PGO, the Postgres Operator from Crunchy Data

Contents

PGO, the Postgres Operator from Crunchy Data

Latest Release: {{< param operatorVersion >}}

Production Postgres Made Easy

PGO, the Postgres Operator from Crunchy Data, gives you a **declarative Postgres** solution that automatically manages your PostgreSQL clusters.

Designed for your GitOps workflows, it is [easy to get started]({{< relref "quickstart/_index.md" >}}) with Postgres on Kubernetes with PGO. Within a few moments, you can have a production grade Postgres cluster complete with high availability, disaster recovery, and monitoring, all over secure TLS communications.Even better, PGO lets you easily customize your Postgres cluster to tailor it to your workload!

With conveniences like cloning Postgres clusters to using rolling updates to roll out disruptive changes with minimal downtime, PGO is ready to support your Postgres data at every stage of your release pipeline. Built for resiliency and uptime, PGO will keep your desired Postgres in a desired state so you do not need to worry about it.

PGO is developed with many years of production experience in automating Postgres management on Kubernetes, providing a seamless cloud native Postgres solution to keep your data always available.

Supported Platforms

PGO, the Postgres Operator from Crunchy Data, is tested on the following platforms:

- Kubernetes 1.19+
- OpenShift 4.6+
- Rancher
- Google Kubernetes Engine (GKE), including Anthos
- Amazon EKS
- Microsoft AKS

- VMware Tanzu

This list only includes the platforms that the Postgres Operator is specifically tested on as part of the release process: PGO works on other Kubernetes distributions as well, such as Rancher.

The PGO Postgres Operator project source code is available subject to the Apache 2.0 license with the PGO logo and branding assets covered by our trademark guidelines.

Can't wait to try out the PGO, the Postgres Operator from Crunchy Data? Let us show you the quickest possible path to getting up and running.

Prerequisites

Please be sure you have the following utilities installed on your host machine:

- `kubectl`
- `git`

Installation

Step 1: Download the Examples

First, go to GitHub and fork the Postgres Operator examples repository:

https://github.com/CrunchyData/postgres-operator-examples/fork

Once you have forked this repo, you can download it to your working environment with a command similar to this:

```
YOUR_GITHUB_UN="<your GitHub username>"
git clone --depth 1 "git@github.com:${YOUR_GITHUB_UN}/postgres-operator-examples.git"
cd postgres-operator-examples
```

Step 2: Install PGO, the Postgres Operator

You can install PGO, the Postgres Operator from Crunchy Data, using the command below:

```
kubectl apply -k kustomize/install
```

This will create a namespace called `postgres-operator` and create all of the objects required to deploy PGO.

To check on the status of your installation, you can run the following command:

```
kubectl -n postgres-operator get pods \
  --selector=postgres-operator.crunchydata.com/control-plane=postgres-operator \
  --field-selector=status.phase=Running
```

If the PGO Pod is healthy, you should see output similar to:

```
NAME                                READY   STATUS    RESTARTS   AGE
postgres-operator-9dd545d64-t4h8d   1/1     Running   0          3s
```

Create a Postgres Cluster

Let's create a simple Postgres cluster. You can do this by executing the following command:

```
kubectl apply -k kustomize/postgres
```

This will create a Postgres cluster named `hippo` in the `postgres-operator` namespace. You can track the progress of your cluster using the following command:

```
kubectl -n postgres-operator describe postgresclusters.postgres-operator.crunchydata.com hippo
```

Connect to the Postgres cluster

As part of creating a Postgres cluster, the Postgres Operator creates a PostgreSQL user account. The credentials for this account are stored in a Secret that has the name `<clusterName>-pguser-<userName>`.

Within this Secret are attributes that provide information to let you log into the PostgreSQL cluster. These include:

- `user`: The name of the user account.
- `password`: The password for the user account.
- `dbname`: The name of the database that the user has access to by default.
- `host`: The name of the host of the database. This references the Service of the primary Postgres instance.
- `port`: The port that the database is listening on.
- `uri`: A PostgreSQL connection URI that provides all the information for logging into the Postgres database.
- `jdbc-uri`: A PostgreSQL JDBC connection URI that provides all the information for logging into the Postgres database via the JDBC driver.

If you deploy your Postgres cluster with the PgBouncer connection pooler, there are additional values that are populated in the user Secret, including:

- `pgbouncer-host`: The name of the host of the PgBouncer connection pooler. This references the Service of the PgBouncer connection pooler.
- `pgbouncer-port`: The port that the PgBouncer connection pooler is listening on.
- `pgbouncer-uri`: A PostgreSQL connection URI that provides all the information for logging into the Postgres database via the PgBouncer connection pooler.
- `pgbouncer-jdbc-uri`: A PostgreSQL JDBC connection URI that provides all the information for logging into the Postgres database via the PgBouncer connection pooler using the JDBC driver.

Note that **all connections use TLS**. PGO sets up a PKI for your Postgres clusters. You can also choose to bring your own PKI / certificate authority; this is covered later in the documentation.

Connect via `psql` in the Terminal

Connect Directly If you are on the same network as your PostgreSQL cluster, you can connect directly to it using the following command:

```
psql $(kubectl -n postgres-operator get secrets hippo-pguser-hippo -o go-template='{{.data.uri |
   base64decode}}')
```

Connect Using a Port-Forward In a new terminal, create a port forward:

```
PG_CLUSTER_PRIMARY_POD=$(kubectl get pod -n postgres-operator -o name \
  -l postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/role=master)
kubectl -n postgres-operator port-forward "${PG_CLUSTER_PRIMARY_POD}" 5432:5432
```

Establish a connection to the PostgreSQL cluster.

```
PG_CLUSTER_USER_SECRET_NAME=hippo-pguser-hippo

PGPASSWORD=$(kubectl get secrets -n postgres-operator "${PG_CLUSTER_USER_SECRET_NAME}" -o
   go-template='{{.data.password | base64decode}}') \
PGUSER=$(kubectl get secrets -n postgres-operator "${PG_CLUSTER_USER_SECRET_NAME}" -o
   go-template='{{.data.user | base64decode}}') \
PGDATABASE=$(kubectl get secrets -n postgres-operator "${PG_CLUSTER_USER_SECRET_NAME}" -o
   go-template='{{.data.dbname | base64decode}}') \
psql -h localhost
```

Connect an Application

The information provided in the user Secret will allow you to connect an application directly to your PostgreSQL database.

For example, let's connect Keycloak. Keycloak is a popular open source identity management tool that is backed by a PostgreSQL database. Using the `hippo` cluster we created, we can deploy the following manifest file:

```
cat <<EOF >> keycloak.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: keycloak
  namespace: postgres-operator
  labels:
    app.kubernetes.io/name: keycloak
spec:
  selector:
    matchLabels:
      app: keycloak
  template:
    metadata:
      labels:
        app.kubernetes.io/name: keycloak
    spec:
      containers:
      - image: quay.io/keycloak/keycloak:latest
        name: keycloak
        env:
        - name: DB_VENDOR
          value: "postgres"
        - name: DB_ADDR
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: host } }
        - name: DB_PORT
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: port } }
        - name: DB_DATABASE
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: dbname } }
        - name: DB_USER
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: user } }
        - name: DB_PASSWORD
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: password } }
        - name: KEYCLOAK_USER
          value: "admin"
        - name: KEYCLOAK_PASSWORD
          value: "admin"
        - name: PROXY_ADDRESS_FORWARDING
          value: "true"
        ports:
        - name: http
          containerPort: 8080
        - name: https
          containerPort: 8443
        readinessProbe:
          httpGet:
            path: /auth/realms/master
            port: 8080
      restartPolicy: Always

EOF

kubectl apply -f keycloak.yaml
```

There is a full example for how to deploy Keycloak with the Postgres Operator in the `kustomize/keycloak` folder.

Next Steps

Congratulations, you've got your Postgres cluster up and running, perhaps with an application connected to it!

You can find out more about the [`postgresclusters` custom resource definition]({{< relref "references/crd.md" >}}) through the [documentation]({{< relref "references/crd.md" >}}) and through `kubectl explain`, i.e:

```
kubectl explain postgresclusters
```

Let’s work through a tutorial together to better understand the various components of PGO, the Postgres Operator, and how you can fine tune your settings to tailor your Postgres cluster to your application.

Ready to get started with PGO, the Postgres Operator from Crunchy Data? Us too!

This tutorial covers several concepts around day-to-day life managing a Postgres cluster with PGO. While going through and looking at various “HOWTOs” with PGO, we will also cover concepts and features that will help you have a successful cloud native Postgres journey!

In this tutorial, you will learn:

- How to create a Postgres cluster
- How to connect to a Postgres cluster
- How to scale and create a high availability (HA) Postgres cluster
- How to resize your cluster
- How to set up proper disaster recovery and manage backups and restores
- How to apply software updates to Postgres and other components
- How to set up connection pooling
- How to delete your cluster

and more.

You will also see:

- How PGO helps your Postgres cluster achieve high availability
- How PGO can heal your Postgres cluster and ensure all objects are present and available
- How PGO sets up disaster recovery
- How to manage working with PGO in a single namespace or in a cluster-wide installation of PGO.

[Let’s get started]({{< relref “./getting-started.md” >}})!

Connection pooling can be helpful for scaling and maintaining overall availability between your application and the database. PGO helps facilitate this by supporting the PgBouncer connection pooler and state manager.

Let’s look at how we can a connection pooler and connect it to our application!

Adding a Connection Pooler

Let’s look at how we can add a connection pooler using the `kustomize/keycloak` example in the Postgres Operator examples repository.

Connection poolers are added using the `spec.proxy` section of the custom resource. Currently, the only connection pooler supported is PgBouncer.

The only required attribute for adding a PgBouncer connection pooler is to set the `spec.proxy.pgBouncer.image` attribute. In the `kustomize/keycloak/postgres.yaml` file, add the following YAML to the spec:

```
proxy:
  pgBouncer:
    image: {{< param imageCrunchyPGBouncer >}}
```

(You can also find an example of this in the `kustomize/examples/high-availability` example).

Save your changes and run:

```
kubectl apply -k kustomize/keycloak
```

PGO will detect the change and create a new PgBouncer Deployment!

That was fairly easy to set up, so now let’s look at how we can connect our application to the connection pooler.

Connecting to a Connection Pooler

When a connection pooler is deployed to the cluster, PGO adds additional information to the user Secrets to allow for applications to connect directly to the connection pooler. Recall that in this example, our user Secret is called `keycloakdb-pguser-keycloakdb`. Describe the user Secret:

```
kubectl -n postgres-operator describe secrets keycloakdb-pguser-keycloakdb
```

You should see that there are several new attributes included in this Secret that allow for you to connect to your Postgres instance via the connection pooler:

- `pgbouncer-host`: The name of the host of the PgBouncer connection pooler. This references the Service of the PgBouncer connection pooler.
- `pgbouncer-port`: The port that the PgBouncer connection pooler is listening on.
- `pgbouncer-uri`: A PostgreSQL connection URI that provides all the information for logging into the Postgres database via the PgBouncer connection pooler.
- `pgbouncer-jdbc-uri`: A PostgreSQL JDBC connection URI that provides all the information for logging into the Postgres database via the PgBouncer connection pooler using the JDBC driver. Note that by default, the connection string disable JDBC managing prepared transactions for optimal use with PgBouncer.

Open up the file in `kustomize/keycloak/keycloak.yaml`. Update the `DB_ADDR` and `DB_PORT` values to be the following:

```
- name: DB_ADDR
  valueFrom: { secretKeyRef: { name: keycloakdb-pguser-keycloakdb, key: pgbouncer-host } }
- name: DB_PORT
  valueFrom: { secretKeyRef: { name: keycloakdb-pguser-keycloakdb, key: pgbouncer-port } }
```

This changes Keycloak's configuration so that it will now connect through the connection pooler.

Apply the changes:

```
kubectl apply -k kustomize/keycloak
```

Kubernetes will detect the changes and begin to deploy a new Keycloak Pod. When it is completed, Keycloak will now be connected to Postgres via the PgBouncer connection pooler!

TLS

PGO deploys every cluster and component over TLS. This includes the PgBouncer connection pooler. If you are using your own [custom TLS setup]({{< relref "./customize-cluster.md" >}}#customize-tls), you will need to provide a Secret reference for a TLS key / certificate pair for PgBouncer in `spec.proxy.pgBouncer.customTLSSecret`.

Your TLS certificate for PgBouncer should have a Common Name (CN) setting that matches the PgBouncer Service name. This is the name of the cluster suffixed with `-pgbouncer`. For example, for our `hippo` cluster this would be `hippo-pgbouncer`. For the `keycloakdb` example, it would be `keycloakdb-pgbouncer`.

To customize the TLS for PgBouncer, you will need to create a Secret in the Namespace of your Postgres cluster that contains the TLS key (`tls.key`), TLS certificate (`tls.crt`) and the CA certificate (`ca.crt`) to use. The Secret should contain the following values:

```
data:
  ca.crt: <value>
  tls.crt: <value>
  tls.key: <value>
```

For example, if you have files named `ca.crt`, `keycloakdb-pgbouncer.key`, and `keycloakdb-pgbouncer.crt` stored on your local machine, you could run the following command:

```
kubectl create secret generic -n postgres-operator keycloakdb-pgbouncer.tls \
  --from-file=ca.crt=ca.crt \
  --from-file=tls.key=keycloakdb-pgbouncer.key \
  --from-file=tls.crt=keycloakdb-pgbouncer.crt
```

You can specify the custom TLS Secret in the `spec.proxy.pgBouncer.customTLSSecret.name` field in your `postgrescluster.postgres-op` custom resource, e.g.:

```
spec:
  proxy:
    pgBouncer:
      customTLSSecret:
        name: keycloakdb-pgbouncer.tls
```

Customizing

The PgBouncer connection pooler is highly customizable, both from a configuration and Kubernetes deployment standpoint. Let's explore some of the customizations that you can do!

Configuration

PgBouncer configuration can be customized through `spec.proxy.pgBouncer.config`. After making configuration changes, PGO will roll them out to any PgBouncer instance and automatically issue a "reload".

There are several ways you can customize the configuration:

- `spec.proxy.pgBouncer.config.global`: Accepts key-value pairs that apply changes globally to PgBouncer.
- `spec.proxy.pgBouncer.config.databases`: Accepts key-value pairs that represent PgBouncer database definitions.
- `spec.proxy.pgBouncer.config.users`: Accepts key-value pairs that represent connection settings applied to specific users.
- `spec.proxy.pgBouncer.config.files`: Accepts a list of files that are mounted in the `/etc/pgbouncer` directory and loaded before any other options are considered using PgBouncer's include directive.

For example, to set the connection pool mode to `transaction`, you would set the following configuration:

```
spec:
  proxy:
    pgBouncer:
      config:
        global:
          pool_mode: transaction
```

For a reference on PgBouncer configuration please see:

https://www.pgbouncer.org/config.html

Replicas

PGO deploys one PgBouncer instance by default. You may want to run multiple PgBouncer instances to have some level of redundancy, though you still want to be mindful of how many connections are going to your Postgres database!

You can manage the number of PgBouncer instances that are deployed through the `spec.proxy.pgBouncer.replicas` attribute.

Resources

You can manage the CPU and memory resources given to a PgBouncer instance through the `spec.proxy.pgBouncer.resources` attribute. The layout of `spec.proxy.pgBouncer.resources` should be familiar: it follows the same pattern as the standard Kubernetes structure for setting container resources.

For example, let's say we want to set some CPU and memory limits on our PgBouncer instances. We could add the following configuration:

```
spec:
  proxy:
    pgBouncer:
      resources:
        limits:
          cpu: 200m
          memory: 128Mi
```

As PGO deploys the PgBouncer instances using a Deployment these changes are rolled out using a rolling update to minimize disruption between your application and Postgres instances!

Annotations / Labels

You can apply custom annotations and labels to your PgBouncer instances through the `spec.proxy.pgBouncer.metadata.annotations` and `spec.proxy.pgBouncer.metadata.labels` attributes respectively. Note that any changes to either of these two attributes take precedence over any other custom labels you have added.

Pod Anti-Affinity / Pod Affinity / Node Affinity

You can control the pod anti-affinity, pod affinity, and node affinity through the `spec.proxy.pgBouncer.affinity` attribute, specifically:

- `spec.proxy.pgBouncer.affinity.nodeAffinity`: controls node affinity for the PgBouncer instances.
- `spec.proxy.pgBouncer.affinity.podAffinity`: controls Pod affinity for the PgBouncer instances.

- `spec.proxy.pgBouncer.affinity.podAntiAffinity`: controls Pod anti-affinity for the PgBouncer instances.

Each of the above follows the standard Kubernetes specification for setting affinity.

For example, to set a preferred Pod anti-affinity rule for the `kustomize/keycloak` example, you would want to add the following to your configuration:

```
spec:
  proxy:
    pgBouncer:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 1
            podAffinityTerm:
              labelSelector:
                matchLabels:
                  postgres-operator.crunchydata.com/cluster: keycloakdb
                  postgres-operator.crunchydata.com/role: pgbouncer
              topologyKey: kubernetes.io/hostname
```

Tolerations

You can deploy PgBouncer instances to Nodes with Taints by setting Tolerations through `spec.proxy.pgBouncer.tolerations`. This attribute follows the Kubernetes standard tolerations layout.

For example, if there were a set of Nodes with a Taint of `role=connection-poolers:NoSchedule` that you want to schedule your PgBouncer instances to, you could apply the following configuration:

```
spec:
  proxy:
    pgBouncer:
      tolerations:
      - effect: NoSchedule
        key: role
        operator: Equal
        value: connection-poolers
```

Note that setting a toleration does not necessarily mean that the PgBouncer instances will be assigned to Nodes with those taints. Tolerations act as a **key**: they allow for you to access Nodes. If you want to ensure that your PgBouncer instances are deployed to specific nodes, you need to combine setting tolerations with node affinity.

Pod Spread Constraints

Besides using affinity, anti-affinity and tolerations, you can also set Topology Spread Constraints through `spec.proxy.pgBouncer.topologyS`
This attribute follows the Kubernetes standard topology spread contraint layout.

For example, since each of of our pgBouncer Pods will have the standard `postgres-operator.crunchydata.com/role: pgbouncer` Label set, we can use this Label when determining the `maxSkew`. In the example below, since we have 3 nodes with a `maxSkew` of 1 and we've set `whenUnsatisfiable` to `ScheduleAnyway`, we should ideally see 1 Pod on each of the nodes, but our Pods can be distributed less evenly if other constraints keep this from happening.

```
  proxy:
    pgBouncer:
      replicas: 3
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: ScheduleAnyway
          labelSelector:
            matchLabels:
              postgres-operator.crunchydata.com/role: pgbouncer
```

If you want to ensure that your PgBouncer instances are deployed more evenly (or not deployed at all), you need to update `whenUnsatisfiable` to `DoNotSchedule`.

Next Steps

Now that we can enable connection pooling in a cluster, let's explore some [administrative tasks]({{< relref "administrative-tasks.md" >}}) such as manually restarting PostgreSQL using PGO. How do we do that?

Postgres is known for its reliability: it is very stable and typically "just works." However, there are many things that can happen in a distributed environment like Kubernetes that can affect Postgres uptime, including:

- The database storage disk fails or some other hardware failure occurs
- The network on which the database resides becomes unreachable
- The host operating system becomes unstable and crashes
- A key database file becomes corrupted
- A data center is lost
- A Kubernetes component (e.g. a Service) is accidentally deleted

There may also be downtime events that are due to the normal case of operations, such as performing a minor upgrade, security patching of operating system, hardware upgrade, or other maintenance.

The good news: PGO is prepared for this, and your Postgres cluster is protected from many of these scenarios. However, to maximize your high availability (HA), let's first scale up your Postgres cluster.

HA Postgres: Adding Replicas to your Postgres Cluster

PGO provides several ways to add replicas to make a HA cluster:

- Increase the `spec.instances.replicas` value
- Add an additional entry in `spec.instances`

For the purposes of this tutorial, we will go with the first method and set `spec.instances.replicas` to `2`. Your manifest should look similar to:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Apply these updates to your Kubernetes cluster with the following command:

```
kubectl apply -k kustomize/postgres
```

Within moment, you should see a new Postgres instance initializing! You can see all of your Postgres Pods for the `hippo` cluster by running the following command:

```
kubectl -n postgres-operator get pods \
  --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instanc
```

Let's test our high availability set up.

Testing Your HA Cluster

An important part of building a resilient Postgres environment is testing its resiliency, so let's run a few tests to see how PGO performs under pressure!

Test #1: Remove a Service

Let's try removing the primary Service that our application is connected to. This test does not actually require a HA Postgres cluster, but it will demonstrate PGO's ability to react to environmental changes and heal things to ensure your applications can stay up.

Recall in the [connecting a Postgres cluster]({{< relref "./connect-cluster.md" >}}) that we observed the Services that PGO creates, e.g:

```
kubectl -n postgres-operator get svc \
  --selector=postgres-operator.crunchydata.com/cluster=hippo
```

yields something similar to:

```
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
hippo-ha          ClusterIP   10.103.73.92    <none>        5432/TCP   4h8m
hippo-ha-config   ClusterIP   None            <none>        <none>     4h8m
hippo-pods        ClusterIP   None            <none>        <none>     4h8m
hippo-primary     ClusterIP   None            <none>        5432/TCP   4h8m
hippo-replicas    ClusterIP   10.98.110.215   <none>        5432/TCP   4h8m
```

We also mentioned that the application is connected to the `hippo-primary` Service. What happens if we were to delete this Service?

```
kubectl -n postgres-operator delete svc hippo-primary
```

This would seem like it could create a downtime scenario, but run the above selector again:

```
kubectl -n postgres-operator get svc \
  --selector=postgres-operator.crunchydata.com/cluster=hippo
```

You should see something similar to:

```
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
hippo-ha          ClusterIP   10.103.73.92    <none>        5432/TCP   4h8m
hippo-ha-config   ClusterIP   None            <none>        <none>     4h8m
hippo-pods        ClusterIP   None            <none>        <none>     4h8m
hippo-primary     ClusterIP   None            <none>        5432/TCP   3s
hippo-replicas    ClusterIP   10.98.110.215   <none>        5432/TCP   4h8m
```

Wow – PGO detected that the primary Service was deleted and it recreated it! Based on how your application connects to Postgres, it may not have even noticed that this event took place!

Now let's try a more extreme downtime event.

Test #2: Remove the Primary StatefulSet

StatefulSets are a Kubernetes object that provide helpful mechanisms for managing Pods that interface with stateful applications, such as databases. They provide a stable mechanism for managing Pods to help ensure data is retrievable in a predictable way.

What happens if we remove the StatefulSet that is pointed to the Pod that represents the Postgres primary? First, let's determine which Pod is the primary. We'll store it in an environmental variable for convenience.

```
PRIMARY_POD=$(kubectl -n postgres-operator get pods \
  --selector=postgres-operator.crunchydata.com/role=master \
  -o jsonpath='{.items[*].metadata.labels.postgres-operator\.crunchydata\.com/instance}')
```

Inspect the environmental variable to see which Pod is the current primary:

```
echo $PRIMARY_POD
```

should yield something similar to:

```
hippo-instance1-zj5s
```

We can use the value above to delete the StatefulSet associated with the current Postgres primary instance:

```
kubectl delete sts -n postgres-operator "${PRIMARY_POD}"
```

Let's see what happens. Try getting all of the StatefulSets for the Postgres instances in the `hippo` cluster:

```
kubectl get sts -n postgres-operator \
  --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instan
```

You should see something similar to:

```
NAME                    READY   AGE
hippo-instance1-6kbw    1/1     15m
hippo-instance1-zj5s    0/1     1s
```

PGO recreated the StatefulSet that was deleted! After this "catastrophic" event, PGO proceeds to heal the Postgres instance so it can rejoin the cluster. We cover the high availability process in greater depth later in the documentation.

What about the other instance? We can see that it became the new primary though the following command:

```
kubectl -n postgres-operator get pods \
  --selector=postgres-operator.crunchydata.com/role=master \
  -o jsonpath='{.items[*].metadata.labels.postgres-operator\.crunchydata\.com/instance}'
```

which should yield something similar to:

```
hippo-instance1-6kbw
```

You can test that the failover successfully occurred in a few ways. You can connect to the example Keycloak application that we [deployed in the previous section]({{< relref "./connect-cluster.md" >}}). Based on Keycloak's connection retry logic, you may need to wait a moment for it to reconnect, but you will see it connected and resume being able to read and write data. You can also connect to the Postgres instance directly and execute the following command:

```
SELECT NOT pg_catalog.pg_is_in_recovery() is_primary;
```

If it returns `true` (or `t`), then the Postgres instance is a primary!

What if PGO was down during the downtime event? Failover would still occur: the Postgres HA system works independently of PGO and can maintain its own uptime. PGO will still need to assist with some of the healing aspects, but your application will still maintain read/write connectivity to your Postgres cluster!

Synchronous Replication

PostgreSQL supports synchronous replication, which is a replication mode designed to limit the risk of transaction loss. Synchronous replication waits for a transaction to be written to at least one additional server before it considers the transaction to be committed. For more information on synchronous replication, please read about PGO's [high availability architecture]({{}}#synchronous-replication-guarding-against-transactions-loss)

To add synchronous replication to your Postgres cluster, you can add the following to your spec:

```
spec:
  patroni:
    dynamicConfiguration:
      synchronous_mode: true
```

While PostgreSQL defaults `synchronous_commit` to `on`, you may also want to explicitly set it, in which case the above block becomes:

```
spec:
  patroni:
    dynamicConfiguration:
      synchronous_mode: true
      postgresql:
        parameters:
          synchronous_commit: "on"
```

Note that Patroni, which manages many aspects of the cluster's availability, will favor availability over synchronicity. This means that if a synchronous replica goes down, Patroni will allow for asynchronous replication to continue as well as writes to the primary. However, if you want to disable all writing if there are no synchronous repliacs available, you would have to enable `synchronous_mode_strict`, i.e.:

```
spec:
  patroni:
    dynamicConfiguration:
      synchronous_mode: true
      synchronous_mode_strict: true
```

Affinity

Kubernetes affinity rules, which include Pod anti-affinity and Node affinity, can help you to define where you want your workloads to reside. Pod anti-affinity is important for high availability: when used correctly, it ensures that your Postgres instances are distributed amongst different Nodes. Node affinity can be used to assign instances to specific Nodes, e.g. to utilize hardware that's optimized for databases.

Understanding Pod Labels

PGO sets up several labels for Postgres cluster management that can be used for Pod anti-affinity or affinity rules in general. These include:

- `postgres-operator.crunchydata.com/cluster`: This is assigned to all managed Pods in a Postgres cluster. The value of this label is the name of your Postgres cluster, in this case: `hippo`.
- `postgres-operator.crunchydata.com/instance-set`: This is assigned to all Postgres instances within a group of `spec.instances`. In the example above, the value of this label is `instance1`. If you do not assign a label, the value is automatically set by PGO using a `NN` format, e.g. `00`.
- `postgres-operator.crunchydata.com/instance`: This is a unique label assigned to each Postgres instance containing the name of the Postgres instance.

Let's look at how we can set up affinity rules for our Postgres cluster to help improve high availability.

Pod Anti-affinity

Kubernetes has two types of Pod anti-affinity:

- Preferred: With preferred (`preferredDuringSchedulingIgnoredDuringExecution`) Pod anti-affinity, Kubernetes will make a best effort to schedule Pods matching the anti-affinity rules to different Nodes. However, if it is not possible to do so, then Kubernetes may schedule one or more Pods to the same Node.
- Required: With required (`requiredDuringSchedulingIgnoredDuringExecution`) Pod anti-affinity, Kubernetes mandates that each Pod matching the anti-affinity rules **must** be scheduled to different Nodes. However, a Pod may not be scheduled if Kubernetes cannot find a Node that does not contain a Pod matching the rules.

There is a tradeoff with these two types of pod anti-affinity: while "required" anti-affinity will ensure that all the matching Pods are scheduled on different Nodes, if Kubernetes cannot find an available Node, your Postgres instance may not be scheduled. Likewise, while "preferred" anti-affinity will make a best effort to scheduled your Pods on different Nodes, Kubernetes may compromise and schedule more than one Postgres instance of the same cluster on the same Node.

By understanding these tradeoffs, the makeup of your Kubernetes cluster, and your requirements, you can choose the method that makes the most sense for your Postgres deployment. We'll show examples of both methods below!

Using Preferred Pod Anti-Affinity First, let's deploy our Postgres cluster with preferred Pod anti-affinity. Note that if you have a single-node Kubernetes cluster, you will not see your Postgres instances deployed to different nodes. However, your Postgres instances *will* be deployed.

We can set up our HA Postgres cluster with preferred Pod anti-affinity like so:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 1
            podAffinityTerm:
              topologyKey: kubernetes.io/hostname
              labelSelector:
                matchLabels:
                  postgres-operator.crunchydata.com/cluster: hippo
                  postgres-operator.crunchydata.com/instance-set: instance1
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Apply those changes in your Kubernetes cluster.

Let's take a closer look at this section:

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 1
      podAffinityTerm:
        topologyKey: kubernetes.io/hostname
        labelSelector:
          matchLabels:
            postgres-operator.crunchydata.com/cluster: hippo
            postgres-operator.crunchydata.com/instance-set: instance1
```

`spec.instances.affinity.podAntiAffinity` follows the standard Kubernetes Pod anti-affinity spec. The values for the `matchLabels` are derived from what we described in the previous section: `postgres-operator.crunchydata.com/cluster` is set to our cluster name of `hippo`, and `postgres-operator.crunchydata.com/instance-set` is set to the instance set name of `instance1`. We choose a `topologyKey` of `kubernetes.io/hostname`, which is standard in Kubernetes clusters.

Preferred Pod anti-affinity will perform a best effort to schedule your Postgres Pods to different nodes. Let's see how you can require your Postgres Pods to be scheduled to different nodes.

Using Required Pod Anti-Affinity Required Pod anti-affinity forces Kubernetes to scheduled your Postgres Pods to different Nodes. Note that if Kubernetes is unable to schedule all Pods to different Nodes, some of your Postgres instances may become unavailable.

Using the previous example, let's indicate to Kubernetes that we want to use required Pod anti-affinity for our Postgres clusters:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - topologyKey: kubernetes.io/hostname
            labelSelector:
              matchLabels:
                postgres-operator.crunchydata.com/cluster: hippo
                postgres-operator.crunchydata.com/instance-set: instance1
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Apply those changes in your Kubernetes cluster.

If you are in a single Node Kubernetes clusters, you will notice that not all of your Postgres instance Pods will be scheduled. This is due to the `requiredDuringSchedulingIgnoredDuringExecution` preference. However, if you have enough Nodes available, you will see the Postgres instance Pods scheduled to different Nodes:

```
kubectl get pods -n postgres-operator -o wide \
  --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instanc
```

Node Affinity

Node affinity can be used to assign your Postgres instances to Nodes with specific hardware or to guarantee a Postgres instance resides in a specific zone. Node affinity can be set within the `spec.instances.affinity.nodeAffinity` attribute, following the standard Kubernetes node affinity spec.

Let's see an example with required Node affinity. Let's say we have a set of Nodes that are reserved for database usage that have a label `workload-role=db`. We can create a Postgres cluster with a required Node affinity rule to scheduled all of the databases to those Nodes using the following configuration:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: workload-role
                operator: In
                values:
                - db
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Pod Topology Spread Constraints

In addition to affinity and anti-affinity settings, Kubernetes Pod Topology Spread Constraints can also help you to define where you want your workloads to reside. However, while PodAffinity allows any number of Pods to be added to a qualifying topology domain, and PodAntiAffinity allows only one Pod to be scheduled into a single topology domain, topology spread constraints allow you to distribute Pods across different topology domains with a finer level of control.

API Field Configuration

The spread constraint API fields can be configured for instance, pgBouncer and pgBackRest repo host pods. The basic configuration is as follows:

```
      topologySpreadConstraints:
      - maxSkew: <integer>
        topologyKey: <string>
        whenUnsatisfiable: <string>
        labelSelector: <object>
```

where "maxSkew" describes the maximum degree to which Pods can be unevenly distributed, "topologyKey" is the key that defines a topology in the Nodes' Labels, "whenUnsatisfiable" specifies what action should be taken when "maxSkew" can't be satisfied, and "labelSelector" is used to find matching Pods.

Example Spread Contraints

To help illustrate how you might use this with your cluster, we can review examples for configuring spread constraints on our Instance and pgBackRest repo host Pods. For this example, assume we have a three node Kubernetes cluster where the first node is labeled with `my-node-label=one`, the second node is labeled with `my-node-label=two` and the final node is labeled `my-node-label=three`. The label key `my-node-label` will function as our `topologyKey`. Note all three nodes in our examples will be schedulable, so a Pod could live on any of the three Nodes.

Instance Pod Spread Constraints To begin, we can set our topology spread contraints on our cluster Instance Pods. Given this configuration

```
  instances:
    - name: instance1
      replicas: 5
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: DoNotSchedule
          labelSelector:
            matchLabels:
              postgres-operator.crunchydata.com/instance-set: instance1
```

we will expect 5 Instance pods to be created. Each of these Pods will have the standard `postgres-operator.crunchydata.com/instance-se` `instance1` Label set, so each Pod will be properly counted when determining the `maxSkew`. Since we have 3 nodes with a `maxSkew` of 1 and we've set `whenUnsatisfiable` to `DoNotSchedule`, we should see 2 Pods on 2 of the nodes and 1 Pod on the remaining Node, thus ensuring our Pods are distributed as evenly as possible.

pgBackRest Repo Pod Spread Constraints We can also set topology spread constraints on our cluster's pgBackRest repo host pod. While we normally will only have a single pod per cluster, we could use a more generic label to add a preference that repo host Pods from different clusters are distributed among our Nodes. For example, by setting our `matchLabel` value to `postgres-operator.crunchydata.com/pgbackrest: ""` and our `whenUnsatisfiable` value to `ScheduleAnyway`, we will allow our repo host Pods to be scheduled no matter what Nodes may be available, but attempt to minimize skew as much as possible.

```
      repoHost:
        topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: ScheduleAnyway
          labelSelector:
            matchLabels:
              postgres-operator.crunchydata.com/pgbackrest: ""
```

Putting it All Together Now that each of our Pods has our desired Topology Spread Constraints defined, let's put together a complete cluster definition:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 5
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: DoNotSchedule
          labelSelector:
            matchLabels:
              postgres-operator.crunchydata.com/instance-set: instance1
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1G
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repoHost:
        topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: ScheduleAnyway
          labelSelector:
            matchLabels:
              postgres-operator.crunchydata.com/pgbackrest: ""
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
```

```
          accessModes:
          - "ReadWriteOnce"
          resources:
            requests:
              storage: 1G
```

You can then apply those changes in your Kubernetes cluster.

Once your cluster finishes deploying, you can check that your Pods are assigned to the correct Nodes:

```
kubectl get pods -n postgres-operator -o wide
  --selector=postgres-operator.crunchydata.com/cluster=hippo
```

Next Steps

We've now seen how PGO helps your application stay "always on" with your Postgres database. Now let's explore how PGO can minimize or eliminate downtime for operations that would normally cause that, such as [resizing your Postgres cluster]({{< relref "./resize-cluster.md" >}}).

Postgres is known for its reliability: it is very stable and typically "just works." However, there are many things that can happen in a distributed environment like Kubernetes that can affect Postgres uptime, including:

- The database storage disk fails or some other hardware failure occurs
- The network on which the database resides becomes unreachable
- The host operating system becomes unstable and crashes
- A key database file becomes corrupted
- A data center is lost
- A Kubernetes component (e.g. a Service) is accidentally deleted

There may also be downtime events that are due to the normal case of operations, such as performing a minor upgrade, security patching of operating system, hardware upgrade, or other maintenance.

The good news: PGO is prepared for this, and your Postgres cluster is protected from many of these scenarios. However, to maximize your high availability (HA), let's first scale up your Postgres cluster.

HA Postgres: Adding Replicas to your Postgres Cluster

PGO provides several ways to add replicas to make a HA cluster:

- Increase the `spec.instances.replicas` value
- Add an additional entry in `spec.instances`

For the purposes of this tutorial, we will go with the first method and set `spec.instances.replicas` to `2`. Your manifest should look similar to:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Apply these updates to your Kubernetes cluster with the following command:

```
kubectl apply -k kustomize/postgres
```

Within moment, you should see a new Postgres instance initializing! You can see all of your Postgres Pods for the `hippo` cluster by running the following command:

```
kubectl -n postgres-operator get pods \
  --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instanc
```

Let's test our high availability set up.

Testing Your HA Cluster

An important part of building a resilient Postgres environment is testing its resiliency, so let's run a few tests to see how PGO performs under pressure!

Test #1: Remove a Service

Let's try removing the primary Service that our application is connected to. This test does not actually require a HA Postgres cluster, but it will demonstrate PGO's ability to react to environmental changes and heal things to ensure your applications can stay up.

Recall in the [connecting a Postgres cluster]({{< relref "./connect-cluster.md" >}}) that we observed the Services that PGO creates, e.g:

```
kubectl -n postgres-operator get svc \
  --selector=postgres-operator.crunchydata.com/cluster=hippo
```

yields something similar to:

```
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
hippo-ha          ClusterIP   10.103.73.92    <none>        5432/TCP   4h8m
hippo-ha-config   ClusterIP   None            <none>        <none>     4h8m
hippo-pods        ClusterIP   None            <none>        <none>     4h8m
hippo-primary     ClusterIP   None            <none>        5432/TCP   4h8m
hippo-replicas    ClusterIP   10.98.110.215   <none>        5432/TCP   4h8m
```

We also mentioned that the application is connected to the `hippo-primary` Service. What happens if we were to delete this Service?

```
kubectl -n postgres-operator delete svc hippo-primary
```

This would seem like it could create a downtime scenario, but run the above selector again:

```
kubectl -n postgres-operator get svc \
  --selector=postgres-operator.crunchydata.com/cluster=hippo
```

You should see something similar to:

```
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
hippo-ha          ClusterIP   10.103.73.92    <none>        5432/TCP   4h8m
hippo-ha-config   ClusterIP   None            <none>        <none>     4h8m
hippo-pods        ClusterIP   None            <none>        <none>     4h8m
hippo-primary     ClusterIP   None            <none>        5432/TCP   3s
hippo-replicas    ClusterIP   10.98.110.215   <none>        5432/TCP   4h8m
```

Wow – PGO detected that the primary Service was deleted and it recreated it! Based on how your application connects to Postgres, it may not have even noticed that this event took place!

Now let's try a more extreme downtime event.

Test #2: Remove the Primary StatefulSet

StatefulSets are a Kubernetes object that provide helpful mechanisms for managing Pods that interface with stateful applications, such as databases. They provide a stable mechanism for managing Pods to help ensure data is retrievable in a predictable way.

