Documentation

1.3.0 (August 4, 2022)

Percona Technical Documentation Team

Percona LLC and/or its affiliates, © 2009 - 2022

Table of contents

1. Percona Operator for PostgreSQL 4
2. Requirements 4
3. Quickstart guides 4
4. Installation guides 4
5. Configuration 4
6. Management 4
7. HOWTOs 5
8. Reference 5
9. Requirements 6
   - 9.1 System Requirements 6
   - 9.2 Design overview 7
10. Quickstart guides 10
    - 10.1 Install Percona Distribution for PostgreSQL on Minikube 10
    - 10.2 Install Percona Distribution for PostgreSQL on Google Kubernetes Engine (GKE) 15
    - 10.3 Install Percona Distribution for PostgreSQL using Helm 18
11. Installation guide 21
    - 11.1 Install Percona Distribution for PostgreSQL on Kubernetes 22
    - 11.2 Install Percona Distribution for PostgreSQL on OpenShift 25
12. Configuration 29
    - 12.1 Users 29
    - 12.2 Changing PostgreSQL Options 31
    - 12.3 Binding Percona Distribution for PostgreSQL components to Specific Kubernetes/OpenShift Nodes 34
    - 12.4 Transport Layer Security (TLS) 37
    - 12.5 Telemetry 42
13. Management 43
    - 13.1 Providing Backups 43
    - 13.2 Update Percona Operator for PostgreSQL 50
    - 13.3 Scale Percona Distribution for PostgreSQL on Kubernetes and OpenShift 54
    - 13.4 Monitoring 55
    - 13.5 Pause/resume PostgreSQL Cluster 57
14. How to 58
    - 14.1 How to deploy a standby cluster for Disaster Recovery 58
    - 14.2 Percona Operator for PostgreSQL single-namespace and multi-namespace deployment 62
    - 14.3 Using PostgreSQL tablespaces with Percona Operator for PostgreSQL 68

Percona LLC and/or its affiliates, © 2009 - 2022

15. Reference 72
15.1 Custom Resource options 72
15.2 Percona certified images 97
15.3 Frequently Asked Questions 100
16. Release Notes 103
16.1 *Percona Operator for PostgreSQL* 1.3.0 103
16.2 *Percona Operator for PostgreSQL* 1.2.0 105
16.3 *Percona Distribution for PostgreSQL Operator* 1.1.0 107
16.4 *Percona Distribution for PostgreSQL Operator* 1.0.0 108
16.5 *Percona Distribution for PostgreSQL Operator* 0.2.0 110
16.6 *Percona Distribution for PostgreSQL Operator* 0.1.0 111

Percona LLC and/or its affiliates, © 2009 - 2022

1. Percona Operator for PostgreSQL

Kubernetes have added a way to manage containerized systems, including database clusters. This management is achieved by controllers, declared in configuration files. These controllers provide automation with the ability to create objects, such as a container or a group of containers called pods, to listen for an specific event and then perform a task.

This automation adds a level of complexity to the container-based architecture and stateful applications, such as a database. A Kubernetes Operator is a special type of controller introduced to simplify complex deployments. The Operator extends the Kubernetes API with custom resources.

The Percona Operator for PostgreSQL is based on best practices for configuration and setup of a Percona Distribution for PostgreSQL cluster. The benefits of the Operator are many, but saving time and delivering a consistent and vetted environment is key.

2. Requirements

- System Requirements
- Design and architecture

3. Quickstart guides

- Install on Minikube
- Install on Google Kubernetes Engine (GKE)
- Install with Helm

4. Installation guides

- Generic Kubernetes installation
- Install on OpenShift

5. Configuration

- Application and system users
- Changing PostgreSQL Options
- Anti-affinity and tolerations
- Transport Encryption (TLS/SSL)
- Telemetry

6. Management

- Backup and restore
- Upgrade Percona Distribution for PostgreSQL and the Operator
- Horizontal and vertical scaling

Percona LLC and/or its affiliates, © 2009 - 2022

- Monitor with Percona Monitoring and Management (PMM)
- Restart or pause the cluster

7. HOWTOs

- How to deploy a standby cluster for Disaster Recovery
- Percona Operator for PostgreSQL single-namespace and multi-namespace deployment
- Using PostgreSQL tablespaces with Percona Operator for PostgreSQL

8. Reference

- Custom Resource options
- Percona certified images
- Frequently Asked Questions
- Release Notes

Last update: 2022-08-03

Percona LLC and/or its affiliates, © 2009 - 2022

9. Requirements

9.1 System Requirements

The Operator is validated for deployment on Kubernetes, GKE and EKS clusters. The Operator is cloud native and storage agnostic, working with a wide variety of storage classes, hostPath, and NFS.

9.1.1 Officially supported platforms

The following platforms were tested and are officially supported by the Operator 1.3.0:

- Google Kubernetes Engine (GKE) 1.21 - 1.24
- Amazon Elastic Container Service for Kubernetes (EKS) 1.20 - 1.22
- OpenShift Container Platform 4.7 - 4.10

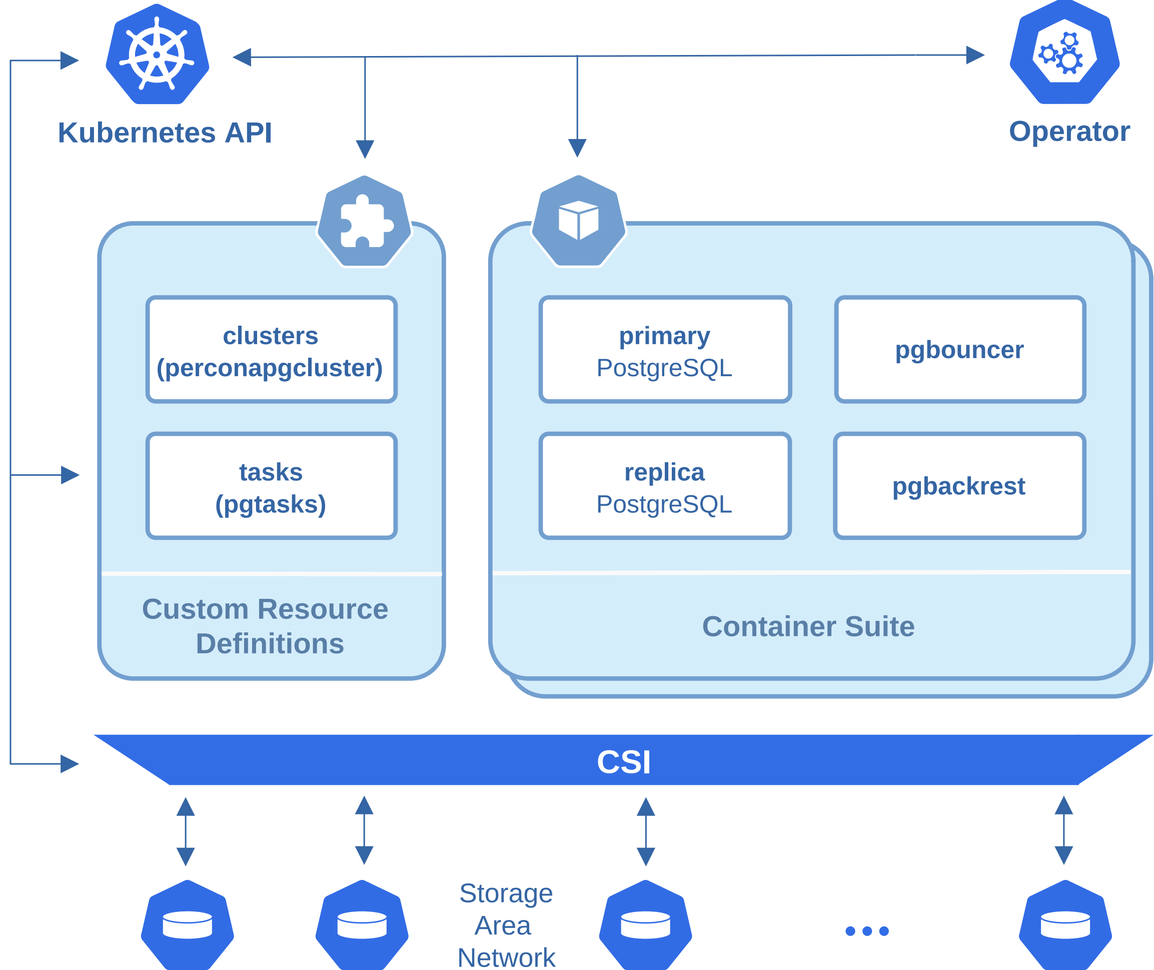
Other Kubernetes platforms may also work but have not been tested.

Last update: 2022-08-04

Percona LLC and/or its affiliates, © 2009 - 2022

9.2 Design overview

The Percona Operator for PostgreSQL automates and simplifies deploying and managing open source PostgreSQL clusters on Kubernetes. The Operator is based on CrunchyData's PostgreSQL Operator.

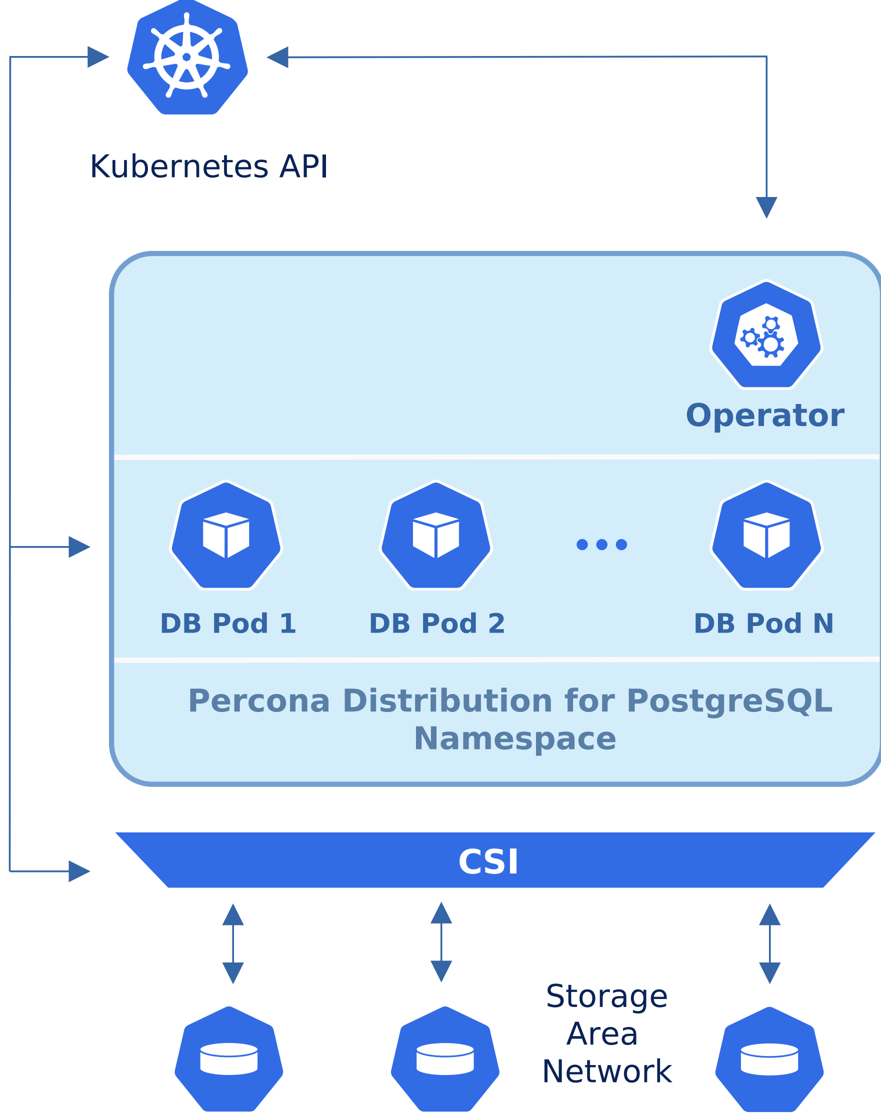


PostgreSQL containers deployed with the Operator include the following components:

- The PostgreSQL database management system, including:
  - PostgreSQL Additional Supplied Modules,
  - pgAudit PostgreSQL auditing extension,
  - PostgreSQL set_user Extension Module,
  - wal2json output plugin,
- The pgBackRest Backup & Restore utility,
- The pgBouncer connection pooler for PostgreSQL,
- The PostgreSQL high-availability implementation based on the Patroni template,
- the pg_stat_monitor PostgreSQL Query Performance Monitoring utility,
- LLVM (for JIT compilation).

To provide high availability the Operator involves node affinity to run PostgreSQL Cluster instances on separate worker nodes if possible. If some node fails, the Pod with it is automatically re-created on another node.

Percona LLC and/or its affiliates, © 2009 - 2022



To provide data storage for stateful applications, Kubernetes uses Persistent Volumes. A *PersistentVolumeClaim* (PVC) is used to implement the automatic storage provisioning to pods. If a failure occurs, the Container Storage Interface (CSI) should be able to re-mount storage on a different node.

The Operator functionality extends the Kubernetes API with Custom Resources Definitions. These CRDs provide extensions to the Kubernetes API, and, in the case of the Operator, allow you to perform actions such as creating a PostgreSQL Cluster, updating PostgreSQL Cluster resource allocations, adding additional utilities to a PostgreSQL cluster, e.g. pgBouncer for connection pooling and more.

When a new Custom Resource is created or an existing one undergoes some changes or deletion, the Operator automatically creates/changes/deletes all needed Kubernetes objects with the appropriate settings to provide a proper Percona PostgreSQL Cluster operation.

Percona LLC and/or its affiliates, © 2009 - 2022

Following CRDs are created while the Operator installation:

- `pgclusters` stores information required to manage a PostgreSQL cluster. This includes things like the cluster name, what storage and resource classes to use, which version of PostgreSQL to run, information about how to maintain a high-availability cluster, etc.
- `pgreplicas` stores information required to manage the replicas within a PostgreSQL cluster. This includes things like the number of replicas, what storage and resource classes to use, special affinity rules, etc.
- `pgtasks` is a general purpose CRD that accepts a type of task that is needed to run against a cluster (e.g. take a backup) and tracks the state of said task through its workflow.

Last update: 2022-07-12

Percona LLC and/or its affiliates, © 2009 - 2022

10. Quickstart guides

10.1 Install Percona Distribution for PostgreSQL on Minikube

Installing the Percona Operator for PostgreSQL on minikube is the easiest way to try it locally without a cloud provider. Minikube runs Kubernetes on GNU/Linux, Windows, or macOS system using a system-wide hypervisor, such as VirtualBox, KVM/QEMU, VMware Fusion or Hyper-V. Using it is a popular way to test the Kubernetes application locally prior to deploying it on a cloud.

Percona LLC and/or its affiliates, © 2009 - 2022

The following steps are needed to run Percona Operator for PostgreSQL on minikube:

Percona LLC and/or its affiliates, © 2009 - 2022

1. Install minikube, using a way recommended for your system. This includes the installation of the following three components:

   a. kubectl tool,

   b. a hypervisor, if it is not already installed,

   c. actual minikube package

   After the installation, run `minikube start` command. Being executed, this command will download needed virtualized images, then initialize and run the cluster. After minikube is successfully started, you can optionally run the Kubernetes dashboard, which visually represents the state of your cluster. Executing `minikube dashboard` will start the dashboard and open it in your default web browser.

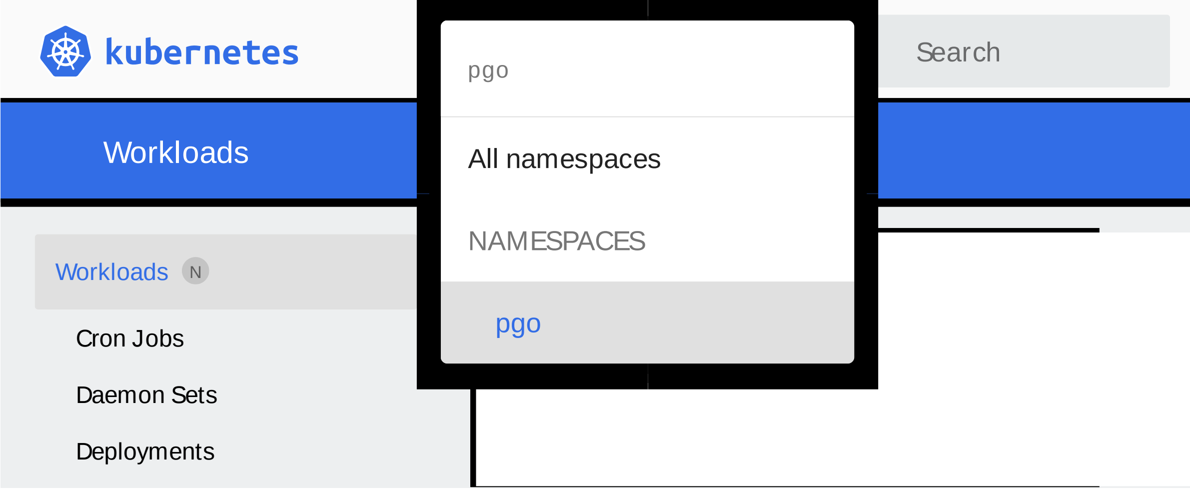
2. The first thing to do is to add the `pgo` namespace to Kubernetes, not forgetting to set the correspondent context for further steps:

```
$ kubectl create namespace pgo
$ kubectl config set-context $(kubectl config current-context) --namespace=pgo
```

Note

To use different namespace, you should edit *all occurrences* of the `namespace: pgo` line in both `deploy/cr.yaml` and `deploy/operator.yaml` configuration files.

If you use Kubernetes dashboard, choose your newly created namespace to be shown instead of the default one:



3. Deploy the operator with the following command:

```
$ kubectl apply -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/
v1.3.0/deploy/operator.yaml
```

4. Deploy Percona Distribution for PostgreSQL:

```
$ kubectl apply -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/
v1.3.0/deploy/cr-minimal.yaml
```

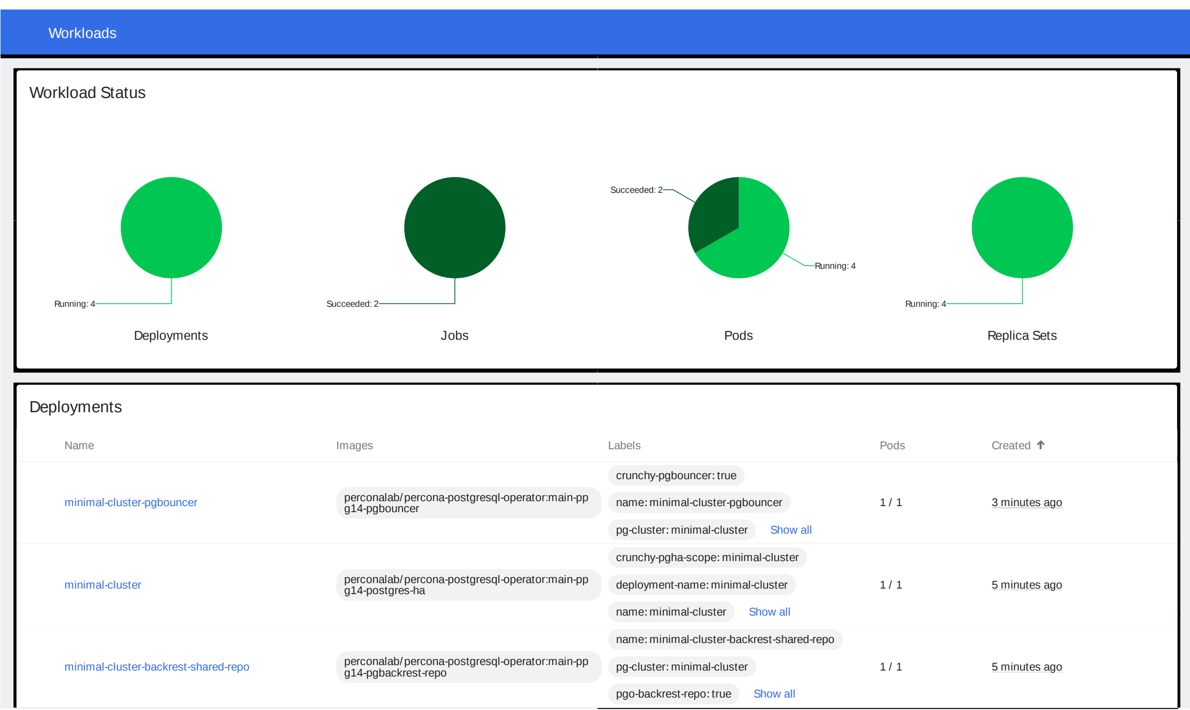
   This deploys PostgreSQL on one node, because `deploy/cr-minimal.yaml` is for minimal non-production deployment. For more configuration options please see `deploy/cr.yaml` and Custom Resource Options.

