did not restrict supported PG4K version to LTS releases. when upgrading PGD4K from 1.0.0 to 1.0.1, if the installed PG4K version exceeds 1.22, we must first uninstall the PG4K operator and reinstall it with a 1.22.x to satisfy the dependency requirements.

4.20 Known issues and limitations

These known issues and limitations are in the current release of EDB Postgres Distributed for Kubernetes.

4.20.1 Postgres major version upgrades

This version of EDB Postgres Distributed for Kubernetes **doesn't support** major version upgrades of Postgres.

4.20.2 Data migration

This version of EDB Postgres Distributed for Kubernetes doesn't support declarative import of data from other Postgres databases. To migrate schemas and data, you can use traditional Postgres migration tools such as EDBLoader or Migration Toolkit / Replication Server . You can also use `pg_dump ...` on the source database and pipe the command's output to your target database with `psql -c` .

4.20.3 Connectivity with PgBouncer

EDB Postgres Distributed for Kubernetes **doesn't support** using PgBouncer to pool client connection requests. This limitation applies to both the open-source and EDB versions of PgBouncer.

4.20.4 Backup operations

To configure an EDB Postgres Distributed for Kubernetes environment, you must apply a `PGDGroup` YAML object to each Kubernetes cluster. Applying this object creates all necessary services for implementing a distributed architecture.

If you added a `spec.backup` section to this `PGDGroup` object with the goal of setting up a backup configuration, the backup will fail unless you also set the `spec.backup.cron.schedule` value.

Error output example:

```
The PGDGroup "region-a" is invalid: spec.backup.cron.schedule: Invalid value: "": Empty␣
↪spec string
```

Workaround

To work around this issue, add a `spec.backup.cron.schedule` section with a schedule that meets your requirements, for example:

```yaml
spec:
  instances: 3
  proxyInstances: 2
  pgd:
    parentGroup:
      create: true
      name: world
  backup:
    configuration:
      barmanObjectStore:
        ...
    cron:
      suspend: false
      immediate: true
      schedule: "0 */5 * * * *"
```

4.20.5 Known issues and limitations in EDB Postgres Distributed

All issues and limitations known for the EDB Postgres Distributed version that you include in your deployment also affect your EDB Postgres Distributed for Kubernetes instance.

For example, if the EDB Postgres Distributed version you're using is 5.x, your EDB Postgres Distributed for Kubernetes instance will be affected by these *5.x known issues* and 5.x limitations .

4.21 Image catalog

A PGD group has its own `ImageCatalog` and `ClusterImageCatalog` . `ImageCatalog` and `ClusterImageCatalog` are essential resources that allow you to define PGD and PGD Proxy images for creating a `PGDGroup` .

The key distinction is in their scope: an `ImageCatalog` is namespaced, while a `ClusterImageCatalog` is cluster scoped.

Both share a common structure, comprising a list of PGD images and a Proxy image. Each PGD image contains a `major` field indicating the major Postgres version of the image.

> **Warning:** The operator trusts the user-defined major version and doesn't conduct any PostgreSQL version detection. It's your responsibility to ensure alignment between the declared major version in the catalog and the PostgreSQL image.

The `major` field' s value must remain unique in a catalog, preventing duplication across images. However, distinct catalogs can expose different images under the same `major` value.

4.21.1 Example of a namespaced `ImageCatalog`

```yaml
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: ImageCatalog
metadata:
  name: pgd-postgres-extended
  namespace: default
spec:
  images:
    - major: 15
      image: docker.enterprisedb.com/k8s_enterprise_pgd/edb-postgres-extended:15.12-5.7.
↪0-1
    - major: 16
      image:  docker.enterprisedb.com/k8s_enterprise_pgd/edb-postgres-extended:16.8-5.7.
↪0-1
  proxyImage: docker.enterprisedb.com/k8s_enterprise_pgd/edb-pgd-proxy:5.7.0-1
```

4.21.2 Example of a cluster-wide catalog using `ClusterImageCatalog` resource

```yaml
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: ClusterImageCatalog
metadata:
  name: pgd-postgres-extended
spec:
  images:
    - major: 15
      image: docker.enterprisedb.com/k8s_enterprise_pgd/edb-postgres-extended:15.12-5.7.
↪0-1
    - major: 16
      image:  docker.enterprisedb.com/k8s_enterprise_pgd/edb-postgres-extended:16.8-5.7.
```

(continues on next page)

(continued from previous page)

```
↪0-1
  proxyImage: docker.enterprisedb.com/k8s_enterprise_pgd/edb-pgd-proxy:5.7.0-1
```