What happens if we remove the StatefulSet that is pointed to the Pod that represents the Postgres primary? First, let's determine which Pod is the primary. We'll store it in an environmental variable for convenience.

```
PRIMARY_POD=$(kubectl -n postgres-operator get pods \
  --selector=postgres-operator.crunchydata.com/role=master \
  -o jsonpath='{.items[*].metadata.labels.postgres-operator\.crunchydata\.com/instance}')
```

Inspect the environmental variable to see which Pod is the current primary:

```
echo $PRIMARY_POD
```

should yield something similar to:

```
hippo-instance1-zj5s
```

We can use the value above to delete the StatefulSet associated with the current Postgres primary instance:

```
kubectl delete sts -n postgres-operator "${PRIMARY_POD}"
```

Let's see what happens. Try getting all of the StatefulSets for the Postgres instances in the `hippo` cluster:

```
kubectl get sts -n postgres-operator \
  --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instan
```

You should see something similar to:

```
NAME                   READY   AGE
hippo-instance1-6kbw   1/1     15m
hippo-instance1-zj5s   0/1     1s
```

PGO recreated the StatefulSet that was deleted! After this "catastrophic" event, PGO proceeds to heal the Postgres instance so it can rejoin the cluster. We cover the high availability process in greater depth later in the documentation.

What about the other instance? We can see that it became the new primary though the following command:

```
kubectl -n postgres-operator get pods \
  --selector=postgres-operator.crunchydata.com/role=master \
  -o jsonpath='{.items[*].metadata.labels.postgres-operator\.crunchydata\.com/instance}'
```

which should yield something similar to:

```
hippo-instance1-6kbw
```

You can test that the failover successfully occurred in a few ways. You can connect to the example Keycloak application that we [deployed in the previous section]({{< relref "./connect-cluster.md" >}}). Based on Keycloak's connection retry logic, you may need to wait a moment for it to reconnect, but you will see it connected and resume being able to read and write data. You can also connect to the Postgres instance directly and execute the following command:

```
SELECT NOT pg_catalog.pg_is_in_recovery() is_primary;
```

If it returns `true` (or `t`), then the Postgres instance is a primary!

What if PGO was down during the downtime event? Failover would still occur: the Postgres HA system works independently of PGO and can maintain its own uptime. PGO will still need to assist with some of the healing aspects, but your application will still maintain read/write connectivity to your Postgres cluster!

Synchronous Replication

PostgreSQL supports synchronous replication, which is a replication mode designed to limit the risk of transaction loss. Synchronous replication waits for a transaction to be written to at least one additional server before it considers the transaction to be committed. For more information on synchronous replication, please read about PGO's [high availability architecture]({{}}#synchronous-replication-guarding-against-transactions-loss)

To add synchronous replication to your Postgres cluster, you can add the following to your spec:

```
spec:
  patroni:
    dynamicConfiguration:
      synchronous_mode: true
```

While PostgreSQL defaults `synchronous_commit` to `on`, you may also want to explicitly set it, in which case the above block becomes:

```
spec:
  patroni:
    dynamicConfiguration:
      synchronous_mode: true
      postgresql:
        parameters:
          synchronous_commit: "on"
```

Note that Patroni, which manages many aspects of the cluster's availability, will favor availability over synchronicity. This means that if a synchronous replica goes down, Patroni will allow for asynchronous replication to continue as well as writes to the primary. However, if you want to disable all writing if there are no synchronous repliacs available, you would have to enable `synchronous_mode_strict`, i.e.:

```
spec:
  patroni:
    dynamicConfiguration:
      synchronous_mode: true
      synchronous_mode_strict: true
```

Affinity

Kubernetes affinity rules, which include Pod anti-affinity and Node affinity, can help you to define where you want your workloads to reside. Pod anti-affinity is important for high availability: when used correctly, it ensures that your Postgres instances are distributed amongst different Nodes. Node affinity can be used to assign instances to specific Nodes, e.g. to utilize hardware that's optimized for databases.

Understanding Pod Labels

PGO sets up several labels for Postgres cluster management that can be used for Pod anti-affinity or affinity rules in general. These include:

- `postgres-operator.crunchydata.com/cluster`: This is assigned to all managed Pods in a Postgres cluster. The value of this label is the name of your Postgres cluster, in this case: `hippo`.
- `postgres-operator.crunchydata.com/instance-set`: This is assigned to all Postgres instances within a group of `spec.instances`. In the example above, the value of this label is `instance1`. If you do not assign a label, the value is automatically set by PGO using a `NN` format, e.g. `00`.
- `postgres-operator.crunchydata.com/instance`: This is a unique label assigned to each Postgres instance containing the name of the Postgres instance.

Let's look at how we can set up affinity rules for our Postgres cluster to help improve high availability.

Pod Anti-affinity

Kubernetes has two types of Pod anti-affinity:

- Preferred: With preferred (`preferredDuringSchedulingIgnoredDuringExecution`) Pod anti-affinity, Kubernetes will make a best effort to schedule Pods matching the anti-affinity rules to different Nodes. However, if it is not possible to do so, then Kubernetes may schedule one or more Pods to the same Node.
- Required: With required (`requiredDuringSchedulingIgnoredDuringExecution`) Pod anti-affinity, Kubernetes mandates that each Pod matching the anti-affinity rules **must** be scheduled to different Nodes. However, a Pod may not be scheduled if Kubernetes cannot find a Node that does not contain a Pod matching the rules.

There is a tradeoff with these two types of pod anti-affinity: while "required" anti-affinity will ensure that all the matching Pods are scheduled on different Nodes, if Kubernetes cannot find an available Node, your Postgres instance may not be scheduled. Likewise, while "preferred" anti-affinity will make a best effort to scheduled your Pods on different Nodes, Kubernetes may compromise and schedule more than one Postgres instance of the same cluster on the same Node.

By understanding these tradeoffs, the makeup of your Kubernetes cluster, and your requirements, you can choose the method that makes the most sense for your Postgres deployment. We'll show examples of both methods below!

Using Preferred Pod Anti-Affinity First, let's deploy our Postgres cluster with preferred Pod anti-affinity. Note that if you have a single-node Kubernetes cluster, you will not see your Postgres instances deployed to different nodes. However, your Postgres instances *will* be deployed.

We can set up our HA Postgres cluster with preferred Pod anti-affinity like so:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 1
            podAffinityTerm:
              topologyKey: kubernetes.io/hostname
              labelSelector:
                matchLabels:
                  postgres-operator.crunchydata.com/cluster: hippo
                  postgres-operator.crunchydata.com/instance-set: instance1
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Apply those changes in your Kubernetes cluster.

Let's take a closer look at this section:

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 1
      podAffinityTerm:
        topologyKey: kubernetes.io/hostname
        labelSelector:
          matchLabels:
            postgres-operator.crunchydata.com/cluster: hippo
            postgres-operator.crunchydata.com/instance-set: instance1
```

`spec.instances.affinity.podAntiAffinity` follows the standard Kubernetes Pod anti-affinity spec. The values for the `matchLabels` are derived from what we described in the previous section: `postgres-operator.crunchydata.com/cluster` is set to our cluster name of `hippo`, and `postgres-operator.crunchydata.com/instance-set` is set to the instance set name of `instance1`. We choose a `topologyKey` of `kubernetes.io/hostname`, which is standard in Kubernetes clusters.

Preferred Pod anti-affinity will perform a best effort to schedule your Postgres Pods to different nodes. Let's see how you can require your Postgres Pods to be scheduled to different nodes.

Using Required Pod Anti-Affinity Required Pod anti-affinity forces Kubernetes to scheduled your Postgres Pods to different Nodes. Note that if Kubernetes is unable to schedule all Pods to different Nodes, some of your Postgres instances may become unavailable.

Using the previous example, let's indicate to Kubernetes that we want to use required Pod anti-affinity for our Postgres clusters:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - topologyKey: kubernetes.io/hostname
            labelSelector:
              matchLabels:
                postgres-operator.crunchydata.com/cluster: hippo
                postgres-operator.crunchydata.com/instance-set: instance1
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Apply those changes in your Kubernetes cluster.

If you are in a single Node Kubernetes clusters, you will notice that not all of your Postgres instance Pods will be scheduled. This is due to the `requiredDuringSchedulingIgnoredDuringExecution` preference. However, if you have enough Nodes available, you will see the Postgres instance Pods scheduled to different Nodes:

```
kubectl get pods -n postgres-operator -o wide \
  --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instan
```

Node Affinity

Node affinity can be used to assign your Postgres instances to Nodes with specific hardware or to guarantee a Postgres instance resides in a specific zone. Node affinity can be set within the `spec.instances.affinity.nodeAffinity` attribute, following the standard Kubernetes node affinity spec.

Let's see an example with required Node affinity. Let's say we have a set of Nodes that are reserved for database usage that have a label `workload-role=db`. We can create a Postgres cluster with a required Node affinity rule to scheduled all of the databases to those Nodes using the following configuration:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: workload-role
                operator: In
                values:
                - db
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Pod Topology Spread Constraints

In addition to affinity and anti-affinity settings, Kubernetes Pod Topology Spread Constraints can also help you to define where you want your workloads to reside. However, while PodAffinity allows any number of Pods to be added to a qualifying topology domain, and PodAntiAffinity allows only one Pod to be scheduled into a single topology domain, topology spread constraints allow you to distribute Pods across different topology domains with a finer level of control.

API Field Configuration

The spread constraint API fields can be configured for instance, pgBouncer and pgBackRest repo host pods. The basic configuration is as follows:

```
      topologySpreadConstraints:
      - maxSkew: <integer>
        topologyKey: <string>
        whenUnsatisfiable: <string>
        labelSelector: <object>
```

where "maxSkew" describes the maximum degree to which Pods can be unevenly distributed, "topologyKey" is the key that defines a topology in the Nodes' Labels, "whenUnsatisfiable" specifies what action should be taken when "maxSkew" can't be satisfied, and "labelSelector" is used to find matching Pods.

Example Spread Contraints

To help illustrate how you might use this with your cluster, we can review examples for configuring spread constraints on our Instance and pgBackRest repo host Pods. For this example, assume we have a three node Kubernetes cluster where the first node is labeled with `my-node-label=one`, the second node is labeled with `my-node-label=two` and the final node is labeled `my-node-label=three`. The label key `my-node-label` will function as our `topologyKey`. Note all three nodes in our examples will be schedulable, so a Pod could live on any of the three Nodes.

Instance Pod Spread Constraints To begin, we can set our topology spread contraints on our cluster Instance Pods. Given this configuration

```
  instances:
    - name: instance1
      replicas: 5
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: DoNotSchedule
          labelSelector:
            matchLabels:
              postgres-operator.crunchydata.com/instance-set: instance1
```

we will expect 5 Instance pods to be created. Each of these Pods will have the standard `postgres-operator.crunchydata.com/instance-se` `instance1` Label set, so each Pod will be properly counted when determining the `maxSkew`. Since we have 3 nodes with a `maxSkew` of 1 and we've set `whenUnsatisfiable` to `DoNotSchedule`, we should see 2 Pods on 2 of the nodes and 1 Pod on the remaining Node, thus ensuring our Pods are distributed as evenly as possible.

pgBackRest Repo Pod Spread Constraints We can also set topology spread constraints on our cluster's pgBackRest repo host pod. While we normally will only have a single pod per cluster, we could use a more generic label to add a preference that repo host Pods from different clusters are distributed among our Nodes. For example, by setting our `matchLabel` value to `postgres-operator.crunchydata.com/pgbackrest: ""` and our `whenUnsatisfiable` value to `ScheduleAnyway`, we will allow our repo host Pods to be scheduled no matter what Nodes may be available, but attempt to minimize skew as much as possible.

```
      repoHost:
        topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: ScheduleAnyway
          labelSelector:
            matchLabels:
              postgres-operator.crunchydata.com/pgbackrest: ""
```

Putting it All Together Now that each of our Pods has our desired Topology Spread Constraints defined, let's put together a complete cluster definition:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 5
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: DoNotSchedule
          labelSelector:
            matchLabels:
              postgres-operator.crunchydata.com/instance-set: instance1
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1G
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repoHost:
        topologySpreadConstraints:
```

```
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: ScheduleAnyway
          labelSelector:
            matchLabels:
              postgres-operator.crunchydata.com/pgbackrest: ""
    repos:
    - name: repo1
      volume:
        volumeClaimSpec:
          accessModes:
          - "ReadWriteOnce"
          resources:
            requests:
              storage: 1G
```

You can then apply those changes in your Kubernetes cluster.

Once your cluster finishes deploying, you can check that your Pods are assigned to the correct Nodes:

```
kubectl get pods -n postgres-operator -o wide
   --selector=postgres-operator.crunchydata.com/cluster=hippo
```

Next Steps

We've now seen how PGO helps your application stay "always on" with your Postgres database. Now let's explore how PGO can minimize or eliminate downtime for operations that would normally cause that, such as [resizing your Postgres cluster]({{< relref "./resize-cluster.md" >}}).

If you have not done so, please install PGO by following the [quickstart]({{< relref "quickstart/_index.md" >}}#installation).

As part of the installation, please be sure that you have done the following:

1. Forked the Postgres Operator examples repository and cloned it to your host machine.
2. Installed PGO to the `postgres-operator` namespace. If you are inside your `postgres-operator-examples` directory, you can run the `kubectl apply -k kustomize/install` command.

Throughout this tutorial, we will be building on the example provided in the `kustomize/postgres`.

When referring to a nested object within a YAML manifest, we will be using the `.` format similar to `kubectl explain`. For example, if we want to refer to the deepest element in this yaml file:

```
spec:
  hippos:
    appetite: huge
```

we would say `spec.hippos.appetite`.

`kubectl explain` is your friend. You can use `kubectl explain postgrescluster` to introspect the `postgrescluster.postgres-operator` custom resource definition. You can also review the [CRD reference]({{< relref "references/crd.md" >}}).

With PGO, the Postgres Operator installed, let's go and [create a Postgres cluster]({{< relref "./create-cluster.md" >}})!

If you came here through the [quickstart]({{< relref "quickstart/_index.md" >}}), you may have already created a cluster. If you created a cluster by using the example in the `kustomize/postgres` directory, feel free to skip to connecting to a cluster, or read onward for a more in depth look into cluster creation!

Create a Postgres Cluster

Creating a Postgres cluster is pretty simple. Using the example in the `kustomize/postgres` directory, all we have to do is run:

```
kubectl apply -k kustomize/postgres
```

and PGO will create a simple Postgres cluster named `hippo` in the `postgres-operator` namespace. You can track the status of your Postgres cluster using `kubectl describe` on the `postgresclusters.postgres-operator.crunchydata.com` custom resource:

```
kubectl -n postgres-operator describe postgresclusters.postgres-operator.crunchydata.com hippo
```

and you can track the state of the Postgres Pod using the following command:

```
kubectl -n postgres-operator get pods \
  --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instan
```

What Just Happened?

PGO created a Postgres cluster based on the information provided to it in the Kustomize manifests located in the `kustomize/postgres` directory. Let's better understand what happened by inspecting the `kustomize/postgres/postgres.yaml` file:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

When we ran the `kubectl apply` command earlier, what we did was create a `PostgresCluster` custom resource in Kubernetes. PGO detected that we added a new `PostgresCluster` resource and started to create all the objects needed to run Postgres in Kubernetes!

What else happened? PGO read the value from `metadata.name` to provide the Postgres cluster with the name `hippo`. Additionally, PGO knew which containers to use for Postgres and pgBackRest by looking at the values in `spec.image` and `spec.backups.pgbackrest.image` respectively. The value in `spec.postgresVersion` is important as it will help PGO track which major version of Postgres you are using.

PGO knows how many Postgres instances to create through the `spec.instances` section of the manifest. While `name` is optional, we opted to give it the name `instance1`. We could have also created multiple replicas and instances during cluster initialization, but we will cover that more when we discuss how to [scale and create a HA Postgres cluster]({{< relref "./high-availability.md" >}}).

A very important piece of your `PostgresCluster` custom resource is the `dataVolumeClaimSpec` section. This describes the storage that your Postgres instance will use. It is modeled after the Persistent Volume Claim. If you do not provide a `spec.instances.dataVolumeClaimSpec.storageClassName`, then the default storage class in your Kubernetes environment is used.

As part of creating a Postgres cluster, we also specify information about our backup archive. PGO uses pgBackRest, an open source backup and restore tool designed to handle terabyte-scale backups. As part of initializing our cluster, we can specify where we want our backups and archives (write-ahead logs or WAL) stored. We will talk about this portion of the `PostgresCluster` spec in greater depth in the [disaster recovery]({{< relref "./backups.md" >}}) section of this tutorial, and also see how we can store backups in Amazon S3, Google GCS, and Azure Blob Storage.

Troubleshooting

PostgreSQL / pgBackRest Pods Stuck in `Pending` Phase

The most common occurrence of this is due to PVCs not being bound. Ensure that you have set up your storage options correctly in any `volumeClaimSpec`. You can always update your settings and reapply your changes with `kubectl apply`.

Also ensure that you have enough persistent volumes available: your Kubernetes administrator may need to provision more.

If you are on OpenShift, you may need to set `spec.openshift` to `true`.

Backups Never Complete

The most common occurrence of this is due to the Kubernetes network blocking SSH connections between Pods. Ensure that your Kubernetes networking layer allows for SSH connections over port 2022 in the Namespace that you are deploying your PostgreSQL clusters into.

Next Steps

We're up and running – now let's [connect to our Postgres cluster]({{< relref "./connect-cluster.md" >}})!

It's one thing to [create a Postgres cluster]({{< relref "./create-cluster.md" >}}); it's another thing to connect to it. Let's explore how PGO makes it possible to connect to a Postgres cluster!

Background: Services, Secrets, and TLS

PGO creates a series of Kubernetes Services to provide stable endpoints for connecting to your Postgres databases. These endpoints make it easy to provide a consistent way for your application to maintain connectivity to your data. To inspect what services are available, you can run the following command:

```
kubectl -n postgres-operator get svc --selector=postgres-operator.crunchydata.com/cluster=hippo
```

will yield something similar to:

```
NAME                TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
hippo-ha            ClusterIP   10.103.73.92    <none>        5432/TCP   3h14m
hippo-ha-config     ClusterIP   None            <none>        <none>     3h14m
hippo-pods          ClusterIP   None            <none>        <none>     3h14m
hippo-primary       ClusterIP   None            <none>        5432/TCP   3h14m
hippo-replicas      ClusterIP   10.98.110.215   <none>        5432/TCP   3h14m
```

You do not need to worry about most of these Services, as they are used to help manage the overall health of your Postgres cluster. For the purposes of connecting to your database, the Service of interest is called `hippo-primary`. Thanks to PGO, you do not need to even worry about that, as that information is captured within a Secret!

When your Postgres cluster is initialized, PGO will bootstrap a database and Postgres user that your application can access. This information is stored in a Secret named with the pattern `<clusterName>-pguser-<userName>`. For our `hippo` cluster, this Secret is called `hippo-pguser-hippo`. This Secret contains the information you need to connect your application to your Postgres database:

- `user`: The name of the user account.
- `password`: The password for the user account.
- `dbname`: The name of the database that the user has access to by default.
- `host`: The name of the host of the database. This references the Service of the primary Postgres instance.
- `port`: The port that the database is listening on.
- `uri`: A PostgreSQL connection URI that provides all the information for logging into the Postgres database.
- `jdbc-uri`: A PostgreSQL JDBC connection URI that provides all the information for logging into the Postgres database via the JDBC driver.

All connections are over TLS. PGO provides its own certificate authority (CA) to allow you to securely connect your applications to your Postgres clusters. This allows you to use the `verify-full` "SSL mode" of Postgres, which provides eavesdropping protection and prevents MITM attacks. You can also choose to bring your own CA, which is described later in this tutorial in the [Customize Cluster]({{< relref "./customize-cluster.md" >}}) section.

Modifying Service Type

By default, PGO deploys Services with the `ClusterIP` Service type. Based on how you want to expose your database, you may want to modify the Services to use a different Service type.

You can modify the Services that PGO manages from the following attributes:

- `spec.service` - this manages the Service for connecting to a Postgres primary.

- `spec.proxy.pgBouncer.service` - this manages the Service for connecting to the PgBouncer connection pooler.

For example, to set the Postgres primary to use a `NodePort` service, you would add the following to your manifest:

```
spec:
  service:
    type: NodePort
```

For our `hippo` cluster, you would see the Service type modification in the . For example:

```
kubectl -n postgres-operator get svc --selector=postgres-operator.crunchydata.com/cluster=hippo
```

will yield something similar to:

```
NAME              TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
hippo-ha          NodePort    10.96.17.210   <none>        5432:32751/TCP   2m37s
hippo-ha-config   ClusterIP   None           <none>        <none>           2m37s
hippo-pods        ClusterIP   None           <none>        <none>           2m37s
hippo-primary     ClusterIP   None           <none>        5432/TCP         2m37s
hippo-replicas    ClusterIP   10.96.151.53   <none>        5432/TCP         2m37s
```

(Note that if you are exposing your Services externally and are relying on TLS verification, you will need to use the [custom TLS]({{< relref "tutorial/customize-cluster.md" >}}#customize-tls) features of PGO).

Connect an Application

For this tutorial, we are going to connect Keycloak, an open source identity management application. Keycloak can be deployed on Kubernetes and is backed by a Postgres database. While we provide an example of deploying Keycloak and a PostgresCluster in the Postgres Operator examples repository, the manifest below deploys it using our `hippo` cluster that is already running:

```
kubectl apply --filename=- <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: keycloak
  namespace: postgres-operator
  labels:
    app.kubernetes.io/name: keycloak
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: keycloak
  template:
    metadata:
      labels:
        app.kubernetes.io/name: keycloak
    spec:
      containers:
      - image: quay.io/keycloak/keycloak:latest
        name: keycloak
        env:
        - name: DB_VENDOR
          value: "postgres"
        - name: DB_ADDR
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: host } }
        - name: DB_PORT
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: port } }
        - name: DB_DATABASE
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: dbname } }
        - name: DB_USER
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: user } }
        - name: DB_PASSWORD
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: password } }
        - name: KEYCLOAK_USER
          value: "admin"
        - name: KEYCLOAK_PASSWORD
```

```
          value: "admin"
        - name: PROXY_ADDRESS_FORWARDING
          value: "true"
        ports:
        - name: http
          containerPort: 8080
        - name: https
          containerPort: 8443
        readinessProbe:
          httpGet:
            path: /auth/realms/master
            port: 8080
      restartPolicy: Always
EOF
```

Notice this part of the manifest:

```
- name: DB_ADDR
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: host } }
- name: DB_PORT
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: port } }
- name: DB_DATABASE
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: dbname } }
- name: DB_USER
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: user } }
- name: DB_PASSWORD
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: password } }
```

The above manifest shows how all of these values are derived from the `hippo-pguser-hippo` Secret. This means that we do not need to know any of the connection credentials or have to insecurely pass them around – they are made directly available to the application!

Using this method, you can tie application directly into your GitOps pipeline that connect to Postgres without any prior knowledge of how PGO will deploy Postgres: all of the information your application needs is propagated into the Secret!

Next Steps

Now that we have seen how to connect an application to a cluster, let's learn how to create a [high availability Postgres]({{< relref "./high-availability.md" >}}) cluster!

Postgres is known for its reliability: it is very stable and typically "just works." However, there are many things that can happen in a distributed environment like Kubernetes that can affect Postgres uptime, including:

- The database storage disk fails or some other hardware failure occurs
- The network on which the database resides becomes unreachable
- The host operating system becomes unstable and crashes
- A key database file becomes corrupted
- A data center is lost
- A Kubernetes component (e.g. a Service) is accidentally deleted

There may also be downtime events that are due to the normal case of operations, such as performing a minor upgrade, security patching of operating system, hardware upgrade, or other maintenance.

The good news: PGO is prepared for this, and your Postgres cluster is protected from many of these scenarios. However, to maximize your high availability (HA), let's first scale up your Postgres cluster.

HA Postgres: Adding Replicas to your Postgres Cluster

PGO provides several ways to add replicas to make a HA cluster:

- Increase the `spec.instances.replicas` value
- Add an additional entry in `spec.instances`

For the purposes of this tutorial, we will go with the first method and set `spec.instances.replicas` to `2`. Your manifest should look similar to:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Apply these updates to your Kubernetes cluster with the following command:

```
kubectl apply -k kustomize/postgres
```

Within moment, you should see a new Postgres instance initializing! You can see all of your Postgres Pods for the `hippo` cluster by running the following command:

```
kubectl -n postgres-operator get pods \
  --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instanc
```

Let's test our high availability set up.

Testing Your HA Cluster

An important part of building a resilient Postgres environment is testing its resiliency, so let's run a few tests to see how PGO performs under pressure!

Test #1: Remove a Service

Let's try removing the primary Service that our application is connected to. This test does not actually require a HA Postgres cluster, but it will demonstrate PGO's ability to react to environmental changes and heal things to ensure your applications can stay up.

Recall in the [connecting a Postgres cluster]({{< relref "./connect-cluster.md" >}}) that we observed the Services that PGO creates, e.g:

```
kubectl -n postgres-operator get svc \
  --selector=postgres-operator.crunchydata.com/cluster=hippo
```

yields something similar to:

```
NAME              TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
hippo-ha          ClusterIP   10.103.73.92   <none>        5432/TCP   4h8m
hippo-ha-config   ClusterIP   None           <none>        <none>     4h8m
hippo-pods        ClusterIP   None           <none>        <none>     4h8m
hippo-primary     ClusterIP   None           <none>        5432/TCP   4h8m
hippo-replicas    ClusterIP   10.98.110.215  <none>        5432/TCP   4h8m
```

We also mentioned that the application is connected to the `hippo-primary` Service. What happens if we were to delete this Service?

```
kubectl -n postgres-operator delete svc hippo-primary
```

This would seem like it could create a downtime scenario, but run the above selector again:

```
kubectl -n postgres-operator get svc \
  --selector=postgres-operator.crunchydata.com/cluster=hippo
```

You should see something similar to:

```
NAME              TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
hippo-ha          ClusterIP   10.103.73.92   <none>        5432/TCP   4h8m
hippo-ha-config   ClusterIP   None           <none>        <none>     4h8m
hippo-pods        ClusterIP   None           <none>        <none>     4h8m
hippo-primary     ClusterIP   None           <none>        5432/TCP   3s
hippo-replicas    ClusterIP   10.98.110.215  <none>        5432/TCP   4h8m
```

Wow – PGO detected that the primary Service was deleted and it recreated it! Based on how your application connects to Postgres, it may not have even noticed that this event took place!

Now let's try a more extreme downtime event.

Test #2: Remove the Primary StatefulSet

StatefulSets are a Kubernetes object that provide helpful mechanisms for managing Pods that interface with stateful applications, such as databases. They provide a stable mechanism for managing Pods to help ensure data is retrievable in a predictable way.

What happens if we remove the StatefulSet that is pointed to the Pod that represents the Postgres primary? First, let's determine which Pod is the primary. We'll store it in an environmental variable for convenience.

```
PRIMARY_POD=$(kubectl -n postgres-operator get pods \
  --selector=postgres-operator.crunchydata.com/role=master \
  -o jsonpath='{.items[*].metadata.labels.postgres-operator\.crunchydata\.com/instance}')
```

Inspect the environmental variable to see which Pod is the current primary:

```
echo $PRIMARY_POD
```

should yield something similar to:

```
hippo-instance1-zj5s
```

We can use the value above to delete the StatefulSet associated with the current Postgres primary instance:

```
kubectl delete sts -n postgres-operator "${PRIMARY_POD}"
```

Let's see what happens. Try getting all of the StatefulSets for the Postgres instances in the `hippo` cluster:

```
kubectl get sts -n postgres-operator \
  --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instan
```

You should see something similar to:

```
NAME                   READY   AGE
hippo-instance1-6kbw   1/1     15m
hippo-instance1-zj5s   0/1     1s
```

PGO recreated the StatefulSet that was deleted! After this "catastrophic" event, PGO proceeds to heal the Postgres instance so it can rejoin the cluster. We cover the high availability process in greater depth later in the documentation.

What about the other instance? We can see that it became the new primary though the following command:

```
kubectl -n postgres-operator get pods \
  --selector=postgres-operator.crunchydata.com/role=master \
  -o jsonpath='{.items[*].metadata.labels.postgres-operator\.crunchydata\.com/instance}'
```

which should yield something similar to:

```
hippo-instance1-6kbw
```

You can test that the failover successfully occurred in a few ways. You can connect to the example Keycloak application that we [deployed in the previous section]({{< relref "./connect-cluster.md" >}}). Based on Keycloak's connection retry logic, you may need to wait a moment for it to reconnect, but you will see it connected and resume being able to read and write data. You can also connect to the Postgres instance directly and execute the following command:

```
SELECT NOT pg_catalog.pg_is_in_recovery() is_primary;
```

If it returns `true` (or `t`), then the Postgres instance is a primary!

What if PGO was down during the downtime event? Failover would still occur: the Postgres HA system works independently of PGO and can maintain its own uptime. PGO will still need to assist with some of the healing aspects, but your application will still maintain read/write connectivity to your Postgres cluster!

Synchronous Replication

PostgreSQL supports synchronous replication, which is a replication mode designed to limit the risk of transaction loss. Synchronous replication waits for a transaction to be written to at least one additional server before it considers the transaction to be committed. For more information on synchronous replication, please read about PGO's [high availability architecture]({{}}#synchronous-replication-guarding-against-transactions-loss)

To add synchronous replication to your Postgres cluster, you can add the following to your spec:

```
spec:
  patroni:
    dynamicConfiguration:
      synchronous_mode: true
```

While PostgreSQL defaults `synchronous_commit` to `on`, you may also want to explicitly set it, in which case the above block becomes:

```
spec:
  patroni:
    dynamicConfiguration:
      synchronous_mode: true
      postgresql:
        parameters:
          synchronous_commit: "on"
```

Note that Patroni, which manages many aspects of the cluster's availability, will favor availability over synchronicity. This means that if a synchronous replica goes down, Patroni will allow for asynchronous replication to continue as well as writes to the primary. However, if you want to disable all writing if there are no synchronous repliacs available, you would have to enable `synchronous_mode_strict`, i.e.:

```
spec:
  patroni:
    dynamicConfiguration:
      synchronous_mode: true
      synchronous_mode_strict: true
```

Affinity

Kubernetes affinity rules, which include Pod anti-affinity and Node affinity, can help you to define where you want your workloads to reside. Pod anti-affinity is important for high availability: when used correctly, it ensures that your Postgres instances are distributed amongst different Nodes. Node affinity can be used to assign instances to specific Nodes, e.g. to utilize hardware that's optimized for databases.

Understanding Pod Labels

PGO sets up several labels for Postgres cluster management that can be used for Pod anti-affinity or affinity rules in general. These include:

- `postgres-operator.crunchydata.com/cluster`: This is assigned to all managed Pods in a Postgres cluster. The value of this label is the name of your Postgres cluster, in this case: `hippo`.
- `postgres-operator.crunchydata.com/instance-set`: This is assigned to all Postgres instances within a group of `spec.instances`. In the example above, the value of this label is `instance1`. If you do not assign a label, the value is automatically set by PGO using a `NN` format, e.g. `00`.
- `postgres-operator.crunchydata.com/instance`: This is a unique label assigned to each Postgres instance containing the name of the Postgres instance.

Let's look at how we can set up affinity rules for our Postgres cluster to help improve high availability.

Pod Anti-affinity

Kubernetes has two types of Pod anti-affinity:

- Preferred: With preferred (`preferredDuringSchedulingIgnoredDuringExecution`) Pod anti-affinity, Kubernetes will make a best effort to schedule Pods matching the anti-affinity rules to different Nodes. However, if it is not possible to do so, then Kubernetes may schedule one or more Pods to the same Node.
- Required: With required (`requiredDuringSchedulingIgnoredDuringExecution`) Pod anti-affinity, Kubernetes mandates that each Pod matching the anti-affinity rules **must** be scheduled to different Nodes. However, a Pod may not be scheduled if Kubernetes cannot find a Node that does not contain a Pod matching the rules.

There is a tradeoff with these two types of pod anti-affinity: while "required" anti-affinity will ensure that all the matching Pods are scheduled on different Nodes, if Kubernetes cannot find an available Node, your Postgres instance may not be scheduled. Likewise, while "preferred" anti-affinity will make a best effort to scheduled your Pods on different Nodes, Kubernetes may compromise and schedule more than one Postgres instance of the same cluster on the same Node.

By understanding these tradeoffs, the makeup of your Kubernetes cluster, and your requirements, you can choose the method that makes the most sense for your Postgres deployment. We'll show examples of both methods below!

Using Preferred Pod Anti-Affinity First, let's deploy our Postgres cluster with preferred Pod anti-affinity. Note that if you have a single-node Kubernetes cluster, you will not see your Postgres instances deployed to different nodes. However, your Postgres instances *will* be deployed.

We can set up our HA Postgres cluster with preferred Pod anti-affinity like so:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 1
            podAffinityTerm:
              topologyKey: kubernetes.io/hostname
              labelSelector:
                matchLabels:
                  postgres-operator.crunchydata.com/cluster: hippo
                  postgres-operator.crunchydata.com/instance-set: instance1
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Apply those changes in your Kubernetes cluster.

Let's take a closer look at this section:

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 1
      podAffinityTerm:
        topologyKey: kubernetes.io/hostname
        labelSelector:
          matchLabels:
            postgres-operator.crunchydata.com/cluster: hippo
            postgres-operator.crunchydata.com/instance-set: instance1
```

`spec.instances.affinity.podAntiAffinity` follows the standard Kubernetes Pod anti-affinity spec. The values for the `matchLabels` are derived from what we described in the previous section: `postgres-operator.crunchydata.com/cluster` is set to our cluster name of `hippo`, and `postgres-operator.crunchydata.com/instance-set` is set to the instance set name of `instance1`. We choose a `topologyKey` of `kubernetes.io/hostname`, which is standard in Kubernetes clusters.

Preferred Pod anti-affinity will perform a best effort to schedule your Postgres Pods to different nodes. Let's see how you can require your Postgres Pods to be scheduled to different nodes.

Using Required Pod Anti-Affinity Required Pod anti-affinity forces Kubernetes to scheduled your Postgres Pods to different Nodes. Note that if Kubernetes is unable to schedule all Pods to different Nodes, some of your Postgres instances may become unavailable.

Using the previous example, let's indicate to Kubernetes that we want to use required Pod anti-affinity for our Postgres clusters:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - topologyKey: kubernetes.io/hostname
            labelSelector:
              matchLabels:
                postgres-operator.crunchydata.com/cluster: hippo
                postgres-operator.crunchydata.com/instance-set: instance1
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Apply those changes in your Kubernetes cluster.

If you are in a single Node Kubernetes clusters, you will notice that not all of your Postgres instance Pods will be scheduled. This is due to the `requiredDuringSchedulingIgnoredDuringExecution` preference. However, if you have enough Nodes available, you will see the Postgres instance Pods scheduled to different Nodes:

```
kubectl get pods -n postgres-operator -o wide \
  --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instanc
```

Node Affinity

Node affinity can be used to assign your Postgres instances to Nodes with specific hardware or to guarantee a Postgres instance resides in a specific zone. Node affinity can be set within the `spec.instances.affinity.nodeAffinity` attribute, following the standard Kubernetes node affinity spec.

Let's see an example with required Node affinity. Let's say we have a set of Nodes that are reserved for database usage that have a label `workload-role=db`. We can create a Postgres cluster with a required Node affinity rule to scheduled all of the databases to those Nodes using the following configuration:

```yaml
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: workload-role
                operator: In
                values:
                - db
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Pod Topology Spread Constraints

In addition to affinity and anti-affinity settings, Kubernetes Pod Topology Spread Constraints can also help you to define where you want your workloads to reside. However, while PodAffinity allows any number of Pods to be added to a qualifying topology domain, and PodAntiAffinity allows only one Pod to be scheduled into a single topology domain, topology spread constraints allow you to distribute Pods across different topology domains with a finer level of control.

API Field Configuration

The spread constraint API fields can be configured for instance, pgBouncer and pgBackRest repo host pods. The basic configuration is as follows:

```
      topologySpreadConstraints:
      - maxSkew: <integer>
        topologyKey: <string>
        whenUnsatisfiable: <string>
        labelSelector: <object>
```

where "maxSkew" describes the maximum degree to which Pods can be unevenly distributed, "topologyKey" is the key that defines a topology in the Nodes' Labels, "whenUnsatisfiable" specifies what action should be taken when "maxSkew" can't be satisfied, and "labelSelector" is used to find matching Pods.

Example Spread Contraints

To help illustrate how you might use this with your cluster, we can review examples for configuring spread constraints on our Instance and pgBackRest repo host Pods. For this example, assume we have a three node Kubernetes cluster where the first node is labeled with `my-node-label=one`, the second node is labeled with `my-node-label=two` and the final node is labeled `my-node-label=three`. The label key `my-node-label` will function as our `topologyKey`. Note all three nodes in our examples will be schedulable, so a Pod could live on any of the three Nodes.