   Creation process will take some time. The process is over when both operator and replica set pod have reached their Running status:

Percona LLC and/or its affiliates, © 2009 - 2022

```
$ kubectl get pods
NAME                                                READY   STATUS      RESTARTS   AGE
backrest-backup-minimal-cluster-dcvkw               0/1     Completed   0          68s
minimal-cluster-6dfd645d94-42xsr                    1/1     Running     0          2m5s
minimal-cluster-backrest-shared-repo-77bd498dfd-9msvp   1/1     Running     0          2m23s
minimal-cluster-pgbouncer-594bf56d-kjwrp            1/1     Running     0          84s
pgo-deploy-lnbv7                                    0/1     Completed   0          4m14s
postgres-operator-6c4c558c5-dkk8v                   4/4     Running     0          3m37s
```

You can also track the progress via the Kubernetes dashboard:



5. During previous steps, the Operator has generated several secrets, including the password for the `pguser` user, which you will need to access the cluster.

   Use `kubectl get secrets` command to see the list of Secrets objects(by default Secrets object you are interested in has `cluster1-pguser-secret` name). Then `kubectl get secret cluster1-pguser-secret -o yaml` will return the YAML file with generated secrets, including the password which should look as follows:

```
...
data:
  ...
  password: cGd1c2VyX3Bhc3N3b3JkCg==
```

   Here the actual password is base64-encoded, and `echo 'cGd1c2VyX3Bhc3N3b3JkCg==' | base64 --decode` will bring it back to a human-readable form (in this example it will be a `pguser_password` string).

6. Check connectivity to a newly created cluster.

   Run new Pod to use it as a client and connect its console output to your terminal (running it may require some time to deploy). When you see the command line prompt of the newly created Pod, run run `psql` tool using the password obtained from the secret:

```
$ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-postgresql:
14.4 --restart=Never -- bash -il
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer -p 5432 -U
pguser pgdb
```


Percona LLC and/or its affiliates, © 2009 - 2022

This command will connect you to the PostgreSQL interactive terminal.

```
psql (14.4)
Type "help" for help.
pgdb=>
```

Last update: 2022-07-19

Percona LLC and/or its affiliates, © 2009 - 2022

10.2 Install Percona Distribution for PostgreSQL on Google Kubernetes Engine (GKE)

Following steps will allow you to install the Operator and use it to manage Percona Distribution for PostgreSQL with the Google Kubernetes Engine. The document assumes some experience with Google Kubernetes Engine (GKE). For more information on the GKE, see the Kubernetes Engine Quickstart.

10.2.1 Prerequisites

All commands from this quickstart can be run either in the **Google Cloud shell** or in **your local shell**.

To use *Google Cloud shell*, you need nothing but a modern web browser.

If you would like to use *your local shell*, install the following:

1. gcloud. This tool is part of the Google Cloud SDK. To install it, select your operating system on the official Google Cloud SDK documentation page and then follow the instructions.
2. kubectl. It is the Kubernetes command-line tool you will use to manage and deploy applications. To install the tool, run the following command:

```
$ gcloud auth login
$ gcloud components install kubectl
```

10.2.2 Configuring default settings for the cluster

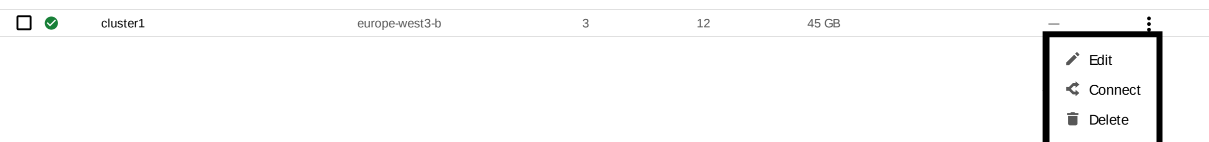
You can configure the settings using the `gcloud` tool. You can run it either in the Cloud Shell or in your local shell (if you have installed Google Cloud SDK locally on the previous step). The following command will create a cluster named `my-cluster-1`:

```
$ gcloud container clusters create cluster-1 --project <project name> --zone us-central1-a --
cluster-version  --machine-type n1-standard-4 --num-nodes=3
```

Note

You must edit the following command and other command-line statements to replace the `<project name>` placeholder with your project name. You may also be required to edit the *zone location*, which is set to `us-central1` in the above example. Other parameters specify that we are creating a cluster with 3 nodes and with machine type of 4 vCPUs and 45 GB memory.

You may wait a few minutes for the cluster to be generated, and then you will see it listed in the Google Cloud console (select *Kubernetes Engine* → *Clusters* in the left menu panel):



Now you should configure the command-line access to your newly created cluster to make `kubectl` be able to use it.

Percona LLC and/or its affiliates, © 2009 - 2022

In the Google Cloud Console, select your cluster and then click the *Connect* shown on the above image. You will see the connect statement configures command-line access. After you have edited the statement, you may run the command in your local shell:

```
$ gcloud container clusters get-credentials cluster-1 --zone us-central1-a --project <project
name>
```

10.2.3 Installing the Operator

1. First of all, use your Cloud Identity and Access Management (Cloud IAM) to control access to the cluster. The following command will give you the ability to create Roles and RoleBindings:

   ```
   $ kubectl create clusterrolebinding cluster-admin-binding --clusterrole cluster-admin --user
   $(gcloud config get-value core/account)
   ```

   The return statement confirms the creation:

   ```
   clusterrolebinding.rbac.authorization.k8s.io/cluster-admin-binding created
   ```

2. Use the following `git clone` command to download the correct branch of the percona-postgresql-operator repository:

   ```
   git clone -b v1.3.0 https://github.com/percona/percona-postgresql-operator
   cd percona-postgresql-operator
   ```

3. The next thing to do is to add the `pgo` namespace to Kubernetes, not forgetting to set the correspondent context for further steps:

   ```
   $ kubectl create namespace pgo
   $ kubectl config set-context $(kubectl config current-context) --namespace=pgo
   ```

   Note

   To use different namespace, you should edit *all occurrences* of the `namespace: pgo` line in both `deploy/cr.yaml` and `deploy/operator.yaml` configuration files.

4. Deploy the operator with the following command:

   ```
   $ kubectl apply -f deploy/operator.yaml
   ```

5. After the operator is started Percona Distribution for PostgreSQL can be created at any time with the following commands:

   ```
   $ kubectl apply -f deploy/cr.yaml
   ```

   Creation process will take some time. The process is over when the Operator and PostgreSQL Pods have reached their Running status:

   ```
   $ kubectl get pods
   NAME                                              READY   STATUS      RESTARTS   AGE
   backrest-backup-cluster1-4nq2x                    0/1     Completed   0          10m
   cluster1-6c9d4f9678-qdfx2                         1/1     Running     0          10m
   cluster1-backrest-shared-repo-7cb4dd8f8f-sh5gg    1/1     Running     0          10m
   ```


Percona LLC and/or its affiliates, © 2009 - 2022

```
cluster1-pgbouncer-6cd69d8966-vlxdt                 1/1     Running    0          10m
pgo-deploy-bp2ts                                    0/1     Completed 0           5m
postgres-operator-67f58bcb8c-9p4tl                  4/4     Running    1          5m
```

Also, you can see the same information when browsing Pods of your cluster in Google Cloud console via the *Object Browser*:

Name	Status	Type	Namespace	Cluster	Location
core		API Group			
Pod		Kind			
backrest-backup-cluster1-t6s42	Succeeded	Pod	pgo	cluster1	europe-west3-b
cluster1-6c9d4f9678-qdfx2	Running	Pod	pgo	cluster1	europe-west3-b
cluster1-backrest-shared-repo-7cb4dd8f8f-sh5gg	Running	Pod	pgo	cluster1	europe-west3-b
cluster1-pgbouncer-6cd69d8966-vlxdt	Running	Pod	pgo	cluster1	europe-west3-b
pgo-deploy-bp2ts	Succeeded	Pod	pgo	cluster1	europe-west3-b
postgres-operator-67f58bcb8c-9p4tl	Running	Pod	pgo	cluster1	europe-west3-b

6. During previous steps, the Operator has generated several secrets, including the password for the `pguser` user, which you will need to access the cluster.

   Use `kubectl get secrets` command to see the list of Secrets objects (by default Secrets object you are interested in has `cluster1-pguser-secret` name). Then `kubectl get secret cluster1-pguser-secret -o yaml` will return the YAML file with generated secrets, including the password which should look as follows:

   ```
   ...
   data:
     ...
     password: cGd1c2VyX3Bhc3N3b3JkCg==
   ```

   Here the actual password is base64-encoded, and `echo 'cGd1c2VyX3Bhc3N3b3JkCg==' | base64 --decode` will bring it back to a human-readable form (in this example it will be a `pguser_password` string).

7. Check connectivity to newly created cluster

   ```
   $ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-postgresql:
   14.4 --restart=Never -- bash -il
   [postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer -p 5432 -U
   pguser pgdb
   ```

   This command will connect you to the PostgreSQL interactive terminal.

   ```
   psql (14.4)
   Type "help" for help.
   pgdb=>
   ```

Last update: 2022-07-19

Percona LLC and/or its affiliates, © 2009 - 2022

10.3 Install Percona Distribution for PostgreSQL using Helm

Helm is the package manager for Kubernetes. Percona Helm charts can be found in percona/percona-helm-charts repository in Github.

10.3.1 Pre-requisites

Install Helm following its official installation instructions.

Note

Helm v3 is needed to run the following steps.

10.3.2 Installation

1. Add the Percona's Helm charts repository and make your Helm client up to date with it:

```
$ helm repo add percona https://percona.github.io/percona-helm-charts/
$ helm repo update
```

2. Install the Percona Operator for PostgreSQL:

```
$ helm install my-operator percona/pg-operator --version 1.3.0
```

The `my-operator` parameter in the above example is the name of a new release object which is created for the Operator when you install its Helm chart (use any name you like).

Note

If nothing explicitly specified, `helm install` command will work with `default` namespace. To use different namespace, provide it with the following additional parameter: `--namespace my-namespace`.

3. Install PostgreSQL:

```
$ helm install my-db percona/pg-db --version 1.3.0 --namespace my-namespace
```

The `my-db` parameter in the above example is the name of a new release object which is created for the Percona Distribution for PostgreSQL when you install its Helm chart (use any name you like).

10.3.3 Installing Percona Distribution for PostgreSQL with customized parameters

The command above installs Percona Distribution for PostgreSQL with default parameters. Custom options can be passed to a `helm install` command as a `--set key=value[,key=value]` argument. The options passed with a chart can be any of the Operator's Custom Resource options.

The following example will deploy a Percona Distribution for PostgreSQL Cluster in the `pgdb` namespace, with enabled Percona Monitoring and Management (PMM) and 20 Gi storage for a Primary PostgreSQL node:

Percona LLC and/or its affiliates, © 2009 - 2022

```
$ helm install my-db percona/pg-db --namespace pgdb \
  --set pgPrimary.volumeSpec.size=20Gi \
  --set pmm.enabled=true
```

Last update: 2022-07-19

Percona LLC and/or its affiliates, © 2009 - 2022

Percona LLC and/or its affiliates, © 2009 - 2022

11. Installation guide

Percona LLC and/or its affiliates, © 2009 - 2022

11.1 Install Percona Distribution for PostgreSQL on Kubernetes

Following steps will allow you to install the Operator and use it to manage Percona Distribution for PostgreSQL in a Kubernetes-based environment.

Percona LLC and/or its affiliates, © 2009 - 2022

1. First of all, clone the percona-postgresql-operator repository:

```
$ git clone -b v1.3.0 https://github.com/percona/percona-postgresql-operator
$ cd percona-postgresql-operator
```

Note

It is crucial to specify the right branch with `-b` option while cloning the code on this step. Please be careful.

2. The next thing to do is to add the `pgo` namespace to Kubernetes, not forgetting to set the correspondent context for further steps:

```
$ kubectl create namespace pgo
$ kubectl config set-context $(kubectl config current-context) --namespace=pgo
```

Note

To use different namespace, you should edit *all occurrences* of the `namespace: pgo` line in both `deploy/cr.yaml` and `deploy/operator.yaml` configuration files.

3. Deploy the operator with the following command:

```
$ kubectl apply -f deploy/operator.yaml
```

4. After the operator is started Percona Distribution for PostgreSQL can be created at any time with the following command:

```
$ kubectl apply -f deploy/cr.yaml
```

Creation process will take some time. The process is over when both operator and replica set pod have reached their Running status:

```
$ kubectl get pods
NAME                                            READY   STATUS     RESTARTS   AGE
backrest-backup-cluster1-j275w                  0/1     Completed  0          10m
cluster1-85486d645f-gpxzb                       1/1     Running    0          10m
cluster1-backrest-shared-repo-6495464548-c8wvl  1/1     Running    0          10m
cluster1-pgbouncer-fc45869f7-s86rf              1/1     Running    0          10m
pgo-deploy-rhv6k                                0/1     Completed  0          5m
postgres-operator-8646c68b57-z8m62              4/4     Running    1          5m
```

5. During previous steps, the Operator has generated several secrets, including the password for the `pguser` user, which you will need to access the cluster.

Use `kubectl get secrets` command to see the list of Secrets objects (by default Secrets object you are interested in has `cluster1-pguser-secret` name). Then `kubectl get secret cluster1-pguser-secret -o yaml` will return the YAML file with generated secrets, including the password which should look as follows:

```
...
data:
  ...
  password: cGd1c2VyX3Bhc3N3b3JkCg==
```


Percona LLC and/or its affiliates, © 2009 - 2022

Here the actual password is base64-encoded, and `echo 'cGd1c2VyX3Bhc3N3b3JkCg==' | base64 --decode` will bring it back to a human-readable form (in this example it will be a `pguser_password` string).

6. Check connectivity to newly created cluster

```
$ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-postgresql:
14.4 --restart=Never -- bash -il
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer -p 5432 -U
pguser pgdb
```

This command will connect you to the PostgreSQL interactive terminal.

```
psql (14.4)
Type "help" for help.
pgdb=>
```

Last update: 2022-07-19

Percona LLC and/or its affiliates, © 2009 - 2022

11.2 Install Percona Distribution for PostgreSQL on OpenShift

Following steps will allow you to install the Operator and use it to manage Percona Distribution for PostgreSQL on Red Hat OpenShift platform. For more information on the OpenShift, see its official documentation.

Percona LLC and/or its affiliates, © 2009 - 2022

Following steps will allow you to install the Operator and use it to manage Percona Distribution for PostgreSQL on OpenShift.

Percona LLC and/or its affiliates, © 2009 - 2022

1. First of all, clone the percona-postgresql-operator repository:

```
git clone -b v1.3.0 https://github.com/percona/percona-postgresql-operator
cd percona-postgresql-operator
```

Note

It is crucial to specify the right branch with `-b` option while cloning the code on this step. Please be careful.

2. The next thing to do is to add the `pgo` namespace to Kubernetes, not forgetting to set the correspondent context for further steps:

```
$ oc create namespace pgo
$ oc config set-context $(kubectl config current-context) --namespace=pgo
```

Note

To use different namespace, you should edit *all occurrences* of the `namespace: pgo` line in both `deploy/cr.yaml` and `deploy/operator.yaml` configuration files.

3. If you are going to use the operator with anyuid https://docs.openshift.com/container-platform/4.9/authentication/managing-security-context-constraints.html security context constraint please execute the following command:

```
$ sed -i '/disable_auto_failover: "false"/a \ \ \ \ disable_fsgroup: "false"' deploy/
operator.yaml
```

4. Deploy the operator with the following command:

```
$ oc apply -f deploy/operator.yaml
```

5. After the operator is started, Percona Distribution for PostgreSQL can be created at any time with the following command:

```
$ oc apply -f deploy/cr.yaml
```

Creation process will take some time. The process is over when both operator and replica set pod have reached their Running status:

```
$ oc get pods
NAME                                              READY   STATUS    RESTARTS   AGE
backrest-backup-cluster1-j275w                    0/1     Completed 0          10m
cluster1-85486d645f-gpxzb                         1/1     Running   0          10m
cluster1-backrest-shared-repo-6495464548-c8wvl    1/1     Running   0          10m
cluster1-pgbouncer-fc45869f7-s86rf                1/1     Running   0          10m
pgo-deploy-rhv6k                                  0/1     Completed 0          5m
postgres-operator-8646c68b57-z8m62                4/4     Running   1          5m
```

6. During previous steps, the Operator has generated several secrets, including the password for the `pguser` user, which you will need to access the cluster.

   Use `oc get secrets` command to see the list of Secrets objects (by default Secrets object you are interested in has `cluster1-pguser-secret` name). Then `kubectl get secret cluster1-pguser-secret -o yaml` will return the YAML file with generated secrets, including the password which should look as follows:

Percona LLC and/or its affiliates, © 2009 - 2022

```
...
data:
  ...
  password: cGd1c2VyX3Bhc3N3b3JkCg==
```

Here the actual password is base64-encoded, and `echo 'cGd1c2VyX3Bhc3N3b3JkCg==' | base64 --decode` will bring it back to a human-readable form (in this example it will be a `pguser_password` string).

7. Check connectivity to newly created cluster

```
$ oc run -i --rm --tty pg-client --image=perconalab/percona-distribution-postgresql:14.4 --
restart=Never -- bash -il
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer -p 5432 -U
pguser pgdb
```

This command will connect you to the PostgreSQL interactive terminal.

```
psql (14.4)
Type "help" for help.
pgdb=>
```

Last update: 2022-07-19

Percona LLC and/or its affiliates, © 2009 - 2022

12. Configuration

12.1 Users

User accounts within the Cluster can be divided into two different groups:

- *application-level users*: the unprivileged user accounts,
- *system-level users*: the accounts needed to automate the cluster deployment and management tasks.

12.1.1 System Users

Credentials for system users are stored as a Kubernetes Secrets object. The Operator requires to be deployed before PostgreSQL Cluster is started. The name of the required secrets (`cluster1-users` by default) should be set in the `spec.secretsName` option of the `deploy/cr.yaml` configuration file.

The following table shows system users' names and purposes.

Warning

These users should not be used to run an application.

The default PostgreSQL instance installation via the Percona Operator for PostgreSQL comes with the following users:

Role name	Attributes
`postgres`	Superuser, Create role, Create DB, Replication, Bypass RLS
`primaryuser`	Replication
`pguser`	Non-privileged user
`pgbouncer`	Administrative user for the pgBouncer connection pooler

The `postgres` user will be the admin user for the database instance. The `primaryuser` is used for replication between primary and replicas. The `pguser` is the default non-privileged user (you can configure different name of this user in the `spec.user` Custom Resource option).

YAML Object Format

The default name of the Secrets object for these users is `cluster1-users` and can be set in the CR for your cluster in `spec.secretName` to something different. When you create the object yourself, it should match the following simple format:

```
apiVersion: v1
kind: Secret
metadata:
  name: cluster1-users
type: Opaque
stringData:
  pgbouncer: pgbouncer_password
  postgres: postgres_password
  primaryuser: primaryuser_password
 pguser: pguser_password
```


Percona LLC and/or its affiliates, © 2009 - 2022

The example above matches what is shipped in the deploy/secrets.yaml file.

As you can see, we use the `stringData` type when creating the Secrets object, so all values for each key/value pair are stated in plain text format convenient from the user's point of view. But the resulting Secrets object contains passwords stored as `data` - i.e., base64-encoded strings. If you want to update any field, you'll need to encode the value into base64 format. To do this, you can run `echo -n "password" | base64` in your local shell to get valid values. For example, setting the PMM Server user's password to `new_password` in the `cluster1-users` object can be done with the following command:

```
kubectl patch secret/cluster1-users -p '{"data":{"pguser": '$(echo -n new_password |
base64)'}}'
```

12.1.2 Application users

By default you can connect to PostgreSQL as non-privileged `pguser` user. Also, you can login as `postgres` (the superuser) to PostgreSQL Pods, but pgBouncer (the connection pooler for PostgreSQL) doesn't allow `postgres` user access by default. That's done for security reasons.