A PGDGroup resource has the flexibility to reference either an ImageCatalog or a ClusterImageCatalog to precisely specify the desired image:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: PGDGroup
metadata:
  name: group-example
spec:
  [...]
  imageCatalogRef:
    apiGroup: pgd.k8s.enterprisedb.io
    kind: ImageCatalog
    name: pgd-postgres-extended
    major: 16
  [...]
```

PGD groups utilizing these catalogs maintain continuous monitoring. Any alterations to the images in a catalog trigger automatic updates for all associated PGD groups that reference that specific entry.

4.22 Join method

When scaling up a PGD group by adding more data nodes or creating cross-region groups, it's crucial for new nodes to join the existing group properly. There are two methods for joining:

- Logical join

This method uses the bdr.join_node_group function to integrate the new node into the PGD group. It's important that the joining node doesn't contain any schemas or data present in the PGD group. We recommend that the new database contain only the BDR extension, as data synchronization occurs during the join.

- Physical join

This method uses the bdr_init_physical command to speed up the joining process. You can prepare data in advance before executing bdr_init_physical .

For more information about join methods, see Creating and joining PGD groups .

4.22.1 The initial group

The PGD4K operator allows you to configure the join methods for nodes joining PGD groups as well as for groups joining other groups.

When creating a single PGD group, you must configure it to create the *parent group* (spec.pgd.parentGroup.create =true). If you intend to create multiple PGD groups that will cross join, at least one of these groups must be designated to form the parent group. This group is referred to as the *initial* PGD group.

The first node in the initial PGD group can either be restored from a backup (as described in the recovery documentation) or created from scratch. This node will always use a logical join to connect to the parent group it created. Conversely, the first node in the *non-initial* PGD group is used for cross joining with other PGD groups.

You can configure the join method for the first node (group join) and the subsequent nodes (node join) separately.

Note: In the scenario where a group joins another group, the first node of the joining group actually creates the group first before joining a different group (either physically or logically).

4.22.2 Configure the join method for node join

You can configure the join method for a node joining a group when scaling up the PGD group and adding new nodes to the local group. In this context, the joining group is the same as the node group.

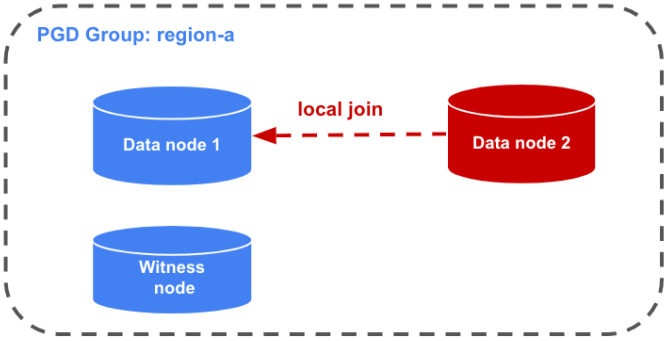

Fig. 3: node join local group

The `spec.cnp.joinMethod` parameter configures how a node joins a local group. By default, a logical join is used. To enable a physical join for a new node, set `spec.cnp.joinMethod =physical` .

In the following example, the second node uses the `pg_basebackup` command to replicate data from the first node in the `region-a` group. Then it uses `bdr_init_physical` to join the `region-a` group.

This example shows a physical join:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-a
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  cnp:
    joinMethod: physical
    storage:
      size: 1Gi
  ...
```

4.22.3 Configure the join method for group join

You can configure the join method for group joins when creating cross-region PGD groups. The first node in the new PGD group joins an existing PGD group. In this context, the joining group is distinct from the node group.

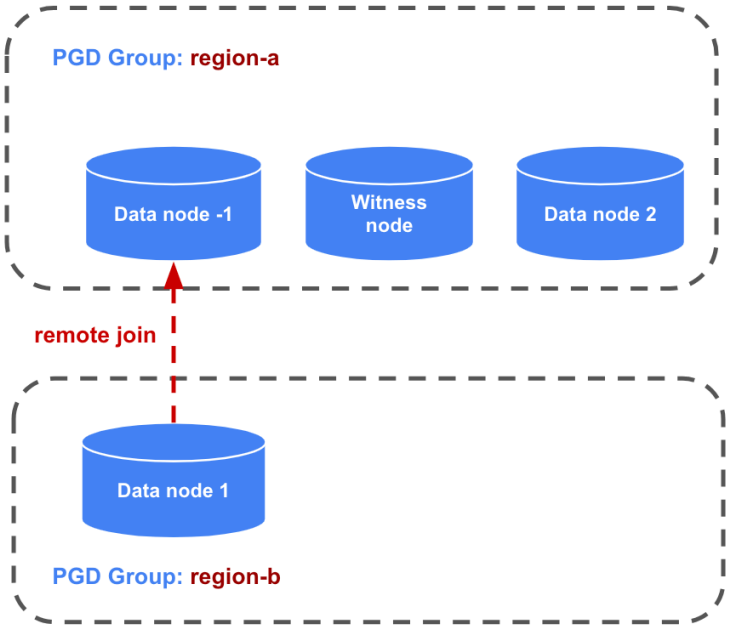

Fig. 4: node join remote group

The PGD4K operator uses the `spec.pgd.joinMethod` section (which defaults to logical) to determine how PGD groups join with each other. It uses the `spec.pgd.discovery` section to specify the target to join.

This example shows a logical join, where `spec.pgd.groupJoinMethod=logical` . The first node in the region-b group waits for all hosts specified in `spec.pgd.discovery` to become available. It then identifies a suitable target to perform the logical join using the `bdr.join_node_group` function. The *group service* of all remote groups is typically defined in the discovery section for this type of join.

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-b
  namespace: region-b
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    groupJoinMethod: logical
    discovery:
      - host: region-a-group
```

(continues on next page)

(continued from previous page)

```
      - host: region-c-group
  cnp:
    storage:
      size: 1Gi
```

This example shows a physical join, configured with `spec.pgd.groupJoinMethod=physical` . The first node in the region-b group will traverse all the nodes defined in `spec.pgd.discovery` , waiting to find one node with a successful Raft consensus to perform the physical join using `bdr_init_physical` . Typically, the *node service* of the remote group is specified in the discovery section for this type of join.

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-b
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    groupJoinMethod: physical
    discovery:
      - host: region-a-3-node
      - host: region-a-1-node
      - host: region-a-2-node
  cnp:
    joinMethod: physical
    storage:
      size: 1Gi
```

Note: A group join is essentially a type of node join, specifically when the first node in the group joins a node from a different group.

Note: Witness nodes always use a logical join to connect to a group.