Instance Pod Spread Constraints To begin, we can set our topology spread contraints on our cluster Instance Pods. Given this configuration

```
  instances:
    - name: instance1
      replicas: 5
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: DoNotSchedule
          labelSelector:
            matchLabels:
              postgres-operator.crunchydata.com/instance-set: instance1
```

we will expect 5 Instance pods to be created. Each of these Pods will have the standard `postgres-operator.crunchydata.com/instance-se` `instance1` Label set, so each Pod will be properly counted when determining the `maxSkew`. Since we have 3 nodes with a `maxSkew` of 1 and we've set `whenUnsatisfiable` to `DoNotSchedule`, we should see 2 Pods on 2 of the nodes and 1 Pod on the remaining Node, thus ensuring our Pods are distributed as evenly as possible.

pgBackRest Repo Pod Spread Constraints We can also set topology spread constraints on our cluster's pgBackRest repo host pod. While we normally will only have a single pod per cluster, we could use a more generic label to add a preference that repo host Pods from different clusters are distributed among our Nodes. For example, by setting our `matchLabel` value to `postgres-operator.crunchydata.com/pgbackrest: ""` and our `whenUnsatisfiable` value to `ScheduleAnyway`, we will allow our repo host Pods to be scheduled no matter what Nodes may be available, but attempt to minimize skew as much as possible.

```
      repoHost:
        topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: ScheduleAnyway
          labelSelector:
            matchLabels:
              postgres-operator.crunchydata.com/pgbackrest: ""
```

Putting it All Together Now that each of our Pods has our desired Topology Spread Constraints defined, let's put together a complete cluster definition:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 5
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: DoNotSchedule
          labelSelector:
            matchLabels:
              postgres-operator.crunchydata.com/instance-set: instance1
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1G
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repoHost:
        topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: ScheduleAnyway
          labelSelector:
            matchLabels:
              postgres-operator.crunchydata.com/pgbackrest: ""
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1G
```

You can then apply those changes in your Kubernetes cluster.

Once your cluster finishes deploying, you can check that your Pods are assigned to the correct Nodes:

```
kubectl get pods -n postgres-operator -o wide
   --selector=postgres-operator.crunchydata.com/cluster=hippo
```

Next Steps

We've now seen how PGO helps your application stay "always on" with your Postgres database. Now let's explore how PGO can minimize or eliminate downtime for operations that would normally cause that, such as [resizing your Postgres cluster]({{< relref "./resize-cluster.md" >}}).

You did it – the application is a success! Traffic is booming, so much so that you need to add more resources to your Postgres cluster. However, you're worried that any resize operation may cause downtime and create a poor experience for your end users.

This is where PGO comes in: PGO will help orchestrate rolling out any potentially disruptive changes to your cluster to minimize or eliminate and downtime for your application. To do so, we will assume that you have [deployed a high availability Postgres cluster]({{< relref "./high-availability.md" >}}) as described in the [previous section]({{< relref "./high-availability.md" >}}).

Let's dive in.

Resize Memory and CPU

Memory and CPU resources are an important component for vertically scaling your Postgres cluster. Couple with [tweaks to your Postgres configuration file]({{< relref "./customize-cluster.md" >}}), allowing your cluster to have more memory and CPU allotted to it can help it to perform better under load.

It's important for instances in the same high availability set to have the same resources. PGO lets you adjust CPU and memory within the `resources` sections of the `postgresclusters.postgres-operator.crunchydata.com` custom resource. These include:

- `spec.instances.resources` section, which sets the resource values for the PostgreSQL container, as well as any init containers in the associated pod and containers created by the `pgDataVolume` and `pgWALVolume` [data migration jobs]({{< relref "guides/data-migration.md" >}}).
- `spec.instances.sidecars.replicacertcopy.resources` section, which sets the resources for the `replica-cert-copy` sidecar container.
- `spec.monitoring.pgmonitor.exporter.resources` section, which sets the resources for the `exporter` sidecar container.
- `spec.backups.pgbackrest.repoHost.resources` section, which sets the resources for the pgBackRest repo host container, as well as any init containers in the associated pod and containers created by the `pgBackRestVolume` [data migration job]({{< relref "guides/data-migration.md" >}}).
- `spec.backups.pgbackrest.sidecars.pgbackrest.resources` section, which sets the resources for the `pgbackrest` sidecar container.
- `spec.backups.pgbackrest.jobs.resources` section, which sets the resources for any pgBackRest backup job.
- `spec.backups.pgbackrest.restore.resources` section, which sets the resources for manual pgBackRest restore jobs.
- `spec.dataSource.postgresCluster.resources` section, which sets the resources for pgBackRest restore jobs created during the [cloning]({{< relref "./disaster-recovery.md" >}}) process.
- `spec.proxy.pgBouncer.resources` section, which sets the resources for the `pgbouncer` container.
- `spec.proxy.pgBouncer.sidecars.pgbouncerconfig.resources` section, which sets the resources for the `pgbouncer-config` sidecar container.

The layout of these `resources` sections should be familiar: they follow the same pattern as the standard Kubernetes structure for setting container resources. Note that these settings also allow for the configuration of QoS classes.

For example, using the `spec.instances.resources` section, let's say we want to update our `hippo` Postgres cluster so that each instance has a limit of `2.0` CPUs and `4Gi` of memory. We can make the following changes to the manifest:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

In particular, we added the following to `spec.instances`:

```
resources:
  limits:
    cpu: 2.0
    memory: 4Gi
```

Apply these updates to your Kubernetes cluster with the following command:

```
kubectl apply -k kustomize/postgres
```

Now, let's watch how the rollout happens:

```
watch "kubectl -n postgres-operator get pods \
  --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instan
     \
  -o=jsonpath='{range
     .items[*]}{.metadata.name}{\"\t\"}{.metadata.labels.postgres-operator\.crunchydata\.com/role}{\"
```

Observe how each Pod is terminated one-at-a-time. This is part of a "rolling update". Because updating the resources of a Pod is a destructive action, PGO first applies the CPU and memory changes to the replicas. PGO ensures that the changes are successfully applied to a replica instance before moving on to the next replica.

Once all of the changes are applied, PGO will perform a "controlled switchover": it will promote a replica to become a primary, and apply the changes to the final Postgres instance.

By rolling out the changes in this way, PGO ensures there is minimal to zero disruption to your application: you are able to successfully roll out updates and your users may not even notice!

Resize PVC

Your application is a success! Your data continues to grow, and it's becoming apparently that you need more disk. That's great: you can resize your PVC directly on your `postgresclusters.postgres-operator.crunchydata.com` custom resource with minimal to zero downtime.

PVC resizing, also known as volume expansion, is a function of your storage class: it must support volume resizing. Additionally, PVCs can only be **sized up**: you cannot shrink the size of a PVC.

You can adjust PVC sizes on all of the managed storage instances in a Postgres instance that are using Kubernetes storage. These include:

- `spec.instances.dataVolumeClaimSpec.resources.requests.storage`: The Postgres data directory (aka your database).
- `spec.backups.pgbackrest.repos.volume.volumeClaimSpec.resources.requests.storage`: The pgBackRest repository when using "volume" storage

The above should be familiar: it follows the same pattern as the standard Kubernetes PVC structure.

For example, let's say we want to update our `hippo` Postgres cluster so that each instance now uses a `10Gi` PVC and our backup repository uses a `20Gi` PVC. We can do so with the following markup:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 10Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
```

```
          accessModes:
          - "ReadWriteOnce"
          resources:
            requests:
              storage: 20Gi
```

In particular, we added the following to `spec.instances`:

```
dataVolumeClaimSpec:
  resources:
    requests:
      storage: 10Gi
```

and added the following to `spec.backups.pgbackrest.repos.volume`:

```
volumeClaimSpec:
  accessModes:
  - "ReadWriteOnce"
  resources:
    requests:
      storage: 20Gi
```

Apply these updates to your Kubernetes cluster with the following command:

```
kubectl apply -k kustomize/postgres
```

Resize PVCs With StorageClass That Does Not Allow Expansion

Not all Kubernetes Storage Classes allow for volume expansion. However, with PGO, you can still resize your Postgres cluster data volumes even if your storage class does not allow it!

Let's go back to the previous example:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 20Gi
```

First, create a new instance that has the larger volume size. Call this instance `instance2`. The manifest would look like this:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
    - name: instance2
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 10Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 20Gi
```

Take note of the block that contains `instance2`:

```
- name: instance2
  replicas: 2
  resources:
    limits:
      cpu: 2.0
      memory: 4Gi
  dataVolumeClaimSpec:
    accessModes:
    - "ReadWriteOnce"
    resources:
      requests:
        storage: 10Gi
```

This creates a second set of two Postgres instances, both of which come up as replicas, that have a larger PVC.

Once this new instance set is available and they are caught to the primary, you can then apply the following manifest:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance2
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 10Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 20Gi
```

This will promote one of the instances with the larger PVC to be the new primary and remove the instances with the smaller PVCs!

This method can also be used to shrink PVCs to use a smaller amount.

Troubleshooting

Postgres Pod Can't Be Scheduled

There are many reasons why a PostgreSQL Pod may not be scheduled:

- **Resources are unavailable**. Ensure that you have a Kubernetes Node with enough resources to satisfy your memory or CPU Request.
- **PVC cannot be provisioned**. Ensure that you request a PVC size that is available, or that your PVC storage class is set up correctly.

PVCs Do Not Resize

Ensure that your storage class supports PVC resizing. You can check that by inspecting the `allowVolumeExpansion` attribute:

```
kubectl get sc
```

If the storage class does not support PVC resizing, you can use the technique described above to resize PVCs using a second instance set.

Next Steps

You've now resized your Postgres cluster, but how can you configure Postgres to take advantage of the new resources? Let's look at how we can [customize the Postgres cluster configuration]({{< relref "./customize-cluster.md" >}}).

Postgres is known for its ease of customization; PGO helps you to roll out changes efficiently and without disruption. After [resizing the resources]({{< relref "./resize-cluster.md" >}}) for our Postgres cluster in the previous step of this tutorial, lets see how we can tweak our Postgres configuration to optimize its usage of them.

Custom Postgres Configuration

Part of the trick of managing multiple instances in a Postgres cluster is ensuring all of the configuration changes are propagated to each of them. This is where PGO helps: when you make a Postgres configuration change for a cluster, PGO will apply the changes to all of the managed instances.

For example, in our previous step we added CPU and memory limits of `2.0` and `4Gi` respectively. Let's tweak some of the Postgres settings to better use our new resources. We can do this in the `spec.patroni.dynamicConfiguration` section. Here is an example updated manifest that tweaks several settings:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
  patroni:
    dynamicConfiguration:
      postgresql:
        parameters:
          max_parallel_workers: 2
          max_worker_processes: 2
          shared_buffers: 1GB
          work_mem: 2MB
```

In particular, we added the following to `spec`:

```
patroni:
  dynamicConfiguration:
    postgresql:
      parameters:
        max_parallel_workers: 2
        max_worker_processes: 2
        shared_buffers: 1GB
        work_mem: 2MB
```

Apply these updates to your Kubernetes cluster with the following command:

```
kubectl apply -k kustomize/postgres
```

PGO will go and apply these settings to all of the Postgres clusters. You can verify that the changes are present using the Postgres `SHOW` command, e.g.

```
SHOW work_mem;
```

should yield something similar to:

```
 work_mem
----------
 2MB
```

Customize TLS

All connections in PGO use TLS to encrypt communication between components. PGO sets up a PKI and certificate authority (CA) that allow you create verifiable endpoints. However, you may want to bring a different TLS infrastructure based upon your organizational requirements. The good news: PGO lets you do this!

If you want to use the TLS infrastructure that PGO provides, you can skip the rest of this section and move on to learning how to [apply software updates]({{< relref "./update-cluster.md" >}}).

How to Customize TLS

There are a few different TLS endpoints that can be customized for PGO, including those of the Postgres cluster and controlling how Postgres instances authenticate with each other. Let's look at how we can customize TLS.

Your TLS certificate should have a Common Name (CN) setting that matches the primary Service name. This is the name of the cluster suffixed with `-primary`. For example, for our `hippo` cluster this would be `hippo-primary`.

To customize the TLS for a Postgres cluster, you will need to create a Secret in the Namespace of your Postgres cluster that contains the TLS key (`tls.key`), TLS certificate (`tls.crt`) and the CA certificate (`ca.crt`) to use. The Secret should contain the following values:

```
data:
  ca.crt: <value>
  tls.crt: <value>
  tls.key: <value>
```

For example, if you have files named `ca.crt`, `hippo.key`, and `hippo.crt` stored on your local machine, you could run the following command:

```
kubectl create secret generic -n postgres-operator hippo.tls \
  --from-file=ca.crt=ca.crt \
  --from-file=tls.key=hippo.key \
  --from-file=tls.crt=hippo.crt
```

You can specify the custom TLS Secret in the `spec.customTLSSecret.name` field in your `postgrescluster.postgres-operator.crunchyda` custom resource, e.g:

```
spec:
  customTLSSecret:
    name: hippo.tls
```

If you're unable to control the key-value pairs in the Secret, you can create a mapping that looks similar to this:

```
spec:
  customTLSSecret:
    name: hippo.tls
    items:
      - key: <tls.crt key>
        path: tls.crt
      - key: <tls.key key>
        path: tls.key
      - key: <ca.crt key>
        path: ca.crt
```

If `spec.customTLSSecret` is provided you **must** also provide `spec.customReplicationTLSSecret` and both must contain the same `ca.crt`.

As with the other changes, you can roll out the TLS customizations with `kubectl apply`.

Labels

There are several ways to add your own custom Kubernetes Labels to your Postgres cluster.

- Cluster: You can apply labels to any PGO managed object in a cluster by editing the `spec.metadata.labels` section of the custom resource.
- Postgres: You can apply labels to a Postgres instance set and its objects by editing `spec.instances.metadata.labels`.
- pgBackRest: You can apply labels to pgBackRest and its objects by editing `postgresclusters.spec.backups.pgbackrest.metadata.`
- PgBouncer: You can apply labels to PgBouncer connection pooling instances by editing `spec.proxy.pgBouncer.metadata.labels`.

Annotations

There are several ways to add your own custom Kubernetes Annotations to your Postgres cluster.

- Cluster: You can apply annotations to any PGO managed object in a cluster by editing the `spec.metadata.annotations` section of the custom resource.
- Postgres: You can apply annotations to a Postgres instance set and its objects by editing `spec.instances.metadata.annotations`.
- pgBackRest: You can apply annotations to pgBackRest and its objects by editing `spec.backups.pgbackrest.metadata.annotations.`
- PgBouncer: You can apply annotations to PgBouncer connection pooling instances by editing `spec.proxy.pgBouncer.metadata.anno`

Pod Priority Classes

PGO allows you to use pod priority classes to indicate the relative importance of a pod by setting a `priorityClassName` field on your Postgres cluster. This can be done as follows:

- Instances: Priority is defined per instance set and is applied to all Pods in that instance set by editing the `spec.instances.priorityCla` section of the custom resource.
- Dedicated Repo Host: Priority defined under the repoHost section of the spec is applied to the dedicated repo host by editing the `spec.backups.pgbackrest.repoHost.priorityClassName` section of the custom resource.
- PgBouncer: Priority is defined under the pgBouncer section of the spec and will apply to all PgBouncer Pods by editing the `spec.proxy.pgBouncer.priorityClassName` section of the custom resource.
- Backup (manual and scheduled): Priority is defined under the `spec.backups.pgbackrest.jobs.priorityClassName` section and applies that priority to all pgBackRest backup Jobs (manual and scheduled).
- Restore (data source or in-place): Priority is defined for either a "data source" restore or an in-place restore by editing the `spec.dataSource.postgresCluster.priorityClassName` section of the custom resource.
- Data Migration: The priority defined for the first instance set in the spec (array position 0) is used for the PGDATA and WAL migration Jobs. The pgBackRest repo migration Job will use the priority class applied to the repoHost.

Separate WAL PVCs

PostgreSQL commits transactions by storing changes in its Write-Ahead Log (WAL). Because the way WAL files are accessed and utilized often differs from that of data files, and in high-performance situations, it can desirable to put WAL files on separate storage volume. With PGO, this can be done by adding the `walVolumeClaimSpec` block to your desired instance in your PostgresCluster spec, either when your cluster is created or anytime thereafter:

```
spec:
  instances:
    - name: instance
      walVolumeClaimSpec:
        accessModes:
        - "ReadWriteMany"
        resources:
          requests:
            storage: 1Gi
```

This volume can be removed later by removing the `walVolumeClaimSpec` section from the instance. Note that when changing the WAL directory, care is taken so as not to lose any WAL files. PGO only deletes the PVC once there are no longer any WAL files on the previously configured volume.

Database Initialization SQL

PGO can run SQL for you as part of the cluster creation and initialization process. PGO runs the SQL using the psql client so you can use meta-commands to connect to different databases, change error handling, or set and use variables. Its capabilities are described in the psql documentation.

Initialization SQL ConfigMap

The Postgres cluster spec accepts a reference to a ConfigMap containing your init SQL file. Update your cluster spec to include the ConfigMap name, `spec.databaseInitSQL.name`, and the data key, `spec.databaseInitSQL.key`, for your SQL file. For example, if you create your ConfigMap with the following command:

```
kubectl -n postgres-operator create configmap hippo-init-sql --from-file=init.sql=/path/to/init.sql
```

You would add the following section to your Postgrescluster spec:

```
spec:
  databaseInitSQL:
    key: init.sql
    name: hippo-init-sql
```

The ConfigMap must exist in the same namespace as your Postgres cluster.

After you add the ConfigMap reference to your spec, apply the change with `kubectl apply -k kustomize/postgres`. PGO will create your `hippo` cluster and run your initialization SQL once the cluster has started. You can verify that your SQL has been run by checking the `databaseInitSQL` status on your Postgres cluster. While the status is set, your init SQL will not be run again. You can check cluster status with the `kubectl describe` command:

```
kubectl -n postgres-operator describe postgresclusters.postgres-operator.crunchydata.com hippo
```

In some cases, due to how Kubernetes treats PostgresCluster status, PGO may run your SQL commands more than once. Please ensure that the commands defined in your init SQL are idempotent.

Now that `databaseInitSQL` is defined in your cluster status, verify database objects have been created as expected. After verifying, we recommend removing the `spec.databaseInitSQL` field from your spec. Removing the field from the spec will also remove `databaseInitSQL` from the cluster status.

PSQL Usage

PGO uses the psql interactive terminal to execute SQL statements in your database. Statements are passed in using standard input and the filename flag (e.g. `psql -f -`).

SQL statements are executed as superuser in the default maintenance database. This means you have full control to create database objects, extensions, or run any SQL statements that you might need.

Integration with User and Database Management If you are creating users or databases, please see the [User/Database Management]({{< relref "tutorial/user-management.md" >}}) documentation. Databases created through the user management section of the spec can be referenced in your initialization sql. For example, if a database `zoo` is defined:

```
spec:
  users:
    - name: hippo
      databases:
       - "zoo"
```

You can connect to `zoo` by adding the following `psql` meta-command to your SQL:

```
\c zoo
create table t_zoo as select s, md5(random()::text) from generate_Series(1,5) s;
```

Transaction support By default, `psql` commits each SQL command as it completes. To combine multiple commands into a single transaction, use the `BEGIN` and `COMMIT` commands.

```
BEGIN;
create table t_random as select s, md5(random()::text) from generate_Series(1,5) s;
COMMIT;
```

PSQL Exit Code and Database Init SQL Status The exit code from psql will determine when the databaseInitSQL status is set. When psql returns 0 the status will be set and SQL will not be run again. When psql returns with an error exit code the status will not be set. PGO will continue attempting to execute the SQL as part of its reconcile loop until psql returns normally. If psql exits with a failure, you will need to edit the file in your ConfigMap to ensure your SQL statements will lead to a successful psql return. The easiest way to make live changes to your ConfigMap is to use the following kubectl edit command:

```
kubectl -n <cluster-namespace> edit configmap hippo-init-sql
```

Be sure to transfer any changes back over to your local file. Another option is to make changes in your local file and use kubectl --dry-run to create a template and pipe the output into kubectl apply:

```
kubectl create configmap hippo-init-sql --from-file=init.sql=/path/to/init.sql --dry-run=client -o
   yaml | kubectl apply -f -
```

If you edit your ConfigMap and your changes aren't showing up, you may be waiting for PGO to reconcile your cluster. After some time, PGO will automatically reconcile the cluster or you can trigger reconciliation by applying any change to your cluster (e.g. with kubectl apply -k kustomize/postgres).

To ensure that psql returns a failure exit code when your SQL commands fail, set the ON_ERROR_STOP variable as part of your SQL file:

```
\set ON_ERROR_STOP
\echo Any error will lead to exit code 3
create table t_random as select s, md5(random()::text) from generate_Series(1,5) s;
```

Troubleshooting

Changes Not Applied

If your Postgres configuration settings are not present, you may need to check a few things. First, ensure that you are using the syntax that Postgres expects. You can see this in the Postgres configuration documentation.

Some settings, such as shared_buffers, require for Postgres to restart. Patroni only performs a reload when parameter changes are identified. Therefore, for parameters that require a restart, the restart can be performed manually by executing into a Postgres instance and running patronictl restart --force <clusterName>-ha.

Next Steps

You've now seen how you can further customize your Postgres cluster, but what about [managing users and atabases]({{< relref "./user-management.md" >}})? That's a great question that is answered in the [next section]({{< relref "./user-management.md" >}}).

PGO comes with some out-of-the-box conveniences for managing users and databases in your Postgres cluster. However, you may have requirements where you need to create additional users, adjust user privileges or add additional databases to your cluster.

For detailed information for how user and database management works in PGO, please see the [User Management]({{< relref "architecture/user-management.md" >}}) section of the architecture guide.

Creating a New User

You can create a new user with the following snippet in the postgrescluster custom resource. Let's add this to our hippo database:

```
spec:
  users:
    - name: rhino
```

You can now apply the changes and see that the new user is created. Note the following:

- The user would only be able to connect to the default postgres database.
- The user will not have any connection credentials populated into the hippo-pguser-rhino Secret.
- The user is unprivileged.

Let's create a new database named zoo that we will let the rhino user access:

```
spec:
  users:
    - name: rhino
      databases:
        - zoo
```

Inspect the `hippo-pguser-rhino` Secret. You should now see that the `dbname` and `uri` fields are now populated!

We can set role privileges by using the standard role attributes that Postgres provides and adding them to the `spec.users.options`. Let's say we want the rhino to become a superuser (be careful about doling out Postgres superuser privileges!). You can add the following to the spec:

```
spec:
  users:
    - name: rhino
      databases:
        - zoo
      options: "SUPERUSER"
```

There you have it: we have created a Postgres user named `rhino` with superuser privileges that has access to the `rhino` database (though a superuser has access to all databases!).

Adjusting Privileges

Let's say you want to revoke the superuser privilege from `rhino`. You can do so with the following:

```
spec:
  users:
    - name: rhino
      databases:
        - zoo
      options: "NOSUPERUSER"
```

If you want to add multiple privileges, you can add each privilege with a space between them in `options`, e.g:

```
spec:
  users:
    - name: rhino
      databases:
        - zoo
      options: "CREATEDB CREATEROLE"
```

Managing the `postgres` User

By default, PGO does not give you access to the `postgres` user. However, you can get access to this account by doing the following:

```
spec:
  users:
    - name: postgres
```

This will create a Secret of the pattern `<clusterName>-pguser-postgres` that contains the credentials of the `postgres` account. For our `hippo` cluster, this would be `hippo-pguser-postgres`.

Deleting a User

As mentioned earlier, PGO does not let you delete a user automatically: if you remove the user from the spec, it will still exist in your cluster. To remove a user and all of its objects, as a superuser you will need to run `DROP OWNED` in each database the user has objects in, and `DROP ROLE` in your Postgres cluster.

For example, with the above `rhino` user, you would run the following:

```
DROP OWNED BY rhino;
DROP ROLE rhino;
```

Note that you may need to run `DROP OWNED BY rhino CASCADE;` based upon your object ownership structure – be very careful with this command!

Once you have removed the user in the database, you can remove the user from the custom resource.

Deleting a Database

As mentioned earlier, PGO does not let you delete a database automatically: if you remove all instances of the database from the spec, it will still exist in your cluster. To completely remove the database, you must run the `DROP DATABASE` command as a Postgres superuser.

For example, to remove the `zoo` database, you would execute the following:

```
DROP DATABASE zoo;
```

Once you have removed the database, you can remove any references to the database from the custom resource.

Next Steps

You now know how to manage users and databases in your cluster and have now a well-rounded set of tools to support your "Day 1" operations. Let's start looking at some of the "Day 2" work you can do with PGO, such as [updating to the next Postgres version]({{< relref "./update-cluster.md" >}}), in the [next section]({{< relref "./update-cluster.md" >}}).

Did you know that Postgres releases bug fixes once every three months? Additionally, we periodically refresh the container images to ensure the base images have the latest software that may fix some CVEs.

It's generally good practice to keep your software up-to-date for stability and security purposes, so let's learn how PGO helps to you accept low risk, "patch" type updates.

The good news: you do not need to update PGO itself to apply component updates: you can update each Postgres cluster whenever you want to apply the update! This lets you choose when you want to apply updates to each of your Postgres clusters, so you can update it on your own schedule. If you have a [high availability Postgres]({{< relref "./high-availability.md" >}}) cluster, PGO uses a rolling update to minimize or eliminate any downtime for your application.

Applying Minor Postgres Updates

The Postgres image is referenced using the `spec.image` and looks similar to the below:

```
spec:
  image: registry.developers.crunchydata.com/crunchydata/crunchy-postgres:centos8-13.4-0
```

Diving into the tag a bit further, you will notice the `13.4-0` portion. This represents the Postgres minor version (`13.4`) and the patch number of the release `0`. If the patch number is incremented (e.g. `13.4-1`), this means that the container is rebuilt, but there are no changes to the Postgres version. If the minor version is incremented (e.g. `13.4-0`), this means that the is a newer bug fix release of Postgres within the container.

To update the image, you just need to modify the `spec.image` field with the new image reference, e.g.

```
spec:
  image: registry.developers.crunchydata.com/crunchydata/crunchy-postgres:centos8-13.4-1
```

You can apply the changes using `kubectl apply`. Similar to the rolling update example when we [resized the cluster]({{< relref "./resize-cluster.md" >}}), the update is first applied to the Postgres replicas, then a controlled switchover occurs, and the final instance is updated.

For the `hippo` cluster, you can see the status of the rollout by running the command below:

```
kubectl -n postgres-operator get pods \
  --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instan
      \
  -o=jsonpath='{range
      .items[*]}{.metadata.name}{"\t"}{.metadata.labels.postgres-operator\.crunchydata\.com/role}{"\t"
```

or by running a watch:

```
watch "kubectl -n postgres-operator get pods \
  --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/instan
      \
  -o=jsonpath='{range
      .items[*]}{.metadata.name}{\"\t\"}{.metadata.labels.postgres-operator\.crunchydata\.com/role}{\"
```

Rolling Back Minor Postgres Updates

This methodology also allows you to rollback changes from minor Postgres updates. You can change the `spec.image` field to your desired container image. PGO will then ensure each Postgres instance in the cluster rolls back to the desired image.

Applying Other Component Updates

There are other components that go into a PGO Postgres cluster. These include pgBackRest, PgBouncer and others. Each one of these components has its own image: for example, you can find a reference to the pgBackRest image in the `spec.backups.pgbackrest.image` attribute.

Applying software updates for the other components in a Postgres cluster works similarly to the above. As pgBackRest and PgBouncer are Kubernetes Deployments, Kubernetes will help manage the rolling update to minimize disruption.

Next Steps

Now that we know how to update our software components, let's look at how PGO handles [disaster recovery]({{< relref "./backups.md" >}})!

An important part of a healthy Postgres cluster is maintaining backups. PGO optimizes its use of open source pgBackRest to be able to support terabyte size databases. What's more, PGO makes it convenient to perform many common and advanced actions that can occur during the lifecycle of a database, including:

- Setting automatic backup schedules and retention policies
- Backing data up to multiple locations
- Support for backup storage in Kubernetes, AWS S3 (or S3-compatible systems like MinIO), Google Cloud Storage (GCS), and Azure Blog Storage
- Taking one-off / ad hoc backups
- Performing a "point-in-time-recovery"
- Cloning data to a new instance

and more.

Let's explore the various disaster recovery features in PGO by first looking at how to set up backups.

Understanding Backup Configuration and Basic Operations

The backup configuration for a PGO managed Postgres cluster resides in the `spec.backups.pgbackrest` section of a custom resource. In addition to indicating which version of pgBackRest to use, this section allows you to configure the fundamental backup settings for your Postgres cluster, including:

- `spec.backups.pgbackrest.configuration` - allows to add additional configuration and references to Secrets that are needed for configuration your backups. For example, this may reference a Secret that contains your S3 credentials.
- `spec.backups.pgbackrest.global` - a convenience to apply global pgBackRest configuration. An example of this may be setting the global pgBackRest logging level (e.g. `log-level-console: info`), or provide configuration to optimize performance.
- `spec.backups.pgbackrest.repos` - information on each specific pgBackRest backup repository. This allows you to configure where and how your backups and WAL archive are stored. You can keep backups in up to four (4) different locations!

You can configure the `repos` section based on the backup storage system you are looking to use. Specifically, you configure your `repos` section according to the storage type you are using. There are four storage types available in `spec.backups.pgbackrest.repos`:

Storage Type	Description
`azure`	For use with Azure Blob Storage.
`gcs`	For use with Google Cloud Storage (GCS).
`s3`	For use with Amazon S3 or any S3 compatible storage system such as MinIO.
`volume`	For use with a Kubernetes Persistent Volume.

Regardless of the backup storage system you select, you **must** assign a name to `spec.backups.pgbackrest.repos.name`, e.g. `repo1`. pgBackRest follows the convention of assigning configuration to a specific repository using a `repoN` format, e.g. `repo1`, `repo2`, etc. You can customize your configuration based upon the name that you assign in the spec. We will cover this topic further in the multi-repository example.

By default, backups are stored in a directory that follows the pattern `pgbackrest/repoN` where `N` is the number of the repo. This typically does not present issues when storing your backup information in a Kubernetes volume, but it can present complications if you are storing all of your backups in the same backup in a blob storage system like S3/GCS/Azure. You can avoid conflicts by setting the `repoN-path` variable in `spec.backups.pgbackrest.global`. The convention we recommend for setting this variable is `/pgbackrest/$NAMESPACE/$CLUSTER_NAME/repoN`. For example, if I have a cluster named `hippo` in the namespace `postgres-operator`, I would set the following:

```
spec:
  backups:
    pgbackrest:
      global:
        repo1-path: /pgbackrest/postgres-operator/hippo/repo1
```

As mentioned earlier, you can store backups in up to four different repositories. You can also mix and match, e.g. you could store your backups in two different S3 repositories. Each storage type does have its own required attributes that you need to set. We will cover that later in this section.

Now that we've covered the basics, let's learn how to set up our backup repositories!

Setting Up a Backup Repository

As mentioned above, PGO, the Postgres Operator from Crunchy Data, supports multiple ways to store backups. Let's look into each method and see how you can ensure your backups and archives are being safely stored!

Using Kubernetes Volumes

The simplest way to get started storing backups is to use a Kubernetes Volume. This was already configure as part of the [create a Postgres cluster]({{< relref "./create-cluster.md">}}) example. Let's take a closer look at some of that configuration:

```
- name: repo1
  volume:
    volumeClaimSpec:
      accessModes:
      - "ReadWriteOnce"
      resources:
        requests:
          storage: 1Gi
```

The one requirement of volume is that you need to fill out the `volumeClaimSpec` attribute. This attribute uses the same format as a persistent volume claim spec! In fact, we performed a similar set up when we [created a Postgres cluster]({{< relref "./create-cluster.md">}}).

In the above example, we assume that the Kubernetes cluster is using a default storage class. If your cluster does not have a default storage class, or you wish to use a different storage class, you will have to set `spec.backups.pgbackrest.repos.volume.volumeClaimSpec.storage`

Using S3

Setting up backups in S3 requires a few additional modifications to your custom resource spec and the use of a Secret to protect your S3 credentials!

There is an example for creating a Postgres cluster that uses S3 for backups in the `kustomize/s3` directory in the Postgres Operator examples repository. In this directory, there is a file called `s3.conf.example`. Copy this example file to `s3.conf`:

```
cp s3.conf.example s3.conf
```

Note that `s3.conf` is protected from commit by a `.gitignore`.

Open up `s3.conf`, you will see something similar to:

```
[global]
repo1-s3-key=<YOUR_AWS_S3_KEY>
repo1-s3-key-secret=<YOUR_AWS_S3_KEY_SECRET>
```

Replace the values with your AWS S3 credentials and save.

Now, open up `kustomize/s3/postgres.yaml`. In the `s3` section, you will see something similar to:

```
s3:
  bucket: "<YOUR_AWS_S3_BUCKET_NAME>"
  endpoint: "<YOUR_AWS_S3_ENDPOINT>"
  region: "<YOUR_AWS_S3_REGION>"
```

Again, replace these values with the values that match your S3 configuration. For `endpoint`, only use the domain and, if necessary, the port (e.g. `s3.us-east-2.amazonaws.com`).

Note that `region` is required by S3, as does pgBackRest. If you are using a storage system with a S3 compatibility layer that does not require `region`, you can fill in region with a random value.

When your configuration is saved, you can deploy your cluster:

```
kubectl apply -k kustomize/s3
```

Watch your cluster: you will see that your backups and archives are now being stored in S3!

Using Google Cloud Storage (GCS)

Similar to S3, setting up backups in Google Cloud Storage (GCS) requires a few additional modifications to your custom resource spec and the use of a Secret to protect your GCS credentials.

There is an example for creating a Postgres cluster that uses GCS for backups in the `kustomize/gcs` directory in the Postgres Operator examples repository. In order to configure this example to use GCS for backups, you will need do two things.

First, copy your GCS key secret (which is a JSON file) into `kustomize/gcs/gcs-key.json`. Note that a `.gitignore` directive prevents you from committing this file.

Next, open the `postgres.yaml` file and edit `spec.backups.pgbackrest.repos.gcs.bucket` to the name of the GCS bucket that you want to back up to.

Save this file, and then run:

```
kubectl apply -k kustomize/gcs
```

Watch your cluster: you will see that your backups and archives are now being stored in GCS!

Using Azure Blob Storage

Similar to the above, setting up backups in Azure Blob Storage requires a few additional modifications to your custom resource spec and the use of a Secret to protect your GCS credentials.

There is an example for creating a Postgres cluster that uses Azure for backups in the `kustomize/azure` directory in the Postgres Operator examples repository. In this directory, there is a file called `azure.conf.example`. Copy this example file to `azure.conf`:

```
cp azure.conf.example azure.conf
```

Note that `azure.conf` is protected from commit by a `.gitignore`.

Open up `azure.conf`, you will see something similar to:

```
[global]
repo1-azure-account=<YOUR_AZURE_ACCOUNT>
repo1-azure-key=<YOUR_AZURE_KEY>
```

Replace the values with your Azure credentials and save.

Now, open up `kustomize/azure/postgres.yaml`. In the `azure` section, you will see something similar to:

```
azure:
  container: "<YOUR_AZURE_CONTAINER>"
```

Again, replace these values with the values that match your Azure configuration.

When your configuration is saved, you can deploy your cluster:

```
kubectl apply -k kustomize/azure
```

Watch your cluster: you will see that your backups and archives are now being stored in Azure!

Set Up Multiple Backup Repositories

It is possible to store backups in multiple locations! For example, you may want to keep your backups both within your Kubernetes cluster and S3. There are many reasons for doing this:

- It is typically faster to heal Postgres instances when your backups are closer
- You can set different backup retention policies based upon your available storage
- You want to ensure that your backups are distributed geographically

and more.

PGO lets you store your backups in up to four locations simultaneously. You can mix and match: for example, you can store backups both locally and in GCS, or store your backups in two different GCS repositories. It's up to you!

There is an example in the Postgres Operator examples repository in the `kustomize/multi-backup-repo` folder that sets up backups in four different locations using each storage type. You can modify this example to match your desired backup topology.

Additional Notes

While storing Postgres archives (write-ahead log [WAL] files) occurs in parallel when saving data to multiple pgBackRest repos, you cannot take parallel backups to different repos at the same time. PGO will ensure that all backups are taken serially. Future work in pgBackRest will address parallel backups to different repos. Please don't confuse this with parallel backup: pgBackRest does allow for backups to use parallel processes when storing them to a single repo!

Encryption

You can encrypt your backups using AES-256 encryption using the CBC mode. This can be used independent of any encryption that may be supported by an external backup system.

To encrypt your backups, you need to set the cipher type and provide a passphrase. The passphrase should be long and random (e.g. the pgBackRest documentation recommends `openssl rand -base64 48`). The passphrase should be kept in a Secret.

Let's use our `hippo` cluster as an example. Let's create a new directory. First, create a file called `pgbackrest-secrets.conf` in this directory. It should look something like this:

```
[global]
repo1-cipher-pass=your-super-secure-encryption-key-passphrase
```

This contains the passphrase used to encrypt your data.

Next, create a `kustomization.yaml` file that looks like this:

```
namespace: postgres-operator

secretGenerator:
- name: hippo-pgbackrest-secrets
  files:
  - pgbackrest-secrets.conf

generatorOptions:
  disableNameSuffixHash: true

resources:
- postgres.yaml
```

Finally, create the manifest for the Postgres cluster in a file named `postgres.yaml` that is similar to the following:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec:
        accessModes:
```

```
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      configuration:
      - secret:
          name: hippo-pgbackrest-secrets
      global:
        repo1-cipher-type: aes-256-cbc
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Notice the reference to the Secret that contains the encryption key:

```
spec:
  backups:
    pgbackrest:
      configuration:
      - secret:
          name: hippo-pgbackrest-secrets
```

as well as the configuration for enabling AES-256 encryption using the CBC mode:

```
spec:
  backups:
    pgbackrest:
      global:
        repo1-cipher-type: aes-256-cbc
```

You can now create a Postgres cluster that has encrypted backups!

Limitations

Currently the encryption settings cannot be changed on backups after they are established.

Custom Backup Configuration

Most of your backup configuration can be configured through the `spec.backups.pgbackrest.global` attribute, or through information that you supply in the ConfigMap or Secret that you refer to in `spec.backups.pgbackrest.configuration`. You can also provide additional Secret values if need be, e.g. `repo1-cipher-pass` for encrypting backups.

The full list of pgBackRest configuration options is available here:

https://pgbackrest.org/configuration.html

Next Steps

We've now seen how to use PGO to get our backups and archives set up and safely stored. Now let's take a look at [backup management]({{< relref "./backup-management.md" >}}) and how we can do things such as set backup frequency, set retention policies, and even take one-off backups!

In the [previous section]({{< relref "./backups.md" >}}), we looked at a brief overview of the full disaster recovery feature set that PGO provides and explored how to [configure backups for our Postgres cluster]({{< relref "./backups.md" >}}).

Now that we have backups set up, lets look at some of the various backup management tasks we can perform. These include:

- Setting up scheduled backups
- Setting backup retention policies
- Taking one-off / ad hoc backups

Managing Scheduled Backups

PGO sets up your Postgres clusters so that they are continuously archiving the write-ahead log: your data is constantly being stored in your backup repository. Effectively, this is a backup!

However, in a [disaster recovery]({{< relref "./disaster-recovery.md" >}}) scenario, you likely want to get your Postgres cluster back up and running as quickly as possible (e.g. a short "recovery time objective (RTO)"). What helps accomplish this is to take periodic backups. This makes it faster to restore!

pgBackRest, the backup management tool used by PGO, provides different backup types to help both from a space management and RTO optimization perspective. These backup types include:

- `full`: A backup of your entire Postgres cluster. This is the largest of all of the backup types.
- `differential`: A backup of all of the data since the last `full` backup.
- `incremental`: A backup of all of the data since the last `full`, `differential`, or `incremental` backup.

Selecting the appropriate backup strategy for your Postgres cluster is outside the scope of this tutorial, but let's look at how we can set up scheduled backups.

Backup schedules are stored in the `spec.backups.pgbackrest.repos.schedules` section. Each value in this section accepts a cron-formatted string that dictates the backup schedule.

Let's say that our backup policy is to take a full backup once a day at 1am and take incremental backups every four hours. We would want to add configuration to our spec that looks similar to:

```
spec:
  backups:
    pgbackrest:
      repos:
      - name: repo1
        schedules:
          full: "0 1 * * *"
          incremental: "0 */4 * * *"
```

To manage scheduled backups, PGO will create several Kubernetes CronJobs that will perform backups on the specified periods. The backups will use the [configuration that you specified]({{< relref "./backups.md" >}}).

Ensuring you take regularly scheduled backups is important to maintaining Postgres cluster health. However, you don't need to keep all of your backups: this could cause you to run out of space! As such, it's also important to set a backup retention policy.