If you still need to provide `postgres` user access to PostgreSQL instances from the outside, set the `pgBouncer.exposePostgresUser` option in the `deploy/cr.yaml` configuration file to `true` and apply changes as usual by the `kubectl apply -f deploy/cr.yaml` command.

Note

Allowing superusers to access to the cluster is not recommended.

Last update: 2022-08-03

Percona LLC and/or its affiliates, © 2009 - 2022

12.2 Changing PostgreSQL Options

You may require a configuration change for your application. PostgreSQL allows customizing the database with configuration files. You can use a ConfigMap to provide the PostgreSQL configuration options specific to the following configuration files:

- PostgreSQL main configuration, postgresql.conf,
- client authentication configuration, pg_hba.conf,
- user name configuration, pg_ident.conf.

Configuration options may be applied in two ways:

- globally to all database servers in the cluster via Patroni Distributed Configuration Store (DCS),
- locally to each database server (Primary and Replica) within the cluster.

Note

PostgreSQL cluster is managed by the Operator, and so there is no need to set custom configuration options in common usage scenarios. Also, changing certain options may cause PostgreSQL cluster malfunction. Do not customize configuration unless you know what you are doing!

Use the `kubectl` command to create the ConfigMap from external resources, for more information, see Configure a Pod to use a ConfigMap.

You can either create a PostgreSQL Cluster With Custom Configuration, or use ConfigMap to set options for the already existing cluster.

To create a cluster with custom options, you should first place these options in a `postgres-ha.yaml` file under specific `bootstrap` section, then use `kubectl create configmap` command with this file to create a ConfigMap, and finally put the ConfigMap name to pgPrimary.customconfig key in the `deploy/cr.yaml` configuration file.

In both cases, the `postgres-ha.yaml` file doesn't fully overwrite PostgreSQL configuration files: options present in `postgres-ha.yaml` will be overwritten, while non-present options will be left intact.

12.2.1 Creating a cluster with custom options

For example, you can create a cluster with a custom `max_connections` option in a `postgresql.conf` configuration file using the following `postgres-ha.yaml` contents:

```yaml
---
bootstrap:
  dcs:
    postgresql:
      parameters:
        max_connections: 30
```

Note

`dsc.postgresql` subsection means that option will be applied globally to `postgresql.conf` of all database servers.

You can create a ConfigMap from this file. The syntax for `kubectl create configmap` command is:

Percona LLC and/or its affiliates, © 2009 - 2022

```
kubectl -n <namespace> create configmap <configmap-name> --from-file=postgres-ha.yaml
```

ConfigMap name should include your cluster name and a dash as a prefix (`cluster1-` by default).

The following example defines `cluster1-custom-config` as the ConfigMap name:

```
$ kubectl create -n pgo configmap cluster1-custom-config --from-file=postgres-ha.yaml
```

To view the created ConfigMap, use the following command:

```
$ kubectl describe configmaps cluster1-custom-config
```

Don't forget to put the name of your ConfigMap to the `deploy/cr.yaml` configuration file:

```
spec:
  ...
  pgPrimary:
    ...
      customconfig: "cluster1-custom-config"
```

Now you can create the cluster following the regular installation instructions.

12.2.2 Modifying options for the existing cluster

If you need to update cluster's configuration settings, you should modify settings in the `<clusterName>-pgha-config` ConfigMap.

Note

This ConfigMap contains `<clusterName>-dcs-config` configuration applied globally to `postgresql.conf` of all database servers, and local configurations for the PostgreSQL cluster database servers: `<clusterName>-local-config` for the current primary, `<clusterName>-repl1-local-config` for the first replica, and so on.

For example, let's change the `max_connections` option in a globally applied `postgresql.conf` configuration file for the cluster named `cluster1`. Edit the `cluster1-pgha-config` ConfigMap with the following command:

```
$ kubectl edit -n pgo configmap cluster1-pgha-config
```

This will open the ConfigMap in a local text editor of your choice. Make sure to modify it as follows:

```
...
cluster1-dcs-config: |
  postgresql:
    parameters:
     ...
     max_connections: 50
     ...
```

Now restart the cluster to ensure the update took effect.

You can check if the changes are applied by querying the appropriate Pods of your cluster using the `kubectl exec` command with a specific Pod name.

First find out names of your Pods in a common way, using the `kubectl get pods` command:

Percona LLC and/or its affiliates, © 2009 - 2022

```
$ kubectl get pods
NAME                                            READY   STATUS    RESTARTS   AGE
backrest-backup-cluster1-j275w                  0/1     Completed 0          10m
cluster1-85486d645f-gpxzb                       1/1     Running   0          10m
cluster1-backrest-shared-repo-6495464548-c8wvl  1/1     Running   0          10m
cluster1-pgbouncer-fc45869f7-s86rf              1/1     Running   0          10m
pgo-deploy-rhv6k                                0/1     Completed 0          5m
postgres-operator-8646c68b57-z8m62              4/4     Running   1          5m
```

Now let's check the `cluster1-85486d645f-gpxzb` Pod for the current `max_connections` value:

```
$ kubectl -n pgo exec -it cluster1-85486d645f-gpxzb -- psql -c 'show max_connections;'
  max_connections
  -----------------
  50
  (1 row)
```

Last update: 2022-07-13

Percona LLC and/or its affiliates, © 2009 - 2022

12.3 Binding Percona Distribution for PostgreSQL components to Specific Kubernetes/OpenShift Nodes

The operator does good job automatically assigning new Pods to nodes with sufficient resources to achieve balanced distribution across the cluster. Still there are situations when it is worth to ensure that pods will land on specific nodes: for example, to get speed advantages of the SSD equipped machine, or to reduce network costs choosing nodes in a same availability zone.

Appropriate sections of the deploy/cr.yaml file (such as `pgPrimary` or `pgReplicas`) contain keys which can be used to do this, depending on what is the best for a particular situation.

12.3.1 Affinity and anti-affinity

Affinity makes Pod eligible (or not eligible - so called “anti-affinity”) to be scheduled on the node which already has Pods with specific labels, or has specific labels itself (so called “Node affinity”). Particularly, Pod anti-affinity is good to reduce costs making sure several Pods with intensive data exchange will occupy the same availability zone or even the same node - or, on the contrary, to make them land on different nodes or even different availability zones for the high availability and balancing purposes. Node affinity is useful to assign PostgreSQL instances to specific Kubernetes Nodes (ones with specific hardware, zone, etc.).

Pod anti-affinity is controlled by the `antiAffinityType` option, which can be put into `pgPrimary`, `pgBouncer`, and `backup` sections of the `deploy/cr.yaml` configuration file. This option can be set to one of two values:

- `preferred` Pod anti-affinity is a sort of a *soft rule*. It makes Kubernetes *trying* to schedule Pods matching the anti-affinity rules to different Nodes. If it is not possible, then one or more Pods are scheduled to the same Node. This variant is used by default.
- `required` Pod anti-affinity is a sort of a *hard rule*. It forces Kubernetes to schedule each Pod matching the anti-affinity rules to different Nodes. If it is not possible, then a Pod will not be scheduled at all.

Node affinity can be controlled by the `pgPrimary.affinity.nodeAffinityType` option in the `deploy/cr.yaml` configuration file. This option can be set to either `preferred` or `required` similarly to the `antiAffinityType` option.

12.3.2 Simple approach - configure Node Affinity based on nodeLabel

The Operator provides the `pgPrimary.affinity.nodeLabel` option, which should contains one or more key-value pairs. If the node is not labeled with each key-value pair and `nodeAffinityType` is set to `required`, the Pod will not be able to land on it.

The following example forces Operator to lend Percona Distribution for PostgreSQL instances on the Nodes having the `kubernetes.io/region: us-central1` label:

```
affinity:
  nodeAffinityType: required
  nodeLabel:
    kubernetes.io/region: us-central1
```

Advanced approach - use standard Kubernetes constraints

Previous way can be used with no special knowledge of the Kubernetes way of assigning Pods to specific Nodes. Still in some cases more complex tuning may be needed. In this case `pgPrimary.affinity.advanced` option placed in the `deploy/cr.yaml` file turns off the effect of the `nodeLabel` and allows to use standard Kubernetes affinity constraints of any complexity:

Percona LLC and/or its affiliates, © 2009 - 2022

```
affinity:
   advanced:
     podAffinity:
       requiredDuringSchedulingIgnoredDuringExecution:
       - labelSelector:
           matchExpressions:
           - key: security
             operator: In
             values:
             - S1
         topologyKey: failure-domain.beta.kubernetes.io/zone
     podAntiAffinity:
       preferredDuringSchedulingIgnoredDuringExecution:
       - weight: 100
         podAffinityTerm:
           labelSelector:
             matchExpressions:
             - key: security
               operator: In
               values:
               - S2
           topologyKey: kubernetes.io/hostname
     nodeAffinity:
       requiredDuringSchedulingIgnoredDuringExecution:
         nodeSelectorTerms:
         - matchExpressions:
           - key: kubernetes.io/e2e-az-name
             operator: In
             values:
             - e2e-az1
             - e2e-az2
       preferredDuringSchedulingIgnoredDuringExecution:
       - weight: 1
         preference:
           matchExpressions:
           - key: another-node-label-key
             operator: In
             values:
             - another-node-label-value
```

You can see the explanation of these affinity options in Kubernetes documentation.

Default Affinity rules

The following anti-affinity rules are applied to all Percona Distribution for PostgreSQL Pods:

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - podAffinityTerm:
        labelSelector:
          matchExpressions:
          - key: vendor
            operator: In
            values:
            - crunchydata
          - key: pg-pod-anti-affinity
            operator: Exists
          - key: pg-cluster
            operator: In
            values:
            - cluster1
```


Percona LLC and/or its affiliates, © 2009 - 2022

```
          topologyKey: kubernetes.io/hostname
        weight: 1
```

You can see the explanation of these affinity options in Kubernetes documentation.

Note

Setting `required` anti-affinity type will result in placing all Pods on separate nodes, so default configuration **will require 7 Kubernetes nodes** to deploy the cluster with separate nodes assigned to one PostgreSQL primary, two PostgreSQL replica instances, three pgBouncer and one pgBackrest Pod.

12.3.3 Tolerations

Tolerations allow Pods having them to be able to land onto nodes with matching *taints*. Toleration is expressed as a `key` with and `operator`, which is either `exists` or `equal` (the latter variant also requires a `value` the key is equal to). Moreover, toleration should have a specified `effect`, which may be a self-explanatory `NoSchedule`, less strict `PreferNoSchedule`, or `NoExecute`. The last variant means that if a *taint* with `NoExecute` is assigned to node, then any Pod not tolerating this *taint* will be removed from the node, immediately or after the `tolerationSeconds` interval, like in the following example.

You can use `pgPrimary.tolerations` key in the `deploy/cr.yaml` configuration file as follows:

```
tolerations:
- key: "node.alpha.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 6000
```

The Kubernetes Taints and Toleratins contains more examples on this topic.

Last update: 2022-08-04

Percona LLC and/or its affiliates, © 2009 - 2022

12.4 Transport Layer Security (TLS)

The Percona Operator for PostgreSQL uses Transport Layer Security (TLS) cryptographic protocol for the following types of communication:

- Internal - communication between PostgreSQL instances in the cluster
- External - communication between the client application and the cluster

The internal certificate is also used as an authorization method for PostgreSQL Replica instances.

TLS security can be configured in several ways:

- the Operator can generate certificates automatically at cluster creation time,
- you can also generate certificates manually.

You can also use pre-generated certificates available in the `deploy/ssl-secrets.yaml` file for test purposes, but we strongly recommend **avoiding their usage on any production system**!

The following subsections explain how to configure TLS security with the Operator yourself, as well as how to temporarily disable it if needed.

12.4.1 Allow the Operator to generate certificates automatically

The Operator is able to generate long-term certificates automatically and turn on encryption at cluster creation time, if there are no certificate secrets available. It generates certificates with the help of cert-manager - a Kubernetes certificate management controller widely used to automate the management and issuance of TLS certificates. Cert-manager is community-driven and open source.

Installation of the *cert-manager*

You can install *cert-manager* as follows:

- Create a namespace,
- Disable resource validations on the cert-manager namespace,
- Install the cert-manager.

The following commands perform all the needed actions:

```
$ kubectl create namespace cert-manager
$ kubectl label namespace cert-manager certmanager.k8s.io/disable-validation=true
$ kubectl_bin apply -f https://github.com/jetstack/cert-manager/releases/download/v1.5.4/cert-
manager.yaml
```

After the installation, you can verify the *cert-manager* by running the following command:

```
$ kubectl get pods -n cert-manager
```

The result should display the *cert-manager* and webhook active and running.

Percona LLC and/or its affiliates, © 2009 - 2022

Turning automatic generation of certificates on

When you have already installed *cert-manager*, the operator is able to request a certificate from it. To make this happend, uncomment `sslCA`, `sslSecretName`, and `sslReplicationSecretName` options in the `deploy/cr.yaml` configuration file:

```
...
spec:
#  secretsName: cluster1-users
  sslCA: cluster1-ssl-ca
  sslSecretName: cluster1-ssl-keypair
  sslReplicationSecretName: cluster1-ssl-keypair
...
```

When done, deploy your cluster as usual, with the `kubectl apply -f deploy/cr.yaml` command. Certificates will be generated if there are no certificate secrets available.

12.4.2 Generate certificates manually

To generate certificates manually, follow these steps:

1. Provision a to generate TLS certificates,
2. Generate a key and certificate file with the server details,
3. Create the server TLS certificates using the keys, certs, and server details.

The set of commands generates certificates with the following attributes:

- `Server-pem` - Certificate
- `Server-key.pem` - the private key
- `ca.pem` - Certificate Authority

You should generate one set of certificates for external communications, and another set for internal ones.

Supposing that your cluster name is `cluster1`, you can use the following commands to generate certificates:

```
$ CLUSTER_NAME=cluster1
$ NAMESPACE=default
$ cat <<EOF | cfssl gencert -initca - | cfssljson -bare ca
{
  "CN": "*",
  "key": {
    "algo": "ecdsa",
    "size": 384
  }
}
EOF

$ cat <<EOF > ca-config.json
{
   "signing": {
     "default": {
        "expiry": "87600h",
        "usages": ["digital signature", "key encipherment", "content commitment"]
      }
   }
}
EOF

$ cat <<EOF | cfssl gencert -ca=ca.pem  -ca-key=ca-key.pem -config=./ca-config.json - |
```


Percona LLC and/or its affiliates, © 2009 - 2022

```
cfssljson -bare server
{
   "hosts": [
     "localhost",
     "${CLUSTER_NAME}",
     "${CLUSTER_NAME}.${NAMESPACE}",
     "${CLUSTER_NAME}.${NAMESPACE}.svc.cluster.local",
     "${CLUSTER_NAME}-pgbouncer",
     "${CLUSTER_NAME}-pgbouncer.${NAMESPACE}",
     "${CLUSTER_NAME}-pgbouncer.${NAMESPACE}.svc.cluster.local",
     "*.${CLUSTER_NAME}",
     "*.${CLUSTER_NAME}.${NAMESPACE}",
     "*.${CLUSTER_NAME}.${NAMESPACE}.svc.cluster.local",
     "*.${CLUSTER_NAME}-pgbouncer",
     "*.${CLUSTER_NAME}-pgbouncer.${NAMESPACE}",
     "*.${CLUSTER_NAME}-pgbouncer.${NAMESPACE}.svc.cluster.local"
   ],
   "CN": "${CLUSTER_NAME}",
   "key": {
     "algo": "ecdsa",
     "size": 384
   }
}
EOF

$ kubectl create secret generic ${CLUSTER_NAME}-ssl-ca --from-file=ca.crt=ca.pem
$ kubectl create secret tls  ${CLUSTER_NAME}-ssl-keypair --cert=server.pem --key=server-key.pem
```

If your PostgreSQL cluster includes replica instances (this feature is on by default), generate certificates for them in a similar way:

```
$ cat <<EOF | cfssl gencert -ca=ca.pem  -ca-key=ca-key.pem -config=./ca-config.json - |
cfssljson -bare replicas
{
   "CN": "primaryuser",
   "key": {
      "algo": "ecdsa",
      "size": 384
   }
}
EOF

$ kubectl create secret tls  ${CLUSTER_NAME}-ssl-replicas --cert=replicas.pem --key=replicas-
key.pem
```

When certificates are generated, set the following keys in the `deploy/cr.yaml` configuration file:

- `spec.sslCA` key should contain the name of the secret with TLS used for both connection encryption (external traffic), and replication (internal traffic),
- `spec.sslSecretName` key should contain the name of the secret created to encrypt **external** communications,
- `spec.secrets.sslReplicationSecretName` key should contain the name of the secret created to encrypt **internal** communications,
- `spec.tlsOnly` is set to `true` by default and enforces encryption

Don't forget to apply changes as usual:

```
$ kubectl apply -f deploy/cr.yaml
```


Percona LLC and/or its affiliates, © 2009 - 2022

12.4.3 Check connectivity to the cluster

You can check TLS communication with use of the `psql`, the standard interactive terminal-based frontend to PostgreSQL. The following command will spawn a new `pg-client` container, which includes needed command and can be used for the check (use your real cluster name instead of the `<cluster-name>` placeholder):

```
$ cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pg-client
spec:
  replicas: 1
  selector:
    matchLabels:
      name: pg-client
  template:
    metadata:
      labels:
        name: pg-client
    spec:
      containers:
        - name: pg-client
          image: perconalab/percona-distribution-postgresql:14.4
          imagePullPolicy: Always
          command:
          - sleep
          args:
          - "100500"
          volumeMounts:
            - name: ca
              mountPath: "/tmp/tls"
      volumes:
      - name: ca
        secret:
          secretName: <cluster_name>-ssl-ca
          items:
          - key: ca.crt
            path: ca.crt
            mode: 0777
EOF
```

Now get shell access to the newly created container, and launch the PostgreSQL interactive terminal to check connectivity over the encrypted channel (please use real cluster-name, PostgreSQL user login and password):

```
$ kubectl exec -it deployment/pg-client -- bash -il
[postgres@pg-client /]$ PGSSLMODE=verify-ca PGSSLROOTCERT=/tmp/tls/ca.crt psql postgres://
<postgresql-user>:<postgresql-password>@<cluster-name>-pgbouncer.<namespace>.svc.cluster.local
```

Now you should see the prompt of PostgreSQL interactive terminal:

```
psql (14.4)
Type "help" for help.
pgdb=>
```

12.4.4 Run Percona Distribution for PostgreSQL without TLS

Omitting TLS is also possible, but we recommend that you run your cluster with the TLS protocol enabled.

Percona LLC and/or its affiliates, © 2009 - 2022

To disable TLS protocol (e.g. for demonstration purposes) set the `spec.tlsOnly` key to `false`, and make sure that there are no certificate secrets configured in the `deploy/cr.yaml` file.

Last update: 2022-07-19

Percona LLC and/or its affiliates, © 2009 - 2022

12.5 Telemetry

The Telemetry function enables the Operator gathering and sending basic anonymous data to Percona, which helps us to determine where to focus the development and what is the uptake for each release of Operator.

The following information is gathered:

- ID of the Custom Resource (the `metadata.uid` field)
- Kubernetes version
- Platform (is it Kubernetes or Openshift)
- PMM Version
- Operator version
- PostgreSQL version
- PgBackRest version

We do not gather anything that identify a system, but the following thing should be mentioned: Custom Resource ID is a unique ID generated by Kubernetes for each Custom Resource.

Telemetry is enabled by default and is sent to the Version Service server when the Operator connects to it at scheduled times to obtain fresh information about version numbers and valid image paths needed for the upgrade.

The landing page for this service, check.percona.com, explains what this service is.

You can disable telemetry with a special option when installing the Operator:

- if you install the Operator with helm, use the following installation command:

```
$ helm install my-db percona/pg-db --version 1.3.0 --namespace my-namespace --set
disable_telemetry="true"
```

- if you don't use helm for installation, you have to edit the `operator.yaml` before applying it with the `kubectl apply -f deploy/operator.yaml` command. Open the `operator.yaml` file with your text editor, find the `disable_telemetry` key and set it to `true`:

```
...
disable_telemetry: "true"
...
```

Last update: 2022-08-03

Percona LLC and/or its affiliates, © 2009 - 2022

13. Management

13.1 Providing Backups

The Operator allows doing backups in two ways. *Scheduled backups* are configured in the deploy/cr.yaml file to be executed automatically in proper time. *On-demand backups* can be done manually at any moment.

The Operator uses the open source pgBackRest backup and restore utility. A special *pgBackRest repository* is created by the Operator along with creating a new PostgreSQL cluster to facilitate the usage of the pgBackRest features in it.

The Operator can store PostgreSQL backups on Amazon S3, any S3-compatible storage and Google Cloud Storage outside the Kubernetes cluster. Storing backups on Persistent Volume attached to the pgBackRest Pod is also possible. At PostgreSQL cluster creation time, you can specify a specific Storage Class for the pgBackRest repository. Additionally, you can also specify the type of the pgBackRest repository that can be used for backups:

- `local` : Uses the storage that is provided by the Kubernetes cluster's Storage Class that you select (for historical reasons this repository type can be alternatively named `posix`),
- `s3` : Use Amazon S3 or an object storage system that uses the S3 protocol,
- `local,s3` : Use both the storage that is provided by the Kubernetes cluster's Storage Class that you select AND Amazon S3 (or equivalent object storage system that uses the S3 protocol).
- `gcs` : Use Google Cloud Storage,
- `local,gcs` : Use both the storage that is provided by the Kubernetes cluster's Storage Class that you select AND Google Cloud Storage.

The pgBackRest repository consists of the following Kubernetes objects:

- A Deployment,
- A Secret that contains information that is specific to the PostgreSQL cluster that it is deployed with (e.g. SSH keys, AWS S3 keys, etc.),
- A Pod with a number of supporting scripts,
- A Service.

The PostgreSQL primary is automatically configured to use the `pgbackrest archive-push` and push the write-ahead log (WAL) archives to the correct repository.

The PostgreSQL Operator supports three types of pgBackRest backups:

- Full (`full`): A full backup of all the contents of the PostgreSQL cluster,
- Differential (`diff`): A backup of only the files that have changed since the last full backup,
- Incremental (`incr`): A backup of only the files that have changed since the last full or differential backup. Incremental backup is the default choice.

The Operator also supports setting pgBackRest retention policies for backups. Backup retention can be controlled by the following pgBackRest options:

- `--repo1-retention-full` the number of full backups to retain,
- `--repo1-retention-diff` the number of differential backups to retain,
- `--repo1-retention-archive` how many sets of write-ahead log archives to retain alongside the full and differential backups that are retained.

You can set both backups type and retention policy when Making on-demand backup.

Percona LLC and/or its affiliates, © 2009 - 2022

Also you should first configure the backup storage in the `deploy/cr.yaml` configuration file to have backups enabled.

13.1.1 Configuring the S3-compatible backup storage

In order to use S3-compatible storage for backups you need to provide some S3-related information, such as proper S3 bucket name, endpoint, etc. This information can be passed to pgBackRest via the following `deploy/cr.yaml` options in the `backup.storages` subsection:

- `bucket` specifies the AWS S3 bucket that should be utilized, for example `my-postgresql-backups-example`,
- `endpointUrl` specifies the S3 endpoint that should be utilized, for example `s3.amazonaws.com`,
- `region` specifies the AWS S3 region that should be utilized, for example `us-east-1`,
- `uriStyle` specifies whether `host` or `path` style URIs should be utilized,
- `verifyTLS` should be set to `true` to enable TLS verification or set to `false` to disable it,
- `type` should be set to `s3`.

You also need to supply pgBackRest with base64-encoded AWS S3 key and AWS S3 key secret stored along with other sensitive information in Kubernetes Secrets (e.g. encoding needed data with the `echo "string-to-encode" | base64` command). Edit the `deploy/backup/cluster1-backrest-repo-config-secret.yaml` configuration file: set there proper cluster name, AWS S3 key, and key secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: <cluster-name>-backrest-repo-config
type: Opaque
data:
  aws-s3-key: <base64-encoded-AWS-S3-key>
  aws-s3-key-secret: <base64-encoded-AWS-S3-key-secret>
```

When done, create the secret as follows:

```
$ kubectl apply -f deploy/backup/cluster1-backrest-repo-config-secret.yaml
```

Finally, create or update the cluster:

```
$ kubectl apply -f deploy/cr.yaml
```

13.1.2 Use Google Cloud Storage for backups

You can configure Google Cloud Storage as an object store for backups similarly to S3 storage.

In order to use Google Cloud Storage (GCS) for backups you need to provide some GCS-related information, such as a proper GCS bucket name. This information can be passed to `pgBackRest` via the following options in the `backup.storages` subsection of the `deploy/cr.yaml` configuration file:

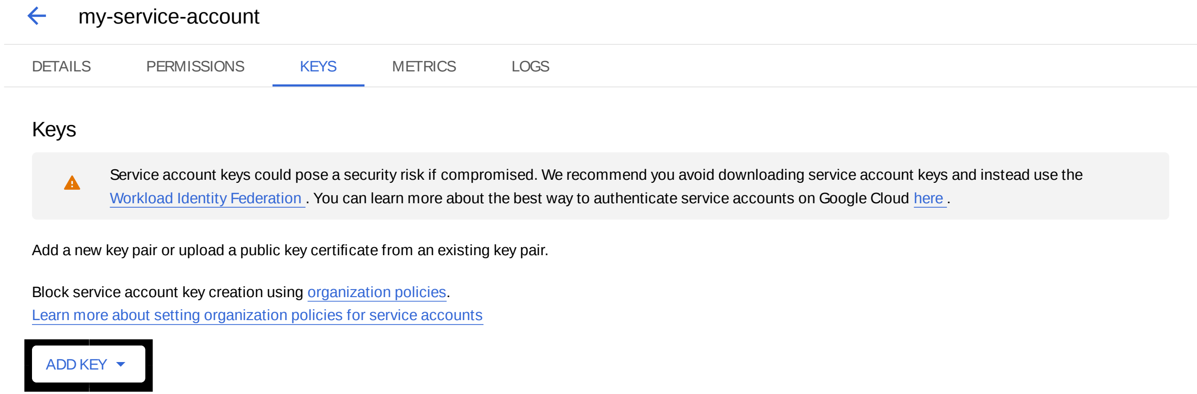
- `bucket` should contain the proper bucket name,
- `type` should be set to `gcs`.

Percona LLC and/or its affiliates, © 2009 - 2022

The Operator will also need your service account key to access storage.

1. Create your service account key following the official Google Cloud instructions.
2. Export this key from your Google Cloud account.

   You can find your key in the Google Cloud console (select *IAM & Admin* → *Service Accounts* in the left menu panel, then click your account and open the *KEYS* tab):



   Click the *ADD KEY* button, chose *Create new key* and chose *JSON* as a key type. These actions will result in downloading a file in JSON format with your new private key and related information.

3. Now you should use a base64-encoded version of this file and to create the Kubernetes Secret. You can encode the file with the `base64 <filename>` command. When done, create the following yaml file with your cluster name and base64-encoded file contents:

```
apiVersion: v1
kind: Secret
metadata:
  name: <cluster-name>-backrest-repo-config
type: Opaque
data:
  gcs-key: <base64-encoded-json-file-contents>
```

   When done, create the secret as follows:

```
$ kubectl apply -f ./my-gcs-account-secret.yaml
```

4. Finally, create or update the cluster:

```
$ kubectl apply -f deploy/cr.yaml
```

13.1.3 Scheduling backups

Backups schedule is defined in the `backup` section of the deploy/cr.yaml file. This section contains following subsections:

- `storages` subsection contains data needed to access the S3-compatible cloud to store backups.
- `schedule` subsection allows to actually schedule backups (the schedule is specified in crontab format).

Here is an example of deploy/cr.yaml which uses Amazon S3 storage for backups:

```
...
backup:
  ...
  schedule:
```


Percona LLC and/or its affiliates, © 2009 - 2022

```
    - name: "sat-night-backup"
      schedule: "0 0 * * 6"
      keep: 3
      type: full
      storage: s3
  ...
```

The schedule is specified in crontab format as explained in Custom Resource options.

13.1.4 Making on-demand backup

To make an on-demand backup, the user should use a backup configuration file. The example of the backup configuration file is deploy/backup/backup.yaml.

The following keys are most important in the parameters section of this file:

- `parameters.backrest-opts` is the string with command line options which will be passed to pgBackRest, for example `--type=full --repo1-retention-full=5`,
- `parameters.pg-cluster` is the name of the PostgreSQL cluster to back up, for example `cluster1`.

When the backup options are configured, execute the actual backup command:

```
$ kubectl apply -f deploy/backup/backup.yaml
```

13.1.5 List existing backups

To get list of all existing backups in the pgBackrest repo, use the following command:

```
$ kubectl exec <name-of-backrest-shared-repo-pod>  -it -- pgbackrest info
```

You can find out the appropriate Pod name using the `` kubectl get pods`` command, as usual. Here is an example of the backups list:

```
$ kubectl exec cluster1-backrest-shared-repo-5ffc465b85-gvhlh -it -- pgbackrest info
stanza: db
    status: ok
    cipher: none

    db (current)
        wal archive min/max (14): 000000010000000000000001/000000010000000000000003

        full backup: 20220614-104859F
            timestamp start/stop: 2022-06-14 10:48:59 / 2022-06-14 10:49:13
            wal start/stop: 000000010000000000000002 / 000000010000000000000002
            database size: 33.5MB, database backup size: 33.5MB
            repo1: backup set size: 4.3MB, backup size: 4.3MB
```

In this example there is only one backup named `20220614-104859F`.

Percona LLC and/or its affiliates, © 2009 - 2022

13.1.6 Restore the cluster from a previously saved backup

The Operator supports the ability to perform a full restore on a PostgreSQL cluster as well as a point-in-time-recovery. There are two types of ways to restore a cluster:

- restore to a new cluster using the pgDataSource.restoreFrom option (and possibly, pgDataSource.restoreOpts for custom pgBackRest options),
- restore in-place, to an existing cluster (note that this is destructive).

Restoring to a new PostgreSQL cluster allows you to take a backup and create a new PostgreSQL cluster that can run alongside an existing one. There are several scenarios where using this technique is helpful:

- Creating a copy of a PostgreSQL cluster that can be used for other purposes. Another way of putting this is *creating a clone*.
- Restore to a point-in-time and inspect the state of the data without affecting the current cluster.

To restore the previously saved backup the user should use a *backup restore* configuration file. The example of the backup configuration file is deploy/backup/restore.yaml:

```
apiVersion: pg.percona.com/v1
kind: Pgtask
metadata:
  labels:
    pg-cluster: cluster1
    pgouser: admin
  name: cluster1-backrest-restore
  namespace: pgo
spec:
  name: cluster1-backrest-restore
  namespace: pgo
  parameters:
    backrest-restore-from-cluster: cluster1
    backrest-restore-opts: --type=time --target="2021-04-16 15:13:32+00"
    backrest-storage-type: local
  tasktype: restore
```

The following keys are the most important in the parameters section of this file:

- `parameters.backrest-restore-cluster` specifies the name of a PostgreSQL cluster which will be restored (this option had name `parameters.backrest-restore-from-cluster` before the Operator 1.2.0). It includes stopping the database and recreating a new primary with the restored data (for example, `cluster1`),
- `parameters.backrest-restore-opts` passes through additional options for pgBackRest,
- `parameters.backrest-storage-type` the type of the pgBackRest repository, (for example, `local`).

The actual restoration process can be started as follows:

```
$ kubectl apply -f deploy/backup/restore.yaml
```

To create a new PostgreSQL cluster from either the active one, or a former cluster whose pgBackRest repository still exists, use the pgDataSource.restoreFrom option.

Percona LLC and/or its affiliates, © 2009 - 2022

The following example will create a new cluster named `cluster2` from an existing one named``cluster1``.

1. First, create the `cluster2-config-secrets.yaml` configuration file with the following content:

```
apiVersion: v1
data:
  password: <base64-encoded-password-for-pguser->
  username: <base64-encoded-pguser-user-name>
kind: Secret
metadata:
  labels:
    pg-cluster: cluster2
    vendor: crunchydata
  name: cluster2-pguser-secret
type: Opaque
---
apiVersion: v1
data:
  password: <base64-encoded-password-for-primaryuser>
  username: <base64-encoded-primaryuser-user-name>
kind: Secret
metadata:
  labels:
    pg-cluster: cluster2
    vendor: crunchydata
  name: cluster2-primaryuser-secret
type: Opaque
---
apiVersion: v1
data:
  password: <base64-encoded-password-for-postgres-user>
  username: <base64-encoded-pguser-postgres-name>
kind: Secret
metadata:
  labels:
    pg-cluster: cluster2
    vendor: crunchydata
  name: cluster2-postgres-secret
type: Opaque
```

2. When done, create the secrets as follows:

```
$ kubectl apply -f ./cluster2-config-secrets.yaml
```

3. Edit the `deploy/cr.yaml` configuration file:
   - set a new cluster name (`cluster2`),
   - set the option pgDataSource.restoreFrom to `cluster1`.

4. Create the cluster as follows:

```
$ kubectl apply -f deploy/cr.yaml
```

13.1.7 Restore the cluster with point-in-time recovery

Point-in-time recovery functionality allows users to revert the database back to a state before an unwanted change had occurred.

You can set up a point-in-time recovery using the normal restore command of pgBackRest with few additional options specified in the `parameters.backrest-restore-opts` key in the backup restore configuration file:

Percona LLC and/or its affiliates, © 2009 - 2022

```
...
spec:
  name: cluster1-backrest-restore
  namespace: pgo
  parameters:
    backrest-restore-from-cluster: cluster1
    backrest-restore-opts: --type=time --target="2021-04-16 15:13:32+00"
```

- set `--type` option to `time`,
- set `--target` to a specific time you would like to restore to. You can use the typical string formatted as `<YYYY-MM-DD HH:MM:DD>`, optionally followed by a timezone offset: `"2021-04-16 15:13:32+00"` (`+00` in the above example means just UTC),
- optional `--set` argument allows you to choose the backup which will be the starting point for point-in-time recovery (look through the available backups to find out the proper backup name). This option must be specified if the target is one or more backups away from the current moment.

After setting these options in the *backup restore* configuration file, follow the standard restore instructions.

Note

Make sure you have a backup that is older than your desired point in time. You obviously can't restore from a time where you do not have a backup. All relevant write-ahead log files must be successfully pushed before you make the restore.

13.1.8 Delete a previously saved backup

The maximum amount of stored backups is controlled by the backup.schedule.keep option (only successful backups are counted). Older backups are automatically deleted, so that amount of stored backups do not exceed this number.

If you want to delete some backup manually, you need to delete both the `pgtask` object and the corresponding job itself. Deletion of the backup object can be done using the same YAML file which was used for the on-demand backup:

```
$ kubectl delete -f deploy/backup/backup.yaml
```

Deletion of the job which corresponds to the backup can be done using `kubectl delete jobs` command with the backup name:

```
$ kubectl delete jobs cluster1-backrest-full-backup
```

Last update: 2022-07-13

Percona LLC and/or its affiliates, © 2009 - 2022

13.2 Update Percona Operator for PostgreSQL

Percona Operator for PostgreSQL allows upgrades to newer versions. This includes upgrades of the Operator itself, and upgrades of the Percona Distribution for PostgreSQL.

13.2.1 Upgrading the Operator

Note

Only the incremental update to a nearest minor version of the Operator is supported. To update to a newer version, which differs from the current version by more than one, make several incremental updates sequentially.

The following steps will allow you to update the Operator to current version (use the name of your cluster instead of the `<cluster-name>` placeholder).

1. Pause the cluster in order to stop all possible activities:

```
$ kubectl patch perconapgcluster/<cluster-name> --type json -p '[{"op": "replace", "path": "/
spec/pause", "value": true},{"op":"replace","path":"/spec/pgBouncer/size","value":0}]'
```

2. If you upgrade the Operator **from a version earlier than 1.1.0**, the following additional step is needed for the 1.0.0 → 1.1.0 upgrade.

```
$ export CLUSTER=<cluster-name>
$ for user in postgres primaryuser $(kubectl get perconapgcluster/${CLUSTER} -o yaml | yq r
- 'spec.user'); do args+="--from-literal=$user=$(kubectl get secret/${CLUSTER}-${user}-o
yaml | yq r - 'data.password' | base64 -d) "; done; eval kubectl create secret generic $
{CLUSTER}-users "${args}"
```

This command creates users' secrets with existing passwords. Otherwise, new secrets with autogenerated passwords will be created automatically, so existing passwords will be overwritten.

Note

The `pgbouncer` user password is stored in encrypted form, and therefore it is not included in the above command. If you know this password and/or would like to update it, please add it as `pgbouncer: base64encodednewpassword` to the resulted Secret manually. Otherwise, this password needs no actions and will be overwritten by the Operator during upgrade.

3. Remove the old Operator and start the new Operator version:

```
$ kubectl delete \
    serviceaccounts/pgo-deployer-sa \
    clusterroles/pgo-deployer-cr \
    configmaps/pgo-deployer-cm \
    configmaps/pgo-config \
    clusterrolebindings/pgo-deployer-crb \
    jobs.batch/pgo-deploy \
    deployment/postgres-operator

$ kubectl create -f https://raw.githubusercontent.com/percona/percona-postgresql-operator/
v1.3.0/deploy/operator.yaml
$ kubectl wait --for=condition=Complete job/pgo-deploy --timeout=90s
```


Percona LLC and/or its affiliates, © 2009 - 2022

13.2.2 Upgrading Percona Distribution for PostgreSQL

Automatic upgrade

Starting from version 1.1.0, the Operator does fully automatic upgrades to the newer versions of Percona PostgreSQL Cluster within the method named *Smart Updates*.

The Operator will carry on upgrades according to the following algorithm. It will query a special *Version Service* server at scheduled times to obtain fresh information about version numbers and valid image paths needed for the upgrade. If the current version should be upgraded, the Operator updates the CR to reflect the new image paths and carries on sequential Pods deletion in a safe order, allowing the cluster Pods to be re-deployed with the new image.

Note

Version Service is in technical preview status and is disabled by default for the Operator version 1.1.0. Disabling Version Service makes Smart Updates rely on the `image` keys in the Operator's Custom Resource.

Percona LLC and/or its affiliates, © 2009 - 2022

The upgrade details are set in the `upgradeOptions` section of the `deploy/cr.yaml` configuration file. Make the following edits to configure updates:

1. Set the `apply` option to one of the following values:
    - `recommended` - automatic upgrades will choose the most recent version of software flagged as recommended (for clusters created from scratch, the Percona Distribution for PostgreSQL 14 version will be selected instead of the Percona Distribution for PostgreSQL 13 or 12 version regardless of the image path; for already existing clusters, 14 vs. 13 or 12 branch choice will be preserved),
    - `14-recommended`, `13-recommended`, `12-recommended` - same as above, but preserves specific major Percona Distribution for PostgreSQL version for newly provisioned clusters (for example, 14 will not be automatically used instead of 13),
    - `latest` - automatic upgrades will choose the most recent version of the software available,
    - `14-latest`, `13-latest`, `12-latest` - same as above, but preserves specific major Percona Distribution for PostgreSQL version for newly provisioned clusters (for example, 14 will not be automatically used instead of 13),
    - *version number* - specify the desired version explicitly,
    - `never` or `disabled` - disable automatic upgrades

    Note

    When automatic upgrades are disabled by the `apply` option, Smart Update functionality will continue working for changes triggered by other events, such as updating a ConfigMap, rotating a password, or changing resource values.

2. Make sure the `versionServiceEndpoint` key is set to a valid Version Server URL (otherwise Smart Updates will not occur).

    Percona's Version Service

    You can use the URL of the official Percona's Version Service (default). Set `versionServiceEndpoint` to `https://check.percona.com`.