Managing Backup Retention

PGO lets you set backup retention on full and differential backups. When a full backup expires, either through your retention policy or through manual expiration, pgBackRest will clean up any backup and WAL files associated with it. For example, if you have a full backup with four associated incremental backups, when the full backup expires, all of its incremental backups also expire.

There are two different types of backup retention you can set:

- `count`: This is based on the number of backups you want to keep. This is the default.
- `time`: This is based on the total number of days you would like to keep a backup.

Let's look at an example where we keep full backups for 14 days. The most convenient way to do this is through the `spec.backups.pgbackrest.global` section:

```
spec:
  backups:
    pgbackrest:
      global:
        repo1-retention-full: "14"
        repo1-retention-full-type: time
```

The full list of available configuration options is in the pgBackRest configuration guide.

Taking a One-Off Backup

There are times where you may want to take a one-off backup, such as before major application changes or updates. This is not your typical declarative action – in fact a one-off backup is imperative in its nature! – but it is possible to take a one-off backup of your Postgres cluster with PGO.

First, you need to configure the `spec.backups.pgbackrest.manual` section to be able to take a one-off backup. This contains information about the type of backup you want to take and any other pgBackRest configuration options.

Let's configure the custom resource to take a one-off full backup:

```
spec:
  backups:
    pgbackrest:
      manual:
        repoName: repo1
        options:
         - --type=full
```

This does not trigger the one-off backup – you have to do that by adding the `postgres-operator.crunchydata.com/pgbackrest-backup` annotation to your custom resource. The best way to set this annotation is with a timestamp, so you know when you initialized the backup.

For example, for our `hippo` cluster, we can run the following command to trigger the one-off backup:

```
kubectl annotate -n postgres-operator postgrescluster hippo \
  postgres-operator.crunchydata.com/pgbackrest-backup="$(date)"
```

PGO will detect this annotation and create a new, one-off backup Job!

If you intend to take one-off backups with similar settings in the future, you can leave those in the spec; just update the annotation to a different value the next time you are taking a backup.

To re-run the command above, you will need to add the `--overwrite` flag so the annotation's value can be updated, i.e.

```
kubectl annotate -n postgres-operator postgrescluster hippo --overwrite \
  postgres-operator.crunchydata.com/pgbackrest-backup="$(date)"
```

Next Steps

We've covered the fundamental tasks with managing backups. What about [restores]({{< relref "./disaster-recovery.md" >}})? Or [cloning data into new Postgres clusters]({{< relref "./disaster-recovery.md" >}})? Let's explore!

Perhaps someone accidentally dropped the `users` table. Perhaps you want to clone your production database to a step-down environment. Perhaps you want to exercise your disaster recovery system (and it is important that you do!).

Regardless of scenario, it's important to know how you can perform a "restore" operation with PGO to be able to recovery your data from a particular point in time, or clone a database for other purposes.

Let's look at how we can perform different types of restore operations. First, let's understand the core restore properties on the custom resource.

Restore Properties

There are several attributes on the custom resource that are important to understand as part of the restore process. All of these attributes are grouped together in the `spec.dataSource.postgresCluster` section of the custom resource.

Please review the table below to understand how each of these attributes work in the context of setting up a restore operation.

- `spec.dataSource.postgresCluster.clusterName`: The name of the cluster that you are restoring from. This corresponds to the `metadata.name` attribute on a different `postgrescluster` custom resource.
- `spec.dataSource.postgresCluster.clusterNamespace`: The namespace of the cluster that you are restoring from. Used when the cluster exists in a different namespace.
- `spec.dataSource.postgresCluster.repoName`: The name of the pgBackRest repository from the `spec.dataSource.postgresCluste` to use for the restore. Can be one of `repo1`, `repo2`, `repo3`, or `repo4`. The repository must exist in the other cluster.
- `spec.dataSource.postgresCluster.options`: Any additional pgBackRest restore options or general options that PGO allows. For example, you may want to set `--process-max` to help improve performance on larger databases; but you will not be able to set`--target-action`, since that option is currently disallowed. (PGO always sets it to `promote` if a `--target` is present, and otherwise leaves it blank.)

- `spec.dataSource.postgresCluster.resources`: Setting resource limits and requests of the restore job can ensure that it runs efficiently.
- `spec.dataSource.postgresCluster.affinity`: Custom Kubernetes affinity rules constrain the restore job so that it only runs on certain nodes.
- `spec.dataSource.postgresCluster.tolerations`: Custom Kubernetes tolerations allow the restore job to run on tainted nodes.

Let's walk through some examples for how we can clone and restore our databases.

Clone a Postgres Cluster

Let's create a clone of our [`hippo`]({{< relref "./create-cluster.md" >}}) cluster that we created previously. We know that our cluster is named `hippo` (based on its `metadata.name`) and that we only have a single backup repository called `repo1`.

Let's call our new cluster `elephant`. We can create a clone of the `hippo` cluster using a manifest like this:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: elephant
spec:
  dataSource:
    postgresCluster:
      clusterName: hippo
      repoName: repo1
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Note this section of the spec:

```
spec:
  dataSource:
    postgresCluster:
      clusterName: hippo
      repoName: repo1
```

This is the part that tells PGO to create the `elephant` cluster as an independent copy of the `hippo` cluster.

The above is all you need to do to clone a Postgres cluster! PGO will work on creating a copy of your data on a new persistent volume claim (PVC) and work on initializing your cluster to spec. Easy!

Perform a Point-in-time-Recovery (PITR)

Did someone drop the user table? You may want to perform a point-in-time-recovery (PITR) to revert your database back to a state before a change occurred. Fortunately, PGO can help you do that.

You can set up a PITR using the restore command of pgBackRest, the backup management tool that powers the disaster recovery capabilities of PGO. You will need to set a few options on `spec.dataSource.postgresCluster.options` to perform a PITR. These options include:

- `--type=time`: This tells pgBackRest to perform a PITR.
- `--target`: Where to perform the PITR to. Any example recovery target is `2021-06-09 14:15:11-04`. The timezone specified here as -04 for EDT. Please see the pgBackRest documentation for other timezone options.
- `--set` (optional): Choose which backup to start the PITR from.

A few quick notes before we begin:

- To perform a PITR, you must have a backup that is older than your PITR time. In other words, you can't perform a PITR back to a time where you do not have a backup!
- All relevant WAL files must be successfully pushed for the restore to complete correctly.
- Be sure to select the correct repository name containing the desired backup!

With that in mind, let's use the `elephant` example above. Let's say we want to perform a point-in-time-recovery (PITR) to `2021-06-09 14:15:11-04`, we can use the following manifest:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: elephant
spec:
  dataSource:
    postgresCluster:
      clusterName: hippo
      repoName: repo1
      options:
      - --type=time
      - --target="2021-06-09 14:15:11-04"
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

The section to pay attention to is this:

```
spec:
  dataSource:
    postgresCluster:
      clusterName: hippo
      repoName: repo1
      options:
      - --type=time
      - --target="2021-06-09 14:15:11-04"
```

Notice how we put in the options to specify where to make the PITR.

Using the above manifest, PGO will go ahead and create a new Postgres cluster that recovers its data up until `2021-06-09 14:15:11-04`. At that point, the cluster is promoted and you can start accessing your database from that specific point in time!

Perform an In-Place Point-in-time-Recovery (PITR)

Similar to the PITR restore described above, you may want to perform a similar reversion back to a state before a change occurred, but without creating another PostgreSQL cluster. Fortunately, PGO can help you do this as well.

You can set up a PITR using the restore command of pgBackRest, the backup management tool that powers the disaster recovery capabilities of PGO. You will need to set a few options on `spec.dataSource.postgresCluster.options` to perform a PITR. These options include:

- `--type=time`: This tells pgBackRest to perform a PITR.
- `--target`: Where to perform the PITR to. Any example recovery target is `2021-06-09 14:15:11-04`.
- `--set` (optional): Choose which backup to start the PITR from.

A few quick notes before we begin:

- To perform a PITR, you must have a backup that is older than your PITR time. In other words, you can't perform a PITR back to a time where you do not have a backup!
- All relevant WAL files must be successfully pushed for the restore to complete correctly.
- Be sure to select the correct repository name containing the desired backup!

To perform an in-place restore, users will first fill out the restore section of the spec as follows:

```
spec:
  backups:
    pgbackrest:
      restore:
        enabled: true
        repoName: repo1
        options:
        - --type=time
        - --target="2021-06-09 14:15:11-04"
```

And to trigger the restore, you will then annotate the PostgresCluster as follows:

```
kubectl annotate postgrescluster hippo --overwrite \
  postgres-operator.crunchydata.com/pgbackrest-restore=id1
```

And once the restore is complete, in-place restores can be disabled:

```
spec:
  backups:
    pgbackrest:
      restore:
        enabled: false
```

Notice how we put in the options to specify where to make the PITR.

Using the above manifest, PGO will go ahead and re-create your Postgres cluster that will recover its data up until `2021-06-09 14:15:11-04`. At that point, the cluster is promoted and you can start accessing your database from that specific point in time!

Standby Cluster

Advanced high-availability and disaster recovery strategies involve spreading your database clusters across multiple data centers to help maximize uptime. In Kubernetes, this technique is known as "federation". Federated Kubernetes clusters are able to communicate with each other, coordinate changes, and provide resiliency for applications that have high uptime requirements.

As of this writing, federation in Kubernetes is still in ongoing development. In the meantime, PGO provides a way to deploy Postgres clusters that can span multiple Kubernetes clusters using an external storage system:

- Amazon S3, or a system that uses the S3 protocol,
- Azure Blob Storage, or
- Google Cloud Storage

Standby Postgres clusters are managed just like any other Postgres cluster in PGO. For example, adding replicas to a standby cluster is a matter of increasing the `spec.instances.replicas` value. The main difference is that PostgreSQL data in the cluster is read-only: one PostgreSQL instance is reading in the database changes from an external repository while the other instances are replicas of it. This is known as cascading replication.

The following manifest defines a Postgres cluster that recovers from WAL files stored in an S3 bucket:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo-standby
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        s3:
          bucket: "my-bucket"
          endpoint: "s3.ca-central-1.amazonaws.com"
          region: "ca-central-1"
  standby:
    enabled: true
    repoName: repo1
```

There comes a time where a standby cluster needs to be promoted to an active cluster. Promoting a standby cluster means that a PostgreSQL instance within it will start accepting both reads and writes. This has the net effect of pushing WAL (transaction archives) to the pgBackRest repository, so we need to take a few steps first to ensure we don't accidentally create a split-brain scenario.

First, if this is not a disaster scenario, you will want to "shutdown" the active PostgreSQL cluster. This can be done with the `spec.shutdown` attribute:

```
spec:
  shutdown: true
```

The effect of this is that all the Kubernetes workloads for this cluster are scaled to 0. You can verify this with the following command:

```
kubectl get deploy,sts,cronjob --selector=postgres-operator.crunchydata.com/cluster=hippo

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/hippo-pgbouncer    0/0     0            0           1h

NAME                              READY   AGE
statefulset.apps/hippo-00-lwgx    0/0     1h

NAME                                       SCHEDULE   SUSPEND   ACTIVE
cronjob.batch/hippo-pgbackrest-repo1-full  @daily     True      0
```

We can then promote the standby cluster by removing or disabling its `spec.standby` section:

```
spec:
  standby:
    enabled: false
```

This change triggers the promotion of the standby leader to a primary PostgreSQL instance, and the cluster begins accepting writes.

Next Steps

Now we've seen how to clone a cluster and perform a point-in-time-recovery, let's see how we can [monitor]({{< relref "./monitoring.md" >}}) our Postgres cluster to detect and prevent issues from occurring.

While having [high availability]({{< relref "tutorial/high-availability.md" >}}) and [disaster recovery]({{< relref "tutorial/disaster-recovery.md" >}}) systems in place helps in the event of something going wrong with your PostgreSQL cluster, monitoring helps you anticipate problems before they happen. Additionally, monitoring can help you diagnose and resolve issues that may cause degraded performance rather than downtime.

Let's look at how PGO allows you to enable monitoring in your cluster.

Adding the Exporter Sidecar

Let's look at how we can add the Crunchy PostgreSQL Exporter sidecar to your cluster using the `kustomize/postgres` example in the Postgres Operator examples repository.

Monitoring tools are added using the `spec.monitoring` section of the custom resource. Currently, the only monitoring tool supported is the Crunchy PostgreSQL Exporter configured with pgMonitor.

The only required attribute for adding the Exporter sidecar is to set `spec.monitoring.pgmonitor.exporter.image`. In the `kustomize/postgres/postgres.yaml` file, add the following YAML to the spec:

```
monitoring:
  pgmonitor:
    exporter:
     image: {{< param imageCrunchyExporter >}}
```

Save your changes and run:

```
kubectl apply -k kustomize/postgres
```

PGO will detect the change and add the Exporter sidecar to all Postgres Pods that exist in your cluster. PGO will also do the work to allow the Exporter to connect to the database and gather metrics that can be accessed using the [PGO Monitoring] stack.

Accessing the Metrics

Once the Crunchy PostgreSQL Exporter has been enabled in your cluster, follow the steps outlined in [PGO Monitoring] to install the monitoring stack. This will allow you to deploy a pgMonitor configuration of Prometheus, Grafana, and Alertmanager monitoring tools in Kubernetes. These tools will be set up by default to connect to the Exporter containers on your Postgres Pods.

Next Steps

Now that we can monitor our cluster, let's explore how [connection pooling]({{< relref "connection-pooling.md" >}}) can be enabled using PGO and how it is helpful.

[PGO Monitoring]: {{< relref "installation/monitoring/_index.md" >}}

Connection pooling can be helpful for scaling and maintaining overall availability between your application and the database. PGO helps facilitate this by supporting the PgBouncer connection pooler and state manager.

Let's look at how we can a connection pooler and connect it to our application!

Adding a Connection Pooler

Let's look at how we can add a connection pooler using the `kustomize/keycloak` example in the Postgres Operator examples repository.

Connection poolers are added using the `spec.proxy` section of the custom resource. Currently, the only connection pooler supported is PgBouncer.

The only required attribute for adding a PgBouncer connection pooler is to set the `spec.proxy.pgBouncer.image` attribute. In the `kustomize/keycloak/postgres.yaml` file, add the following YAML to the spec:

```
proxy:
  pgBouncer:
    image: {{< param imageCrunchyPGBouncer >}}
```

(You can also find an example of this in the `kustomize/examples/high-availability` example).

Save your changes and run:

```
kubectl apply -k kustomize/keycloak
```

PGO will detect the change and create a new PgBouncer Deployment!

That was fairly easy to set up, so now let's look at how we can connect our application to the connection pooler.

Connecting to a Connection Pooler

When a connection pooler is deployed to the cluster, PGO adds additional information to the user Secrets to allow for applications to connect directly to the connection pooler. Recall that in this example, our user Secret is called `keycloakdb-pguser-keycloakdb`. Describe the user Secret:

```
kubectl -n postgres-operator describe secrets keycloakdb-pguser-keycloakdb
```

You should see that there are several new attributes included in this Secret that allow for you to connect to your Postgres instance via the connection pooler:

- `pgbouncer-host`: The name of the host of the PgBouncer connection pooler. This references the Service of the PgBouncer connection pooler.
- `pgbouncer-port`: The port that the PgBouncer connection pooler is listening on.
- `pgbouncer-uri`: A PostgreSQL connection URI that provides all the information for logging into the Postgres database via the PgBouncer connection pooler.
- `pgbouncer-jdbc-uri`: A PostgreSQL JDBC connection URI that provides all the information for logging into the Postgres database via the PgBouncer connection pooler using the JDBC driver. Note that by default, the connection string disable JDBC managing prepared transactions for optimal use with PgBouncer.

Open up the file in `kustomize/keycloak/keycloak.yaml`. Update the `DB_ADDR` and `DB_PORT` values to be the following:

```
- name: DB_ADDR
  valueFrom: { secretKeyRef: { name: keycloakdb-pguser-keycloakdb, key: pgbouncer-host } }
- name: DB_PORT
  valueFrom: { secretKeyRef: { name: keycloakdb-pguser-keycloakdb, key: pgbouncer-port } }
```

This changes Keycloak's configuration so that it will now connect through the connection pooler.

Apply the changes:

```
kubectl apply -k kustomize/keycloak
```

Kubernetes will detect the changes and begin to deploy a new Keycloak Pod. When it is completed, Keycloak will now be connected to Postgres via the PgBouncer connection pooler!

TLS

PGO deploys every cluster and component over TLS. This includes the PgBouncer connection pooler. If you are using your own [custom TLS setup]({{< relref "./customize-cluster.md" >}}#customize-tls), you will need to provide a Secret reference for a TLS key / certificate pair for PgBouncer in `spec.proxy.pgBouncer.customTLSSecret`.

Your TLS certificate for PgBouncer should have a Common Name (CN) setting that matches the PgBouncer Service name. This is the name of the cluster suffixed with `-pgbouncer`. For example, for our `hippo` cluster this would be `hippo-pgbouncer`. For the `keycloakdb` example, it would be `keycloakdb-pgbouncer`.

To customize the TLS for PgBouncer, you will need to create a Secret in the Namespace of your Postgres cluster that contains the TLS key (`tls.key`), TLS certificate (`tls.crt`) and the CA certificate (`ca.crt`) to use. The Secret should contain the following values:

```
data:
  ca.crt: <value>
  tls.crt: <value>
  tls.key: <value>
```

For example, if you have files named `ca.crt`, `keycloakdb-pgbouncer.key`, and `keycloakdb-pgbouncer.crt` stored on your local machine, you could run the following command:

```
kubectl create secret generic -n postgres-operator keycloakdb-pgbouncer.tls \
  --from-file=ca.crt=ca.crt \
  --from-file=tls.key=keycloakdb-pgbouncer.key \
  --from-file=tls.crt=keycloakdb-pgbouncer.crt
```

You can specify the custom TLS Secret in the `spec.proxy.pgBouncer.customTLSSecret.name` field in your `postgrescluster.postgres-op` custom resource, e.g.:

```
spec:
  proxy:
    pgBouncer:
      customTLSSecret:
        name: keycloakdb-pgbouncer.tls
```

Customizing

The PgBouncer connection pooler is highly customizable, both from a configuration and Kubernetes deployment standpoint. Let's explore some of the customizations that you can do!

Configuration

PgBouncer configuration can be customized through `spec.proxy.pgBouncer.config`. After making configuration changes, PGO will roll them out to any PgBouncer instance and automatically issue a "reload".

There are several ways you can customize the configuration:

- `spec.proxy.pgBouncer.config.global`: Accepts key-value pairs that apply changes globally to PgBouncer.
- `spec.proxy.pgBouncer.config.databases`: Accepts key-value pairs that represent PgBouncer database definitions.
- `spec.proxy.pgBouncer.config.users`: Accepts key-value pairs that represent connection settings applied to specific users.
- `spec.proxy.pgBouncer.config.files`: Accepts a list of files that are mounted in the `/etc/pgbouncer` directory and loaded before any other options are considered using PgBouncer's include directive.

For example, to set the connection pool mode to `transaction`, you would set the following configuration:

```
spec:
  proxy:
    pgBouncer:
      config:
        global:
          pool_mode: transaction
```

For a reference on PgBouncer configuration please see:

https://www.pgbouncer.org/config.html

Replicas

PGO deploys one PgBouncer instance by default. You may want to run multiple PgBouncer instances to have some level of redundancy, though you still want to be mindful of how many connections are going to your Postgres database!

You can manage the number of PgBouncer instances that are deployed through the `spec.proxy.pgBouncer.replicas` attribute.

Resources

You can manage the CPU and memory resources given to a PgBouncer instance through the `spec.proxy.pgBouncer.resources` attribute. The layout of `spec.proxy.pgBouncer.resources` should be familiar: it follows the same pattern as the standard Kubernetes structure for setting container resources.

For example, let's say we want to set some CPU and memory limits on our PgBouncer instances. We could add the following configuration:

```
spec:
  proxy:
    pgBouncer:
      resources:
        limits:
          cpu: 200m
          memory: 128Mi
```

As PGO deploys the PgBouncer instances using a Deployment these changes are rolled out using a rolling update to minimize disruption between your application and Postgres instances!

Annotations / Labels

You can apply custom annotations and labels to your PgBouncer instances through the `spec.proxy.pgBouncer.metadata.annotations` and `spec.proxy.pgBouncer.metadata.labels` attributes respectively. Note that any changes to either of these two attributes take precedence over any other custom labels you have added.

Pod Anti-Affinity / Pod Affinity / Node Affinity

You can control the pod anti-affinity, pod affinity, and node affinity through the `spec.proxy.pgBouncer.affinity` attribute, specifically:

- `spec.proxy.pgBouncer.affinity.nodeAffinity`: controls node affinity for the PgBouncer instances.
- `spec.proxy.pgBouncer.affinity.podAffinity`: controls Pod affinity for the PgBouncer instances.
- `spec.proxy.pgBouncer.affinity.podAntiAffinity`: controls Pod anti-affinity for the PgBouncer instances.

Each of the above follows the standard Kubernetes specification for setting affinity.

For example, to set a preferred Pod anti-affinity rule for the `kustomize/keycloak` example, you would want to add the following to your configuration:

```
spec:
  proxy:
    pgBouncer:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 1
            podAffinityTerm:
              labelSelector:
                matchLabels:
                  postgres-operator.crunchydata.com/cluster: keycloakdb
                  postgres-operator.crunchydata.com/role: pgbouncer
              topologyKey: kubernetes.io/hostname
```

Tolerations

You can deploy PgBouncer instances to Nodes with Taints by setting Tolerations through `spec.proxy.pgBouncer.tolerations`. This attribute follows the Kubernetes standard tolerations layout.

For example, if there were a set of Nodes with a Taint of `role=connection-poolers:NoSchedule` that you want to schedule your PgBouncer instances to, you could apply the following configuration:

```
spec:
  proxy:
    pgBouncer:
      tolerations:
      - effect: NoSchedule
        key: role
        operator: Equal
        value: connection-poolers
```

Note that setting a toleration does not necessarily mean that the PgBouncer instances will be assigned to Nodes with those taints. Tolerations act as a **key**: they allow for you to access Nodes. If you want to ensure that your PgBouncer instances are deployed to specific nodes, you need to combine setting tolerations with node affinity.

Pod Spread Constraints

Besides using affinity, anti-affinity and tolerations, you can also set Topology Spread Constraints through `spec.proxy.pgBouncer.topologyS`
This attribute follows the Kubernetes standard topology spread contraint layout.

For example, since each of of our pgBouncer Pods will have the standard `postgres-operator.crunchydata.com/role: pgbouncer` Label set, we can use this Label when determining the `maxSkew`. In the example below, since we have 3 nodes with a `maxSkew` of 1 and we've set `whenUnsatisfiable` to `ScheduleAnyway`, we should ideally see 1 Pod on each of the nodes, but our Pods can be distributed less evenly if other constraints keep this from happening.

```
  proxy:
    pgBouncer:
      replicas: 3
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: ScheduleAnyway
          labelSelector:
            matchLabels:
              postgres-operator.crunchydata.com/role: pgbouncer
```

If you want to ensure that your PgBouncer instances are deployed more evenly (or not deployed at all), you need to update `whenUnsatisfiable` to `DoNotSchedule`.

Next Steps

Now that we can enable connection pooling in a cluster, let's explore some [administrative tasks]({{< relref "administrative-tasks.md" >}}) such as manually restarting PostgreSQL using PGO. How do we do that?

Manually Restarting PostgreSQL

There are times when you might need to manually restart PostgreSQL. This can be done by adding or updating a custom annotation to the cluster's `spec.metadata.annotations` section. PGO will notice the change and perform a [rolling restart]({{< relref "/architecture/high-availability.md" >}}#rolling-update).

For example, if you have a cluster named `hippo` in the namespace `postgres-operator`, all you need to do is patch the hippo postgrescluster with the following:

```
kubectl patch postgrescluster/hippo -n postgres-operator --type merge \
  --patch '{"spec":{"metadata":{"annotations":{"restarted":"'"$(date)"'"}}}}'
```

Watch your hippo cluster: you will see the rolling update has been triggered and the restart has begun.

Shutdown

You can shut down a Postgres cluster by setting the `spec.shutdown` attribute to `true`. You can do this by editing the manifest, or, in the case of the `hippo` cluster, executing a comand like the below:

```
kubectl patch postgrescluster/hippo -n postgres-operator --type merge \
  --patch '{"spec":{"shutdown": true}}'
```

Shutting down a cluster will terminate all of the active Pods. Any Statefulsets or Deployments are scaled to `0`.

To turn a Postgres cluster that is shut down back on, you can set `spec.shutdown` to `false`.

Rotating TLS Certificates

Credentials should be invalidated and replaced (rotated) as often as possible to minimize the risk of their misuse. Unlike passwords, every TLS certificate has an expiration, so replacing them is inevitable. When you use your own TLS certificates with PGO, you are responsible for replacing them appropriately. Here's how.

PGO automatically detects and loads changes to the contents of PostgreSQL server and replication Secrets without downtime. You or your certificate manager need only replace the values in the Secret referenced by `spec.customTLSSecret`.

If instead you change `spec.customTLSSecret` to refer to a new Secret or new fields, PGO will perform a [rolling restart]({{< relref "/architecture/high-availability.md" >}}#rolling-update).

When changing the PostgreSQL certificate authority, make sure to update [`customReplicationTLSSecret`]({{< relref "/tutorial/customize-cluster.md" >}}#customize-tls) as well.

PgBouncer needs to be restarted after its certificates change. There are a few ways to do it:

1. Store the new certificates in a new Secret. Edit the PostgresCluster object to refer to the new Secret, and PGO will perform a rolling restart of PgBouncer. `yaml spec: proxy: pgBouncer: customTLSSecret: name: hippo.pgbouncer.new.tls`

or

2. Replace the old certificates in the current Secret. PGO doesn't notice when the contents of your Secret change, so you need to trigger a rolling restart of PgBouncer. Edit the PostgresCluster object to add a unique annotation. The name and value are up to you, so long as the value differs from the previous value. `yaml spec: proxy: pgBouncer: metadata: annotations: restarted: Q1-certs`

This `kubectl patch` command uses your local date and time:

`shell kubectl patch postgrescluster/hippo --type merge \ --patch '{"spec":{"proxy":{"pgBouncer":{"metadata":{"anno`

Next Steps

We've covered a lot in terms of building, maintaining, scaling, customizing, restarting, and expanding our Postgres cluster. However, there may come a time where we need to [delete our Postgres cluster]({{< relref "delete-cluster.md" >}}). How do we do that?

There comes a time when it is necessary to delete your cluster. If you have been following along with the example, you can delete your Postgres cluster by simply running:

```
kubectl delete -k kustomize/postgres
```

PGO will remove all of the objects associated with your cluster.

With data retention, this is subject to the retention policy of your PVC. For more information on how Kubernetes manages data retention, please refer to the Kubernetes docs on volume reclaiming.

This section provides detailed instructions for anything and everything related to installing PGO in your Kubernetes environment. This includes instructions for installing PGO according to a variety of supported installation methods, along with information for customizing the installation of PGO according your specific needs.

Additionally, instructions are provided for installing and configuring [PGO Monitoring]({{< relref "./monitoring" >}}).

Installing PGO

- [PGO Kustomize Install]({{< relref "./kustomize.md" >}})
- [PGO Helm Install]({{< relref "./helm.md" >}})

Installing PGO Monitoring

- [PGO Monitoring Kustomize Install]({{< relref "./monitoring/kustomize.md" >}})

Installing PGO Using Kustomize

This section provides instructions for installing and configuring PGO using Kustomize.

Prerequisites

First, go to GitHub and fork the Postgres Operator examples repository, which contains the PGO Kustomize installer.

https://github.com/CrunchyData/postgres-operator-examples/fork

Once you have forked this repo, you can download it to your working environment with a command similar to this:

```
YOUR_GITHUB_UN="<your GitHub username>"
git clone --depth 1 "git@github.com:${YOUR_GITHUB_UN}/postgres-operator-examples.git"
cd postgres-operator-examples
```

The PGO installation project is located in the `kustomize/install` directory.

Configuration

While the default Kustomize install should work in most Kubernetes environments, it may be necessary to further customize the Kustomize project(s) according to your specific needs.

For instance, to customize the image tags utilized for the PGO Deployment, the `images` setting in the `kustomize/install/bases/kustomiza` file can be modified:

```
images:
- name: postgres-operator
  newName: {{< param repository >}}
  newTag: {{< param postgresOperatorTag >}}
```

Additionally, please note that the Kustomize install project will also create a namespace for PGO by default (though it is possible to install without creating the namespace, as shown below). To modify the name of namespace created by the installer, the `kustomize/install/namespace.yaml` should be modified:

```
apiVersion: v1
kind: Namespace
metadata:
  name: custom-namespace
```

Additionally, the `namespace` setting in `kustomize/install/bases/kustomization.yaml` should be modified accordingly.

```
namespace: custom-namespace
```

Additional Kustomize overlays can then also be created to further patch and customize the installation according to your specific needs.

Installation Mode

When PGO is installed, it can be configured to manage PostgreSQL clusters in all namespaces within the Kubernetes cluster, or just those within a single namespace. When managing PostgreSQL clusters in all namespaces, a ClusterRole and ClusterRoleBinding is created to ensure PGO has the permissions it requires to properly manage PostgreSQL clusters across all namespaces. However, when PGO is configured to manage PostgreSQL clusters within a single namespace only, a Role and RoleBinding is created instead.

By default, the Kustomize installer will configure PGO to manage PostgreSQL clusters in all namespaces, which means a ClusterRole and ClusterRoleBinding will also be created by default. To instead configure PGO to manage PostgreSQL clusters in only a single namespace, simply modify the `bases` section of the `kustomize/install/bases/kustomization.yaml` file as follows:

```
bases:
- crd
- rbac/namespace
- manager
```

Note that `rbac/cluster` has been changed to `rbac/namespace`.

Add the PGO_TARGET_NAMESPACE environment variable to the env section of the `kustomize/install/bases/manager/manager.yaml` file to facilitate the ability to specify the single namespace as follows:

```
        env:
        - name: PGO_TARGET_NAMESPACE
          valueFrom: { fieldRef: { apiVersion: v1, fieldPath: metadata.namespace } }
```

With these configuration changes, PGO will create a Role and RoleBinding, and will therefore only manage PostgreSQL clusters created within the namespace defined using the `namespace` setting in the `kustomize/install/bases/kustomization.yaml` file:

```
namespace: postgres-operator
```

Install

Once the Kustomize project has been modified according to your specific needs, PGO can then be installed using `kubectl` and Kustomize. To create both the target namespace for PGO and then install PGO itself, the following command can be utilized:

```
kubectl apply -k kustomize/install
```

However, if the namespace has already been created, the following command can be utilized to install PGO only:

```
kubectl apply -k kustomize/install/bases
```

Uninstall

Once PGO has been installed, it can also be uninstalled using `kubectl` and Kustomize. To uninstall PGO and then also delete the namespace it had been deployed into (assuming the namespace was previously created using the Kustomize installer as described above), the following command can be utilized:

```
kubectl delete -k kustomize/install
```

To uninstall PGO only (e.g. if Kustomize was not initially utilized to create the PGO namespace), the following command can be utilized:

```
kubectl delete -k kustomize/install/bases
```

Installing PGO Using Helm

This section provides instructions for installing and configuring PGO using Helm.

Prerequisites

First, go to GitHub and fork the Postgres Operator examples repository, which contains the PGO Helm installer.

https://github.com/CrunchyData/postgres-operator-examples/fork

Once you have forked this repo, you can download it to your working environment with a command similar to this:

```
YOUR_GITHUB_UN="<your GitHub username>"
git clone --depth 1 "git@github.com:${YOUR_GITHUB_UN}/postgres-operator-examples.git"
cd postgres-operator-examples
```

The PGO Helm chart is located in the `helm/install` directory of this repository.

Configuration

The `values.yaml` file for the Helm chart contains all of the available configuration settings for PGO. The default `values.yaml` settings should work in most Kubernetes environments, but it may require some customization depending on your specific environment and needs.

For instance, it might be necessary to customize the image tags that are utilized using the `image` setting:

```
image:
  repository: {{< param repository >}}
  tag: "{{< param postgresOperatorTag >}}"
```

Please note that the `values.yaml` file is located in `helm/install`.

Installation Mode

When PGO is installed, it can be configured to manage PostgreSQL clusters in all namespaces within the Kubernetes cluster, or just those within a single namespace. When managing PostgreSQL clusters in all namespaces, a ClusterRole and ClusterRoleBinding is created to ensure PGO has the permissions it requires to properly manage PostgreSQL clusters across all namespaces. However, when PGO is configured to manage PostgreSQL clusters within a single namespace only, a Role and RoleBinding is created instead.

In order to select between these two modes when installing PGO using Helm, the `singleNamespace` setting in the `values.yaml` file can be utilized:

```
singleNamespace: false
```

Specifically, if this setting is set to `false` (which is the default), then a ClusterRole and ClusterRoleBinding will be created, and PGO will manage PostgreSQL clusters in all namespaces. However, if this setting is set to `true`, then a Role and RoleBinding will be created instead, allowing PGO to only manage PostgreSQL clusters in the same namespace utilized when installing the PGO Helm chart.

Install

Once you have configured the Helm chart according to your specific needs, it can then be installed using `helm`:

```
helm install <name> -n <namespace> helm/install
```

Upgrade and Uninstall

And once PGO has been installed, it can then be upgraded and uninstalled using applicable `helm` commands:

```
helm upgrade <name> -n <namespace> helm/install
```

```
helm uninstall <name> -n <namespace>
```

Overview

Upgrading to a new version of PGO is typically as simple as following the various installation guides defined within the PGO documentation:

- [PGO Kustomize Install]({{< relref "./kustomize.md" >}})
- [PGO Helm Install]({{< relref "./helm.md" >}})

However, when upgrading to or from certain versions of PGO, extra steps may be required in order to ensure a clean and successful upgrade. This page will therefore document any additional steps that must be completed when upgrading PGO.

Upgrading from PGO v5.0.0 Using Kustomize

Starting with PGO v5.0.1, both the Deployment and ServiceAccount created when installing PGO via the installers in the Postgres Operator examples repository have been renamed from `postgres-operator` to `pgo`. As a result of this change, if using Kustomize to install PGO and upgrading from PGO v5.0.0, the following step must be completed prior to upgrading. This will ensure multiple versions of PGO are not installed and running concurrently within your Kubernetes environment.

Prior to upgrading PGO, first manually delete the PGO v5.0.0 `postgres-operator` Deployment and ServiceAccount:

```
kubectl -n postgres-operator delete deployment,serviceaccount postgres-operator
```

Then, once both the Deployment and ServiceAccount have been deleted, proceed with upgrading PGO by applying the new version of the Kustomize installer:

```
kubectl apply -k kustomize/install/bases
```

Upgrading from PGO v5.0.2 and Below

As a result of changes to pgBackRest dedicated repository host deployments in PGO v5.0.3 (please see the [PGO v5.0.3 release notes]({{< relref "../releases/5.0.3.md" >}}) for more details), reconciliation of a pgBackRest dedicated repository host might become stuck with the following error (as shown in the PGO logs) following an upgrade from PGO versions v5.0.0 through v5.0.2:

```
StatefulSet.apps \"hippo-repo-host\" is invalid: spec: Forbidden: updates to statefulset spec for
   fields other than 'replicas', 'template', 'updateStrategy' and 'minReadySeconds' are forbidden
```

If this is the case, proceed with deleting the pgBackRest dedicated repository host StatefulSet, and PGO will then proceed with recreating and reconciling the dedicated repository host normally:

```
kubectl delete sts hippo-repo-host
```

Additionally, please be sure to update and apply all PostgresCluster custom resources in accordance with any applicable spec changes described in the [PGO v5.0.3 release notes]({{< relref "../releases/5.0.3.md" >}}).

The PGO Monitoring stack is a fully integrated solution for monitoring and visualizing metrics captured from PostgreSQL clusters created using PGO. By leveraging pgMonitor to configure and integrate the various tools, components and metrics needed to effectively monitor PostgreSQL clusters, PGO Monitoring provides an powerful and easy-to-use solution to effectively monitor and visualize pertinent PostgreSQL database and container metrics. Included in the monitoring infrastructure are the following components:

- pgMonitor - Provides the configuration needed to enable the effective capture and visualization of PostgreSQL database metrics using the various tools comprising the PostgreSQL Operator Monitoring infrastructure
- Grafana - Enables visual dashboard capabilities for monitoring PostgreSQL clusters, specifically using Crunchy PostgreSQL Exporter data stored within Prometheus
- Prometheus - A multi-dimensional data model with time series data, which is used in collaboration with the Crunchy PostgreSQL Exporter to provide and store metrics
- Alertmanager - Handles alerts sent by Prometheus by deduplicating, grouping, and routing them to receiver integrations.

By leveraging the installation method described in this section, PGO Monitoring can be deployed alongside PGO.

Installing PGO Monitoring Using Kustomize

This section provides instructions for installing and configuring PGO Monitoring using Kustomize.

Prerequisites

First, go to GitHub and fork the Postgres Operator examples repository, which contains the PGO Monitoring Kustomize installer.

https://github.com/CrunchyData/postgres-operator-examples/fork

Once you have forked this repo, you can download it to your working environment with a command similar to this:

```
YOUR_GITHUB_UN="<your GitHub username>"
git clone --depth 1 "git@github.com:${YOUR_GITHUB_UN}/postgres-operator-examples.git"
cd postgres-operator-examples
```

The PGO Monitoring project is located in the `kustomize/monitoring` directory.

Configuration

While the default Kustomize install should work in most Kubernetes environments, it may be necessary to further customize the project according to your specific needs.

For instance, by default `fsGroup` is set to `26` for the `securityContext` defined for the various Deployments comprising the PGO Monitoring stack:

```
securityContext:
  fsGroup: 26
```

In most Kubernetes environments this setting is needed to ensure processes within the container have the permissions needed to write to any volumes mounted to each of the Pods comprising the PGO Monitoring stack. However, when installing in an OpenShift environment (and more specifically when using the `restricted` Security Context Constraint), the `fsGroup` setting should be removed since OpenShift will automatically handle setting the proper `fsGroup` within the Pod's `securityContext`.

Additionally, within this same section it may also be necessary to modify the `supplmentalGroups` setting according to your specific storage configuration:

```
securityContext:
  supplementalGroups : 65534
```

Therefore, the following files (located under `kustomize/monitoring`) should be modified and/or patched (e.g. using additional overlays) as needed to ensure the `securityContext` is properly defined for your Kubernetes environment:

- `deploy-alertmanager.yaml`
- `deploy-grafana.yaml`
- `deploy-prometheus.yaml`

And to modify the configuration for the various storage resources (i.e. PersistentVolumeClaims) created by the PGO Monitoring installer, the `kustomize/monitoring/pvcs.yaml` file can also be modified.

Additionally, it is also possible to further customize the configuration for the various components comprising the PGO Monitoring stack (Grafana, Prometheus and/or AlertManager) by modifying the following configuration resources:

- `alertmanager-config.yaml`
- `alertmanager-rules-config.yaml`
- `grafana-datasources.yaml`
- `prometheus-config.yaml`

Finally, please note that the default username and password for Grafana can be updated by modifying the Grafana Secret in file `kustomize/monitoring/grafana-secret.yaml`.

Install

Once the Kustomize project has been modified according to your specific needs, PGO Monitoring can then be installed using `kubectl` and Kustomize:

```
kubectl apply -k kustomize/monitoring
```

Uninstall

And similarly, once PGO Monitoring has been installed, it can uninstalled using `kubectl` and Kustomize:

```
kubectl delete -k kustomize/monitoring
```

This section contains guides on handling various scenarios when managing Postgres clusters using PGO, the Postgres Operator. These include step-by-step instructions for situations such as migrating data to a PGO managed Postgres cluster or upgrading from an older version of PGO.

These guides are in no particular order: choose the guide that is most applicable to your situation.

If you are looking for how to manage most day-to-day Postgres scenarios, we recommend first going through the [Tutorial]({{< relref "tutorial/_index.md" >}}).

You can upgrade from PGO v4 to PGO v5 through a variety of methods by following this guide. There are several methods that can be used to upgrade: we present these methods based upon a variety of factors, including:

- Redundancy / ability to roll back
- Available resources
- Downtime preferences

and others.

These methods include:

- *Migrating Using Data Volumes*. This allows you to migrate from v4 to v5 using the existing data volumes that you created in v4. This is the simplest method for upgrade and is the most resource efficient, but you will have a greater potential for downtime using this method.
- *Migrate From Backups*. This allows you to create a Postgres cluster with v5 from the backups taken with v4. This provides a way for you to create a preview of your Postgres cluster through v5, but you would need to take your applications offline to ensure all the data is migrated.
- *Migrate Using a Standby Cluster*. This allows you to run a v4 and a v5 Postgres cluster in parallel, with data replicating from the v4 cluster to the v5 cluster. This method minimizes downtime and lets you preview your v5 environment, but is the most resource intensive.

You should choose the method that makes the most sense for your environment. Each method is described in detail below.

Prerequisites

There are several prerequisites for using any of these upgrade methods.

- PGO v4 is currently installed within the Kubernetes cluster, and is actively managing any existing v4 PostgreSQL clusters.
- Any PGO v4 clusters being upgraded have been properly initialized using PGO v4, which means the v4 `pgcluster` custom resource should be in a `pgcluster Initialized` status:

```
$ kubectl get pgcluster hippo -o jsonpath='{ .status }'
{"message":"Cluster has been initialized","state":"pgcluster Initialized"}
```

- The PGO v4 `pgo` client is properly configured and available for use.
- PGO v5 is currently [installed]({{< relref "installation/_index.md" >}}) within the Kubernetes cluster.

For these examples, we will use a Postgres cluster named `hippo`.

Upgrade Method #1: Data Volumes

This upgrade method allows you to migrate from PGO v4 to PGO v5 using the existing data volumes that were created in PGO v4. Note that this is an "in place" migration method: this will immediately move your Postgres clusters from being managed by PGO v4 and PGO v5. If you wish to have some failsafes in place, please use one of the other migration methods. Please also note that you will need to perform the cluster upgrade in the same namespace as the original cluster in order for your v5 cluster to access the existing PVCs.

Step 1: Prepare the PGO v4 Cluster for Migration

You will need to set up your PGO v4 Postgres cluster so that it can be migrated to a PGO v5 cluster. The following describes how to set up a PGO v4 cluster for using this migration method.

1. Scale down any existing replicas within the cluster. This will ensure that the primary PVC does not change again prior to the upgrade.

You can get a list of replicas using the `pgo scaledown --query` command, e.g.:

```
pgo scaledown hippo --query
```

If there are any replicas, you will see something similar to:

```
Cluster: hippo
REPLICA                  STATUS          NODE ...
hippo                    running         node01 ...
```

Scaledown any replicas that are running in this cluser, e.g.:

```
pgo scaledown hippo --target=hippo
```

2. Once all replicas are removed and only the primary remains, proceed with deleting the cluster while retaining the data and backups. You can do this `--keep-data` and `--keep-backups` flags:

You MUST run this command with the `--keep-data` and `--keep-backups` flag otherwise you risk deleting ALL of your data.