    Version Service inside your cluster

    Alternatively, you can run Version Service inside your cluster. This can be done with the `kubectl` command as follows:

    ```
    $ kubectl run version-service --image=perconalab/version-service --env="SERVE_HTTP=true" --
    port 11000 --expose
    ```

    Note

    Version Service is never checked if automatic updates are disabled. If automatic updates are enabled, but Version Service URL can not be reached, upgrades will not occur.

3. Use the `schedule` option to specify the update checks time in CRON format.

    The following example sets the midnight update checks with the official Percona's Version Service:

    ```
    spec:
      upgradeOptions:
        apply: recommended
        versionServiceEndpoint: https://check.percona.com
        schedule: "0 4 * * *"
    ...
    ```


Percona LLC and/or its affiliates, © 2009 - 2022

Semi-automatic upgrade

Semi-automatic update of Percona Distribution for PostgreSQL should be used with the Operator version 1.0.0 or earlier. For all newer versions, use automatic update instead.

The following steps will allow you to update the Operator to current version (use the name of your cluster instead of the `<cluster-name>` placeholder).

1. Pause the cluster in order to stop all possible activities:

```
$ kubectl patch perconapgcluster/<cluster-name> --type json -p '[{"op": "replace", "path": "/
spec/pause", "value": true},{"op":"replace","path":"/spec/pgBouncer/size","value":0}]'
```

2. Now you can switch the cluster to a new version:

```
$ kubectl patch perconapgcluster/<cluster-name> --type json -p '[{"op": "replace", "path": "/
spec/backup/backrestRepoImage", "value": "percona/percona-postgresql-operator:1.3.0-ppg14-
pgbackrest-repo"},{"op":"replace","path":"/spec/backup/image","value":"percona/percona-
postgresql-operator:1.3.0-ppg13-pgbackrest"},{"op":"replace","path":"/spec/pgBadger/
image","value":"percona/percona-postgresql-operator:1.3.0-ppg14-pgbadger"},
{"op":"replace","path":"/spec/pgBouncer/image","value":"percona/percona-postgresql-operator:
1.3.0-ppg14-pgbouncer"},{"op":"replace","path":"/spec/pgPrimary/image","value":"percona/
percona-postgresql-operator:1.3.0-ppg14-postgres-ha"},{"op":"replace","path":"/spec/
userLabels/pgo-version","value":"v1.3.0"},{"op":"replace","path":"/metadata/labels/pgo-
version","value":"v1.3.0"},{"op": "replace", "path": "/spec/pause", "value": false}]'
```

> **Note**
>
> The above example is composed in asumption of using PostgreSQL 14 as a database management system. For PostgreSQL 13 you should change all occurrences of the `ppg14` substring to `ppg13`.

This will carry on the image update, cluster version update and the pause status switch.

3. Now you can enable the `pgbouncer` again:

```
$ kubectl patch perconapgcluster/<cluster-name --type json -p \
    '[
        {"op":"replace","path":"/spec/pgBouncer/size","value":1}
    ]'
```

Wait until the cluster is ready.

Last update: 2022-07-19

Percona LLC and/or its affiliates, © 2009 - 2022

13.3 Scale Percona Distribution for PostgreSQL on Kubernetes and OpenShift

One of the great advantages brought by Kubernetes and the OpenShift platform is the ease of an application scaling. Scaling an application results in adding or removing the Pods and scheduling them to available Kubernetes nodes.

Size of the cluster is dynamically controlled by a pgReplicas.REPLICA-NAME.size key in the Custom Resource options configuration. That's why scaling the cluster needs nothing more but changing this option and applying the updated configuration file. This may be done in a specifically saved config, or on the fly, using the following command:

```
$ kubectl scale --replicas=5 perconapgcluster/cluster1
```

In this example we have changed the number of PostgreSQL Replicas to `5` instances.

Last update: 2022-07-12

Percona LLC and/or its affiliates, © 2009 - 2022

13.4 Monitoring

Percona Monitoring and Management (PMM) provides an excellent solution to monitor Percona Distribution for PostgreSQL.

Note

Only PMM 2.x versions are supported by the Operator.

PMM is a client/server application. PMM Client runs on each node with the database you wish to monitor: it collects needed metrics and sends gathered data to PMM Server. As a user, you connect to PMM Server to see database metrics on a number of dashboards.

That's why PMM Server and PMM Client need to be installed separately.

13.4.1 Installing the PMM Server

PMM Server runs as a *Docker image*, a *virtual appliance*, or on an *AWS instance*. Please refer to the official PMM documentation for the installation instructions.

Percona LLC and/or its affiliates, © 2009 - 2022

13.4.2 Installing the PMM Client

The following steps are needed for the PMM client installation in your Kubernetes-based environment:

1. The PMM client installation is initiated by updating the `pmm` section in the deploy/cr.yaml file.
    - set `pmm.enabled=true`
    - set the `pmm.serverHost` key to your PMM Server hostname,
    - check that the `serverUser` key contains your PMM Server user name (`admin` by default),
    - make sure the `pmmserver` key in the deploy/pmm-secret.yaml secrets file contains the password specified for the PMM Server during its installation.

   Apply changes with the `kubectl apply -f deploy/pmm-secret.yaml` command.

   Info

   You use `deploy/pmm-secret.yaml` file to *create* Secrets Object. The file contains all values for each key/value pair in a convenient plain text format. But the resulting Secrets contain passwords stored as base64-encoded strings. If you want to *update* password field, you'll need to encode the value into base64 format. To do this, you can run `echo -n "password" | base64` in your local shell to get valid values. For example, setting the PMM Server user's password to `new_password` in the `cluster1-pmm-secret` object can be done with the following command:

   ```
   kubectl patch secret/cluster1-pmm-secret -p '{"data":{"pmmserver": '$(echo -n new_password | base64)'}}'
   ```

   When done, apply the edited `deploy/cr.yaml` file:

   ```
   $ kubectl apply -f deploy/cr.yaml
   ```

2. Check that corresponding Pods are not in a cycle of stopping and restarting. This cycle occurs if there are errors on the previous steps:

   ```
   $ kubectl get pods
   $ kubectl logs cluster1-7b7f7898d5-7f5pz -c pmm-client
   ```

3. Now you can access PMM via *https* in a web browser, with the login/password authentication, and the browser is configured to show Percona Distribution for PostgreSQL metrics.

Last update: 2022-07-13

Percona LLC and/or its affiliates, © 2009 - 2022

13.5 Pause/resume PostgreSQL Cluster

There may be external situations when it is needed to pause your Cluster for a while and then start it back up (some works related to the maintenance of the enterprise infrastructure, etc.).

The `deploy/cr.yaml` file contains a special `spec.pause` key for this. Setting it to `true` gracefully stops the cluster:

```
spec:
  .......
  pause: true
```

To start the cluster after it was paused just revert the `spec.pause` key to `false`.

Note

There is an option also to put the cluster into a standby (read-only) mode instead of completely shutting it down. This is done by a special `spec.standby` key, which should be set to `true` for read-only state or should be set to `false` for normal cluster operation:

```
spec:
  .......
  standby: false
```

Last update: 2022-07-13

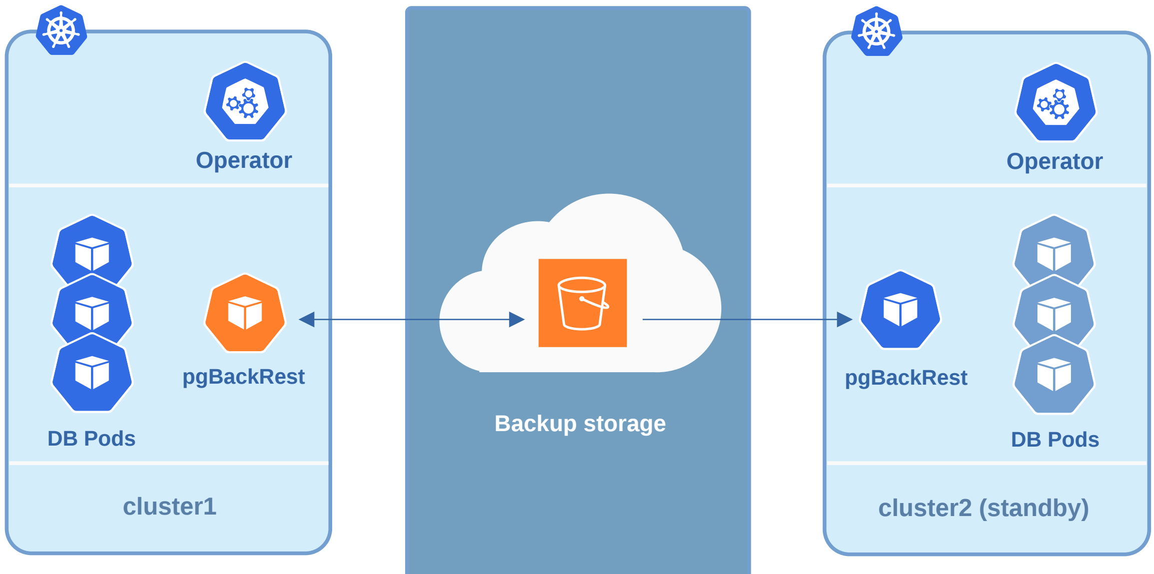
Percona LLC and/or its affiliates, © 2009 - 2022

14. How to

14.1 How to deploy a standby cluster for Disaster Recovery

Deployment of a standby PostgreSQL cluster is mainly targeted for Disaster Recovery (DR), though it can also be used for migrations.

In both cases, it involves using some object storage system for backups, such as AWS S3 or GCP Cloud Storage, which the standby cluster can access:



- there is a *primary cluster* with configured `pgbackrest` tool, which pushes the write-ahead log (WAL) archives to the correct remote repository,
- the *standby cluster* is built from one of these backups, and it is kept in sync with the *primary cluster* by consuming the WAL files copied from the remote repository.

Note

The primary node in the *standby cluster* is **not a streaming replica** from any of the nodes in the *primary cluster*. It relies only on WAL archives to replicate events. For this reason, this approach cannot be used as a High Availability solution.

Percona LLC and/or its affiliates, © 2009 - 2022

Creating such a standby cluster involves the following steps:

Percona LLC and/or its affiliates, © 2009 - 2022

- Copy needed passwords from the *primary cluster* Secrets and adjust them to use the *standby cluster* name.

You need the yq tool installed.

The following commands save the secrets files from `cluster1` under `/tmp/copied-secrets` directory and prepare them to be used in `cluster2`:

```
$ mkdir -p /tmp/copied-secrets/
$ export primary_cluster_name=cluster1
$ export standby_cluster_name=cluster2
$ export secrets="${primary_cluster_name}-users"
$ kubectl get secret/$secrets -o yaml \
yq eval 'del(.metadata.creationTimestamp)' - \
yq eval 'del(.metadata.uid)' - \
yq eval 'del(.metadata.selfLink)' - \
yq eval 'del(.metadata.resourceVersion)' - \
yq eval 'del(.metadata.namespace)' - \
yq eval 'del(.metadata.annotations."kubectl.kubernetes.io/last-applied-configuration")' -
\
yq eval '.metadata.name = "'"${secrets/$primary_cluster_name/$standby_cluster_name}"'"' -
\
yq eval '.metadata.labels.pg-cluster = "'"${standby_cluster_name}"'"' - \
>/tmp/copied-secrets/${secrets/$primary_cluster_name/$standby_cluster_name}
```

- Create the Operator in the Kubernetes environment for the *standby cluster*, if not done:

```
$ kubectl apply -f deploy/operator.yaml
```

- Apply the Adjusted Kubernetes Secrets:

```
$ export standby_cluster_name=cluster2
$ kubectl create -f /tmp/copied-secrets/${standby_cluster_name}-users
```

- Set the backup.repoPath option in the `deploy/cr.yaml` file of your *standby cluster* to the actual place where the *primary cluster* stores backups. If this option is not set in `deploy/cr.yaml` of your *primary cluster*, then the following default naming is used: `/backrestrepo/<primary-cluster-name>-backrest-shared-repo`. For example, in case of `myPrimaryCluster` and `myStandbyCluster` clusters, it should look as follows:

```
...
  name: myStandbyCluster
...
  backup:
    ...
    repoPath: "/backrestrepo/myPrimaryCluster-backrest-shared-repo"
```

- Supply your *standby cluster* with the Kubernetes Secret used by pgBackRest of the *primary cluster* to Access the Storage Bucket. The name of this Secret is `<cluster-name>-backrest-repo-config`, and its content depends on the cloud used for backups (refer to the Operator's backups documentation for this step). The contents of the Secret needs to be the same for both *primary* and *standby* clusters except for the name: e.g. `cluster1-backrest-repo-config` should be recreated as `cluster2-backrest-repo-config`.
- Enable the standby option in your *standby cluster's* `deploy/cr.yaml` file:

```
standby: true
```

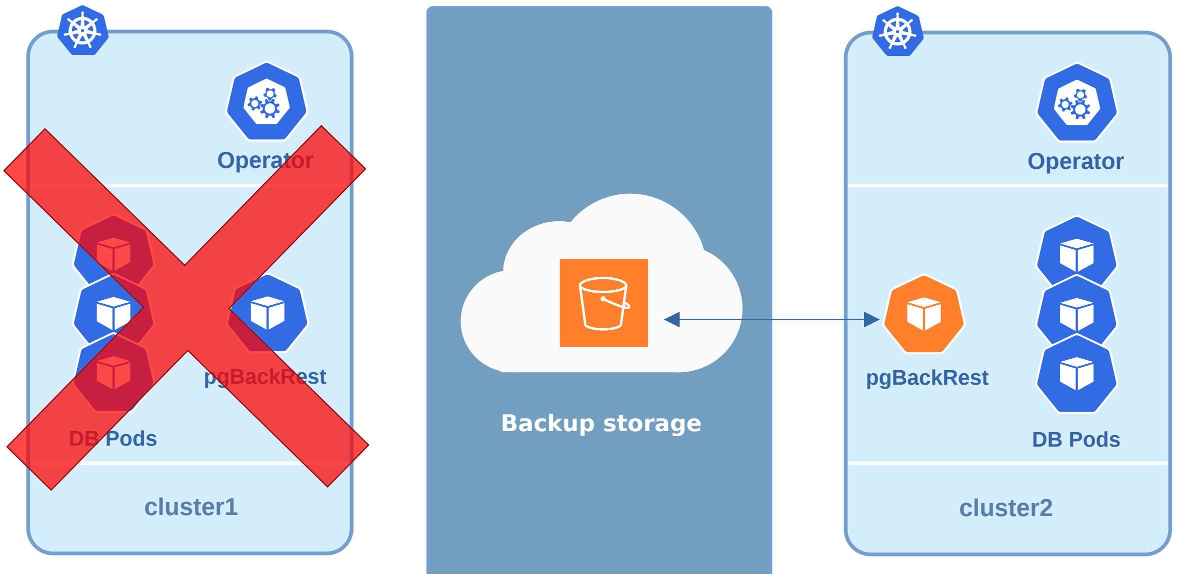

Percona LLC and/or its affiliates, © 2009 - 2022

When you have applied your new cluster configuration with the usual `kubectl -f deploy/cr.yaml` command, it starts the synchronization via pgBackRest, and your Disaster Recovery preparations are over.

When you need to actually use your new cluster, get it out from standby mode, changing the standby option in your `deploy/cr.yaml` file:

```
standby: false
```

Please take into account, that your `cluster1` cluster should not exist at the moment when you get out your `cluster2` from standby:



Note

If `cluster1` still exists for some reason, **make sure it can not connect** to backup storage. Otherwise, both clusters sending WAL archives to it would cause data corruption!

Last update: 2022-07-13

Percona LLC and/or its affiliates, © 2009 - 2022

14.2 Percona Operator for PostgreSQL single-namespace and multi-namespace deployment

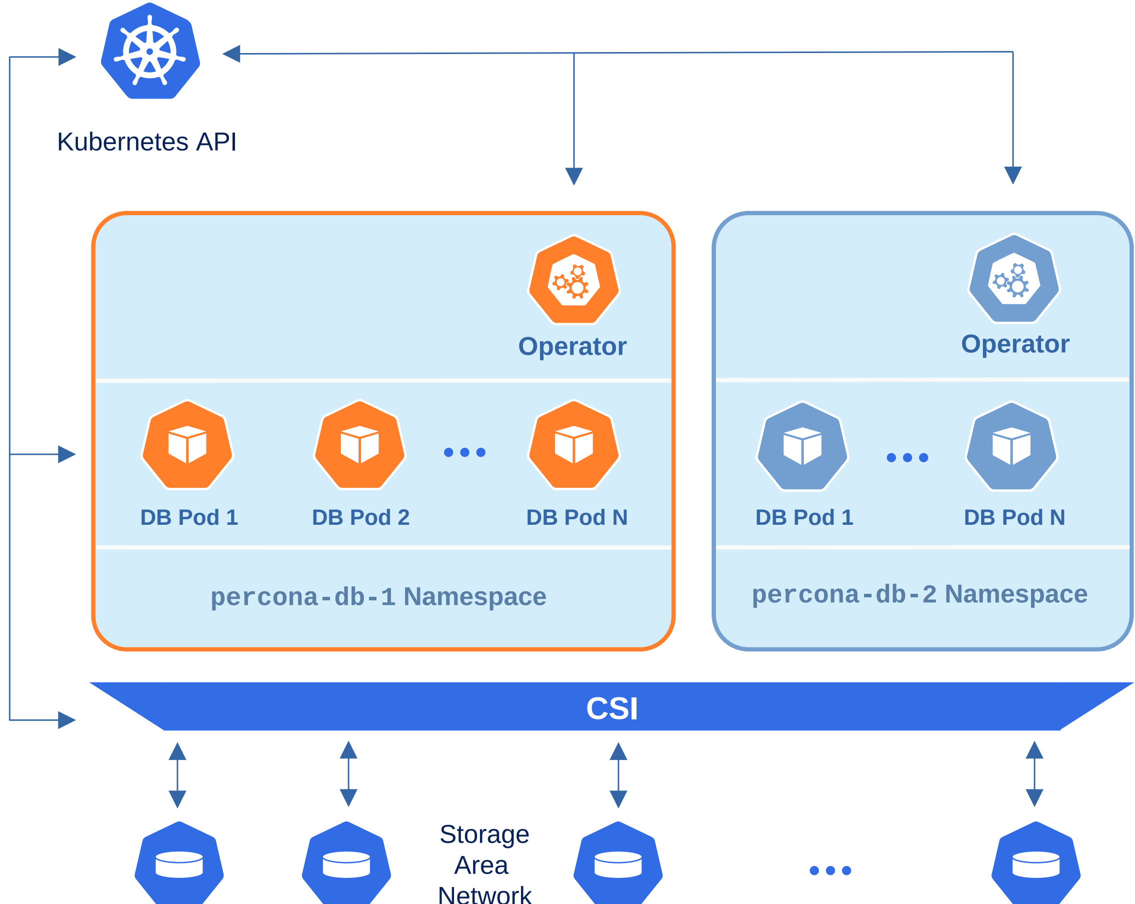
There are two design patterns that you can choose from when deploying Percona Operator for PostgreSQL and PostgreSQL clusters in Kubernetes:

- Namespace-scope - one Operator per Kubernetes namespace,
- Cluster-wide - one Operator can manage clusters in multiple namespaces.

This how-to explains how to configure Percona Operator for PostgreSQL for each scenario.

14.2.1 Namespace-scope

By default, Percona Operator for PostgreSQL functions in a specific Kubernetes namespace. You can create default `pgo` one or some other Namespace during installation (like it is shown in the installation instructions). This approach allows several Operators to co-exist in one Kubernetes-based environment, being separated in different namespaces:



Normally this is a recommended approach, as isolation minimizes impact in case of various failure scenarios. This is the default configuration of our Operator.

Percona LLC and/or its affiliates, © 2009 - 2022

Let's say you have a Namespace in your Kubernetes cluster called `percona-db-1`.

1. Edit the following lines in your deploy/operator.yaml:

```
apiVersion: v1
kind: ConfigMap
metadata:
name: pgo-deployer-cm
data:
  values.yaml: |-
  ...
    namespace: "percona-db-1"
    pgo_operator_namespace: "percona-db-1"

...
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: pgo-deployer-crb
subjects:
...

  - kind: ServiceAccount
    namespace: percona-db-1
```

2. Deploy the Operator:

```
$ kubectl apply -f deploy/operator.yaml -n percona-db-1
```

3. Once Operator is up and running, deploy the database cluster itself:

```
$ kubectl apply -f deploy/cr.yaml -n percona-db-1
```

You can deploy multiple clusters in this namespace.

Percona LLC and/or its affiliates, © 2009 - 2022

Add more namespaces

What if there is a need to deploy clusters in another namespace? The solution for namespace-scope deployment is to have more than one Operator in the corresponding namespace. We will use the `percona-db-2` namespace as an example.

1. Edit or copy `operator.yaml`:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: pgo-deployer-cm
data:
  values.yaml: |-
...
     namespace: "percona-db-2"
     pgo_operator_namespace: "percona-db-2"

...
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: pgo-deployer-crb
subjects:
...
   - kind: ServiceAccount
    namespace: percona-db-2
```

2. Deploy the Operator:

```
$ kubectl apply -f deploy/operator.yaml -n percona-db-2
```

3. Once Operator is up and running deploy the database cluster itself:

```
$ kubectl apply -f deploy/cr.yaml -n percona-db-2
```

Note

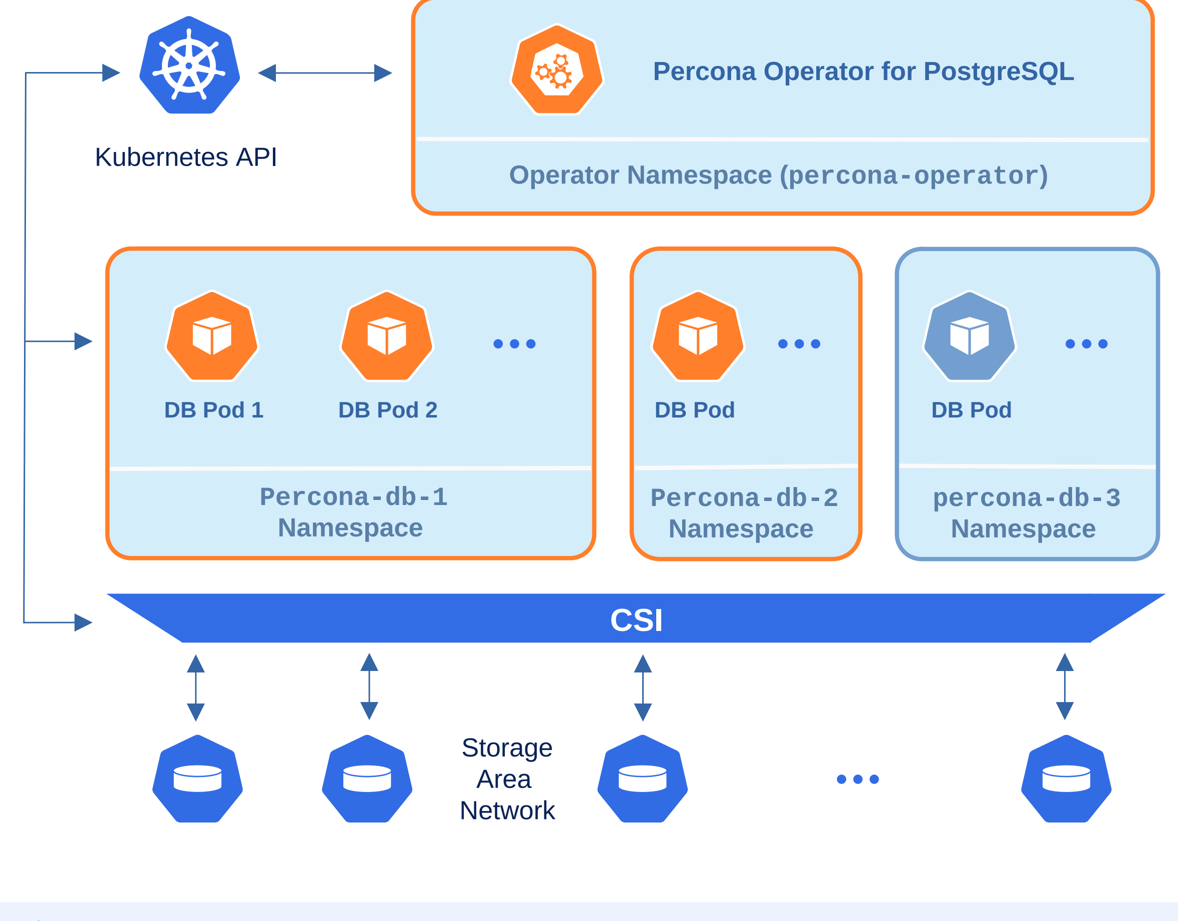
Cluster names may be the same in different namespaces.

14.2.2 Install the Operator cluster-wide

Sometimes it is more convenient to have one Operator watching for Percona Distribution for PostgreSQL custom resources in several namespaces.

We recommend running Percona Operator for PostgreSQL in a traditional way, limited to a specific namespace. But it is possible to run it in so-called *cluster-wide* mode, one Operator watching several namespaces, if needed:

Percona LLC and/or its affiliates, © 2009 - 2022



Note

Please take into account that if several Operators are configured to watch the same namespace, it is entirely unpredictable which one will get ownership of the Custom Resource in it, so this situation should be avoided.

Percona LLC and/or its affiliates, © 2009 - 2022

The following simple example shows how to install Operator cluster-wide on Kubernetes. It does the following:

- deploys Operator into a separate `percona-operator` Namespace,
- allows Operator to control databases in two Namespaces: `percona-db-1` and `percona-db-2`.
- Edit the following lines in your deploy/operator.yaml:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: pgo-deployer-cm
data:
  values.yaml: |-
  ...
    namespace: "percona-db-1,percona-db-2"
    pgo_operator_namespace: "percona-operator"

...
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: pgo-deployer-crb
subjects:
...
  - kind: ServiceAccount
    namespace: percona-operator
```

Note

Before deploying the Operator, please ensure that all Namespaces exist.

- Deploy the Operator:

```
$ kubectl apply -f deploy/operator.yaml -n percona-operator
```

- You can now deploy databases into the namespaces listed in the `namespace:` variable.

```
$ kubectl apply -f deploy/cr.yaml -n percona-db-1
$ kubectl apply -f deploy/cr.yaml -n percona-db-2
```


Percona LLC and/or its affiliates, © 2009 - 2022

Add more namespaces

Let's say we want the Operator to manage databases in one more Namespace: `percona-db-3`.

1. Edit the `operator.yaml` and add one more Namespace into the corresponding field:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: pgo-deployer-cm
data:
  values.yaml: |-
  ...
    namespace: "percona-db-1,percona-db-2,percona-db-3"
```

2. Delete the Operator deployment and deploy job:

```
$ kubectl -n percona-operator delete -f deploy/operator.yaml
$ kubectl -n percona-operator delete deploy postgres-operator
```

Note

Deletion of the Operator does not affect your existing clusters' availability, but limits your ability to manage them. For example, you will not be able to scale the clusters or take backups.

3. Deploy the Operator again with the new Namespace added:

```
$ kubectl apply -f deploy/operator.yaml -n percona-operator
```

4. You can now deploy databases into the new Namespace:

```
$ kubectl apply -f deploy/cr.yaml -n percona-db-3
```

Last update: 2022-07-19

Percona LLC and/or its affiliates, © 2009 - 2022

14.3 Using PostgreSQL tablespaces with Percona Operator for PostgreSQL

Tablespaces allow DBAs to store a database on multiple file systems within the same server and to control where (on which file systems) specific parts of the database are stored. You can think about it as if you were giving names to your disk mounts and then using those names as additional parameters when creating database objects.

PostgreSQL supports this feature, allowing you to *store data outside of the primary data directory*, and Percona Operator for PostgreSQL is a good option to bring this to your Kubernetes environment when needed.

14.3.1 Possible use cases

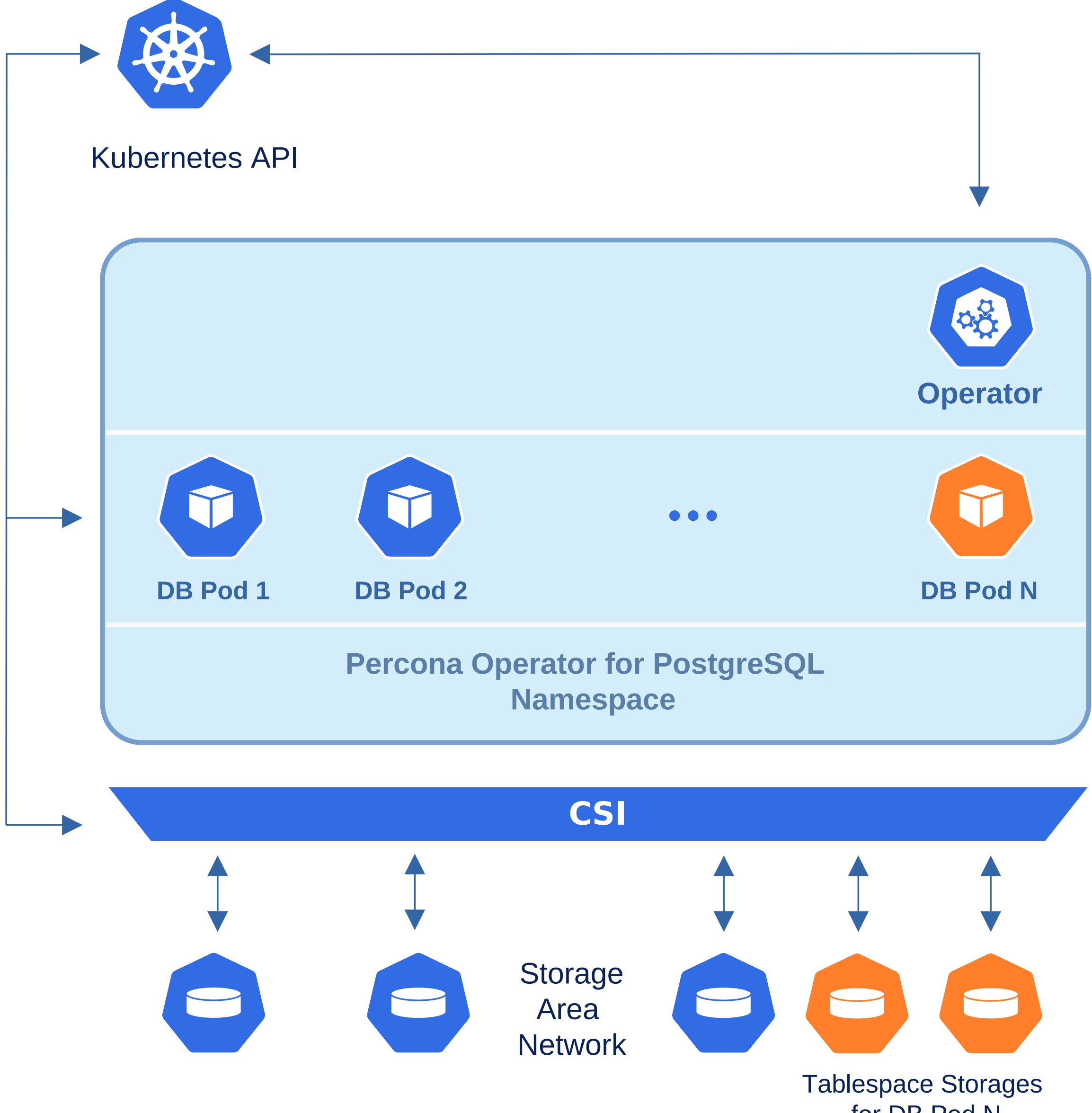
The most obvious use case for tablespaces is performance optimization. You place appropriate parts of the database on fast but expensive storage and engage slower but cheaper storage for lesser-used database objects. The classic example would be using an SSD for heavily-used indexes and using a large slow HDD for archive data.

Of course, the Operator already provides you with traditional Kubernetes approaches to achieve this on a per-Pod basis (Tolerations, etc.). But if you would like to go deeper and make such differentiation at the level of your database objects (tables and indexes), tablespaces are exactly what you would need for that.

Another well-known use case for tablespaces is quickly adding a new partition to the database cluster when you run out of space on the initially used one and cannot extend it (which may look less typical for cloud storage). Finally, you may need tablespaces when migrating your existing architecture to the cloud.

Each tablespace created by Percona Operator for PostgreSQL corresponds to a separate Persistent Volume, mounted in a container to the `/tablespaces` directory.

Percona LLC and/or its affiliates, © 2009 - 2022



14.3.2 Creating a new tablespace

Providing a new tablespace for your database in Kubernetes involves two parts:

1. Configure the new tablespace storage with the Operator,
2. Create database objects in this tablespace with PostgreSQL.

The first part is done in the traditional way of Percona Operators, by modifying Custom Resource via the `deploy/cr.yaml` configuration file. It has a special spec.tablespaceStorages section with subsections names equal to PostgreSQL tablespace names.

The example already present in `deploy/cr.yaml` shows how to create tablespace storage named `lake` 1Gb in size with dynamic provisioning (you can see official Kubernetes documentation on Persistent Volumes for details):

```
spec:
...
  tablespaceStorages:
```


Percona LLC and/or its affiliates, © 2009 - 2022

```
    lake:
      volumeSpec:
        size: 1G
        accessmode: ReadWriteOnce
        storagetype: dynamic
        storageclass: ""
        matchLabels: ""
```

After you apply this by running the `kubectl apply -f deploy/cr.yaml` command, the new `lake` tablespace will appear within your database. Please take into account that if you add your new tablespace to the already existing PostgreSQL cluster, it may take time for the Operator to create Persistent Volume Claims and get Persistent Volumes actually mounted.

Now you can append `TABLESPACE <tablespace_name>` to your `CREATE` SQL statements to implicitly create tables, indexes, or even entire databases in specific tablespaces (of course, your user should have appropriate `CREATE` privileges to make it possible).

Let's create an example table in the already mentioned `lake` tablespace:

```
CREATE TABLE products (
    product_sku character(10),
    quantity int,
    manufactured_date timestamptz)
TABLESPACE lake;
```

It is also possible to set a default tablespace with the `SET default_tablespace = <tablespace_name>;` statement. It will affect all further `CREATE TABLE` and `CREATE INDEX` commands without an explicit tablespace specifier, until you unset it with an empty string.

As you can see, Percona Operator for PostgreSQL simplifies tablespace creation by carrying on all necessary modifications with Persistent Volumes and Pods. The same would not be true for the deletion of an already existing tablespace, which is not automated, neither by the Operator nor by PostgreSQL.

14.3.3 Deleting an existing tablespace

Deleting an existing tablespace from your database in Kubernetes also involves two parts:

- Delete related database objects and tablespace with PostgreSQL,
- Delete tablespace storage in Kubernetes.

To make tablespace deletion with PostgreSQL possible, you should make this tablespace empty (it is impossible to drop a tablespace until *all objects in all databases using this tablespace* have been removed). Tablespaces are listed in the `pg_tablespace` table, and you can use it to find out which objects are stored in a specific tablespace. The example command for the `lake` tablespace will look as follows:

```
SELECT relname FROM pg_class WHERE reltablespace=(SELECT oid FROM pg_tablespace WHERE
spcname='lake');
```

When your tablespace is empty, you can log in to the *PostgreSQL Primary instance* as a *superuser*, and then execute the `DROP TABLESPACE <tablespace_name>;` command.

Now, when the PostgreSQL part is finished, you can remove the tablespace entry from the `tablespaceStorages` section (don't forget to run the `kubectl apply -f deploy/cr.yaml` command to apply changes).

However, Persistent Volumes will still be mounted to the `/tablespaces` directory in PostgreSQL Pods. To remove these mounts, you should edit *all Deployment objects* for `pgPrimary` and `pgReplica` instances in your Kubernetes cluster and remove the `Volume` and `VolumeMount` entries related to your tablespace.

Percona LLC and/or its affiliates, © 2009 - 2022

You can see the list of Deployment objects with the kubectl `get deploy` command. Running it for a default cluster named `cluster1` results in the following output:

```
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
cluster1                      1/1     1            1           156m
cluster1-backrest-shared-repo 1/1     1            1           156m
cluster1-pgbouncer            3/3     3            3           154m
cluster1-repl1                1/1     1            1           154m
cluster1-repl2                1/1     1            1           154m
postgres-operator             1/1     1            1           157m
```

Now run `kubectl edit deploy <oblect_name>` for `cluster1`, `cluster1-repl1`, and `cluster1-repl2` objects consequently. Each command will open a text editor, where you should remove the appropriate lines, which in case of the `lake` tablespace will look as follows:

```
...
spec:
    ...
    containers:
      - name: database
        ...
        volumeMounts:
          - name: tablespace-lake
            mountPath: /tablespaces/lake
    volumes:
      ...
      - name: tablespace-lake
        persistentVolumeClaim:
          claimName: cluster1-tablespace-lake
      ...
```

Finishing the edit causes Pods to be recreated without tablespace mounts.

Last update: 2022-07-13

Percona LLC and/or its affiliates, © 2009 - 2022

15. Reference

15.1 Custom Resource options

The Cluster is configured via the deploy/cr.yaml file.

The metadata part of this file contains the following keys:

- `name` (`cluster1` by default) sets the name of your Percona Distribution for PostgreSQL Cluster; it should include only URL-compatible characters, not exceed 22 characters, start with an alphabetic character, and end with an alphanumeric character;

Percona LLC and/or its affiliates, © 2009 - 2022

The spec part of the deploy/cr.yaml file contains the following sections:

Key	Value type	Default	Description
pause	boolean	`false`	Pause/resume: setting it to `true` gracefully stops the cluster, and setting it to `false` after shut down starts the cluster back.
upgradeOptions	subdoc		Percona Distribution for PostgreSQL upgrade options section
pgPrimary	subdoc		PostgreSQL Primary instance options section
walStorage	subdoc		Tablespaces Storage Section
walStorage	subdoc		Write-ahead Log Storage Section
backup	subdoc		Section to configure backups and pgBackRest
pmm	subdoc		Percona Monitoring and Management section
pgBouncer	subdoc		The pgBouncer connection pooler section
pgReplicas	subdoc		Section required to manage the replicas within a PostgreSQL cluster
pgBadger	subdoc		The pgBadger PostgreSQL log analyzer section

Percona LLC and/or its affiliates, © 2009 - 2022

Key	database
Value	string
Example	`pgdb`
Description	The name of a database that the PostgreSQL user can log into after the PostgreSQL cluster is created

Key	disableAutofail
Value	boolean
Example	`false`
Description	Turns high availability on or off. By default, every cluster can have high availability if there is at least one replica

Key	tlsOnly
Value	boolean
Example	`false`
Description	Enforece Operator to use only Transport Layer Security (TLS) for both internal and external communications

Key	sslCA
Value	string
Example	`cluster1-ssl-ca`
Description	The name of the secret with TLS used for both connection encryption (external traffic), and replication (internal traffic)

Key	sslSecretName
Value	string
Example	`cluster1-ssl-keypair`
Description	The name of the secret created to encrypt external communications

Key	sslReplicationSecretName
Value	string
Example	`cluster1-ssl-keypair"`
Description	The name of the secret created to encrypt internal communications

Key	keepData
Value	boolean
Example	`true`
Description	If `true`, PVCs will be kept after the cluster deletion

Key	keepBackups

Percona LLC and/or its affiliates, © 2009 - 2022

Value	boolean
Example	`true`
Description	If `true`, local backups will be kept after the cluster deletion

Key	pgDataSource.restoreFrom
Value	string
Example	`""`
Description	The name of a data source PostgreSQL cluster, which is used to restore backup to a new cluster

Key	pgDataSource.restoreOpts
Value	string
Example	`""`
Description	Custom pgBackRest options to restore backup to a new cluster

15.1.1 Upgrade Options Section

The `upgradeOptions` section in the deploy/cr.yaml file contains various configuration options to control Percona Distribution for PostgreSQL upgrades.

Key	upgradeOptions.versionServiceEndpoint
Value	string
Example	`https://check.percona.com`
Description	The Version Service URL used to check versions compatibility for upgrade

Key	upgradeOptions.apply
Value	string
Example	`disabled`
Description	Specifies how updates are processed by the Operator. `Never` or `Disabled` will completely disable automatic upgrades, otherwise it can be set to `Latest` or `Recommended` or to a specific version number of Percona Distribution for PostgreSQL to have it version-locked (so that the user can control the version running, but use automatic upgrades to move between them).

Key	upgradeOptions.schedule
Value	string
Example	`0 2 * * *`
Description	Scheduled time to check for updates, specified in the crontab format

Percona LLC and/or its affiliates, © 2009 - 2022

15.1.2 pgPrimary Section

The pgPrimary section controls the PostgreSQL Primary instance.

Percona LLC and/or its affiliates, © 2009 - 2022

Key	pgPrimary.image
Value	string
Example	`perconalab/percona-postgresql-operator:main-ppg13-postgres-ha`
Description	The Docker image of the PostgreSQL Primary instance

Key	pgPrimary.imagePullPolicy
Value	string
Example	`Always`
Description	This option is used to set the policy for updating pgPrimary and pgReplicas images

Key	pgPrimary.resources.requests.memory
Value	int
Example	`256Mi`
Description	The Kubernetes memory requests for a PostgreSQL Primary container

Key	pgPrimary.resources.requests.cpu
Value	string
Example	`500m`
Description	Kubernetes CPU requests for a PostgreSQL Primary container

Key	pgPrimary.resources.limits.cpu
Value	string
Example	`500m`
Description	Kubernetes CPU limits for a PostgreSQL Primary container

Key	pgPrimary.resources.limits.memory
Value	string
Example	`256Mi`
Description	The Kubernetes memory limits for a PostgreSQL Primary container

Key	pgPrimary.affinity.antiAffinityType
Value	string
Example	`preferred`
Description	Pod anti-affinity type, can be either `preferred` or `required`

Key	pgPrimary.affinity.nodeAffinityType
Value	string
Example	`preferred`
Description	Node affinity type, can be either `preferred` or `required`

Percona LLC and/or its affiliates, © 2009 - 2022

Key	pgPrimary.affinity.nodeLabel
Value	label
Example	`kubernetes.io/region: us-central1`
Description	Set labels for PostgreSQL instances Node affinity

Key	pgPrimary.affinity.advanced
Value	subdoc
Example	
Description	Allows using standard Kubernetes affinity constraints for advanced affinity and anti-affinity tuning

Key	pgPrimary.volumeSpec.size
Value	int
Example	`1G`
Description	The Kubernetes PersistentVolumeClaim size for the PostgreSQL Primary storage

Key	pgPrimary.tolerations
Value	subdoc
Example	`node.alpha.kubernetes.io/unreachable`
Description	Kubernetes Pod tolerations

Key	pgPrimary.volumeSpec.size
Value	int
Example	`1G`
Description	The Kubernetes PersistentVolumeClaim size for the PostgreSQL Primary storage

Key	pgPrimary.volumeSpec.accessmode
Value	string
Example	`ReadWriteOnce`
Description	The Kubernetes PersistentVolumeClaim access modes for the PostgreSQL Primary storage

Key	pgPrimary.volumeSpec.storagetype
Value	string
Example	`dynamic`
Description	Type of the PostgreSQL Primary storage provisioning: `create` (the default variant; used if storage is provisioned, e.g. using hostpath) or `dynamic` (for a dynamic storage provisioner, e.g. via a StorageClass)

Key	pgPrimary.volumeSpec.storageclass

Percona LLC and/or its affiliates, © 2009 - 2022

Value	string
Example	`""`
Description	Optionally sets the Kubernetes storage class to use with the PostgreSQL Primary storage PersistentVolumeClaim

Key	pgPrimary.volumeSpec.matchLabels
Value	string
Example	`""`
Description	A PostgreSQL Primary storage label selector

Key	pgPrimary.imagePullPolicy
Value	string
Example	`Always`
Description	This option is used to set the policy for updating pgPrimary and pgReplicas images

Key	pgPrimary.customconfig
Value	string
Example	`""`
Description	Name of the Custom configuration options ConfigMap for PostgreSQL cluster

Percona LLC and/or its affiliates, © 2009 - 2022

15.1.3 Tablespaces Storage Section

The `tablespaceStorages` section in the deploy/cr.yaml file contains configuration options for PostgreSQL Tablespace.

Key	tablespaceStorages.<storage-name>.volumeSpec.size
Value	int
Example	`1G`
Description	The Kubernetes PersistentVolumeClaim size for the PostgreSQL Tablespaces storage

Key	tablespaceStorages.<storage-name>.volumeSpec.accessmode
Value	string
Example	`ReadWriteOnce`
Description	The Kubernetes PersistentVolumeClaim access modes for the PostgreSQL Tablespaces storage

Key	tablespaceStorages.<storage-name>.volumeSpec.storagetype
Value	string
Example	`dynamic`
Description	Type of the PostgreSQL Tablespaces storage provisioning: `create` (the default variant; used if storage is provisioned, e.g. using hostpath) or `dynamic` (for a dynamic storage provisioner, e.g. via a StorageClass)

Key	tablespaceStorages.<storage-name>.volumeSpec.storageclass
Value	string
Example	`""`
Description	Optionally sets the Kubernetes storage class to use with the PostgreSQL Tablespaces storage PersistentVolumeClaim

Key	tablespaceStorages.<storage-name>.volumeSpec.matchLabels
Value	string
Example	`""`
Description	A PostgreSQL Tablespaces storage label selector

Percona LLC and/or its affiliates, © 2009 - 2022

15.1.4 Write-ahead Log Storage Section

The `walStorage` section in the deploy/cr.yaml file contains configuration options for PostgreSQL write-ahead logging.

Key	walStorage.volumeSpec.size
Value	int
Example	`1G`
Description	The Kubernetes PersistentVolumeClaim size for the PostgreSQL Write-ahead Log storage

Key	walStorage.volumeSpec.accessmode
Value	string
Example	`ReadWriteOnce`
Description	The Kubernetes PersistentVolumeClaim access modes for the PostgreSQL Write-ahead Log storage

Key	walStorage.volumeSpec.storagetype
Value	string
Example	`dynamic`
Description	Type of the PostgreSQL Write-ahead Log storage provisioning: `create` (the default variant; used if storage is provisioned, e.g. using hostpath) or `dynamic` (for a dynamic storage provisioner, e.g. via a StorageClass)

Key	walStorage.volumeSpec.storageclass
Value	string
Example	`""`
Description	Optionally sets the Kubernetes storage class to use with the PostgreSQL Write-ahead Log storage PersistentVolumeClaim

Key	walStorage.volumeSpec.matchLabels
Value	string
Example	`""`
Description	A PostgreSQL Write-ahead Log storage label selector

Percona LLC and/or its affiliates, © 2009 - 2022

15.1.5 Backup Section

The `backup` section in the deploy/cr.yaml file contains the following configuration options for the regular Percona Distribution for PostgreSQL backups.

Percona LLC and/or its affiliates, © 2009 - 2022

Key	backup.image
Value	string
Example	`perconalab/percona-postgresql-operator:main-ppg13-pgbackrest`
Description	The Docker image for pgBackRest

Key	backup.backrestRepoImage
Value	string
Example	`perconalab/percona-postgresql-operator:main-ppg13-pgbackrest-repo`
Description	The Docker image for the BackRest repository

Key	backup.resources.requests.cpu
Value	string
Example	`500m`
Description	Kubernetes CPU requests for a pgBackRest container

Key	backup.resources.requests.memory
Value	int
Example	`48Mi`
Description	The Kubernetes memory requests for a pgBackRest container

Key	backup.resources.limits.cpu
Value	int
Example	`1`
Description	Kubernetes CPU limits for a pgBackRest container

Key	backup.resources.limits.memory
Value	int
Example	`64Mi`
Description	The Kubernetes memory limits for a pgBackRest container

Key	backup.affinity.antiAffinityType
Value	string
Example	`preferred`
Description	Pod anti-affinity type, can be either `preferred` or `required`

Key	backup.volumeSpec.size
Value	int
Example	`1G`
Description	The Kubernetes PersistentVolumeClaim size for the pgBackRest Storage

Percona LLC and/or its affiliates, © 2009 - 2022

Key	backup.volumeSpec.accessmode
Value	string
Example	`ReadWriteOnce`
Description	The Kubernetes PersistentVolumeClaim access modes for the pgBackRest Storage

Key	backup.volumeSpec.storagetype
Value	string
Example	`dynamic`
Description	Type of the pgBackRest storage provisioning: `create` (the default variant; used if storage is provisioned, e.g. using hostpath) or `dynamic` (for a dynamic storage provisioner, e.g. via a StorageClass)

Key	backup.volumeSpec.storageclass
Value	string
Example	`""`
Description	Optionally sets the Kubernetes storage class to use with the pgBackRest Storage PersistentVolumeClaim

Key	backup.volumeSpec.matchLabels
Value	string
Example	`""`
Description	A pgBackRest storage label selector

Key	backup.storages.<storage-name>.type
Value	string
Example	`s3`
Description	Type of the storage used for backups

Key	backup.storages.<storage-name>.endpointURL
Value	string
Example	`minio-gateway-svc:9000`
Description	The endpoint URL of the S3-compatible storage to be used for backups (not needed for the original Amazon S3 cloud)

Key	backup.storages.<storage-name>.bucket
Value	string
Example	`""`
Description	The Amazon S3 bucket or Google Cloud Storage bucket

Percona LLC and/or its affiliates, © 2009 - 2022

name used for backups

Key	backup.storages.<storage-name>.region
Value	boolean
Example	`us-east-1`
Description	The AWS region to use for Amazon and all S3-compatible storages

Key	backup.storages.<storage-name>.uriStyle
Value	string
Example	`path`
Description	Optional parameter that specifies if pgBackRest should use the path or host S3 URI style

Key	backup.storages.<storage-name>.verifyTLS
Value	boolean
Example	`false`
Description	Enables or disables TLS verification for pgBackRest

Key	backup.storageTypes
Value	array
Example	`[ "s3" ]`
Description	The backup storage types for the pgBackRest repository

Key	backup.repoPath
Value	string
Example	`""`
Description	Custom path for pgBackRest repository backups

Key	backup.schedule.name
Value	string
Example	`sat-night-backup`
Description	The backup name

Key	backup.schedule.schedule
Value	string
Example	`0 0 * * 6`
Description	Scheduled time to make a backup specified in the
crontab format	

Key	backup.schedule.keep

Percona LLC and/or its affiliates, © 2009 - 2022

Value	int
Example	`3`
Description	The amount of most recent backups to store. Older backups are automatically deleted. Set `keep` to zero or completely remove it to disable automatic deletion of backups

Key	backup.schedule.type
Value	string
Example	`full`
Description	The type of the pgBackRest backup

Key	backup.schedule.storage
Value	string
Example	`local`
Description	The type of the pgBackRest repository

Key	backup.customconfig
Value	string
Example	`""`
Description	Name of the ConfigMap to pass custom pgBackRest configuration options

Key	backup.imagePullPolicy
Value	string
Example	`Always`
Description	This option is used to set the policy for updating pgBackRest images

Percona LLC and/or its affiliates, © 2009 - 2022

15.1.6 PMM Section

The `pmm` section in the deploy/cr.yaml file contains configuration options for Percona Monitoring and Management.

Percona LLC and/or its affiliates, © 2009 - 2022

Key	pmm.enabled
Value	boolean
Example	`false`
Description	Enables or disables monitoring Percona Distribution for PostgreSQL cluster with PMM

Key	pmm.image
Value	string
Example	`percona/pmm-client:2.29.0`
Description	Percona Monitoring and Management (PMM) Client Docker image

Key	pmm.serverHost
Value	string
Example	`monitoring-service`
Description	Address of the PMM Server to collect data from the cluster

Key	pmm.serverUser
Value	string
Example	`admin`
Description	The PMM Server User. The PMM Server password should be configured using Secrets

Key	pmm.pmmSecret
Value	string
Example	`cluster1-pmm-secret`
Description	Name of the Kubernetes Secret object for the PMM Server password

Key	pmm.resources.requests.memory
Value	string
Example	`200M`
Description	The Kubernetes memory requests for a PMM container

Key	pmm.resources.requests.cpu
Value	string
Example	`500m`
Description	Kubernetes CPU requests for a PMM container

Key	pmm.resources.limits.cpu
Value	string
Example	`500m`
Description	Kubernetes CPU limits for a PMM container

Percona LLC and/or its affiliates, © 2009 - 2022

Key	pmm.resources.limits.memory
Value	string
Example	`200M`
Description	The Kubernetes memory limits for a PMM container

Key	pmm.imagePullPolicy
Value	string
Example	`Always`
Description	This option is used to set the policy for updating PMM Client images

Percona LLC and/or its affiliates, © 2009 - 2022

15.1.7 pgBouncer Section

The `pgBouncer` section in the deploy/cr.yaml file contains configuration options for the pgBouncer connection pooler for PostgreSQL.

Percona LLC and/or its affiliates, © 2009 - 2022

Key	pgBouncer.image
Value	string
Example	`perconalab/percona-postgresql-operator:main-ppg13-pgbouncer`
Description	Docker image for the pgBouncer connection pooler

Key	pgBouncer.exposePostgresUser
Value	boolean
Example	`false`
Description	Enables or disables exposing postgres user through pgBouncer

Key	pgBouncer.size
Value	int
Example	`1G`
Description	The number of the pgBouncer Pods to provide connection pooling

Key	pgBouncer.resources.requests.cpu
Value	int
Example	`1`
Description	Kubernetes CPU requests for a pgBouncer container

Key	pgBouncer.resources.requests.memory
Value	int
Example	`128Mi`
Description	The Kubernetes memory requests for a pgBouncer container

Key	pgBouncer.resources.limits.cpu
Value	int
Example	`2`
Description	Kubernetes CPU limits for a pgBouncer container

Key	pgBouncer.resources.limits.memory
Value	int
Example	`512Mi`
Description	The Kubernetes memory limits for a pgBouncer container

Key	pgBouncer.affinity.antiAffinityType
Value	string
Example	`preferred`

Percona LLC and/or its affiliates, © 2009 - 2022

Description	Pod anti-affinity type, can be either `preferred` or `required`
Key	pgBouncer.expose.serviceType
Value	string
Example	`ClusterIP`
Description	Specifies the type of Kubernetes Service for pgBouncer

Key	pgBouncer.expose.loadBalancerSourceRanges
Value	string
Example	`"10.0.0.0/8"`
Description	The range of client IP addresses from which the load balancer should be reachable (if not set, there is no limitations)

Key	pgBouncer.expose.annotations
Value	label
Example	`pg-cluster-annot: cluster1`
Description	The Kubernetes annotations metadata for pgBouncer

Key	pgBouncer.expose.labels
Value	label
Example	`pg-cluster-label: cluster1`
Description	Set labels for the pgBouncer Service

Key	pgBouncer.imagePullPolicy
Value	string
Example	`Always`
Description	This option is used to set the policy for updating pgBouncer images

Percona LLC and/or its affiliates, © 2009 - 2022

15.1.8 pgReplicas Section

The `pgReplicas` section in the deploy/cr.yaml file stores information required to manage the replicas within a PostgreSQL cluster.

Percona LLC and/or its affiliates, © 2009 - 2022

Key	pgReplicas..size
Value	int
Example	`1G`
Description	The number of the PostgreSQL Replica Pods

Key	pgReplicas..resources.requests.cpu
Value	int
Example	`500m`
Description	Kubernetes CPU requests for a PostgreSQL Replica container

Key	pgReplicas..resources.requests.memory
Value	int
Example	`256Mi`
Description	The Kubernetes memory requests for a PostgreSQL Replica container

Key	pgReplicas..resources.limits.cpu
Value	int
Example	`500m`
Description	Kubernetes CPU limits for a PostgreSQL Replica container

Key	pgReplicas..resources.limits.memory
Value	int
Example	`256Mi`
Description	The Kubernetes memory limits
for a PostgreSQL Replica container	

Key	pgReplicas..volumeSpec.accessmode
Value	string
Example	`ReadWriteOnce`
Description	The Kubernetes PersistentVolumeClaim access modes for the PostgreSQL Replica storage

Key	pgReplicas..volumeSpec.size
Value	int
Example	`1G`
Description	The Kubernetes PersistentVolumeClaim size for the PostgreSQL Replica storage

Key	pgReplicas..volumeSpec.storagetype
Value	string

Percona LLC and/or its affiliates, © 2009 - 2022

Example	`dynamic`
Description	Type of the PostgreSQL Replica storage provisioning: `create` (the default variant; used if storage is provisioned, e.g. using hostpath) or `dynamic` (for a dynamic storage provisioner, e.g. via a StorageClass)

Key	pgReplicas..volumeSpec.storageclass
Value	string
Example	`standard`
Description	Optionally sets the Kubernetes storage class to use with the PostgreSQL Replica storage PersistentVolumeClaim

Key	pgReplicas..volumeSpec.matchLabels
Value	string
Example	`""`
Description	A PostgreSQL Replica storage label selector

Key	pgReplicas..labels
Value	label
Example	`pg-cluster-label: cluster1`
Description	Set labels for PostgreSQL Replica Pods

Key	pgReplicas..annotations
Value	label
Example	`pg-cluster-annot: cluster1-1`
Description	The Kubernetes annotations metadata for PostgreSQL Replica

Key	pgReplicas..expose.serviceType
Value	string
Example	`ClusterIP`
Description	Specifies the type of Kubernetes Service for for PostgreSQL Replica

Key	pgReplicas..expose.loadBalancerSourceRanges
Value	string
Example	`"10.0.0.0/8"`
Description	The range of client IP addresses from which the load balancer should be reachable (if not set, there is no limitations)

Key	pgReplicas..expose.annotations
Value	label
Example	`pg-cluster-annot: cluster1`

Percona LLC and/or its affiliates, © 2009 - 2022

Description	The Kubernetes annotations metadata for PostgreSQL Replica

Key	pgReplicas..expose.labels
Value	label
Example	`pg-cluster-label: cluster1`
Description	Set labels for the PostgreSQL Replica Service

15.1.9 pgBadger Section

The `pgBadger` section in the deploy/cr.yaml file contains configuration options for the pgBadger PostgreSQL log analyzer.

Key	pgBadger.enabled
Value	boolean
Example	`false`
Description	Enables or disables the pgBadger PostgreSQL log analyzer

Key	pgBadger.image
Value	string
Example	`perconalab/percona-postgresql-operator:main-ppg13-pgbadger`
Description	pgBadger PostgreSQL log analyzer Docker image

Key	pgBadger.port
Value	int
Example	`10000`
Description	The port number for pgBadger

Key	pgBadger.imagePullPolicy
Value	string
Example	`Always`
Description	This option is used to set the policy for updating pgBadger images

Last update: 2022-08-04

Percona LLC and/or its affiliates, © 2009 - 2022

15.2 Percona certified images

Following table presents Percona’s certified docker images to be used with the Percona Operator for PostgreSQL:

Percona LLC and/or its affiliates, © 2009 - 2022

Image	Digest
percona/percona-postgresql-operator:1.3.0-pgo-deployer	sha256:4f1e7292db27fcd7cbd96066d9e22c38a5cdfbefc5701d003cef8c631587401c
percona/percona-postgresql-operator:1.3.0-postgres-operator	sha256:c997d3b902bd95185252ae6727bca1fef5424ff1d8c6c716e2f419b88946716a
percona/percona-postgresql-operator:1.3.0-pgo-scheduler	sha256:5e6a484f7a559e90e79aa7a034bdcf9415848015c08d244b176de17d47deeb12
percona/percona-postgresql-operator:1.3.0-pgo-rmdata	sha256:3c7688d4ad7bbfda10a773862b58a070f6eb11106a003fd909081ed51edd520d
percona/percona-postgresql-operator:1.3.0-pgo-event	sha256:8089fb1d73bf16757a28d94b1c08649556457d5f61cdfdc33bd75cea1147f012
percona/percona-postgresql-operator:1.3.0-pgo-apiserver	sha256:362eb3c33eaa99ab39fdf42962a38f2818253e1841aa55f656a16bcb79ac5489
percona/percona-postgresql-operator:1.3.0-ppg12-pgbadger	sha256:7f32c664f332bdbd31ef3b85a5c00ebb073012f9c24a1f4a172e6d41686bc660
percona/percona-postgresql-operator:1.3.0-ppg13-pgbadger	sha256:4ed1cfeb88cbe97ca714e013e4f04011f648c6336b699cdc04844cb40a97b453
percona/percona-postgresql-operator:1.3.0-ppg14-pgbadger	sha256:b885df785f1ec057e04d9e7fefc0805d93f479ac812d6b8df4161a178cab21a1
percona/percona-postgresql-operator:1.3.0-ppg12-postgres-ha	sha256:d592459c6d30d84157050d4032ad8676a272f98a54a192ea64dfa58ecc87d1a2
percona/percona-postgresql-operator:1.3.0-ppg13-postgres-ha	sha256:535fe86580cc45cdc2f5de5f0b470d642ba7e358c67c0056364d1d28e347db32
percona/percona-postgresql-operator:1.3.0-ppg14-postgres-ha	sha256:65746063ec9fab1869065f48394bb40b164c33e8102c48c42439812b2a6ba43e
percona/percona-postgresql-operator:1.3.0-ppg12-pgbouncer	sha256:e39b74afd2cb02b290d54d925348c1c8219cdcfbb98ecc5a53767d310476ed7b
percona/percona-postgresql-operator:1.3.0-ppg13-pgbouncer	sha256:90cd9cff47c1c9bb81712cd2159eb4cc9de83c9fb13eca59f5564b58c9e8c672

Percona LLC and/or its affiliates, © 2009 - 2022

Image	Digest
percona/percona-postgresql-operator:1.3.0-ppg14-pgbouncer	sha256:0863a651319c5339898343adb4bbf09ef6e50478badd5927a1582abd5e3c3b74
percona/percona-postgresql-operator:1.3.0-ppg12-pgbackrest	sha256:1581f4a8733a24f82d59b113c396dda2a908e7084f252e561cae49ee8e1c4969
percona/percona-postgresql-operator:1.3.0-ppg13-pgbackrest	sha256:92f78c38b7192880ef4bb63b419fb4de72723073ff1410c4cd244e0458ec08dc
percona/percona-postgresql-operator:1.3.0-ppg14-pgbackrest	sha256:8cfc2320d663152ec46adb79355935e161879249796e4689b3243a75653089ee
percona/percona-postgresql-operator:1.3.0-ppg12-pgbackrest-repo	sha256:87ce6a7ae2c40d8080b0724614e7c53e4767064d04211dad03d3492a2757e5fe
percona/percona-postgresql-operator:1.3.0-ppg13-pgbackrest-repo	sha256:8ad77cadc3f3c76afb5bf231ff0a629aa1f832299eaf7a6e82b298534b5f86ee
percona/percona-postgresql-operator:1.3.0-ppg14-pgbackrest-repo	sha256:0bde250fde02d7e0a234277780d83785f5ef04c48dbf1e57c127d0f2ccbaa34a

Last update: 2022-08-04

Percona LLC and/or its affiliates, © 2009 - 2022

15.3 Frequently Asked Questions

15.3.1 Why do we need to follow "the Kubernetes way" when Kubernetes was never intended to run databases?

As it is well known, the Kubernetes approach is targeted at stateless applications but provides ways to store state (in Persistent Volumes, etc.) if the application needs it. Generally, a stateless mode of operation is supposed to provide better safety, sustainability, and scalability, it makes the already-deployed components interchangeable. You can find more about substantial benefits brought by Kubernetes to databases in this blog post.

The architecture of state-centric applications (like databases) should be composed in a right way to avoid crashes, data loss, or data inconsistencies during hardware failure. Percona Operator for PostgreSQL provides out-of-the-box functionality to automate provisioning and management of highly available PostgreSQL database clusters on Kubernetes.

15.3.2 How can I contact the developers?

The best place to discuss Percona Operator for PostgreSQL with developers and other community members is the community forum.

If you would like to report a bug, use the Percona Operator for PostgreSQL project in JIRA.

15.3.3 How can I analyze PostgreSQL logs with pgBadger?

pgBadger is a report generator for PostgreSQL, which can analyze PostgreSQL logs and provide you web-based representation with charts and various statistics. You can configure it via the pgBadger Section in the deploy/cr.yaml file. The most important option there is pgBadger.enabled, which is off by default. When enabled, a separate pgBadger sidecar container with a specialized HTTP server is added to each PostgreSQL Pod.

You can generate the log report and access it through an exposed port (10000 by default) and an `/api/badgergenerate` endpoint: `http://<Pod-address>:10000/api/badgergenerate`. Also, this report is available in the appropriate pgBadger container as a `/report/index.html` file.

15.3.4 How can I set the Operator to control PostgreSQL in several namespaces?

Sometimes it is convenient to have one Operator watching for PostgreSQL Cluster custom resources in several namespaces.

Percona LLC and/or its affiliates, © 2009 - 2022

You can set additional namespace to be watched by the Operator as follows:

1. First of all clean up the installer artifacts:

```
$ kubectl delete -f deploy/operator.yaml
```

2. Make changes in the `deploy/operator.yaml` file:

   - Find the `pgo-deployer-cm` ConfigMap. It contains the `values.yaml` configuration file. Find the `namespace` key in this file (it is set to `"pgo"` by default) and append your additional namespace to it in a comma-separated list.

```
...
apiVersion: v1
kind: ConfigMap
metadata:
  name: pgo-deployer-cm
data:
  values.yaml: |-
    ...
    namespace: "pgo,myadditionalnamespace"
    ...
```

   - Find the `pgo-deploy` container template in the `pgo-deploy` job spec. It has `env` element named `DEPLOY_ACTION`, which you should change from `install` to `update`:

```
...
apiVersion: batch/v1
kind: Job
metadata:
name: pgo-deploy
...
    containers:
      - name: pgo-deploy
      ...
      env:
        - name: DEPLOY_ACTION
          value: update
          ...
```

3. Now apply your changes as usual:

```
$ kubectl apply -f deploy/operator.yaml
```

Note

You need to perform cleanup between each `DEPLOY_ACTION` activity, which can be either `install`, `update`, or `uninstall`.

15.3.5 How can I store backups on S3-compatible storage with self-issued certificates?

The Operator allows you to store backups on any S3-compatible storage including your private one (for example, a local MinIO installation). Backup and restore with a private S3-compatible storage can be done following the official instruction except the case when you use self-signed certificates and would like to skip TLS verification (which can be reasonable when both your database and storage are located in the same Kubernetes cluster or in the same protected intranet segment).

Percona LLC and/or its affiliates, © 2009 - 2022

The backup.storages. option in the `deploy/cr.yaml` configuration file allows you to skip TLS verification for specific S3-compatible storage. Setting it to `true` is enough to *make a backup*.

Restoring a backup without TLS requires you to make two changes in the `parameters` subsection of the `deploy/restore.yaml` file:

- set `backrest-s3-verify-tls` option to `false`,
- add `--no-repo1-storage-verify-tls` value to `backrest-restore-opts` field.

The following example shows how the resulting `parameters` section may look like:

```
...
parameters:
 backrest-restore-from-cluster: cluster1
 backrest-restore-opts: --type=time --target="2022-05-03 15:22:42" --no-repo1-storage-verify-
tls
 backrest-storage-type: "s3"
 backrest-s3-verify-tls: "false"
tasktype: restore
```

Last update: 2022-07-13

Percona LLC and/or its affiliates, © 2009 - 2022

16. Release Notes

16.1 *Percona Operator for PostgreSQL* 1.3.0

- **Date**

  August 4, 2022

- **Installation**

  Percona Operator for PostgreSQL

16.1.1 Release Highlights

- The automated upgrade is now disabled by default to prevent an unplanned downtimes for user applications and to provide defaults more focused on strict user's control over the cluster
- Flexible anti-affinity configuration is now available, which allows the Operator to isolate PostgreSQL cluster instances on different Kubernetes nodes or to increase its availability by placing PostgreSQL instances in different availability zones

16.1.2 Improvements

- K8SPG-155: Flexible anti-affinity configuration
- K8SPG-196: Add possibility for postgres user to connect to PostgreSQL through PgBouncer with a new `pgBouncer.exposePostgresUser` Custom Resource option
- K8SPG-218: The automated upgrade is now disabled by default to prevent an unplanned downtimes for user applications and to provide defaults more focused on strict user's contol over the cluster; also the user is now able to turn off sending data to the Version Service server
- K8SPG-226: A new build and testing guide allows user to easily experiment with the source code of the Operator

16.1.3 Bugs Fixed

- K8SPG-178: Fix the bug in the instruction on passing custom configuration options for PostgreSQL which made it usable for the new cluster only
- K8SPG-193: Fix the bug which caused the Operator crash without pgReplicas section in Custom Resource definition
- K8SPG-197: Fix the bug which caused the Operator to make connection requests to Version Service even with disabled Smart Update
- K8SPG-207: Fix the bug due to which restoring S3 backup from storage with self-signed certificates didn't work, by introducing the special `backup.storages.verifyTLS` option to address this issue

16.1.4 Supported platforms

The following platforms were tested and are officially supported by the Operator 1.3.0:

- Google Kubernetes Engine (GKE) 1.21 - 1.24
- Amazon Elastic Container Service for Kubernetes (EKS) 1.20 - 1.22
- OpenShift 4.7 - 4.10

This list only includes the platforms that the Percona Operators are specifically tested on as part of the release process. Other Kubernetes flavors and versions depend on the backward compatibility offered by Kubernetes itself.

Percona LLC and/or its affiliates, © 2009 - 2022

Last update: 2022-08-04

Percona LLC and/or its affiliates, © 2009 - 2022

16.2 *Percona Operator for PostgreSQL* 1.2.0

- **Date**

  April 6, 2022

- **Installation**

  Percona Operator for PostgreSQL

16.2.1 Release Highlights

- With this release, the Operator turns to a simplified naming convention and changes its official name to **Percona Operator for PostgreSQL**
- Starting from this release, the Operator automatically generates TLS certificates and turns on encryption by default at cluster creation time. This includes both external certificates which allow users to connect to pgBouncer and PostgreSQL via the encrypted channel, and internal ones used for communication between PostgreSQL cluster nodes
- Various cleanups in the deploy/cr.yaml configuration file simplify the deployment of the cluster, making no need in going into YAML manifests and tuning them

16.2.2 Improvements

- K8SPG-149: It is now possible to explicitly set the version of PostgreSQL for newly provisioned clusters. Before that, all new clusters were started with the latest PostgreSQL version if Version Service was enabled
- K8SPG-148: Add possibility of specifying `imagePullPolicy` option for all images in the Custom Resource of the cluster to run in air-gapped environments
- K8SPG-147: Users now can pass additional customizations to pgBackRest with the pgBackRest configuration options provided via ConfigMap
- K8SPG-142: Introduce deploy/cr-minimal.yaml configuration file to deploy minimal viable clusters - useful for developers to deploy PostgreSQL on local Kubernetes clusters, such as Minikube
- K8SPG-141: YAML manifest cleanup simplifies cluster deployment, reducing it to just two commands
- K8SPG-112: Enable automated generation of TLS certificates and provide encryption for all new clusters by default
- K8SPG-161: The Operator documentation now has a how-to that covers deploying a standby PostgreSQL cluster on Kubernetes

16.2.3 Bugs Fixed

- K8SPG-115: Fix the bug that caused creation a "cloned" cluster with `pgDataSource` to fail due to missing Secrets
- K8SPG-163: Fix the security vulnerability CVE-2021-40346 by removing the unused dependency in the Operator images
- K8SPG-152: Fix the bug that prevented deploying the Operator in disabled/readonly namespace mode. It is now possible to deploy several operators in different namespaces in the same cluster

16.2.4 Options Changes

- K8SPG-116: The `backrest-restore-from-cluster` parameter was renamed to `backrest-restore-cluster` for clarity in the deploy/backup/restore.yaml file used to restore the cluster from a previously saved backup

Percona LLC and/or its affiliates, © 2009 - 2022

16.2.5 Supported platforms

The following platforms were tested and are officially supported by the Operator 1.2.0:

- Google Kubernetes Engine (GKE) 1.19 - 1.22
- Amazon Elastic Container Service for Kubernetes (EKS) 1.19 - 1.21
- OpenShift 4.7 - 4.9

This list only includes the platforms that the Percona Operators are specifically tested on as part of the release process. Other Kubernetes flavors and versions depend on the backward compatibility offered by Kubernetes itself.

Last update: 2022-07-19

Percona LLC and/or its affiliates, © 2009 - 2022

16.3 *Percona Distribution for PostgreSQL Operator* 1.1.0

- **Date**

  December 7, 2021

- **Installation**

  Installing Percona Distribution for PostgreSQL Operator

16.3.1 Release Highlights

- A Kubernetes-native horizontal scaling capability was added to the Custom Resource to unblock Horizontal Pod Autoscaler and Kubernetes Event-driven Autoscaling (KEDA) usage
- The Smart Upgrade functionality along with the technical preview of the Version Service allows users to automatically get the latest version of the software compatible with the Operator and apply it safely
- Percona Distribution for PostgreSQL Operator now supports PostgreSQL 14

16.3.2 New Features

- K8SPG-101: Add support for Kubernetes horizontal scaling to set the number of Replicas dynamically via the `kubectl scale` command or Horizontal Pod Autoscaler
- K8SPG-77: Add support for PostgreSQL 14 in the Operator
- K8SPG-75: Manage Operator's system users hrough a single Secret resource even after cluster creation
- K8SPG-71: Add Smart Upgrade functionality to automate Percona Distribution for PostgreSQL upgrades

16.3.3 Improvements

- K8SPG-96: PMM container does not cause the crash of the whole database Pod if pmm-agent is not working properly

16.3.4 Bugs Fixed

- K8SPG-120: The Operator default behavior is now to keep backups and PVCs when the cluster is deleted

Supported platforms

The following platforms were tested and are officially supported by the Operator 1.1.0:

- Google Kubernetes Engine (GKE) 1.19 - 1.22
- Amazon Elastic Container Service for Kubernetes (EKS) 1.18 - 1.21
- OpenShift 4.7 - 4.9

This list only includes the platforms that the Percona Operators are specifically tested on as part of the release process. Other Kubernetes flavors and versions depend on the backward compatibility offered by Kubernetes itself.

Last update: 2022-07-20

Percona LLC and/or its affiliates, © 2009 - 2022

16.4 *Percona Distribution for PostgreSQL Operator* 1.0.0

- **Date**

  October 7, 2021

- **Installation**

  Installing Percona Distribution for PostgreSQL Operator

Percona announces the general availability of Percona Distribution for PostgreSQL Operator 1.0.0.

The Percona Distribution for PostgreSQL Operator automates the lifecycle, simplifies deploying and managing open source PostgreSQL clusters on Kubernetes.

The Operator follows best practices for configuration and setup of the Percona Distribution for PostgreSQL. The Operator provides a consistent way to package, deploy, manage, and perform a backup and a restore for a Kubernetes application. Operators deliver automation advantages in cloud-native applications.

The advantages are the following:

- Deploy a Percona Distribution for PostgreSQL with no single point of failure and environment which can span multiple availability zones
- Modify the Percona Distribution for PostgreSQL size parameter to add or remove PostgreSQL instances
- Use single Custom Resource as a universal entry point to configure the cluster, similar to other Percona Operators
- Carry on semi-automatic upgrades of the Operator and PostgreSQL to newer versions
- Integrate with Percona Monitoring and Management (PMM) to seamlessly monitor your Percona Distribution for PostgreSQL
- Automate backups or perform on-demand backups as needed with support for performing an automatic restore
- Use cloud storage with S3-compatible APIs or Google Cloud for backups
- Use Transport Layer Security (TLS) for the replication and client traffic
- Support advanced Kubernetes features such as pod disruption budgets, node selector, constraints, tolerations, priority classes, and affinity/anti-affinity

Percona Distribution for PostgreSQL Operator is based on Postgres Operator developed by Crunchy Data.

16.4.1 Release Highlights

- It is now possible to configure scheduled backups following the declarative approach in the `deploy/cr.yaml` file, similar to other Percona Kubernetes Operators
- OpenShift compatibility allows running Percona Distribution for PostgreSQL on Red Hat OpenShift Container Platform
- For the first time, the main functionality of the Operator is covered by functional tests, which ensure the overall quality and stability

16.4.2 New Features and Improvements

- K8SPG-96: PMM Client container does not cause the crash of the whole database Pod if `pmm-agent` is not working properly
- K8SPG-86: The Operator is now compatible with the OpenShift platform
- K8SPG-62: Configuring scheduled backups through the main Custom Resource is now supported

Percona LLC and/or its affiliates, © 2009 - 2022

- K8SPG-99, K8SPG-131: The Operator documentation was substantially improved, and now it covers among other things the usage of Transport Layer Security (TLS) for internal and external communications, and cluster upgrades

16.4.3 Supported Platforms

The following platforms were tested and are officially supported by Operator 1.0.0:

- OpenShift 4.6 - 4.8
- Google Kubernetes Engine (GKE) 1.17 - 1.21
- Amazon Elastic Container Service for Kubernetes (EKS) 1.21

This list only includes the platforms that the Operator is specifically tested on as a part of the release process. Other Kubernetes flavors and versions depend on the backward compatibility offered by Kubernetes itself.

Last update: 2022-07-20

Percona LLC and/or its affiliates, © 2009 - 2022

16.5 *Percona Distribution for PostgreSQL Operator* 0.2.0

- **Date**

  August 12, 2021

- **Installation**

  Installing Percona Distribution for PostgreSQL Operator

Version 0.2.0 of the Percona Distribution for PostgreSQL Operator is a Beta release, and it is not recommended for production environments.

16.5.1 New Features and Improvements

- K8SPG-80: The Custom Resource structure was reworked to provide the same look and feel as in other Percona Operators. Read more about Custom Resource options in the documentation and review the default `deploy/cr.yaml` configuration file on GitHub.
- K8SPG-53: Merged upstream CrunchyData Operator v4.7.0 made it possible to use Google Cloud Storage as an object store for backups without using third-party tools
- K8SPG-42: There is no need to specify the name of the pgBackrest Pod in the backup manifest anymore as it is detected automatically by the Operator
- K8SPG-30: Replicas management is now performed through a main Custom Resource manifest instead of creating separate Kubernetes resources. This also adds the possibility of scaling up/scaling down replicas via the 'deploy/cr.yaml' configuration file
- K8SPG-66: Helm chart is now officially provided with the Operator

Last update: 2022-07-20

Percona LLC and/or its affiliates, © 2009 - 2022

16.6 *Percona Distribution for PostgreSQL Operator* 0.1.0

- **Date**

  May 10, 2021

- **Installation**

  Installing Percona Distribution for PostgreSQL Operator

The Percona Operator is based on best practices for configuration and setup of a Percona Distribution for PostgreSQL on Kubernetes. The benefits of the Operator are many, but saving time and delivering a consistent and vetted environment is key.

Kubernetes provides users with a distributed orchestration system that automates the deployment, management, and scaling of containerized applications. The Operator extends the Kubernetes API with a new custom resource for deploying, configuring, and managing the application through the whole life cycle. You can compare the Kubernetes Operator to a System Administrator who deploys the application and watches the Kubernetes events related to it, taking administrative/operational actions when needed.

Version 0.1.0 of the Percona Distribution for PostgreSQL Operator is a tech preview release and it is not recommended for production environments.

You can install *Percona Distribution for PostgreSQL Operator* on Kubernetes, Google Kubernetes Engine (GKE), and Amazon Elastic Kubernetes Service (EKS) clusters. The Operator is based on Postgres Operator developed by Crunchy Data.

Here are the main differences between v 0.1.0 and the original Operator:

- Percona Distribution for PostgreSQL is now used as the main container image.
- It is possible to specify custom images for all components separately. For example, users can easily build and use custom images for one or several components (e.g. pgBouncer) while all other images will be the official ones. Also, users can build and use all custom images.
- All container images are reworked and simplified. They are built on Red Hat Universal Base Image (UBI) 8.
- The Operator has built-in integration with Percona Monitoring and Management v2.
- A build/test infrastructure was created, and we have started adding e2e tests to be sure that all pieces of the cluster work together as expected.
- We have phased out the `pgo` CLI tool, and the Custom Resource UX will be completely aligned with other Percona Operators in the following release.

Once Percona Operator is promoted to GA, users would be able to get the full package of services from Percona teams.

While the Operator is in its very first release, instructions on how to install and configure it are already available along with the source code hosted in our Github repository.

Help us improve our software quality by reporting any bugs you encounter using our bug tracking system.

Last update: 2022-07-12

Percona LLC and/or its affiliates, © 2009 - 2022