```
pgo delete cluster hippo --keep-data --keep-backups
```

3. The PVC for the primary Postgres instance and the pgBackRest repository should still remain. You can verify this with the command below:

```
kubectl get pvc --selector=pg-cluster=hippo
```

This should yield something similar to:

```
NAME               STATUS   VOLUME ...
hippo-jgut         Bound    pvc-a0b89bdb- ...
hippo-pgbr-repo    Bound    pvc-25501671- …
```

A third PVC used to store write-ahead logs (WAL) may also be present if external WAL volumes were enabled for the cluster.

Step 2: Migrate to PGO v5

With the PGO v4 cluster's volumes prepared for the move to PGO v5, you can now create a [`PostgresCluster`]({{< relref "references/crd.md" >}}) custom resource using these volumes. This migration method does not carry over any specific configurations or customizations from PGO v4: you will need to create the specific `PostgresCluster` configuration that you need.

Additional steps are required to set proper file permissions when using certain storage options, such as NFS and HostPath storage, due to a known issue with how fsGroups are applied. When migrating from PGO v4, this will require the user to manually set the group value of the pgBackRest repo directory, and all subdirectories, to `26` to match the `postgres` group used in PGO v5. Please see here for more information.

To complete the upgrade process, your `PostgresCluster` custom resource **MUST** include the following:

1. A `volumes` data source that points to the PostgreSQL data, PostgreSQL WAL (if applicable) and pgBackRest repository PVCs identified in the `spec.dataSource.volumes` section.

For example, using the `hippo` cluster:

```
spec:
  dataSource:
    volumes:
      pgDataVolume:
        pvcName: hippo-jgut
        directory: "hippo-jgut"
      pgBackRestVolume:
        pvcName: hippo-pgbr-repo
        directory: "hippo-backrest-shared-repo"
      # only specify external WAL PVC if enabled in PGO v4 cluster
      # pgWALVolume:
      #  pvcName: hippo-jgut-wal
```

Please see the [Data Migration]({{< relref "guides/data-migration.md" >}}) section of the [tutorial]({{< relref "tutorial/_index.md" >}}) for more details on how to properly populate this section of the spec when migrating from a PGO v4 cluster.

2. If you customized Postgres parameters, you will need to ensure they match in the PGO v5 cluster. For more information, please review the tutorial on [customizing a Postgres cluster]({{< relref "tutorial/customize-cluster.md" >}}).

3. Once the `PostgresCluster` spec is populated according to these guidelines, you can create the `PostgresCluster` custom resource. For example, if the `PostgresCluster` you're creating is a modified version of the `postgres` example in the PGO examples repo, you can run the following command:

```
kubectl apply -k examples/postgrescluster
```

Your upgrade is now complete! You should now remove the `spec.dataSource.volumes` section from your `PostgresCluster`. For more information on how to use PGO v5, we recommend reading through the [PGO v5 tutorial]({{< relref "tutorial/_index.md" >}}).

Upgrade Method #2: Backups

This upgrade method allows you to migrate from PGO v4 to PGO v5 by creating a new PGO v5 Postgres cluster using a backup from a PGO v4 cluster. This method allows you to preserve the data in your PGO v4 cluster while you transition to PGO v5. To fully move the data over, you will need to incur downtime and shut down your PGO v4 cluster.

Step 1: Prepare the PGO v4 Cluster for Migration

1. Ensure you have a recent backup of your cluster. You can do so with the `pgo backup` command, e.g.:

```
pgo backup hippo
```

Please ensure that the backup completes. You will see the latest backup appear using the `pgo show backup` command.

2. Next, delete the cluster while keeping backups (using the `--keep-backups` flag):

```
pgo delete cluster hippo --keep-backups
```

Additional steps are required to set proper file permissions when using certain storage options, such as NFS and HostPath storage, due to a known issue with how fsGroups are applied. When migrating from PGO v4, this will require the user to manually set the group value of the pgBackRest repo directory, and all subdirectories, to `26` to match the `postgres` group used in PGO v5. Please see here for more information.

Step 2: Migrate to PGO v5

With the PGO v4 Postgres cluster's backup repository prepared, you can now create a [`PostgresCluster`]({{< relref "references/crd.md" >}}) custom resource. This migration method does not carry over any specific configurations or customizations from PGO v4: you will need to create the specific `PostgresCluster` configuration that you need.

To complete the upgrade process, your `PostgresCluster` custom resource **MUST** include the following:

1. You will need to configure your pgBackRest repository based upon whether you are using a PVC to store your backups, or an object storage system such as S3/GCS. Please follow the directions based upon the repository type you are using as part of the migration.

PVC-based Backup Repository

When migrating from a PVC-based backup repository, you will need to configure a pgBackRest repo of a `spec.backups.pgbackrest.repos.volume` under the `spec.backups.pgbackrest.repos.name` of `repo1`. The `volumeClaimSpec` should match the attributes of the pgBackRest repo PVC being used as part of the migration, i.e. it must have the same `storageClassName`, `accessModes`, `resources`, etc. Please note that you will need to perform the cluster upgrade in the same namespace as the original cluster in order for your v5 cluster to access the existing PVCs. For example:

```
spec:
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              storageClassName: standard-wffc
              accessModes:
              - "ReadWriteOnce"
              resources:
                requests:
                  storage: 1Gi
```

S3 / GCS Backup Repository When migrating from a S3 or GCS based backup repository, you will need to configure your `spec.backups.pgbackrest.repos.volume` to point to the backup storage system. For instance, if AWS S3 storage is being utilized, the repo would be defined similar to the following:

```
spec:
  backups:
    pgbackrest:
      repos:
        - name: repo1
          s3:
            bucket: hippo
            endpoint: s3.amazonaws.com
            region: us-east-1
```

Any required secrets or desired custom pgBackRest configuration should be created and configured as described in the [backup tutorial]({{< relref "tutorial/backups.md" >}}).

You will also need to ensure that the "pgbackrest-repo-path" configured for the repository matches the path used by the PGO v4 cluster. The default repository path follows the pattern `/backrestrepo/<clusterName>-backrest-shared-repo`. Note that the path name here is different than migrating from a PVC-based repository.

Using the `hippo` Postgres cluster as an example, you would set the following in the `spec.backups.pgbackrest.global` section:

```
spec:
  backups:
    pgbackrest:
      global:
        repo1-path: /backrestrepo/hippo-backrest-shared-repo
```

2. Set the `spec.dataSource` section to restore from the backups used for this migration. For example:

```
spec:
  dataSource:
    postgresCluster:
      repoName: repo1
```

You can also provide other pgBackRest restore options, e.g. if you wish to restore to a specific point-in-time (PITR).

3. If you are using a PVC-based pgBackRest repository, then you will also need to specify a pgBackRestVolume data source that references the PGO v4 pgBackRest repository PVC:

```
spec:
  dataSource:
    volumes:
      pgBackRestVolume:
        pvcName: hippo-pgbr-repo
        directory: "hippo-backrest-shared-repo"
    postgresCluster:
      repoName: repo1
```

4. If you customized other Postgres parameters, you will need to ensure they match in the PGO v5 cluster. For more information, please review the tutorial on [customizing a Postgres cluster]({{< relref "tutorial/customize-cluster.md" >}}).

5. Once the `PostgresCluster` spec is populated according to these guidelines, you can create the `PostgresCluster` custom resource. For example, if the `PostgresCluster` you're creating is a modified version of the `postgres` example in the PGO examples repo, you can run the following command:

```
kubectl apply -k examples/postgrescluster
```

WARNING: Once the PostgresCluster custom resource is created, it will become the owner of the PVC. *This means that if the PostgresCluster is then deleted (e.g. if attempting to revert back to a PGO v4 cluster), then the PVC will be deleted as well.*

If you wish to protect against this, first remove the reference to the pgBackRest PVC in the PostgresCluster spec:

```
kubectl patch postgrescluster hippo-pgbr-repo --type='json' -p='[{"op": "remove", "path":
   "/spec/dataSource/volumes"}]'
```

Then relabel the PVC prior to deleting the PostgresCluster custom resource. Below uses the `hippo` Postgres cluster as an example:

```
kubectl label pvc hippo-pgbr-repo \
  postgres-operator.crunchydata.com/cluster- \
  postgres-operator.crunchydata.com/pgbackrest-repo- \
  postgres-operator.crunchydata.com/pgbackrest-volume- \
  postgres-operator.crunchydata.com/pgbackrest-
```

You will also need to remove all ownership references from the PVC:

```
kubectl patch pvc hippo-pgbr-repo --type='json' -p='[{"op": "remove", "path":
   "/metadata/ownerReferences"}]'
```

It is recommended to set the reclaim policy for any PV's bound to existing PVC's to `Retain` to ensure data is retained in the event a PVC is accidentally deleted during the upgrade.

Your upgrade is now complete! For more information on how to use PGO v5, we recommend reading through the [PGO v5 tutorial]({{< relref "tutorial/_index.md" >}}).

Upgrade Method #3: Standby Cluster

This upgrade method allows you to migrate from PGO v4 to PGO v5 by creating a new PGO v5 Postgres cluster in a "standby" mode, allowing it to mirror the PGO v4 cluster and continue to receive data updates in real time. This has the advantage of being able to fully inspect your PGO v5 Postgres cluster while leaving your PGO v4 cluster up and running, thus minimizing downtime when you cut over. The tradeoff is that you will temporarily use more resources while this migration is occurring.

This method only works if your PGO v4 cluster uses S3 or an S3-compatible storage system, or GCS. For more information on standby clusters, please refer to the [tutorial]({{< relref "tutorial/disaster-recovery.md" >}}#standby-cluster).

Step 1: Migrate to PGO v5

Create a [`PostgresCluster`]({{< relref "references/crd.md" >}}) custom resource. This migration method does not carry over any specific configurations or customizations from PGO v4: you will need to create the specific `PostgresCluster` configuration that you need.

To complete the upgrade process, your `PostgresCluster` custom resource **MUST** include the following:

1. Configure your pgBackRest to use an object storage system such as S3/GCS. You will need to configure your `spec.backups.pgbackrest.re` to point to the backup storage system. For instance, if AWS S3 storage is being utilized, the repo would be defined similar to the following:

```
spec:
  backups:
    pgbackrest:
      repos:
        - name: repo1
          s3:
            bucket: hippo
            endpoint: s3.amazonaws.com
            region: us-east-1
```

Any required secrets or desired custom pgBackRest configuration should be created and configured as described in the [backup tutorial]({{< relref "tutorial/backups.md" >}}).

You will also need to ensure that the "pgbackrest-repo-path" configured for the repository matches the path used by the PGO v4 cluster. The default repository path follows the pattern `/backrestrepo/<clusterName>-backrest-shared-repo`. Note that the path name here is different than migrating from a PVC-based repository.

Using the `hippo` Postgres cluster as an example, you would set the following in the `spec.backups.pgbackrest.global` section:

```
spec:
  backups:
    pgbackrest:
      global:
        repo1-path: /backrestrepo/hippo-backrest-shared-repo
```

2. A `spec.standby` cluster configuration within the spec that is populated according to the name of pgBackRest repo configured in the spec. For example:

```
spec:
  standby:
    enabled: true
    repoName: repo1
```

3. If you customized other Postgres parameters, you will need to ensure they match in the PGO v5 cluster. For more information, please review the tutorial on [customizing a Postgres cluster]({{< relref "tutorial/customize-cluster.md" >}}).

4. Once the `PostgresCluster` spec is populated according to these guidelines, you can create the `PostgresCluster` custom resource. For example, if the `PostgresCluster` you're creating is a modified version of the `postgres` example in the PGO examples repo, you can run the following command:

```
kubectl apply -k examples/postgrescluster
```

5. Once the standby cluster is up and running and you are satisfied with your set up, you can promote it.

First, you will need to shut down your PGO v4 cluster. You can do so with the following command, e.g.:

```
pgo update cluster hippo --shutdown
```

You can then update your PGO v5 cluster spec to promote your standby cluster:

```
spec:
  standby:
    enabled: false
```

Note: When the v5 cluster is running in non-standby mode, you will not be able to restart the v4 cluster, as that data is now being managed by the v5 cluster.

Your upgrade is now complete! Once you verify that the PGO v5 cluster is running and you have recorded the user credentials from the v4 cluster, you can remove the old cluster:

```
pgo delete cluster hippo
```

For more information on how to use PGO v5, we recommend reading through the [PGO v5 tutorial]({{< relref "tutorial/_index.md" >}}).

Additional Considerations

Upgrading to PGO v5 may result in a base image upgrade from EL-7 (UBI / CentOS) to EL-8 (UBI / CentOS). Based on the contents of your Postgres database, you may need to perform additional steps.

Due to changes in the GNU C library (`glibc`) in EL-8, you may need to reindex certain indexes in your Postgres cluster. For more information, please read the PostgreSQL Wiki on Locale Data Changes, how you can determine if your indexes are affected, and how to fix them.

There are certain cases where you may want to migrate existing volumes to a new cluster. If so, read on for an in depth look at the steps required.

Configure your PostgresCluster CRD

In order to use existing pgData, pg_wal or pgBackRest repo volumes in a new PostgresCluster, you will need to configure the `spec.dataSource.volumes` section of your PostgresCluster CRD. As shown below, there are three possible volumes you may configure, `pgDataVolume`, `pgWALVolume` and `pgBackRestVolume`. Under each, you must define the PVC name to use in the new cluster. A directory may also be defined, as needed, for cases where the existing directory name does not match the v5 directory.

To help explain how these fields are used, we will consider a `pgcluster` from PGO v4, `oldhippo`. We will assume that the `pgcluster` has been deleted and only the PVCs have been left in place.

Please note that any differences in configuration or other datasources will alter this procedure significantly and that certain storage options require additional steps (see *Considerations* below)!

In a standard PGO v4.7 cluster, a primary database pod with a separate pg_wal PVC will mount its pgData PVC, named "oldhippo", at `/pgdata` and its pg_wal PVC, named "oldhippo-wal", at `/pgwal` within the pod's file system. In this pod, the standard pgData directory will be `/pgdata/oldhippo` and the standard pg_wal directory will be `/pgwal/oldhippo-wal`. The pgBackRest repo pod will mount its PVC at `/backrestrepo` and the repo directory will be `/backrestrepo/oldhippo-backrest-shared-repo`.

With the above in mind, we need to reference the three PVCs we wish to migrate in the `dataSource.volumes` portion of the PostgresCluster spec. Additionally, to accommodate the PGO v5 file structure, we must also reference the pgData and pgBackRest repo directories. Note that the pg_wal directory does not need to be moved when migrating from v4 to v5!

Now, we just need to populate our CRD with the information described above:

```
spec:
  dataSource:
    volumes:
      pgDataVolume:
        pvcName: oldhippo
        directory: oldhippo
      pgWALVolume:
        pvcName: oldhippo-wal
      pgBackRestVolume:
        pvcName: oldhippo-pgbr-repo
        directory: oldhippo-backrest-shared-repo
```

Lastly, it is very important that the PostgreSQL version and storage configuration in your PostgresCluster match *exactly* the existing volumes being used.

If the volumes were used with PostgreSQL 13, the `spec.postgresVersion` value should be `13` and the associated `spec.image` value should refer to a PostgreSQL 13 image.

Similarly, the configured data volume definitions in your PostgresCluster spec should match your existing volumes. For example, if the existing pgData PVC has a RWO access mode and is 1 Gigabyte, the relevant `dataVolumeClaimSpec` should be configured as

```
dataVolumeClaimSpec:
  accessModes:
  - "ReadWriteOnce"
  resources:
    requests:
      storage: 1G
```

With the above configuration in place, your existing PVC will be used when creating your PostgresCluster. They will be given appropriate Labels and ownership references, and the necessary directory updates will be made so that your cluster is able to find the existing directories.

Considerations

- Additional steps are required to set proper file permissions when using certain storage options, such as NFS and HostPath storage due to a known issue with how fsGroups are applied. When migrating from PGO v4, this will require the user to manually set the group value of the pgBackRest repo directory, and all subdirectories, to `26` to match the `postgres` group used in PGO v5. Please see here for more information.
- An existing pg_wal volume is not required when the pg_wal directory is located on the same PVC as the pgData directory.
- When using existing pg_wal volumes, an existing pgData volume **must** also be defined to ensure consistent naming and proper bootstrapping.
- When migrating from PGO v4 volumes, it is recommended to use the most recently available version of PGO v4.
- As there are many factors that may impact this procedure, it is strongly recommended that a test run be completed beforehand to ensure successful operation.

Putting it all together

Now that we've identified all of our volumes and required directories, we're ready to create our new cluster!

Below is a complete PostgresCluster that includes everything we've talked about. After your `PostgreCluster` is created, you should remove the `spec.dataSource.volumes` section.

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: oldhippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  dataSource:
    volumes:
      pgDataVolume:
        pvcName: oldhippo
        directory: oldhippo
      pgWALVolume:
        pvcName: oldhippo-wal
      pgBackRestVolume:
        pvcName: oldhippo-pgbr-repo
        directory: oldhippo-backrest-shared-repo
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1G
      walVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1G
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1G
```

PGO uses persistent volumes to store Postgres data and, based on your configuration, data for backups, archives, etc. There are cases where you may want to retain your volumes for [later use]({{< relref "./data-migration.md" >}}).

The below guide shows how to configure your persistent volumes (PVs) to remain after a Postgres cluster managed by PGO is deleted and to deploy the retained PVs to a new Postgres cluster.

For the purposes of this exercise, we will use a Postgres cluster named `hippo`.

Modify Persistent Volume Retention

Retention of persistent volumes is set using a reclaim policy. By default, more persistent volumes have a policy of `Delete`, which removes any data on a persistent volume once there are no more persistent volume claims (PVCs) associated with it.

To retain a persistent volume you will need to set the reclaim policy to `Retain`. Note that persistent volumes are cluster-wide objects, so you will need to appropriate permissions to be able to modify a persistent volume.

To retain the persistent volume associated with your Postgres database, you must first determine which persistent volume is associated with the persistent volume claim for your database. First, local the persistent volume claim. For example, with the `hippo` cluster, you can do so with the following command:

```
kubectl get pvc -n postgres-operator
   --selector=postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/data=
```

This will yield something similar to the below, which are the PVCs associated with any Postgres instance:

```
NAME                          STATUS   VOLUME                                     CAPACITY
   ACCESS MODES   STORAGECLASS   AGE
hippo-instance1-x9vq-pgdata   Bound    pvc-aef7ee64-4495-4813-b896-8a67edc53e58   1Gi        RWO
               standard       6m53s
```

The `VOLUME` column contains the name of the persistent volume. You can inspect it using `kubectl get pv`, e.g.:

```
kubectl get pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58
```

which should yield:

```
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
   CLAIM                                           STORAGECLASS   REASON   AGE
pvc-aef7ee64-4495-4813-b896-8a67edc53e58   1Gi        RWO            Delete           Bound
   postgres-operator/hippo-instance1-x9vq-pgdata   standard                8m10s
```

To modify the reclaim policy set it to `Retain`, you can run a command similar to this:

```
kubectl patch pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58  -p
   '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
```

Verify that the change occurred:

```
kubectl get pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58
```

should show that `Retain` is set in the `RECLAIM POLICY` column:

```
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
   CLAIM                                          STORAGECLASS   REASON   AGE
pvc-aef7ee64-4495-4813-b896-8a67edc53e58   1Gi        RWO            Retain           Bound
   postgres-operator/hippo-instance1-x9vq-pgdata   standard                9m53s
```

Delete Postgres Cluster, Retain Volume

This is a potentially destructive action. Please be sure that your volume retention is set correctly and/or you have backups in place to restore your data.

[Delete your Postgres cluster]({{< relref "tutorial/delete-cluster.md" >}}). You can delete it using the manifest or with a command similar to:

```
kubectl -n postgres-operator delete postgrescluster hippo
```

Wait for the Postgres cluster to finish deleting. You should then verify that the persistent volume is still there:

```
kubectl get pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58
```

should yield:

```
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
   CLAIM                                          STORAGECLASS   REASON   AGE
pvc-aef7ee64-4495-4813-b896-8a67edc53e58   1Gi        RWO            Retain           Released
   postgres-operator/hippo-instance1-x9vq-pgdata   standard                21m
```

Create Postgres Cluster With Retained Volume

You can now create a new Postgres cluster with the retained volume. First, to aid the process, you will want to provide a label that is unique for your persistent volumes so we can identify it in the manifest. For example:

```
kubectl label pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58
   pgo-postgres-cluster=postgres-operator-hippo
```

(This label uses the format `<namespace>-<clusterName>`).

Next, you will need to reference this persistent volume in your Postgres cluster manifest. For example:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
        selector:
          matchLabels:
            pgo-postgres-cluster: postgres-operator-hippo
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Wait for the Pods to come up. You may see the Postgres Pod is in a `Pending` state. You will need to go in and clear the claim on the persistent volume that you want to use for this Postgres cluster, e.g.:

```
kubectl patch pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58  -p '{"spec":{"claimRef": null}}'
```

After that, your Postgres cluster will come up and will be using the previously used persistent volume!

If you ultimately want the volume to be deleted, you will need to revert the reclaim policy to `Delete`, e.g.:

```
kubectl patch pv pvc-aef7ee64-4495-4813-b896-8a67edc53e58  -p
   '{"spec":{"persistentVolumeReclaimPolicy":"Delete"}}'
```

After doing that, the next time you delete your Postgres cluster, the volume and your data will be deleted.

Additional Notes on Storage Retention

Systems using "hostpath" storage or a storage class that does not support label selectors may not be able to use the label selector method for using a retained volume volume. You would have to specify the `volumeName` directly, e.g.:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
        volumeName: "pvc-aef7ee64-4495-4813-b896-8a67edc53e58"
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Additionally, to add additional replicas to your Postgres cluster, you will have to make changes to your spec. You can do one of the following:

1. Remove the volume-specific configuration from the volume claim spec (e.g. delete `spec.instances.selector` or `spec.instances.volu`
2. Add a new instance set specifically for your replicas, e.g.:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
      selector:
        matchLabels:
          pgo-postgres-cluster: postgres-operator-hippo
    - name: instance2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Logical replication is a Postgres feature that provides a convenient way for moving data between databases, particularly Postgres clusters that are in an active state.

You can set up your PGO managed Postgres clusters to use logical replication. This guide provides an example for how to do so.

Set Up Logical Replication

This example creates two separate Postgres clusters named `hippo` and `rhino`. We will logically replicate data from `rhino` to `hippo`. We can create these two Postgres clusters using the manifests below:

```
---
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
---
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: rhino
spec:
  image: {{< param imageCrunchyPostgres >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imageCrunchyPGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
  users:
    - name: logic
      databases:
        - zoo
      options: "REPLICATION"
```

The key difference between the two Postgres clusters is this section in the `rhino` manifest:

```
users:
  - name: logic
    databases:
      - zoo
    options: "REPLICATION"
```

This creates a database called `zoo` and a user named `logic` with `REPLICATION` privileges. This will allow for replicating data logically to the `hippo` Postgres cluster.

Create these two Postgres clusters. When the `rhino` cluster is ready, [log into the `zoo` database]({{< relref "tutorial/connect-cluster.md" >}}). For convenience, you can use the `kubectl exec` method of logging in:

```
kubectl exec -it -n postgres-operator -c database \
  $(kubectl get pods -n postgres-operator
     --selector='postgres-operator.crunchydata.com/cluster=rhino,postgres-operator.crunchydata.com/ro
     -o name) -- psql zoo
```

Let's create a simple table called `abc` that contains just integer data. We will also populate this table:

```
CREATE TABLE abc (id int PRIMARY KEY);
INSERT INTO abc SELECT * FROM generate_series(1,10);
```

We need to grant SELECT privileges to the logic user in order for it to perform an initial data synchronization during logical replication. You can do so with the following command:

```
GRANT SELECT ON abc TO logic;
```

Finally, create a publication that allows for the replication of data from abc:

```
CREATE PUBLICATION zoo FOR ALL TABLES;
```

Quit out of the rhino Postgres cluster.

For the next step, you will need to get the connection information for how to connection as the logic user to the rhino Postgres database. You can get the key information from the following commands, which return the hostname, username, and password:

```
kubectl -n postgres-operator get secrets rhino-pguser-logic -o jsonpath={.data.host} | base64 -d
kubectl -n postgres-operator get secrets rhino-pguser-logic -o jsonpath={.data.user} | base64 -d
kubectl -n postgres-operator get secrets rhino-pguser-logic -o jsonpath={.data.password} | base64
   -d
```

The host will be something like rhino-primary.postgres-operator.svc and the user will be logic. Further down, the guide references the password as <LOGIC-PASSWORD>. You can substitute the actual password there.

Log into the hippo Postgres cluster. Note that we are logging into the postgres database within the hippo cluster:

```
kubectl exec -it -n postgres-operator -c database \
  $(kubectl get pods -n postgres-operator
     --selector='postgres-operator.crunchydata.com/cluster=hippo,postgres-operator.crunchydata.com/ro
     -o name) -- psql
```

Create a table called abc that is identical to the table in the rhino database:

```
CREATE TABLE abc (id int PRIMARY KEY);
```

Finally, create a subscription that will manage the data replication from rhino into hippo:

```
CREATE SUBSCRIPTION zoo
    CONNECTION 'host=rhino-primary.postgres-operator.svc user=logic dbname=zoo
       password=<LOGIC-PASSWORD>'
    PUBLICATION zoo;
```

In a few moments, you should see the data replicated into your table:

```
TABLE abc;
```

which yields:

```
 id
----
  1
  2
  3
  4
  5
  6
  7
  8
  9
  10
(10 rows)
```

You can further test that logical replication is working by modifying the data on rhino in the abc table, and the verifying that it is replicated into hippo.

Extensions combine functions, data types, casts, etc. – everything you need to add some new feature to PostgreSQL in an easy to install package. How easy to install? For many extensions, like the fuzzystrmatch extension, it's as easy as connecting to the database and running a command like this:

```
CREATE EXTENSION fuzzystrmatch;
```

However, in other cases, an extension might require additional configuration management. PGO lets you add those configurations to the `PostgresCluster` spec easily.

PGO also allows you to add a custom databse initialization script in case you would like to automate how and where the extension is installed.

This guide will walk through adding custom configuration for an extension and automating installation, using the example of Crunchy Data's own `pgnodemx` extension.

- pgnodemx

pgnodemx

`pgnodemx` is a PostgreSQL extension that is able to pull container-specific metrics (e.g. CPU utilization, memory consumption) from the container itself via SQL queries.

In order to do this, `pgnodemx` requires information from the Kubernetes DownwardAPI to be mounted on the PostgreSQL pods. Please see the `pgnodemx and the DownwardAPI` section of the [backup architecture]]({{< relref "architecture/backups.md" >}}) page for more information on where and how the DownwardAPI is mounted.

`pgnodemx` Configuration

To enable the `pdnodemx` extension, we need to set certain configurations. Luckily, this can all be done directly through the spec:

```
spec:
  patroni:
    dynamicConfiguration:
      postgresql:
        parameters:
          shared_preload_libraries: pgnodemx
          pgnodemx.kdapi_enabled: on
          pgnodemx.kdapi_path: /etc/database-containerinfo
```

Those three settings will

- load `pgnodemx` at start;
- enable the `kdapi` functions (which are specific to the capture of Kubernetes DownwardAPI information);
- tell `pgnodemx` where those DownwardAPI files are mounted (at the `/etc/dabatase-containerinfo` path).

If you create a `PostgresCluster` with those configurations, you will be able to connect, create the extension in a database, and run the functions installed by that extension:

```
CREATE EXTENSION pgnodemx;
SELECT * FROM proc_diskstats();
```

Automating `pgnodemx` Creation

Now that you know how to configure `pgnodemx`, let's say you want to automate the creation of the extension in a particular database, or in all databases. We can do that through a custom database initialization.

First, we have to create a ConfigMap with the initialization SQL. Let's start with the case where we want `pgnodemx` created for us in the `hippo` database. Our initialization SQL file might be named `init.sql` and look like this:

```
\c hippo\\
CREATE EXTENSION pgnodemx;
```

Now we create the ConfigMap from that file in the same namespace as our PostgresCluster will be created:

```
kubectl create configmap hippo-init-sql -n postgres-operator --from-file=init.sql=path/to/init.sql
```

You can check that the ConfigMap was created and has the right information:

```
kubectl get configmap -n postgres-operator hippo-init-sql -o yaml

apiVersion: v1
data:
  init.sql: |-
    \c hippo\\
    CREATE EXTENSION pgnodemx;
kind: ConfigMap
metadata:
  name: hippo-init-sql
  namespace: postgres-operator
```

Now, in addition to the spec changes we made above to allow `pgnodemx` to run, we add that ConfigMap's information to the PostgresCluster spec: the name of the ConfigMap (`hippo-init-sql`) and the key for the data (`init.sql`):

```
spec:
  databaseInitSQL:
    key: init.sql
    name: hippo-init-sql
```

Apply that spec to a new or existing PostgresCluster, and the pods should spin up with `pgnodemx` already installed in the `hippo` database.

The goal of PGO, the Postgres Operator from Crunchy Data is to provide a means to quickly get your applications up and running on Postgres for both development and production environments. To understand how PGO does this, we want to give you a tour of its architecture, with explains both the architecture of the PostgreSQL Operator itself as well as recommended deployment models for PostgreSQL in production!

PGO Architecture

The Crunchy PostgreSQL Operator extends Kubernetes to provide a higher-level abstraction for rapid creation and management of PostgreSQL clusters. The Crunchy PostgreSQL Operator leverages a Kubernetes concept referred to as "Custom Resources" to create several custom resource definitions (CRDs) that allow for the management of PostgreSQL clusters.

The main custom resource definition is [`postgresclusters.postgres-operator.crunchydata.com`]({{< relref "references/crd.md" >}}). This allows you to control all the information about a Postgres cluster, including:

- General information
- Resource allocation
- High availability
- Backup management
- Where and how it is deployed (affinity, tolerations, topology spread constraints)
- Disaster Recovery / standby clusters
- Monitoring

and more.

PGO itself runs as a Deployment and is composed of a single container.

- `operator` (image: postgres-operator) - This is the heart of the PostgreSQL Operator. It contains a series of Kubernetes controllers that place watch events on a series of native Kubernetes resources (Jobs, Pods) as well as the Custom Resources that come with the PostgreSQL Operator (Pgcluster, Pgtask)

The main purpose of PGO is to create and update information around the structure of a Postgres Cluster, and to relay information about the overall status and health of a PostgreSQL cluster. The goal is to also simplify this process as much as possible for users. For example, let's say we want to create a high-availability PostgreSQL cluster that has a single replica, supports having backups in both a local storage area and Amazon S3 and has built-in metrics and connection pooling, similar to:

This can be accomplished with a relatively simple manifest. Please refer to the [tutorial]({{< relref "tutorial/_index.md" >}}) for how to accomplish this, or see the Postgres Operator examples repo.

The Postgres Operator handles setting up all of the various StatefulSets, Deployments, Services and other Kubernetes objects.

You will also notice that **high-availability is enabled by default** if you deploy at least one Postgres replica. The Crunchy PostgreSQL Operator uses a distributed-consensus method for PostgreSQL cluster high-availability, and as such delegates the management of each cluster's availability to the clusters themselves. This removes the PostgreSQL Operator from being a single-point-of-failure, and has benefits such as faster recovery times for each PostgreSQL cluster. For a detailed discussion on high-availability, please see the [High-Availability]({{< relref "architecture/high-availability.md" >}}) section.

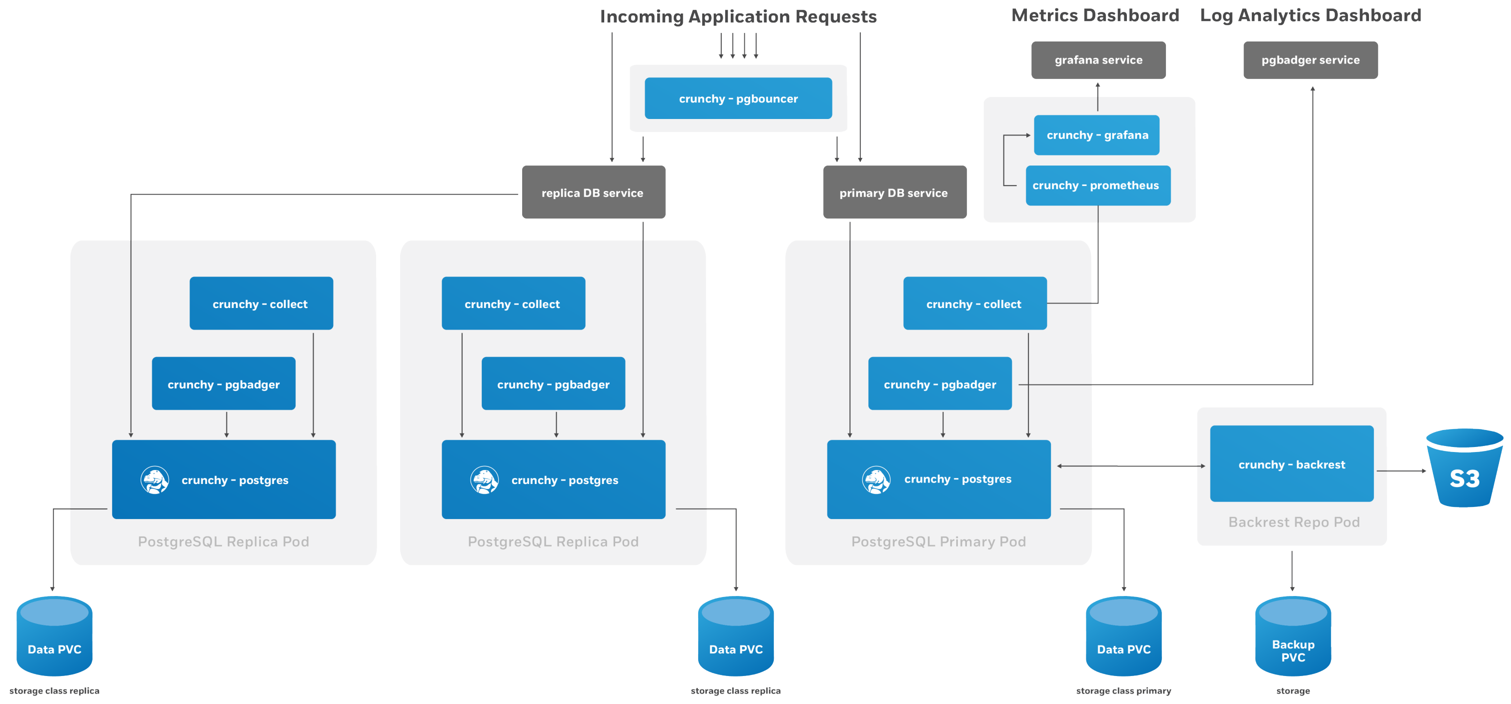


Figure 1: PostgreSQL HA Cluster

Kubernetes StatefulSets: The PGO Deployment Model

PGO, the Postgres Operator from Crunchy Data, uses Kubernetes StatefulSets for running Postgres instances, and will use Deployments for more ephemeral services.

PGO deploys Kubernetes Statefulsets in a way to allow for creating both different Postgres instance groups and be able to support advanced operations such as rolling updates that minimize or eliminate Postgres downtime. Additional components in our PostgreSQL cluster, such as the pgBackRest repository or an optional pgBouncer, are deployed with Kubernetes Deployments.

With the PGO architecture, we can also leverage Statefulsets to apply affinity and toleration rules across every Postgres instance or individual ones. For instance, we may want to force one or more of our PostgreSQL replicas to run on Nodes in a different region than our primary PostgreSQL instances.

What's great about this is that PGO manages this for you so you don't have to worry! Being aware of this model can help you understand how the Postgres Operator gives you maximum flexibility for your PostgreSQL clusters while giving you the tools to troubleshoot issues in production.

The last piece of this model is the use of Kubernetes Services for accessing your PostgreSQL clusters and their various components. The PostgreSQL Operator puts services in front of each Deployment to ensure you have a known, consistent means of accessing your PostgreSQL components.

Note that in some production environments, there can be delays in accessing Services during transition events. The PostgreSQL Operator attempts to mitigate delays during critical operations (e.g. failover, restore, etc.) by directly accessing the Kubernetes Pods to perform given actions.

Additional Architecture Information

There is certainly a lot to unpack in the overall architecture of PGO. Understanding the architecture will help you to plan the deployment model that is best for your environment. For more information on the architectures of various components of the PostgreSQL Operator, please read onward!

One of the great things about PostgreSQL is its reliability: it is very stable and typically "just works." However, there are certain things that can happen in the environment that PostgreSQL is deployed in that can affect its uptime, including:

- The database storage disk fails or some other hardware failure occurs
- The network on which the database resides becomes unreachable
- The host operating system becomes unstable and crashes

- A key database file becomes corrupted
- A data center is lost

There may also be downtime events that are due to the normal case of operations, such as performing a minor upgrade, security patching of operating system, hardware upgrade, or other maintenance.

Fortunately, PGO, the Postgres Operator from Crunchy Data, is prepared for this.

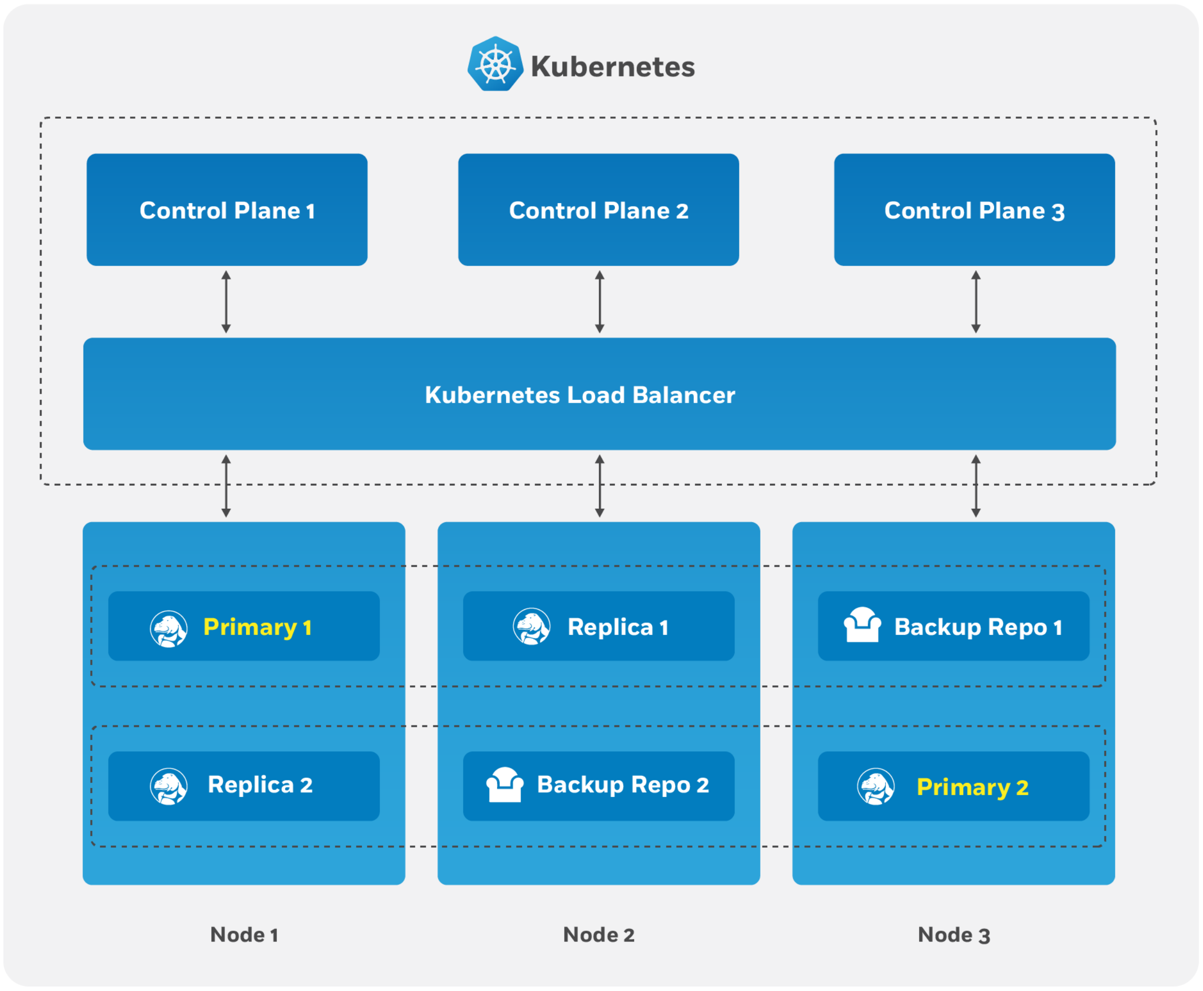


Figure 2: PostgreSQL Operator high availability Overview

The Crunchy PostgreSQL Operator supports a distributed-consensus based high availability (HA) system that keeps its managed PostgreSQL clusters up and running, even if the PostgreSQL Operator disappears. Additionally, it leverages Kubernetes specific features such as Pod Anti-Affinity to limit the surface area that could lead to a PostgreSQL cluster becoming unavailable. The PostgreSQL Operator also supports automatic healing of failed primaries and leverages the efficient pgBackRest "delta restore" method, which eliminates the need to fully reprovision a failed cluster!

The Crunchy PostgreSQL Operator also maintains high availability during a routine task such as a PostgreSQL minor version upgrade.

For workloads that are sensitive to transaction loss, PGO supports PostgreSQL synchronous replication.

The high availability backing for your PostgreSQL cluster is only as good as your high availability backing for Kubernetes. To learn more about creating a high availability Kubernetes cluster, please review the Kubernetes documentation or consult your systems administrator.

The Crunchy Postgres Operator High Availability Algorithm

A critical aspect of any production-grade PostgreSQL deployment is a reliable and effective high availability (HA) solution. Organizations want to know that their PostgreSQL deployments can remain available despite various issues that have the potential to disrupt operations, including hardware failures, network outages, software errors, or even human mistakes.

The key portion of high availability that the PostgreSQL Operator provides is that it delegates the management of HA to the PostgreSQL clusters themselves. This ensures that the PostgreSQL Operator is not a single-point of failure for the availability of any of the PostgreSQL clusters that it manages, as the PostgreSQL Operator is only maintaining the definitions of what should be in the cluster (e.g. how many instances in the cluster, etc.).

Each HA PostgreSQL cluster maintains its availability using concepts that come from the Raft algorithm to achieve distributed consensus. The Raft algorithm ("Reliable, Replicated, Redundant, Fault-Tolerant") was developed for systems that have one "leader" (i.e. a primary) and one-to-many followers (i.e. replicas) to provide the same fault tolerance and safety as the PAXOS algorithm while being easier to implement.

For the PostgreSQL cluster group to achieve distributed consensus on who the primary (or leader) is, each PostgreSQL cluster leverages the distributed etcd key-value store that is bundled with Kubernetes. After it is elected as the leader, a primary will place a lock in the distributed etcd cluster to indicate that it is the leader. The "lock" serves as the method for the primary to provide a heartbeat: the primary will periodically update the lock with the latest time it was able to access the lock. As long as each replica sees that the lock was updated within the allowable automated failover time, the replicas will continue to follow the leader.

The "log replication" portion that is defined in the Raft algorithm is handled by PostgreSQL in two ways. First, the primary instance will replicate changes to each replica based on the rules set up in the provisioning process. For PostgreSQL clusters that leverage "synchronous replication," a transaction is not considered complete until all changes from those transactions have been sent to all replicas that are subscribed to the primary.

In the above section, note the key word that the transaction are sent to each replica: the replicas will acknowledge receipt of the transaction, but they may not be immediately replayed. We will address how we handle this further down in this section.

During this process, each replica keeps track of how far along in the recovery process it is using a "log sequence number" (LSN), a built-in PostgreSQL serial representation of how many logs have been replayed on each replica. For the purposes of HA, there are two LSNs that need to be considered: the LSN for the last log received by the replica, and the LSN for the changes replayed for the replica. The LSN for the latest changes received can be compared amongst the replicas to determine which one has replayed the most changes, and an important part of the automated failover process.

The replicas periodically check in on the lock to see if it has been updated by the primary within the allowable automated failover timeout. Each replica checks in at a randomly set interval, which is a key part of Raft algorithm that helps to ensure consensus during an election process. If a replica believes that the primary is unavailable, it becomes a candidate and initiates an election and votes for itself as the new primary. A candidate must receive a majority of votes in a cluster in order to be elected as the new primary.

There are several cases for how the election can occur. If a replica believes that a primary is down and starts an election, but the primary is actually not down, the replica will not receive enough votes to become a new primary and will go back to following and replaying the changes from the primary.

In the case where the primary is down, the first replica to notice this starts an election. Per the Raft algorithm, each available replica compares which one has the latest changes available, based upon the LSN of the latest logs received. The replica with the latest LSN wins and receives the vote of the other replica. The replica with the majority of the votes wins. In the event that two replicas' logs have the same LSN, the tie goes to the replica that initiated the voting request.

Once an election is decided, the winning replica is immediately promoted to be a primary and takes a new lock in the distributed etcd cluster. If the new primary has not finished replaying all of its transactions logs, it must do so in order to reach the desired state based on the LSN. Once the logs are finished being replayed, the primary is able to accept new queries.

At this point, any existing replicas are updated to follow the new primary.

When the old primary tries to become available again, it realizes that it has been deposed as the leader and must be healed. The old primary determines what kind of replica it should be based upon the CRD, which allows it to set itself up with appropriate attributes. It is then restored from the pgBackRest backup archive using the "delta restore" feature, which heals the instance and makes it ready to follow the new primary, which is known as "auto healing."

How The Crunchy PostgreSQL Operator Uses Pod Anti-Affinity

Kubernetes has two types of Pod anti-affinity:

- Preferred: With preferred (`preferredDuringSchedulingIgnoredDuringExecution`) Pod anti-affinity, Kubernetes will make a best effort to schedule Pods matching the anti-affinity rules to different Nodes. However, if it is not possible to do so, then Kubernetes may schedule one or more Pods to the same Node.
- Required: With required (`requiredDuringSchedulingIgnoredDuringExecution`) Pod anti-affinity, Kubernetes mandates that each Pod matching the anti-affinity rules **must** be scheduled to different Nodes. However, a Pod may not be scheduled if Kubernetes cannot find a Node that does not contain a Pod matching the rules.

There is a tradeoff with these two types of pod anti-affinity: while "required" anti-affinity will ensure that all the matching Pods are scheduled on different Nodes, if Kubernetes cannot find an available Node, your Postgres instance may not be scheduled. Likewise, while "preferred" anti-affinity will make a best effort to scheduled your Pods on different Nodes, Kubernetes may compromise and schedule more than one Postgres instance of the same cluster on the same Node.

By understanding these tradeoffs, the makeup of your Kubernetes cluster, and your requirements, you can choose the method that makes the most sense for your Postgres deployment. We'll show examples of both methods below!

For an example for how pod anti-affinity works with PGO, please see the [high availability tutorial]({{< relref "tutorial/high-availability.md" >}}#pod-anti-affinity).

Synchronous Replication: Guarding Against Transactions Loss

Clusters managed by the Crunchy PostgreSQL Operator can be deployed with synchronous replication, which is useful for workloads that are sensitive to losing transactions, as PostgreSQL will not consider a transaction to be committed until it is committed to all synchronous replicas connected to a primary. This provides a higher guarantee of data consistency and, when a healthy synchronous replica is present, a guarantee of the most up-to-date data during a failover event.

This comes at a cost of performance: PostgreSQL has to wait for a transaction to be committed on all synchronous replicas, and a connected client will have to wait longer than if the transaction only had to be committed on the primary (which is how asynchronous replication works). Additionally, there is a potential impact to availability: if a synchronous replica crashes, any writes to the primary will be blocked until a replica is promoted to become a new synchronous replica of the primary.

Node Affinity

Kubernetes Node Affinity can be used to scheduled Pods to specific Nodes within a Kubernetes cluster. This can be useful when you want your PostgreSQL instances to take advantage of specific hardware (e.g. for geospatial applications) or if you want to have a replica instance deployed to a specific region within your Kubernetes cluster for high availability purposes.

For an example for how node affinity works with PGO, please see the [high availability tutorial]({{< relref "tutorial/high-availability.md" >}}##node-affinity).

Tolerations

Kubernetes Tolerations can help with the scheduling of Pods to appropriate nodes. There are many reasons that a Kubernetes administrator may want to use tolerations, such as restricting the types of Pods that can be assigned to particular Nodes. Reasoning and strategy for using taints and tolerations is outside the scope of this documentation.

You can configure the tolerations for your Postgres instances on the `postgresclusters` custom resource.

Pod Topology Spread Constraints

Kubernetes Pod Topology Spread Constraints can also help you efficiently schedule your workloads by ensuring your Pods are not scheduled in only one portion of your Kubernetes cluster. By spreading your Pods across your Kubernetes cluster among your various failure-domains, such as regions, zones, nodes, and other user-defined topology domains, you can achieve high availability as well as efficient resource utilization.

For an example of how pod topology spread constraints work with PGO, please see the [high availability tutorial]({{< relref "tutorial/high-availability.md" >}}#pod-topology-spread-constraints).

Rolling Updates

During the lifecycle of a PostgreSQL cluster, there are certain events that may require a planned restart, such as an update to a "restart required" PostgreSQL configuration setting (e.g. `shared_buffers`) or a change to a Kubernetes Deployment template (e.g. [changing the memory request]({{< relref "tutorial/resize-cluster.md">}}#customize-cpu-memory)). Restarts can be disruptive in a high availability deployment, which is why many setups employ a "rolling update" strategy (aka a "rolling restart") to minimize or eliminate downtime during a planned restart.

Because PostgreSQL is a stateful application, a simple rolling restart strategy will not work: PostgreSQL needs to ensure that there is a primary available that can accept reads and writes. This requires following a method that will minimize the amount of downtime when the primary is taken offline for a restart.

The PostgreSQL Operator uses the following algorithm to perform the rolling restart to minimize any potential interruptions:

1. Each replica is updated in sequential order. This follows the following process:

2. The replica is explicitly shut down to ensure any outstanding changes are flushed to disk.

3. If requested, the PostgreSQL Operator will apply any changes to the Deployment.

4. The replica is brought back online. The PostgreSQL Operator waits for the replica to become available before it proceeds to the next replica.

5. The above steps are repeated until all of the replicas are restarted.

6. A controlled switchover is performed. The PostgreSQL Operator determines which replica is the best candidate to become the new primary. It then demotes the primary to become a replica and promotes the best candidate to become the new primary.

7. The former primary follows a process similar to what is described in step 1.

The downtime is thus constrained to the amount of time the switchover takes.

PGO will automatically detect when to apply a rolling update.

When using the PostgreSQL Operator, the answer to the question "do you take backups of your database" is automatically "yes!"

The PostgreSQL Operator uses the open source pgBackRest backup and restore utility that is designed for working with databases that are many terabytes in size. As described in the [tutorial]({{< relref "/tutorial/backups.md" >}}), pgBackRest is enabled by default as it permits the PostgreSQL Operator to automate some advanced as well as convenient behaviors, including:

- Efficient provisioning of new replicas that are added to the PostgreSQL cluster
- Preventing replicas from falling out of sync from the PostgreSQL primary by allowing them to replay old WAL logs
- Allowing failed primaries to automatically and efficiently heal using the "delta restore" feature
- Serving as the basis for the cluster cloning feature
- …and of course, allowing for one to take full, differential, and incremental backups and perform full and point-in-time restores

Below is one example of how PGO manages backups with both a local storage and a Amazon S3 configuration.

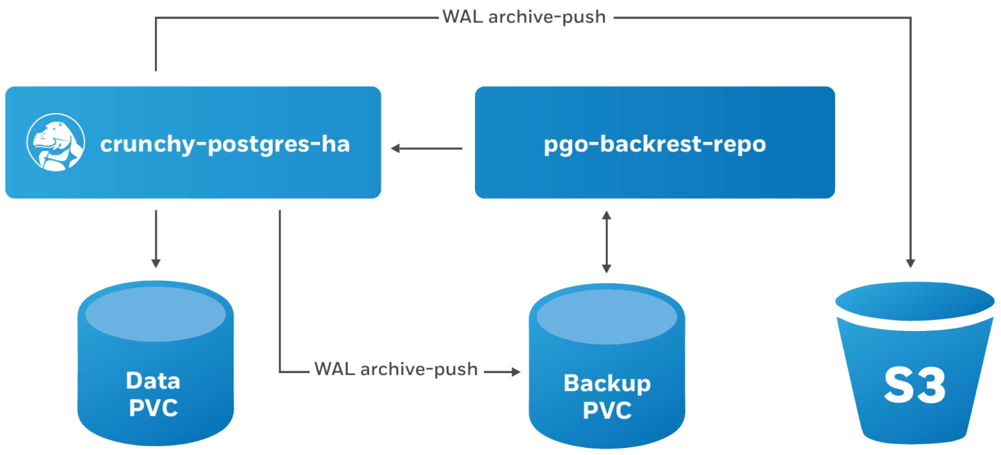


Figure 3: PostgreSQL Operator pgBackRest Integration

The PostgreSQL Operator leverages a pgBackRest repository to facilitate the usage of the pgBackRest features in a PostgreSQL cluster. When a new PostgreSQL cluster is created, it simultaneously creates a pgBackRest repository.

You can store your pgBackRest backups in up to four different locations and using four different storage types:

- Any Kubernetes supported storage class
- Amazon S3 (or S3 equivalents like MinIO)
- Google Cloud Storage (GCS)
- Azure Blob Storage

PostgreSQL is automatically configured to use the `pgbackrest archive-push` command to archive the write-ahead log (WAL) in all repositories.

Backups

PGO supports three types of pgBackRest backups:

- Full: A full backup of all the contents of the PostgreSQL cluster
- Differential: A backup of only the files that have changed since the last full backup
- Incremental: A backup of only the files that have changed since the last full, differential, or incremental backup

Scheduling Backups

Any effective disaster recovery strategy includes having regularly scheduled backups. PGO enables this by managing a series of Kubernetes CronJobs to ensure that backups are executed at scheduled times.

Note that pgBackRest presently only supports taking one backup at a time. This may change in a future release, but for the time being we suggest that you stagger your backup times.

Please see the [backup management tutorial]({{< relref "/tutorial/backup-management.md" >}}) for how to set up backup schedules and configure retention policies.

Restores

The PostgreSQL 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
- Restore in-place

For examples of this, please see the [disaster recovery tutorial]({{< relref "/tutorial/disaster-recovery.md" >}})

Deleting a Backup

If you delete a backup that is *not* set to expire, you may be unable to meet your retention requirements. If you are deleting backups to free space, it is recommended to delete your oldest backups first.

A backup can be deleted by running the `pgbackrest expire` command directly on the pgBackRest repository Pod or a Postgres instance.

Deploying to your Kubernetes cluster may allow for greater reliability than other environments, but that's only the case when it's configured correctly. Fortunately, PGO, the Postgres Operator from Crunchy Data, is ready to help with helpful default settings to ensure you make the most out of your Kubernetes environment!

High Availability By Default

As shown in the [high availability tutorial]({{< relref "tutorial/high-availability.md" >}}#pod-topology-spread-constraints), PGO supports the use of Pod Topology Spread Constraints to customize your Pod deployment strategy, but useful defaults are already in place for you without any additional configuration required!

PGO's default scheduling constraints for HA is implemented for the various Pods comprising a PostgreSQL cluster, specifically to ensure the Operator always deploys a High-Availability cluster architecture by default.

Using Pod Topology Spread Constraints, the general scheduling guidelines are as follows:

- Pods are only considered from the same cluster.
- PgBouncer pods are only considered amongst other PgBouncer pods.
- Postgres pods are considered amongst all Postgres pods and pgBackRest repo host Pods.
- pgBackRest repo host Pods are considered amongst all Postgres pods and pgBackRest repo hosts Pods.
- Pods are scheduled across the different `kubernetes.io/hostname` and `topology.kubernetes.io/zone` failure domains.
- Pods are scheduled when there are fewer nodes than pods, e.g. single node.

With the above configuration, your data is distributed as widely as possible throughout your Kubernetes cluster to maximize safety.

Customization

While the default scheduling settings are designed to meet the widest variety of environments, they can be customized or removed as needed. Assuming a PostgresCluster named 'hippo', the default Pod Topology Spread Constraints applied on Postgres Instance and pgBackRest Repo Host Pods are as follows:

```
topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: kubernetes.io/hostname
    whenUnsatisfiable: ScheduleAnyway
    labelSelector:
      matchLabels:
        postgres-operator.crunchydata.com/cluster: hippo
      matchExpressions:
      - key: postgres-operator.crunchydata.com/data
        operator: In
        values:
        - postgres
        - pgbackrest
  - maxSkew: 1
    topologyKey: topology.kubernetes.io/zone
    whenUnsatisfiable: ScheduleAnyway
    labelSelector:
      matchLabels:
        postgres-operator.crunchydata.com/cluster: hippo
      matchExpressions:
      - key: postgres-operator.crunchydata.com/data
        operator: In
        values:
        - postgres
        - pgbackrest
```

Similarly, for pgBouncer Pods they will be:

```
topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: kubernetes.io/hostname
    whenUnsatisfiable: ScheduleAnyway
    labelSelector:
      matchLabels:
        postgres-operator.crunchydata.com/cluster: hippo
        postgres-operator.crunchydata.com/role: pgbouncer
  - maxSkew: 1
    topologyKey: topology.kubernetes.io/zone
    whenUnsatisfiable: ScheduleAnyway
    labelSelector:
      matchLabels:
        postgres-operator.crunchydata.com/cluster: hippo
        postgres-operator.crunchydata.com/role: pgbouncer
```

Which, as described in the API documentation, means that there should be a maximum of one Pod difference within the `kubernetes.io/hostname` and `topology.kubernetes.io/zone` failure domains when considering either `data` Pods, i.e. Postgres Instance or pgBackRest repo host Pods from a single PostgresCluster or when considering pgBouncer Pods from a single PostgresCluster.

Any other scheduling configuration settings, such as Affinity, Anti-affinity, Taints, Tolerations, or other Pod Topology Spread Constraints will be added in addition to these defaults. Care should be taken to ensure the combined effect of these settings are appropriate for your Kubernetes cluster.

In cases where these defaults are not desired, PGO does provide a method to disable the default Pod scheduling by setting the `spec.disableDefaultPodScheduling` to ‘true’.

PGO manages PostgreSQL users that you define in [`PostgresCluster.spec.users`]({{< relref “/references/crd#postgresclusterspecusersind
>}}). There, you can list their role attributes and which databases they can access.

Below is some information on how the user and database management systems work. To try out some examples, please see the [user and database management]({{< relref “tutorial/user-management.md” >}}) section of the [tutorial]({{< relref “tutorial/_index.md” >}}).

Understanding Default User Management

When you create a Postgres cluster with PGO and do not specify any additional users or databases, PGO will do the following:

- Create a database that matches the name of the Postgres cluster.
- Create an unprivileged Postgres user with the name of the cluster. This user has access to the database created in the previous step.
- Create a Secret with the login credentials and connection details for the Postgres user in relation to the database. This is stored in a Secret named `<clusterName>-pguser-<clusterName>`. These credentials include:
- `user`: The name of the user account.
- `password`: The password for the user account.

- `dbname`: The name of the database that the user has access to by default.
- `host`: The name of the host of the database. This references the Service of the primary Postgres instance.
- `port`: The port that the database is listening on.
- `uri`: A PostgreSQL connection URI that provides all the information for logging into the Postgres database.
- `jdbc-uri`: A PostgreSQL JDBC connection URI that provides all the information for logging into the Postgres database via the JDBC driver.

You can see this default behavior in the [connect to a cluster]({{< relref "tutorial/connect-cluster.md" >}}) portion of the tutorial.

As an example, using our `hippo` Postgres cluster, we would see the following created:

- A database named `hippo`.
- A Postgres user named `hippo`.
- A Secret named `hippo-pguser-hippo` that contains the user credentials and connection information.

While the above defaults may work for your application, there are certain cases where you may need to customize your user and databases:

- You may require access to the `postgres` superuser.
- You may need to define privileges for your users.
- You may need multiple databases in your cluster, e.g. in a multi-tenant application.
- Certain users may only be able to access certain databases.

Custom Users and Databases

Users and databases can be customized in the `spec.users` section of the custom resource. These can be adding during cluster creation and adjusted over time, but it's important to note the following:

- If `spec.users` is set during cluster creation, PGO will **not** create any default users or databases except for `postgres`. If you want additional databases, you will need to specify them.
- For any users added in `spec.users`, PGO will created a Secret of the format `<clusterName>-pguser-<userName>`. This will contain the user credentials.
- If no databases are specified, `dbname` and `uri` will not be present in the Secret.
- If at least one `spec.users.databases` is specified, the first database in the list will be populated into the connection credentials.
- To prevent accidental data loss, PGO will not automatically drop users. We will see how to drop a user below.
- Similarly, to prevent accidental data loss PGO will not automatically drop databases. We will see how to drop a database below.
- Role attributes are not automatically dropped if you remove them. You will have to set the inverse attribute to drop them (e.g. `NOSUPERUSER`).
- The special `postgres` user can be added as one of the custom users; however, the privileges of the users cannot be adjusted.

For specific examples for how to manage users, please see the [user and database management]({{< relref "tutorial/user-management.md" >}}) section of the [tutorial]({{< relref "tutorial/_index.md" >}}).

Custom Passwords

There are cases where you may want to explicitly provide your own password for a Postgres user. PGO determines the password from an attribute in the user Secret called `verifier`. This contains a hashed copy of your password. When `verifier` changes, PGO will load the contents of the verifier into your Postgres cluster. This method allows for the secure transmission of the password into the Postgres database.

Postgres provides two methods for hashing password: SCRAM-SHA-256 and md5. The preferred (and as of PostgreSQL 14, default) method is to use SCRAM, which is also what PGO uses as a default.

There are two ways you can set a custom password for a user. You can provide a plaintext password in the `password` field and remove the `verifier`. When PGO detects a password without a verifier it will generate the SCRAM `verifier` for you. Optionally, you can generate your own password and verifier. When both values are found in the user secret PGO will not generate anything. Once the password and verifier are found PGO will ensure the provided credential is properly set in postgres.

Example

For example, let's say we have a Postgres cluster named `hippo` and a Postgres user named `hippo`. The Secret then would be called `hippo-pguser-hippo`. We want to set the password for `hippo` to be `datalake` and we can achieve this with a simple `kubectl patch` command. The below assumes that the Secret is stored in the `postgres-operator` namespace:

```
PASSWORD=datalake
kubectl patch secret -n postgres-operator hippo-pguser-hippo -p \
   "{\"stringData\":{\"password\":\"${PASSWORD}\",\"verifier\":\"\"}}"
```

We can take advantage of the Kubernetes Secret `stringData` field to specify non-binary secret data in string form.

PGO generates the SCRAM verifier and applies the updated password to Postgres, and you will be able to log in with the password `datalake`.

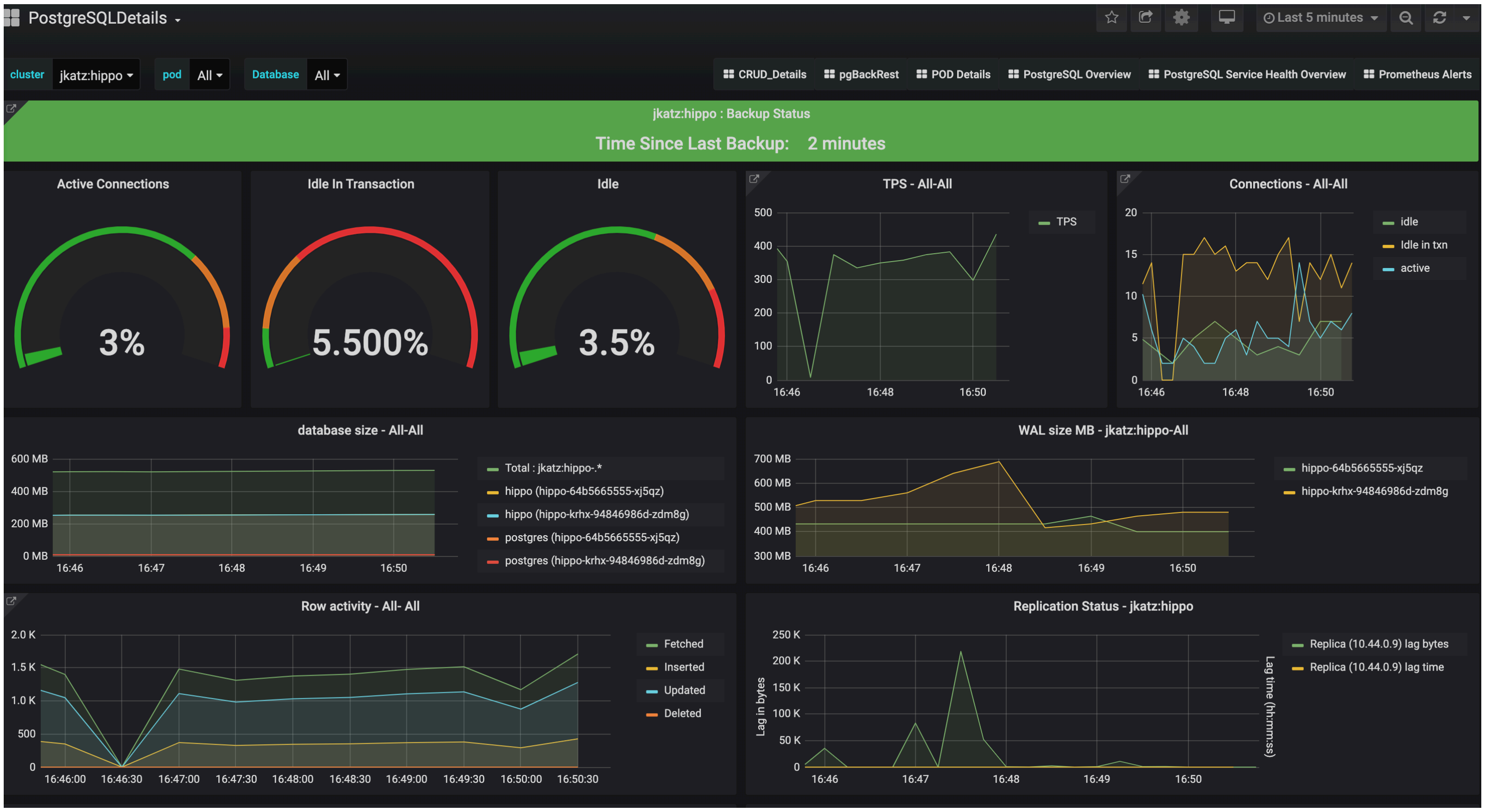


Figure 4: PostgreSQL Operator Monitoring

While having [high availability]({{< relref "architecture/high-availability.md" >}}), [backups]({{< relref "architecture/backups.md" >}}), and disaster recovery systems in place helps in the event of something going wrong with your PostgreSQL cluster, monitoring helps you anticipate problems before they happen. Additionally, monitoring can help you diagnose and resolve additional issues that may not result in downtime, but cause degraded performance.

There are many different ways to monitor systems within Kubernetes, including tools that come with Kubernetes itself. This is by no means to be a comprehensive on how to monitor everything in Kubernetes, but rather what the PostgreSQL Operator provides to give you an [out-of-the-box monitoring solution]({{< relref "installation/monitoring/_index.md" >}}).

Getting Started

If you want to install the metrics stack, please visit the [installation]({{< relref "installation/monitoring/_index.md" >}}) instructions for the [PostgreSQL Operator Monitoring]({{< relref "installation/monitoring/_index.md" >}}) stack.

Components

The [PostgreSQL Operator Monitoring]({{< relref "installation/monitoring/_index.md" >}}) stack is made up of several open source components:

- pgMonitor, which provides the core of the monitoring infrastructure including the following components:

- postgres_exporter, which provides queries used to collect metrics information about a PostgreSQL instance.
- Prometheus, a time-series database that scrapes and stores the collected metrics so they can be consumed by other services.
- Grafana, a visualization tool that provides charting and other capabilities for viewing the collected monitoring data.
- Alertmanager, a tool that can send alerts when metrics hit a certain threshold that require someone to intervene.
- pgnodemx, a PostgreSQL extension that is able to pull container-specific metrics (e.g. CPU utilization, memory consumption) from the container itself via SQL queries.

pgnodemx and the DownwardAPI

pgnodemx is able to pull and format container-specific metrics by accessing several Kubernetes fields that are mounted from the pod to the `database` container's filesystem. By default, these fields include the pod's labels and annotations, as well as the `database` pod's CPU and memory. These fields are mounted at the `/etc/database-containerinfo` path.

Visualizations

Below is a brief description of all the visualizations provided by the [PostgreSQL Operator Monitoring]({{< relref "installation/monitoring/_index.md" >}}) stack. Some of the descriptions may include some directional guidance on how to interpret the charts, though this is only to provide a starting point: actual causes and effects of issues can vary between systems.

Many of the visualizations can be broken down based on the following groupings:

- Cluster: which PostgreSQL cluster should be viewed
- Pod: the specific Pod or PostgreSQL instance

Overview

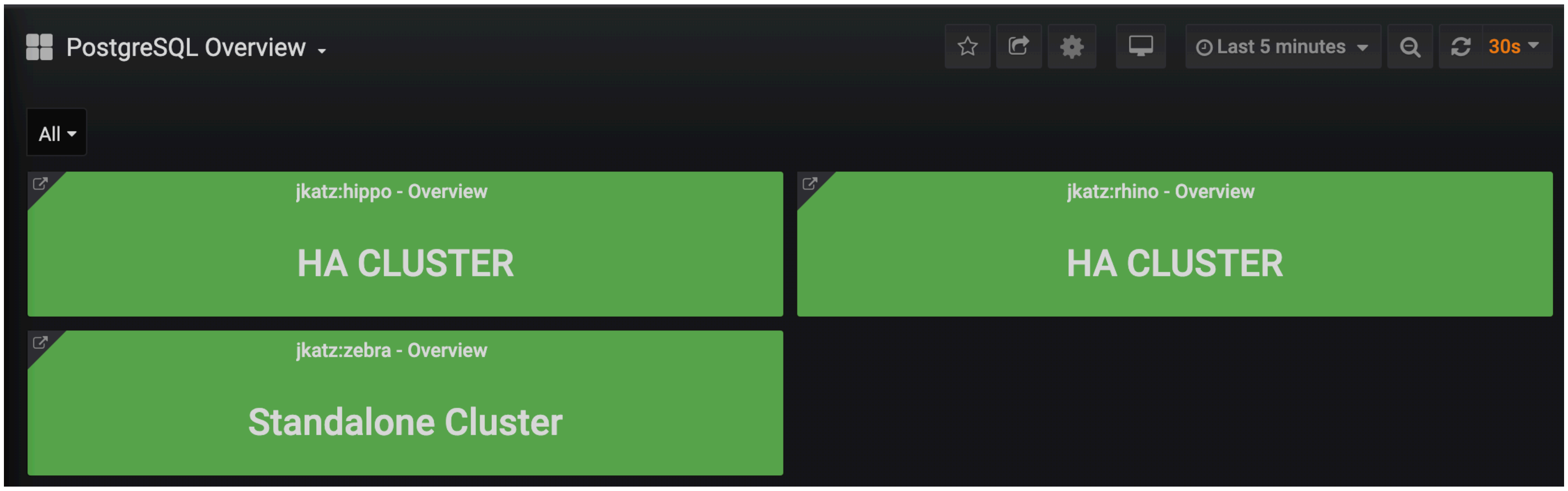


Figure 5: PostgreSQL Operator Monitoring - Overview

The overview provides an overview of all of the PostgreSQL clusters that are being monitoring by the PostgreSQL Operator Monitoring stack. This includes the following information:

- The name of the PostgreSQL cluster and the namespace that it is in
- The type of PostgreSQL cluster (HA [high availability] or standalone)
- The status of the cluster, as indicate by color. Green indicates the cluster is available, red indicates that it is not.

Each entry is clickable to provide additional cluster details.

PostgreSQL Details

The PostgreSQL Details view provides more information about a specific PostgreSQL cluster that is being managed and monitored by the PostgreSQL Operator. These include many key PostgreSQL-specific metrics that help make decisions around managing a PostgreSQL cluster. These include:

- Backup Status: The last time a backup was taken of the cluster. Green is good. Orange means that a backup has not been taken in more than a day and may warrant investigation.

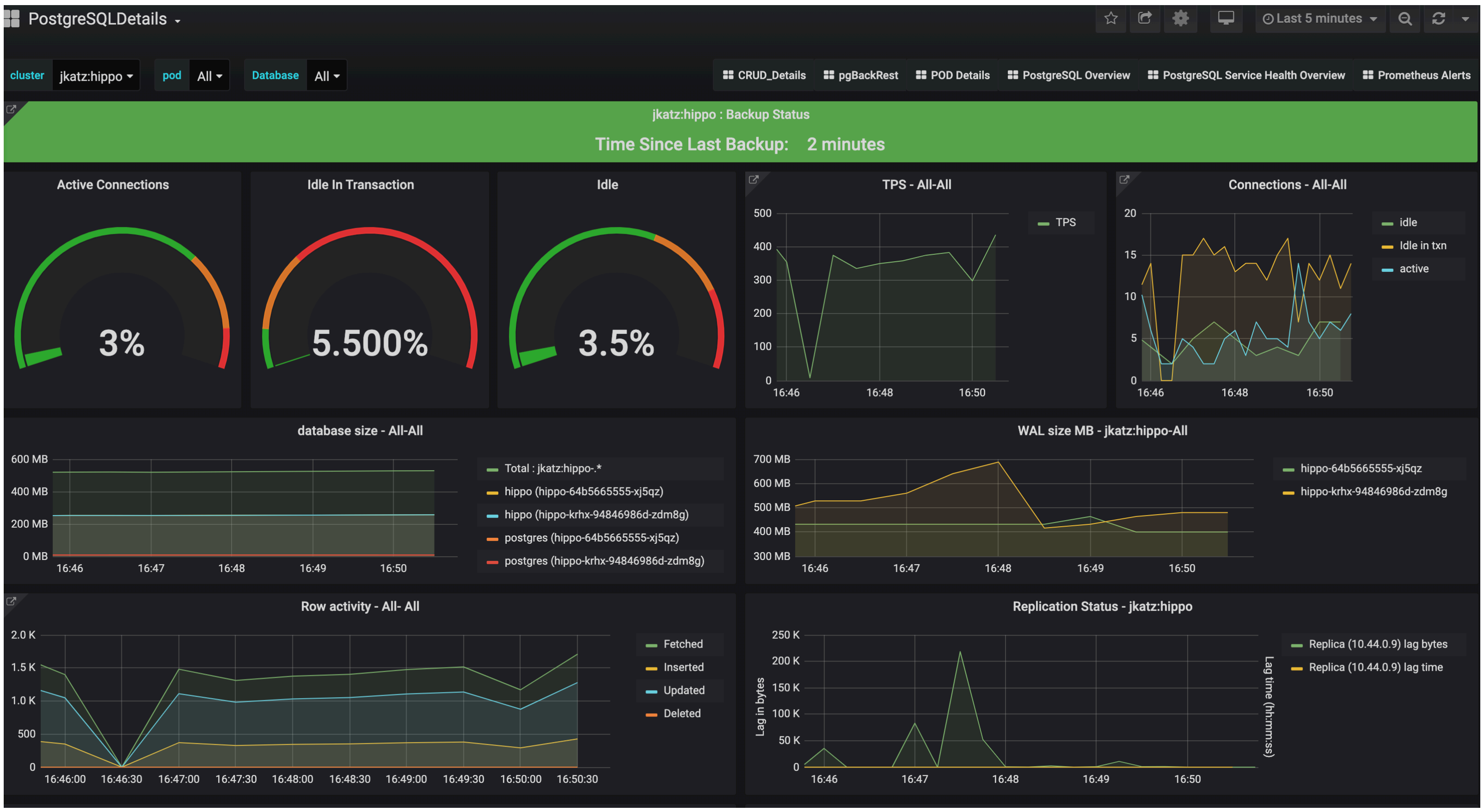


Figure 6: PostgreSQL Operator Monitoring - Cluster Cluster Details

- Active Connections: How many clients are connected to the database. Too many clients connected could impact performance and, for values approaching 100%, can lead to clients being unable to connect.
- Idle in Transaction: How many clients have a connection state of "idle in transaction". Too many clients in this state can cause performance issues and, in certain cases, maintenance issues.
- Idle: How many clients are connected but are in an "idle" state.
- TPS: The number of "transactions per second" that are occurring. Usually needs to be combined with another metric to help with analysis. "Higher is better" when performing benchmarking.
- Connections: An aggregated view of active, idle, and idle in transaction connections.
- Database Size: How large databases are within a PostgreSQL cluster. Typically combined with another metric for analysis. Helps keep track of overall disk usage and if any triage steps need to occur around PVC size.
- WAL Size: How much space write-ahead logs (WAL) are taking up on disk. This can contribute to extra space being used on your data disk, or can give you an indication of how much space is being utilized on a separate WAL PVC. If you are using replication slots, this can help indicate if a slot is not being acknowledged if the numbers are much larger than the `max_wal_size` setting (the PostgreSQL Operator does not use slots by default).
- Row Activity: The number of rows that are selected, inserted, updated, and deleted. This can help you determine what percentage of your workload is read vs. write, and help make database tuning decisions based on that, in conjunction with other metrics.
- Replication Status: Provides guidance information on how much replication lag there is between primary and replica PostgreSQL instances, both in bytes and time. This can provide an indication of how much data could be lost in the event of a failover.

- Conflicts / Deadlocks: These occur when PostgreSQL is unable to complete operations, which can result in transaction loss. The goal is for these numbers to be `0`. If these are occurring, check your data access and writing patterns.
- Cache Hit Ratio: A measure of how much of the "working data", e.g. data that is being accessed and manipulated, resides in memory. This is used to understand how much PostgreSQL is having to utilize the disk. The target number of this should be as high as possible. How to achieve this is the subject of books, but certain takes efforts on your applications use PostgreSQL.
- Buffers: The buffer usage of various parts of the PostgreSQL system. This can be used to help understand the overall throughput between various parts of the system.
- Commit & Rollback: How many transactions are committed and rolled back.
- Locks: The number of locks that are present on a given system.

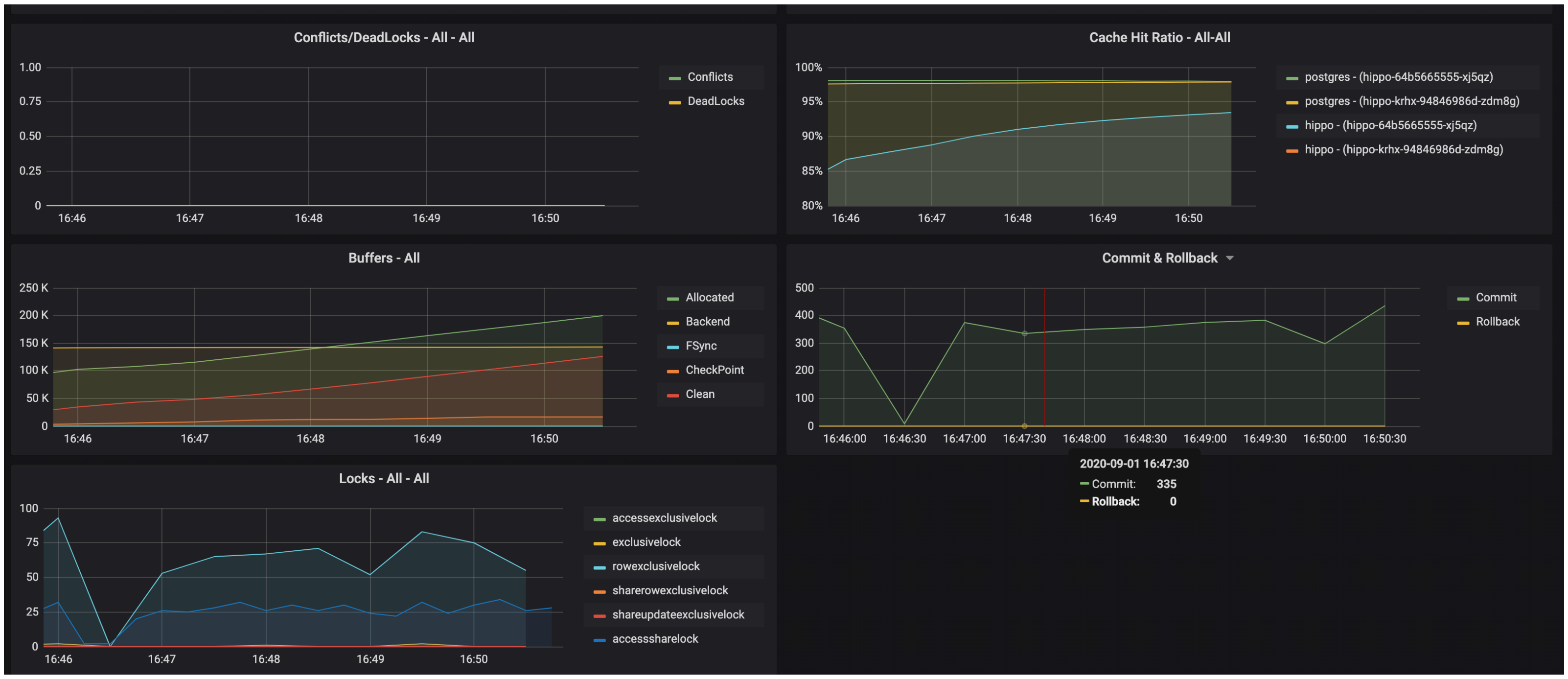


Figure 7: PostgreSQL Operator Monitoring - Cluster Cluster Details 2

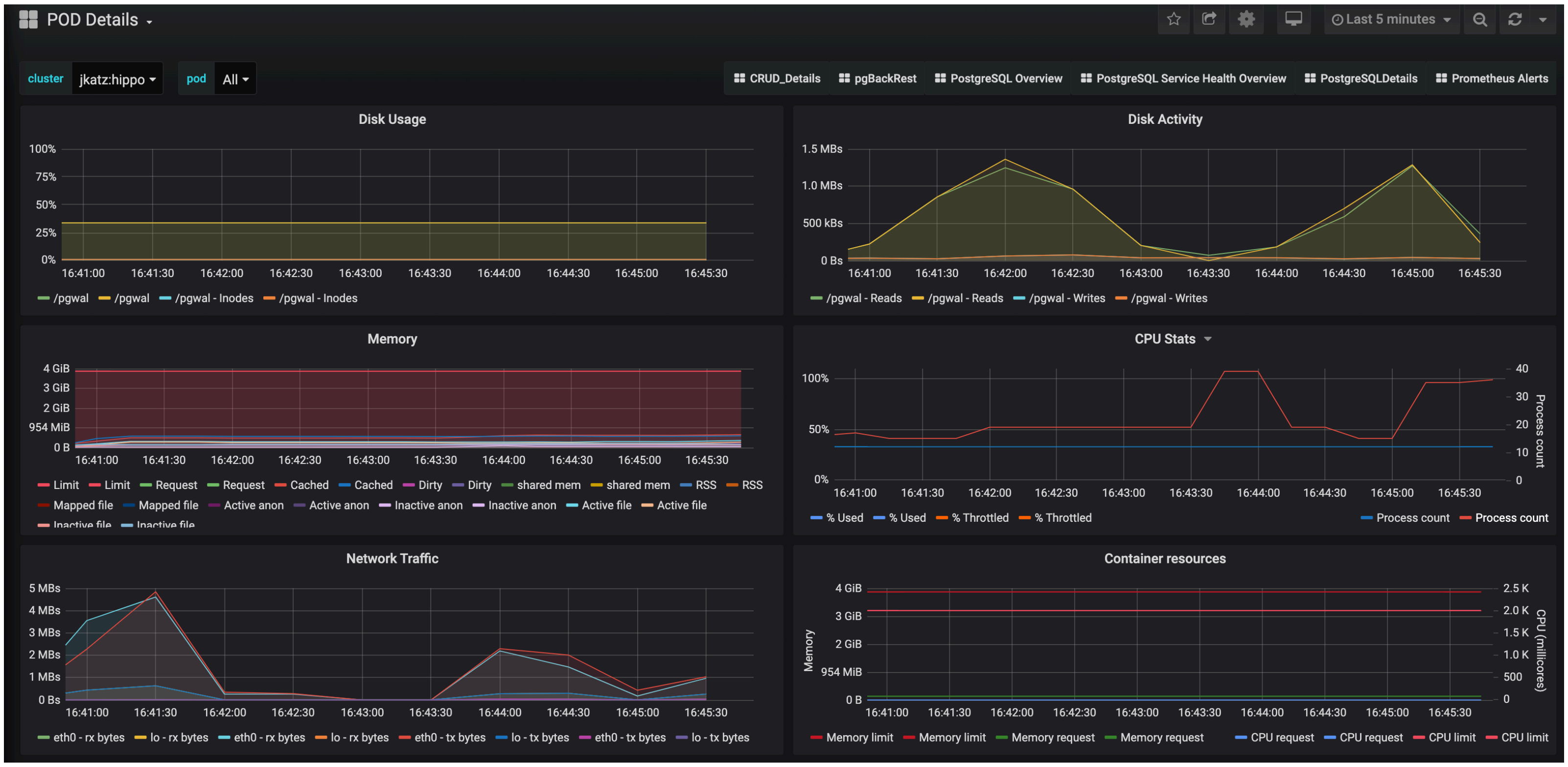


Figure 8: PostgreSQL Operator Monitoring - Pod Details

Pod Details

Pod details provide information about a given Pod or Pods that are being used by a PostgreSQL cluster. These are similar to "operating system" or "node" metrics, with the differences that these are looking at resource utilization by a container, not the entire node.

It may be helpful to view these metrics on a "pod" basis, by using the Pod filter at the top of the dashboard.

- Disk Usage: How much space is being consumed by a volume.
- Disk Activity: How many reads and writes are occurring on a volume.
- Memory: Various information about memory utilization, including the request and limit as well as actually utilization.
- CPU: The amount of CPU being utilized by a Pod
- Network Traffic: The amount of networking traffic passing through each network device.
- Container ResourceS: The CPU and memory limits and requests.

Backups

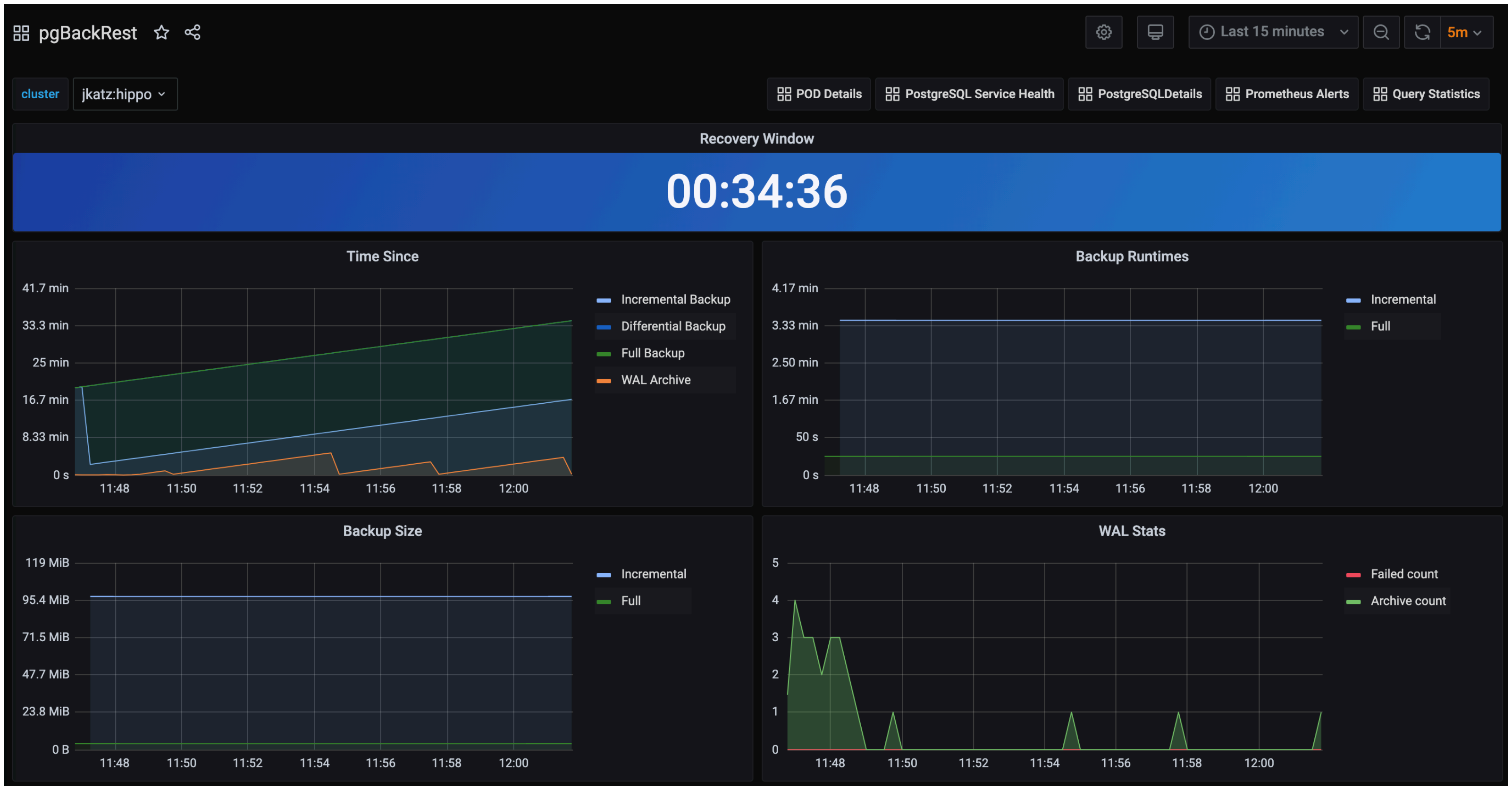


Figure 9: PostgreSQL Operator - Monitoring - Backup Health

There are a variety of reasons why you need to monitoring your backups, starting from answering the fundamental question of "do I have backups available?" Backups can be used for a variety of situations, from cloning new clusters to restoring clusters after a disaster. Additionally, Postgres can run into issues if your backup repository is not healthy, e.g. if it cannot push WAL archives. If your backups are set up properly and healthy, you will be set up to mitigate the risk of data loss!

The backup, or pgBackRest panel, will provide information about the overall state of your backups. This includes:

- Recovery Window: This is an indicator of how far back you are able to restore your data from. This represents all of the backups and archives available in your backup repository. Typically, your recovery window should be close to your overall data retention specifications.
- Time Since Last Backup: this indicates how long it has been since your last backup. This is broken down into pgBackRest backup type (full, incremental, differential) as well as time since the last WAL archive was pushed.
- Backup Runtimes: How long the last backup of a given type (full, incremental differential) took to execute. If your backups are slow, consider providing more resources to the backup jobs and tweaking pgBackRest's performance tuning settings.
- Backup Size: How large the backups of a given type (full, incremental, differential).
- WAL Stats: Shows the metrics around WAL archive pushes. If you have failing pushes, you should to see if there is a transient or permanent error that is preventing WAL archives from being pushed. If left untreated, this could end up causing issues for your Postgres cluster.

PostgreSQL Service Health Overview

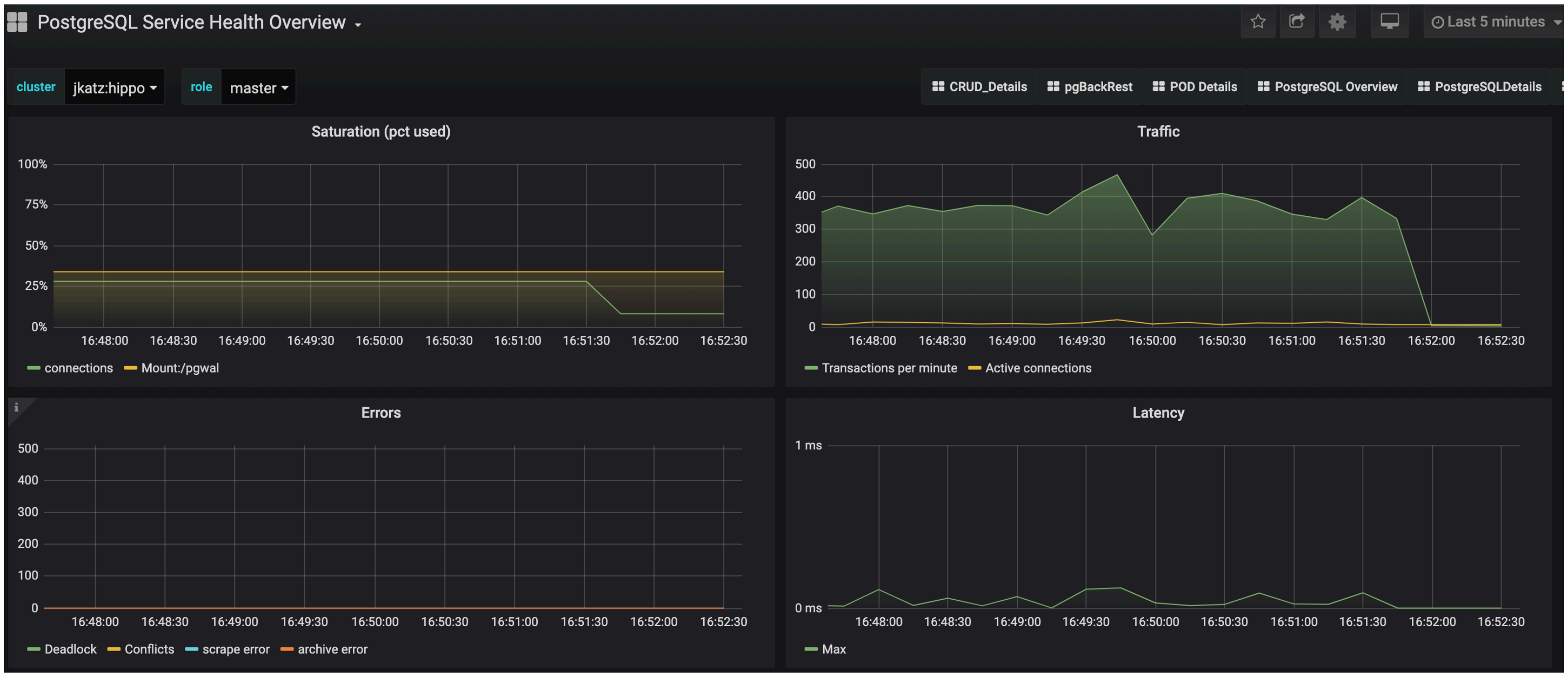


Figure 10: PostgreSQL Operator Monitoring - Service Health Overview

The Service Health Overview provides information about the Kubernetes Services that sit in front of the PostgreSQL Pods. This provides information about the status of the network.

- Saturation: How much of the available network to the Service is being consumed. High saturation may cause degraded performance to clients or create an inability to connect to the PostgreSQL cluster.
- Traffic: Displays the number of transactions per minute that the Service is handling.
- Errors: Displays the total number of errors occurring at a particular Service.
- Latency: What the overall network latency is when interfacing with the Service.

Query Runtime

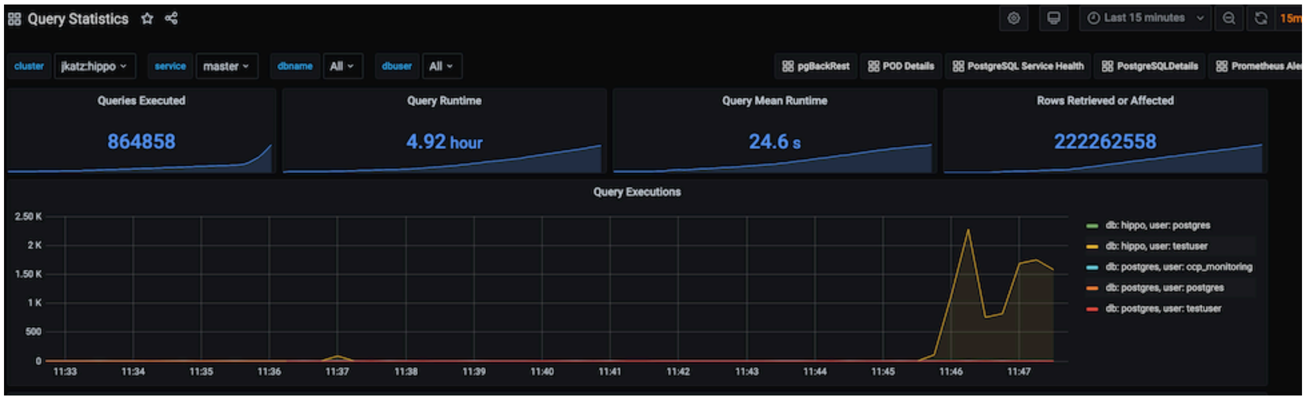
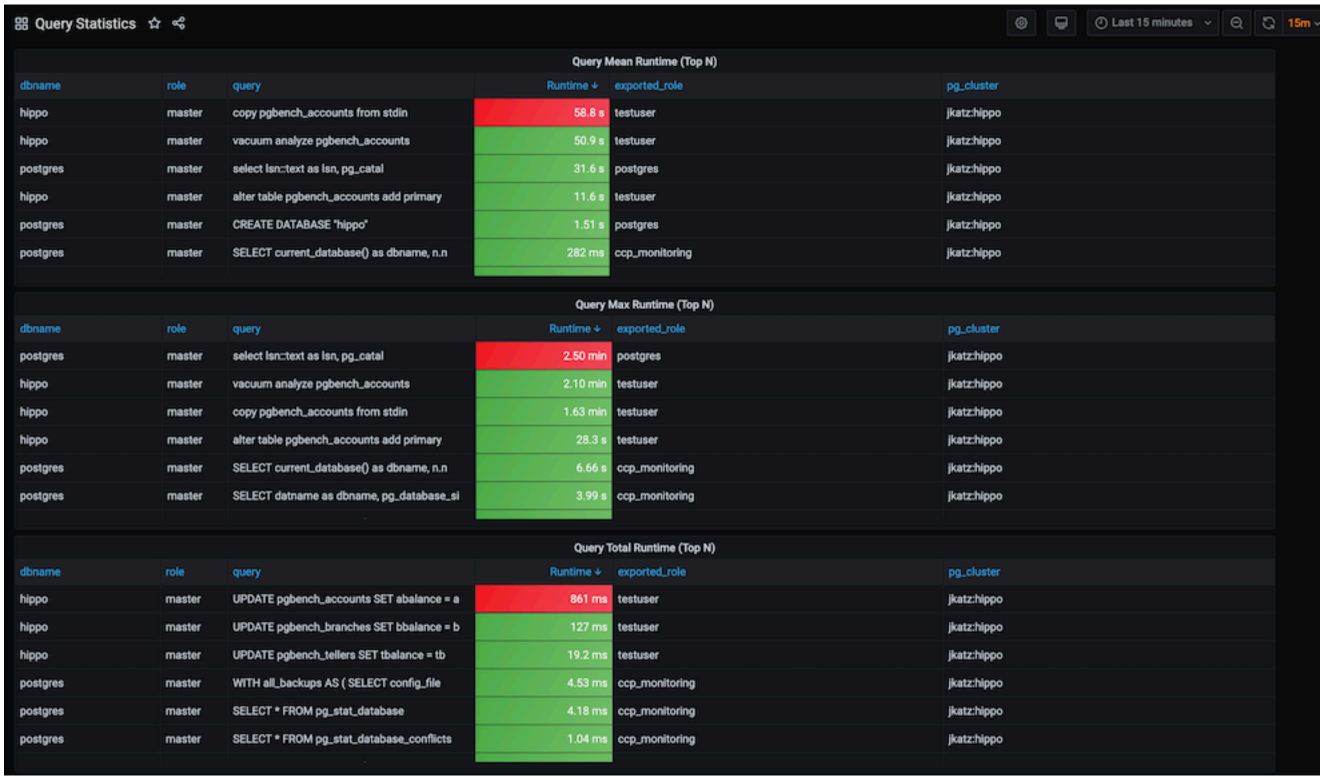


Figure 11: PostgreSQL Operator Monitoring - Query Performance

Looking at the overall performance of queries can help optimize a Postgres deployment, both from [providing resources]({{< relref "tutorial/customize-cluster.md" >}}) to query tuning in the application itself.

You can get a sense of the overall activity of a PostgreSQL cluster from the chart that is visualized above:

- Queries Executed: The total number of queries executed on a system during the period.
- Query runtime: The aggregate runtime of all the queries combined across the system that were executed in the period.
- Query mean runtime: The average query time across all queries executed on the system in the given period.
- Rows retrieved or affected: The total number of rows in a database that were either retrieved or had modifications made to them.



Query Statistics

Query Mean Runtime (Top N)

dbname	role	query	Runtime ↓	exported_role	pg_cluster
hippo	master	copy pgbench_accounts from stdin	58.8 s	testuser	jkatz:hippo
hippo	master	vacuum analyze pgbench_accounts	50.9 s	testuser	jkatz:hippo
postgres	master	select lsn::text as lsn, pg_catal	31.6 s	postgres	jkatz:hippo
hippo	master	alter table pgbench_accounts add primary	11.6 s	testuser	jkatz:hippo
postgres	master	CREATE DATABASE "hippo"	1.51 s	postgres	jkatz:hippo
postgres	master	SELECT current_database() as dbname, n.n	282 ms	ccp_monitoring	jkatz:hippo

Query Max Runtime (Top N)

dbname	role	query	Runtime ↓	exported_role	pg_cluster
postgres	master	select lsn::text as lsn, pg_catal	2.50 min	postgres	jkatz:hippo
hippo	master	vacuum analyze pgbench_accounts	2.10 min	testuser	jkatz:hippo
hippo	master	copy pgbench_accounts from stdin	1.63 min	testuser	jkatz:hippo
hippo	master	alter table pgbench_accounts add primary	28.3 s	testuser	jkatz:hippo
postgres	master	SELECT current_database() as dbname, n.n	6.66 s	ccp_monitoring	jkatz:hippo
postgres	master	SELECT datname as dbname, pg_database_si	3.99 s	ccp_monitoring	jkatz:hippo

Query Total Runtime (Top N)

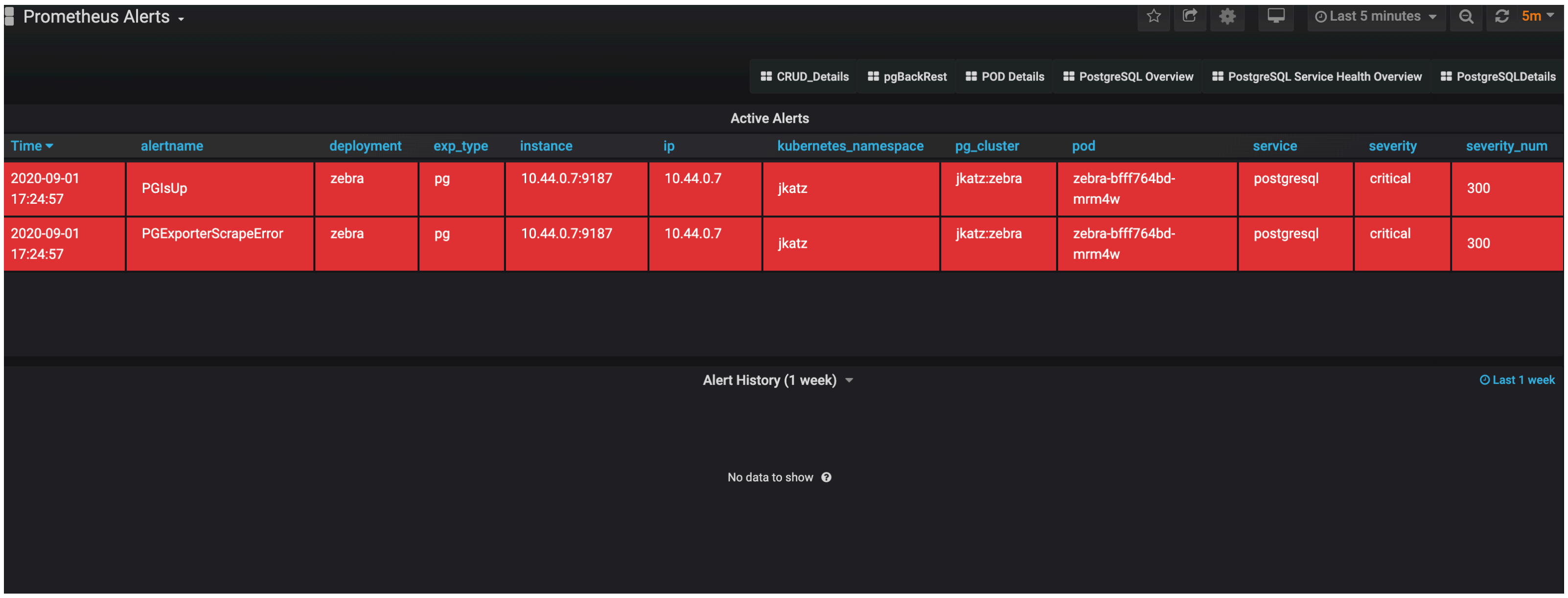
dbname	role	query	Runtime ↓	exported_role	pg_cluster
hippo	master	UPDATE pgbench_accounts SET abalance = a	861 ms	testuser	jkatz:hippo
hippo	master	UPDATE pgbench_branches SET bbalance = b	127 ms	testuser	jkatz:hippo
hippo	master	UPDATE pgbench_tellers SET tbalance = tb	19.2 ms	testuser	jkatz:hippo
postgres	master	WITH all_backups AS (SELECT config_file	4.53 ms	ccp_monitoring	jkatz:hippo
postgres	master	SELECT * FROM pg_stat_database	4.18 ms	ccp_monitoring	jkatz:hippo
postgres	master	SELECT * FROM pg_stat_database_conflicts	1.04 ms	ccp_monitoring	jkatz:hippo

Figure 12: PostgreSQL Operator Monitoring - Query Analysis

PostgreSQL Operator Monitoring also further breaks down the queries so you can identify queries that are being executed too frequently or are taking up too much time.

- Query Mean Runtime (Top N): This highlights the N number of slowest queries by average runtime on the system. This might indicate you are missing an index somewhere, or perhaps the query could be rewritten to be more efficient.
- Query Max Runtime (Top N): This highlights the N number of slowest queries by absolute runtime. This could indicate that a specific query or the system as a whole may need more resources.
- Query Total Runtime (Top N): This highlights the N of slowest queries by aggregate runtime. This could indicate that a ORM is looping over a single query and executing it many times that could possibly be rewritten as a single, faster query.

Alerts



Prometheus Alerts

Active Alerts

Time	alertname	deployment	exp_type	instance	ip	kubernetes_namespace	pg_cluster	pod	service	severity	severity_num
2020-09-01 17:24:57	PGIsUp	zebra	pg	10.44.0.7:9187	10.44.0.7	jkatz	jkatz:zebra	zebra-bfff764bd-mrm4w	postgresql	critical	300
2020-09-01 17:24:57	PGExporterScrapeError	zebra	pg	10.44.0.7:9187	10.44.0.7	jkatz	jkatz:zebra	zebra-bfff764bd-mrm4w	postgresql	critical	300

Alert History (1 week)

No data to show

Figure 13: PostgreSQL Operator Monitoring - Alerts

Alerting lets one view and receive alerts about actions that require intervention, for example, a HA cluster that cannot self-heal. The alerting system is powered by Alertmanager.

The alerts that come installed by default include:

- `PGExporterScrapeError`: The Crunchy PostgreSQL Exporter is having issues scraping statistics used as part of the monitoring stack.
- `PGIsUp`: A PostgreSQL instance is down.
- `PGIdleTxn`: There are too many connections that are in the "idle in transaction" state.
- `PGQueryTime`: A single PostgreSQL query is taking too long to run. Issues a warning at 12 hours and goes critical after 24.
- `PGConnPerc`: Indicates that there are too many connection slots being used. Issues a warning at 75% and goes critical above 90%.
- `PGDiskSize`: Indicates that a PostgreSQL database is too large and could be in danger of running out of disk space. Issues a warning at 75% and goes critical at 90%.
- `PGReplicationByteLag`: Indicates that a replica is too far behind a primary instance, which could risk data loss in a failover scenario. Issues a warning at 50MB an goes critical at 100MB.
- `PGReplicationSlotsInactive`: Indicates that a replication slot is inactive. Not attending to this can lead to out-of-disk errors.
- `PGXIDWraparound`: Indicates that a PostgreSQL instance is nearing transaction ID wraparound. Issues a warning at 50% and goes critical at 75%. It's important that you vacuum your database to prevent this.
- `PGEmergencyVacuum`: Indicates that autovacuum is not running or cannot keep up with ongoing changes, i.e. it's past its "freeze" age. Issues a warning at 110% and goes critical at 125%.
- `PGArchiveCommandStatus`: Indicates that the archive command, which is used to ship WAL archives to pgBackRest, is failing.
- `PGSequenceExhaustion`: Indicates that a sequence is over 75% used.
- `PGSettingsPendingRestart`: Indicates that there are settings changed on a PostgreSQL instance that requires a restart.

Optional alerts that can be enabled:

- `PGMinimumVersion`: Indicates if PostgreSQL is below a desired version.
- `PGRecoveryStatusSwitch_Replica`: Indicates that a replica has been promoted to a primary.
- `PGConnectionAbsent_Prod`: Indicates that metrics collection is absent from a PostgresQL instance.
- `PGSettingsChecksum`: Indicates that PostgreSQL settings have changed from a previous state.
- `PGDataChecksum`: Indicates that there are data checksum failures on a PostgreSQL instance. This could be a sign of data corruption.

You can modify these alerts as you see fit, and add your own alerts as well! Please see the [installation instructions]({{< relref "installation/monitoring/_index.md" >}}) for general setup of the PostgreSQL Operator Monitoring stack.

Advanced [high-availability]({{< relref "architecture/high-availability.md" >}}) and [backup management]({{< relref "architecture/backups.md" >}}) strategies involve spreading your database clusters across multiple data centers to help maximize uptime. In Kubernetes, this technique is known as "federation". Federated Kubernetes clusters are able to communicate with each other, coordinate changes, and provide resiliency for applications that have high uptime requirements.

As of this writing, federation in Kubernetes is still in ongoing development and is something we monitor with intense interest. As Kubernetes federation continues to mature, we wanted to provide a way to deploy PostgreSQL clusters managed by the PostgreSQL Operator that can span multiple Kubernetes clusters. This can be accomplished with a few environmental setups:

- Two Kubernetes clusters
- An external storage system, using one of the following:
- S3, or an external storage system that uses the S3 protocol
- GCS
- Azure Blog Storage
- A Kubernetes storage system that can span multiple clusters

At a high-level, the PostgreSQL Operator follows the "active-standby" data center deployment model for managing the PostgreSQL clusters across Kubernetes clusters. In one Kubernetes cluster, the PostgreSQL Operator deploy PostgreSQL as an "active" PostgreSQL cluster, which means it has one primary and one-or-more replicas. In another Kubernetes cluster, the PostgreSQL cluster is deployed as a "standby" cluster: every PostgreSQL instance is a replica.

A side-effect of this is that in each of the Kubernetes clusters, the PostgreSQL Operator can be used to deploy both active and standby PostgreSQL clusters, allowing you to mix and match! While the mixing and matching may not ideal for how you deploy your PostgreSQL clusters, it does allow you to perform online moves of your PostgreSQL data to different Kubernetes clusters as well as manual online upgrades.

Lastly, while this feature does extend high-availability, promoting a standby cluster to an active cluster is **not** automatic. While the PostgreSQL clusters within a Kubernetes cluster do support self-managed high-availability, a cross-cluster deployment requires someone to specifically promote the cluster from standby to active.

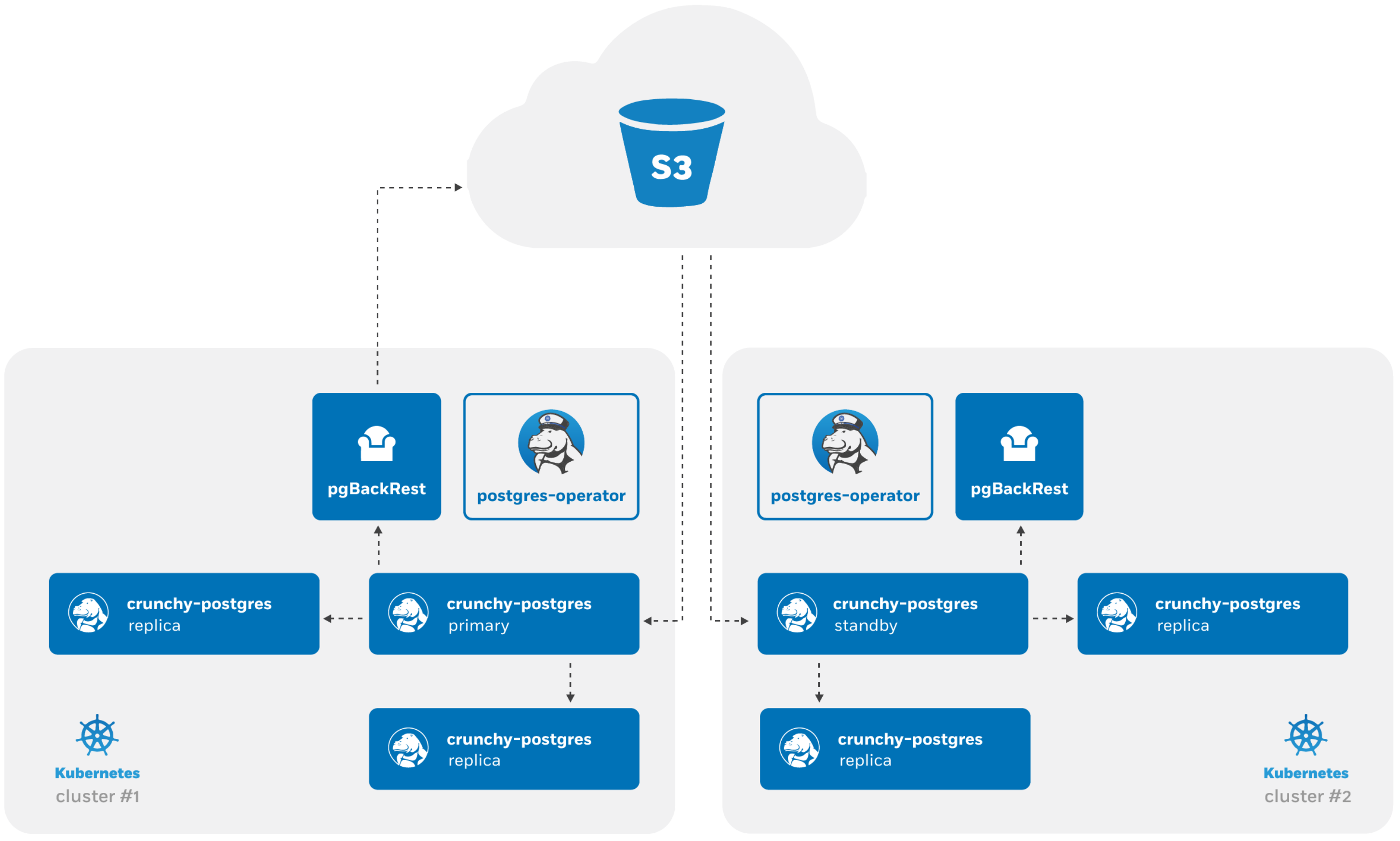


Figure 14: PostgreSQL Operator High-Availability Overview

Standby Cluster Overview

Standby PostgreSQL clusters are managed just like any other PostgreSQL cluster that is managed by the PostgreSQL Operator. For example, adding replicas to a standby cluster is identical as adding them to a primary cluster.

As the architecture diagram above shows, the main difference is that there is no primary instance: one PostgreSQL instance is reading in the database changes from the backup repository, while the other replicas are replicas of that instance. This is known as cascading replication. replicas are cascading replicas, i.e. replicas replicating from a database server that itself is replicating from another database server.

Because standby clusters are effectively read-only, certain functionality that involves making changes to a database, e.g. PostgreSQL user changes, is blocked while a cluster is in standby mode. Additionally, backups and restores are blocked as well. While pgBackRest does support backups from standbys, this requires direct access to the primary database, which cannot be done until the PostgreSQL Operator supports Kubernetes federation.

Creating a Standby PostgreSQL Cluster

For creating a standby Postgres cluster with PGO, please see the [disaster recovery tutorial]({{< relref "tutorial/disaster-recovery.md" >}}#standby-cluster)

Promoting a Standby Cluster

There comes a time where a standby cluster needs to be promoted to an active cluster. Promoting a standby cluster means that a PostgreSQL instance within it will become a primary and start accepting both reads and writes. This has the net effect of pushing WAL (transaction archives) to the pgBackRest repository, so we need to take a few steps first to ensure we don't accidentally create a split-brain scenario.

First, if this is not a disaster scenario, you will want to "shutdown" the active PostgreSQL cluster. This can be done by setting:

```
spec:
  shutdown: true
```

The effect of this is that all the Kubernetes Statefulsets and Deployments for this cluster are scaled to 0.

We can then promote the standby cluster using the following:

```
spec:
  standby:
    enabled: false
```

This command essentially removes the standby configuration from the Kubernetes cluster's DCS, which triggers the promotion of the current standby leader to a primary PostgreSQL instance. You can view this promotion in the PostgreSQL standby leader's (soon to be active leader's) logs:

With the standby cluster now promoted, the cluster with the original active PostgreSQL cluster can now be turned into a standby PostgreSQL cluster. This is done by deleting and recreating all PVCs for the cluster and re-initializing it as a standby using the backup repository. Being that this is a destructive action (i.e. data will only be retained if any Storage Classes and/or Persistent Volumes have the appropriate reclaim policy configured) a warning is shown when attempting to enable standby.

The cluster will reinitialize from scratch as a standby, just like the original standby that was created above. Therefore any transactions written to the original standby, should now replicate back to this cluster.

Packages:

- postgres-operator.crunchydata.com/v1beta1

postgres-operator.crunchydata.com/v1beta1

Resource Types:

- PostgresCluster

PostgresCluster

PostgresCluster is the Schema for the postgresclusters API

Name

Type

		Description	Required
apiVersion	string	postgres-operator.crunchydata.com/v1beta1	true
kind	string	PostgresCluster	true
metadata	object	Refer to the Kubernetes API documentation for the fields of the `metadata` field.	true
spec	object	PostgresClusterSpec defines the desired state of PostgresCluster	false
status	object	PostgresClusterStatus defines the observed state of PostgresCluster	false

PostgresCluster.spec   Parent

PostgresClusterSpec defines the desired state of PostgresCluster

Name	Type	Description	Required
backups	object	PostgreSQL backup configuration	true
instances	[]object	Specifies one or more sets of PostgreSQL pods that replicate data for this cluster.	true
postgresVersion	integer	The major version of PostgreSQL installed in the PostgreSQL image	true
customReplicationTLSSecret	object	The secret containing the replication client certificates and keys for secure connections to the PostgreSQL server. It will need to contain the client TLS certificate, TLS key and the Certificate Authority certificate with the data keys set to tls.crt, tls.key and ca.crt, respectively.	

NOTE: If CustomReplicationClientTLSSecret is provided, CustomTLSSecret MUST be provided and the ca.crt provided must be the same.

false

customTLSSecret

object

The secret containing the Certificates and Keys to encrypt PostgreSQL traffic will need to contain the server TLS certificate, TLS key and the Certificate Authority certificate with the data keys set to tls.crt, tls.key and ca.crt, respectively. It will then be mounted as a volume projection to the '/pgconf/tls' directory. For more information on Kubernetes secret projections, please see https://k8s.io/docs/concepts/configuration/secret/#projection-of-secret-keys-to-specific-paths NOTE: If CustomTLSSecret is provided, CustomReplicationClientTLSSecret MUST be provided and the ca.crt provided must be the same.

false

dataSource

object

Specifies a data source for bootstrapping the PostgreSQL cluster.

false

databaseInitSQL

object

DatabaseInitSQL defines a ConfigMap containing custom SQL that will be run after the cluster is initialized. This ConfigMap must be in the same namespace as the cluster.

false

disableDefaultPodScheduling

boolean

Whether or not the PostgreSQL cluster should use the defined default scheduling constraints. If the field is unset or false, the default scheduling constraints will be used in addition to any custom constraints provided.

false

image

string

The image name to use for PostgreSQL containers. When omitted, the value comes from an operator environment variable. For standard PostgreSQL images, the format is RELATED_IMAGE_POSTGRES_{postgresVersion}, e.g. RELATED_IMAGE_POSTGRES_13. For PostGIS enabled PostgreSQL images, the format is RELATED_IMAGE_POSTGRES_{postgresVersion}*GIS*{postGISVersion}, e.g. RELATED_IMAGE_POSTGRES_13_GIS_3.1.

false

imagePullPolicy

enum

ImagePullPolicy is used to determine when Kubernetes will attempt to pull (download) container images. More info: https://kubernetes.io/do
pull-policy

false

imagePullSecrets

[]object

The image pull secrets used to pull from a private registry Changing this value causes all running pods to restart. https://k8s.io/docs/tasks/co
pod-container/pull-image-private-registry/

false

metadata

object

Metadata contains metadata for PostgresCluster resources

false

monitoring

object

The specification of monitoring tools that connect to PostgreSQL

false

openshift

boolean

Whether or not the PostgreSQL cluster is being deployed to an OpenShift environment. If the field is unset, the operator will automatically detect the environment.

false

patroni

object

false

port

integer

The port on which PostgreSQL should listen.

false

postGISVersion

string

The PostGIS extension version installed in the PostgreSQL image. When image is not set, indicates a PostGIS enabled image will be used.

false

proxy

object

The specification of a proxy that connects to PostgreSQL.

false

service

object

Specification of the service that exposes the PostgreSQL primary instance.

false

shutdown

boolean

Whether or not the PostgreSQL cluster should be stopped. When this is true, workloads are scaled to zero and CronJobs are suspended. Other resources, such as Services and Volumes, remain in place.

false

standby

object

Run this cluster as a read-only copy of an existing cluster or archive.

false

supplementalGroups

[]integer

A list of group IDs applied to the process of a container. These can be useful when accessing shared file systems with constrained permissions. More info: https://kubernetes.io/docs/reference/kubernetes-api/workload-resources/pod-v1/#security-context

false

users

[]object

Users to create inside PostgreSQL and the databases they should access. The default creates one user that can access one database matching the PostgresCluster name. An empty list creates no users. Removing a user from this list does NOT drop the user nor revoke their access.

false

PostgresCluster.spec.backups   Parent

PostgreSQL backup configuration

Name

Type

Description

Required

pgbackrest

object

pgBackRest archive configuration

true

PostgresCluster.spec.backups.pgbackrest   Parent

pgBackRest archive configuration

Name

Type

Description

Required

repos

[]object

Defines a pgBackRest repository

true

configuration

[]object

Projected volumes containing custom pgBackRest configuration. These files are mounted under “/etc/pgbackrest/conf.d” alongside any pgBackRest configuration generated by the PostgreSQL Operator: https://pgbackrest.org/configuration.html

false

global

map[string]string

Global pgBackRest configuration settings. These settings are included in the “global” section of the pgBackRest configuration generated by the PostgreSQL Operator, and then mounted under “/etc/pgbackrest/conf.d”: https://pgbackrest.org/configuration.html

false

image

string

The image name to use for pgBackRest containers. Utilized to run pgBackRest repository hosts and backups. The image may also be set using the RELATED_IMAGE_PGBACKREST environment variable

false

jobs

object

Jobs field allows configuration for all backup jobs

false

manual

object

Defines details for manual pgBackRest backup Jobs

false

metadata

object

Metadata contains metadata for PostgresCluster resources

false

repoHost

object

Defines configuration for a pgBackRest dedicated repository host. This section is only applicable if at least one “volume” (i.e. PVC-based) repository is defined in the “repos” section, therefore enabling a dedicated repository host Deployment.

false

restore

object

Defines details for performing an in-place restore using pgBackRest

false

sidecars

object

Configuration for pgBackRest sidecar containers

false

PostgresCluster.spec.backups.pgbackrest.repos[index]   Parent

PGBackRestRepo represents a pgBackRest repository. Only one of its members may be specified.

Name

Type

Description

Required

name

string

The name of the the repository

true

azure

object

Represents a pgBackRest repository that is created using Azure storage

false

gcs

object

Represents a pgBackRest repository that is created using Google Cloud Storage

false

s3

object

RepoS3 represents a pgBackRest repository that is created using AWS S3 (or S3-compatible) storage

false

schedules

object

Defines the schedules for the pgBackRest backups Full, Differential and Incremental backup types are supported: https://pgbackrest.org/user-guide.html#concept/backup

false

volume

	object	Represents a pgBackRest repository that is created using a PersistentVolumeClaim	false

PostgresCluster.spec.backups.pgbackrest.repos[index].azure Parent

Represents a pgBackRest repository that is created using Azure storage

Name	Type	Description	Required
container	string	The Azure container utilized for the repository	true

PostgresCluster.spec.backups.pgbackrest.repos[index].gcs Parent

Represents a pgBackRest repository that is created using Google Cloud Storage

Name	Type	Description	Required
bucket	string	The GCS bucket utilized for the repository	true

PostgresCluster.spec.backups.pgbackrest.repos[index].s3 Parent

RepoS3 represents a pgBackRest repository that is created using AWS S3 (or S3-compatible) storage

Name	Type	Description	Required
bucket	string	The S3 bucket utilized for the repository	true
endpoint	string	A valid endpoint corresponding to the specified region	true
region	string	The region corresponding to the S3 bucket	true

PostgresCluster.spec.backups.pgbackrest.repos[index].schedules Parent

Defines the schedules for the pgBackRest backups Full, Differential and Incremental backup types are supported: https://pgbackrest.org/user-guide.html#concept/backup

Name	Type	Description	Required
differential	string	Defines the Cron schedule for a differential pgBackRest backup. Follows the standard Cron schedule syntax: https://k8s.io/docs/concepts/wo jobs/#cron-schedule-syntax	false
full	string	Defines the Cron schedule for a full pgBackRest backup. Follows the standard Cron schedule syntax: https://k8s.io/docs/concepts/workloads jobs/#cron-schedule-syntax	false
incremental	string	Defines the Cron schedule for an incremental pgBackRest backup. Follows the standard Cron schedule syntax: https://k8s.io/docs/concepts/ jobs/#cron-schedule-syntax	false

PostgresCluster.spec.backups.pgbackrest.repos[index].volume Parent

Represents a pgBackRest repository that is created using a PersistentVolumeClaim

Name	Type	Description	Required
volumeClaimSpec	object	Defines a PersistentVolumeClaim spec used to create and/or bind a volume	true

PostgresCluster.spec.backups.pgbackrest.repos[index].volume.volumeClaimSpec Parent

Defines a PersistentVolumeClaim spec used to create and/or bind a volume

Name	Type	Description	Required
accessModes	[]string	AccessModes contains the desired access modes the volume should have. More info: https://kubernetes.io/docs/concepts/storage/persistent-volumes#access-modes-1	true
resources	object	Resources represents the minimum resources the volume should have. More info: https://kubernetes.io/docs/concepts/storage/persistent-volumes#resources	true

dataSource

object

This field can be used to specify either: * An existing VolumeSnapshot object (snapshot.storage.k8s.io/VolumeSnapshot) * An existing PVC (PersistentVolumeClaim) * An existing custom resource that implements data population (Alpha) In order to use custom resource types that implement data population, the AnyVolumeDataSource feature gate must be enabled. If the provisioner or an external controller can support the specified data source, it will create a new volume based on the contents of the specified data source.

false

selector

object

A label query over volumes to consider for binding.

false

storageClassName

string

Name of the StorageClass required by the claim. More info: https://kubernetes.io/docs/concepts/storage/persistent-volumes#class-1

false

volumeMode

string

volumeMode defines what type of volume is required by the claim. Value of Filesystem is implied when not included in claim spec.

false

volumeName

string

VolumeName is the binding reference to the PersistentVolume backing this claim.

false

PostgresCluster.spec.backups.pgbackrest.repos[index].volume.volumeClaimSpec.resources Parent

Resources represents the minimum resources the volume should have. More info: https://kubernetes.io/docs/concepts/storage/persistent-volumes#resources

Name

Type

Description

Required

requests

map[string]int or string

Requests describes the minimum amount of compute resources required. If Requests is omitted for a container, it defaults to Limits if that is explicitly specified, otherwise to an implementation-defined value. More info: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

true

limits

map[string]int or string

Limits describes the maximum amount of compute resources allowed. More info: https://kubernetes.io/docs/concepts/configuration/manage
compute-resources-container/

false

PostgresCluster.spec.backups.pgbackrest.repos[index].volume.volumeClaimSpec.dataSource Parent

This field can be used to specify either: * An existing VolumeSnapshot object (snapshot.storage.k8s.io/VolumeSnapshot) * An existing PVC (PersistentVolumeClaim) * An existing custom resource that implements data population (Alpha) In order to use custom resource types that implement data population, the AnyVolumeDataSource feature gate must be enabled. If the provisioner or an external controller can support the specified data source, it will create a new volume based on the contents of the specified data source.

Name

	Type	Description	Required
kind	string	Kind is the type of resource being referenced	true
name	string	Name is the name of resource being referenced	true
apiGroup	string	APIGroup is the group for the resource being referenced. If APIGroup is not specified, the specified Kind must be in the core API group. For any other third-party types, APIGroup is required.	false

PostgresCluster.spec.backups.pgbackrest.repos[index].volume.volumeClaimSpec.selector   Parent

A label query over volumes to consider for binding.

Name	Type	Description	Required
matchExpressions	[]object	matchExpressions is a list of label selector requirements. The requirements are ANDed.	false
matchLabels	map[string]string	matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.	false

PostgresCluster.spec.backups.pgbackrest.repos[index].volume.volumeClaimSpec.selector.matchExpressions[index]   Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name	Type	Description	Required
key	string	key is the label key that the selector applies to.	true
operator	string	operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.	

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.configuration[index]   Parent

Projection that may be projected along with other supported volume types

Name

Type

Description

Required

configMap

object

information about the configMap data to project

false

downwardAPI

object

information about the downwardAPI data to project

false

secret

object

information about the secret data to project

false

serviceAccountToken

object

information about the serviceAccountToken data to project

false

PostgresCluster.spec.backups.pgbackrest.configuration[index].configMap   Parent

information about the configMap data to project

Name

Type

Description

Required

items

[]object

If unspecified, each key-value pair in the Data field of the referenced ConfigMap will be projected into the volume as a file whose name is the key and content is the value. If specified, the listed keys will be projected into the specified paths, and unlisted keys will not be present. If a key is specified which is not present in the ConfigMap, the volume setup will error unless it is marked optional. Paths must be relative and may not contain the '..' path or start with '..'.

false

name

string

Name of the referent. More info: https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#names TODO: Add other useful fields. apiVersion, kind, uid?

false

optional

boolean

Specify whether the ConfigMap or its keys must be defined

false

PostgresCluster.spec.backups.pgbackrest.configuration[index].configMap.items[index]   Parent

Maps a string key to a path within a volume.

Name

Type

Description

Required

key

string

The key to project.

true

path

string

The relative path of the file to map the key to. May not be an absolute path. May not contain the path element '..'. May not start with the string '..'.

true

mode

integer

Optional: mode bits used to set permissions on this file. Must be an octal value between 0000 and 0777 or a decimal value between 0 and 511. YAML accepts both octal and decimal values, JSON requires decimal values for mode bits. If not specified, the volume defaultMode will be used. This might be in conflict with other options that affect the file mode, like fsGroup, and the result can be other mode bits set.

false

PostgresCluster.spec.backups.pgbackrest.configuration[index].downwardAPI   Parent

information about the downwardAPI data to project

Name

Type

Description

Required

items

[]object

Items is a list of DownwardAPIVolume file

false

PostgresCluster.spec.backups.pgbackrest.configuration[index].downwardAPI.items[index]   Parent

DownwardAPIVolumeFile represents information to create the file containing the pod field

Name

Type

Description

Required

path

string

Required: Path is the relative path name of the file to be created. Must not be absolute or contain the '..' path. Must be utf-8 encoded. The first item of the relative path must not start with '..'

true

fieldRef

object

Required: Selects a field of the pod: only annotations, labels, name and namespace are supported.

false

mode

integer

Optional: mode bits used to set permissions on this file, must be an octal value between 0000 and 0777 or a decimal value between 0 and 511. YAML accepts both octal and decimal values, JSON requires decimal values for mode bits. If not specified, the volume defaultMode will be used. This might be in conflict with other options that affect the file mode, like fsGroup, and the result can be other mode bits set.

false

resourceFieldRef

object

Selects a resource of the container: only resources limits and requests (limits.cpu, limits.memory, requests.cpu and requests.memory) are currently supported.

false

PostgresCluster.spec.backups.pgbackrest.configuration[index].downwardAPI.items[index].fieldRef   Parent

Required: Selects a field of the pod: only annotations, labels, name and namespace are supported.

Name

Type

Description

Required

fieldPath

string

Path of the field to select in the specified API version.

true

apiVersion

string

Version of the schema the FieldPath is written in terms of, defaults to "v1".

false

PostgresCluster.spec.backups.pgbackrest.configuration[index].downwardAPI.items[index].resourceFieldRef   Parent

Selects a resource of the container: only resources limits and requests (limits.cpu, limits.memory, requests.cpu and requests.memory) are currently supported.

Name

Type

Description

Required

resource

string

Required: resource to select

true

containerName

string

Container name: required for volumes, optional for env vars

false

divisor

int or string

Specifies the output format of the exposed resources, defaults to “1”

false

PostgresCluster.spec.backups.pgbackrest.configuration[index].secret   Parent

information about the secret data to project

Name

Type

Description

Required

items

[]object

If unspecified, each key-value pair in the Data field of the referenced Secret will be projected into the volume as a file whose name is the key and content is the value. If specified, the listed keys will be projected into the specified paths, and unlisted keys will not be present. If a key is specified which is not present in the Secret, the volume setup will error unless it is marked optional. Paths must be relative and may not contain the ‘..’ path or start with ‘..’.

false

name

string

Name of the referent. More info: https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#names TODO: Add other useful fields. apiVersion, kind, uid?

false

optional

boolean

Specify whether the Secret or its key must be defined

false

PostgresCluster.spec.backups.pgbackrest.configuration[index].secret.items[index]   Parent

Maps a string key to a path within a volume.

Name

Type

Description

Required

key

string

The key to project.

true

path

string

The relative path of the file to map the key to. May not be an absolute path. May not contain the path element ‘..’. May not start with the string ‘..’.

true

mode

integer

Optional: mode bits used to set permissions on this file. Must be an octal value between 0000 and 0777 or a decimal value between 0 and 511. YAML accepts both octal and decimal values, JSON requires decimal values for mode bits. If not specified, the volume defaultMode will be used. This might be in conflict with other options that affect the file mode, like fsGroup, and the result can be other mode bits set.

false

PostgresCluster.spec.backups.pgbackrest.configuration[index].serviceAccountToken   Parent

information about the serviceAccountToken data to project

Name	Type	Description	Required
path	string	Path is the path relative to the mount point of the file to project the token into.	true
audience	string	Audience is the intended audience of the token. A recipient of a token must identify itself with an identifier specified in the audience of the token, and otherwise should reject the token. The audience defaults to the identifier of the apiserver.	false
expirationSeconds	integer	ExpirationSeconds is the requested duration of validity of the service account token. As the token approaches expiration, the kubelet volume plugin will proactively rotate the service account token. The kubelet will start trying to rotate the token if the token is older than 80 percent of its time to live or if the token is older than 24 hours.Defaults to 1 hour and must be at least 10 minutes.	false

PostgresCluster.spec.backups.pgbackrest.jobs   Parent

Jobs field allows configuration for all backup jobs

Name	Type	Description	Required
priorityClassName	string	Priority class name for the pgBackRest backup Job pods. Changing this value causes PostgreSQL to restart. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/	false
resources	object	Resource limits for backup jobs. Includes manual, scheduled and replica create backups	false

PostgresCluster.spec.backups.pgbackrest.jobs.resources   Parent

Resource limits for backup jobs. Includes manual, scheduled and replica create backups

Name	Type	Description	Required

limits	map[string]int or string	Limits describes the maximum amount of compute resources allowed. More info: https://kubernetes.io/docs/concepts/configuration/manage compute-resources-container/	false
requests	map[string]int or string	Requests describes the minimum amount of compute resources required. If Requests is omitted for a container, it defaults to Limits if that is explicitly specified, otherwise to an implementation-defined value. More info: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/	false

PostgresCluster.spec.backups.pgbackrest.manual   Parent

Defines details for manual pgBackRest backup Jobs

Name	Type	Description	Required
repoName	string	The name of the pgBackRest repo to run the backup command against.	true
options	[]string	Command line options to include when running the pgBackRest backup command. https://pgbackrest.org/command.html#command-backup	false

PostgresCluster.spec.backups.pgbackrest.metadata   Parent

Metadata contains metadata for PostgresCluster resources

Name	Type	Description	Required
annotations	map[string]string		false
labels	map[string]string		false

PostgresCluster.spec.backups.pgbackrest.repoHost   Parent

Defines configuration for a pgBackRest dedicated repository host. This section is only applicable if at least one “volume” (i.e. PVC-based) repository is defined in the “repos” section, therefore enabling a dedicated repository host Deployment.

Name	Type	Description	Required

affinity

object

Scheduling constraints of the Dedicated repo host pod. Changing this value causes repo host to restart. More info: https://kubernetes.io/docs
eviction/assign-pod-node

false

priorityClassName

string

Priority class name for the pgBackRest repo host pod. Changing this value causes PostgreSQL to restart. More info: https://kubernetes.io/do
eviction/pod-priority-preemption/

false

resources

object

Resource requirements for a pgBackRest repository host

false

sshConfigMap

object

ConfigMap containing custom SSH configuration

false

sshSecret

object

Secret containing custom SSH keys

false

tolerations

[]object

Tolerations of a PgBackRest repo host pod. Changing this value causes a restart. More info: https://kubernetes.io/docs/concepts/scheduling-
eviction/taint-and-toleration

false

topologySpreadConstraints

[]object

Topology spread constraints of a Dedicated repo host pod. Changing this value causes the repo host to restart. More info: https://kubernetes.io/docs/concepts/workloads/pods/pod-topology-spread-constraints/

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity Parent

Scheduling constraints of the Dedicated repo host pod. Changing this value causes repo host to restart. More info: https://kubernetes.io/docs
eviction/assign-pod-node

Name

Type

Description

Required

nodeAffinity

object

Describes node affinity scheduling rules for the pod.

false

podAffinity

object

Describes pod affinity scheduling rules (e.g. co-locate this pod in the same node, zone, etc. as some other pod(s)).

false

podAntiAffinity

object

Describes pod anti-affinity scheduling rules (e.g. avoid putting this pod in the same node, zone, etc. as some other pod(s)).

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.nodeAffinity Parent

Describes node affinity scheduling rules for the pod.

Name

Type

Description

Required

preferredDuringSchedulingIgnoredDuringExecution

[]object

The scheduler will prefer to schedule pods to nodes that satisfy the affinity expressions specified by this field, but it may choose a node that violates one or more of the expressions. The node that is most preferred is the one with the greatest sum of weights, i.e. for each node that meets all of the scheduling requirements (resource request, requiredDuringScheduling affinity expressions, etc.), compute a sum by iterating through the elements of this field and adding "weight" to the sum if the node matches the corresponding matchExpressions; the node(s) with the highest sum are the most preferred.

false

requiredDuringSchedulingIgnoredDuringExecution

object

If the affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to an update), the system may or may not try to eventually evict the pod from its node.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index] Parent

An empty preferred scheduling term matches all objects with implicit weight 0 (i.e. it's a no-op). A null preferred scheduling term matches no objects (i.e. is also a no-op).

Name

Type

Description

Required

preference

object

A node selector term, associated with the corresponding weight.

true

weight

integer

Weight associated with matching the corresponding nodeSelectorTerm, in the range 1-100.

true

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].preferenc
Parent

A node selector term, associated with the corresponding weight.

Name

Type

Description

Required

matchExpressions

[]object

A list of node selector requirements by node’s labels.

false

matchFields

[]object

A list of node selector requirements by node’s fields.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].preferenc
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].preferenc
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution Parent

If the affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to an update), the system may or may not try to eventually evict the pod from its node.

Name

Type

Description

Required

nodeSelectorTerms

[]object

Required. A list of node selector terms. The terms are ORed.

true

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTer
Parent

A null or empty node selector term matches no objects. The requirements of them are ANDed. The TopologySelectorTerm type implements a subset of the NodeSelectorTerm.

Name

Type

Description

Required

matchExpressions

[]object

A list of node selector requirements by node's labels.

false

matchFields

[]object

A list of node selector requirements by node's fields.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTer
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTer
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.podAffinity Parent

Describes pod affinity scheduling rules (e.g. co-locate this pod in the same node, zone, etc. as some other pod(s)).

Name

Type

Description

Required

preferredDuringSchedulingIgnoredDuringExecution

[]object

The scheduler will prefer to schedule pods to nodes that satisfy the affinity expressions specified by this field, but it may choose a node that violates one or more of the expressions. The node that is most preferred is the one with the greatest sum of weights, i.e. for each node that meets all of the scheduling requirements (resource request, requiredDuringScheduling affinity expressions, etc.), compute a sum by iterating through the elements of this field and adding "weight" to the sum if the node has pods which matches the corresponding podAffinityTerm; the node(s) with the highest sum are the most preferred.

false

requiredDuringSchedulingIgnoredDuringExecution

[]object

If the affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to a pod label update), the system may or may not try to eventually evict the pod from its node. When there are multiple elements, the lists of nodes corresponding to each podAffinityTerm are intersected, i.e. all terms must be satisfied.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index]   Parent

The weights of all of the matched WeightedPodAffinityTerm fields are added per-node to find the most preferred node(s)

Name

Type

Description

Required

podAffinityTerm

object

Required. A pod affinity term, associated with the corresponding weight.

true

weight

integer

weight associated with matching the corresponding podAffinityTerm, in the range 1-100.

true

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinit
Parent

Required. A pod affinity term, associated with the corresponding weight.

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means “this pod’s namespace”

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinit
Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinit
 Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution[index] Parent

Defines a set of pods (namely those matching the labelSelector relative to the given namespace(s)) that this pod should be co-located (affinity) or not co-located (anti-affinity) with, where co-located is defined as running on a node whose value of the label with key matches that of any node on which a pod of the set of pods is running

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means “this pod’s namespace”

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSelecto
Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSelecto
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.podAntiAffinity Parent

Describes pod anti-affinity scheduling rules (e.g. avoid putting this pod in the same node, zone, etc. as some other pod(s)).

Name

Type

Description

Required

preferredDuringSchedulingIgnoredDuringExecution

[]object

The scheduler will prefer to schedule pods to nodes that satisfy the anti-affinity expressions specified by this field, but it may choose a node that violates one or more of the expressions. The node that is most preferred is the one with the greatest sum of weights, i.e. for each node that meets all of the scheduling requirements (resource request, requiredDuringScheduling anti-affinity expressions, etc.), compute a sum by iterating through the elements of this field and adding "weight" to the sum if the node has pods which matches the corresponding podAffinityTerm; the node(s) with the highest sum are the most preferred.

false

requiredDuringSchedulingIgnoredDuringExecution

[]object

If the anti-affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the anti-affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to a pod label update), the system may or may not try to eventually evict the pod from its node. When there are multiple elements, the lists of nodes corresponding to each podAffinityTerm are intersected, i.e. all terms must be satisfied.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index]
Parent

The weights of all of the matched WeightedPodAffinityTerm fields are added per-node to find the most preferred node(s)

Name

Type

Description

Required

podAffinityTerm

object

Required. A pod affinity term, associated with the corresponding weight.

true

weight

integer

weight associated with matching the corresponding podAffinityTerm, in the range 1-100.

true

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podA
Parent

Required. A pod affinity term, associated with the corresponding weight.

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means “this pod’s namespace”

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAı
Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAı
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution[index]
Parent

Defines a set of pods (namely those matching the labelSelector relative to the given namespace(s)) that this pod should be co-located (affinity) or not co-located (anti-affinity) with, where co-located is defined as running on a node whose value of the label with key matches that of any node on which a pod of the set of pods is running

Name	Type	Description	Required
topologyKey	string	This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.	true
labelSelector	object	A label query over a set of resources, in this case pods.	false
namespaces	[]string	namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means "this pod's namespace"	false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSe
Parent

A label query over a set of resources, in this case pods.

Name	Type	Description	Required
matchExpressions	[]object	matchExpressions is a list of label selector requirements. The requirements are ANDed.	false
matchLabels	map[string]string	matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is "key", the operator is "In", and the values array contains only "value". The requirements are ANDed.	false

PostgresCluster.spec.backups.pgbackrest.repoHost.affinity.podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSe
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name	Type	Description	Required
key	string		

key is the label key that the selector applies to.

true

operator

string

operator represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.resources Parent

Resource requirements for a pgBackRest repository host

Name

Type

Description

Required

limits

map[string]int or string

Limits describes the maximum amount of compute resources allowed. More info: https://kubernetes.io/docs/concepts/configuration/manage
compute-resources-container/

false

requests

map[string]int or string

Requests describes the minimum amount of compute resources required. If Requests is omitted for a container, it defaults to Limits if that is explicitly specified, otherwise to an implementation-defined value. More info: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

false

PostgresCluster.spec.backups.pgbackrest.repoHost.sshConfigMap Parent

ConfigMap containing custom SSH configuration

Name

Type

Description

Required

items

[]object

If unspecified, each key-value pair in the Data field of the referenced ConfigMap will be projected into the volume as a file whose name is the key and content is the value. If specified, the listed keys will be projected into the specified paths, and unlisted keys will not be present. If a key is specified which is not present in the ConfigMap, the volume setup will error unless it is marked optional. Paths must be relative and may not contain the '..' path or start with '..'.

false

name

string

Name of the referent. More info: https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#names TODO: Add other useful fields. apiVersion, kind, uid?

false

optional

boolean

Specify whether the ConfigMap or its keys must be defined

false

PostgresCluster.spec.backups.pgbackrest.repoHost.sshConfigMap.items[index]   Parent

Maps a string key to a path within a volume.

Name

Type

Description

Required

key

string

The key to project.

true

path

string

The relative path of the file to map the key to. May not be an absolute path. May not contain the path element '..'. May not start with the string '..'.

true

mode

integer

Optional: mode bits used to set permissions on this file. Must be an octal value between 0000 and 0777 or a decimal value between 0 and 511. YAML accepts both octal and decimal values, JSON requires decimal values for mode bits. If not specified, the volume defaultMode will be used. This might be in conflict with other options that affect the file mode, like fsGroup, and the result can be other mode bits set.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.sshSecret   Parent

Secret containing custom SSH keys

Name

Type

Description

Required

items

[]object

If unspecified, each key-value pair in the Data field of the referenced Secret will be projected into the volume as a file whose name is the key and content is the value. If specified, the listed keys will be projected into the specified paths, and unlisted keys will not be present. If a key is specified which is not present in the Secret, the volume setup will error unless it is marked optional. Paths must be relative and may not contain the '..' path or start with '..'.

false

name

string

Name of the referent. More info: https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#names TODO: Add other useful fields. apiVersion, kind, uid?

false

optional

boolean

Specify whether the Secret or its key must be defined

false

PostgresCluster.spec.backups.pgbackrest.repoHost.sshSecret.items[index]   Parent

Maps a string key to a path within a volume.

Name	Type	Description	Required
key	string	The key to project.	true
path	string	The relative path of the file to map the key to. May not be an absolute path. May not contain the path element ‘..’. May not start with the string ‘..’.	true
mode	integer	Optional: mode bits used to set permissions on this file. Must be an octal value between 0000 and 0777 or a decimal value between 0 and 511. YAML accepts both octal and decimal values, JSON requires decimal values for mode bits. If not specified, the volume defaultMode will be used. This might be in conflict with other options that affect the file mode, like fsGroup, and the result can be other mode bits set.	false

PostgresCluster.spec.backups.pgbackrest.repoHost.tolerations[index]   Parent

The pod this Toleration is attached to tolerates any taint that matches the triple using the matching operator .

Name	Type	Description	Required
effect	string	Effect indicates the taint effect to match. Empty means match all taint effects. When specified, allowed values are NoSchedule, PreferNoSchedule and NoExecute.	false
key	string	Key is the taint key that the toleration applies to. Empty means match all taint keys. If the key is empty, operator must be Exists; this combination means to match all values and all keys.	false
operator	string	Operator represents a key’s relationship to the value. Valid operators are Exists and Equal. Defaults to Equal. Exists is equivalent to wildcard for value, so that a pod can tolerate all taints of a particular category.	false
tolerationSeconds	integer		

TolerationSeconds represents the period of time the toleration (which must be of effect NoExecute, otherwise this field is ignored) tolerates the taint. By default, it is not set, which means tolerate the taint forever (do not evict). Zero and negative values will be treated as 0 (evict immediately) by the system.

false

value

string

Value is the taint value the toleration matches to. If the operator is Exists, the value should be empty, otherwise just a regular string.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.topologySpreadConstraints[index]   Parent

TopologySpreadConstraint specifies how to spread matching pods among the given topology.

Name

Type

Description

Required

maxSkew

integer

MaxSkew describes the degree to which pods may be unevenly distributed. When `whenUnsatisfiable=DoNotSchedule`, it is the maximum permitted difference between the number of matching pods in the target topology and the global minimum. For example, in a 3-zone cluster, MaxSkew is set to 1, and pods with the same labelSelector spread as 1/1/0: | zone1 | zone2 | zone3 | | P | P | | - if MaxSkew is 1, incoming pod can only be scheduled to zone3 to become 1/1/1; scheduling it onto zone1(zone2) would make the ActualSkew(2-0) on zone1(zone2) violate MaxSkew(1). - if MaxSkew is 2, incoming pod can be scheduled onto any zone. When `whenUnsatisfiable=ScheduleAnyway`, it is used to give higher precedence to topologies that satisfy it. It's a required field. Default value is 1 and 0 is not allowed.

true

topologyKey

string

TopologyKey is the key of node labels. Nodes that have a label with this key and identical values are considered to be in the same topology. We consider each as a "bucket", and try to put balanced number of pods into each bucket. It's a required field.

true

whenUnsatisfiable

string

WhenUnsatisfiable indicates how to deal with a pod if it doesn't satisfy the spread constraint. - DoNotSchedule (default) tells the scheduler not to schedule it. - ScheduleAnyway tells the scheduler to schedule the pod in any location, but giving higher precedence to topologies that would help reduce the skew. A constraint is considered "Unsatisfiable" for an incoming pod if and only if every possible node assigment for that pod would violate "MaxSkew" on some topology. For example, in a 3-zone cluster, MaxSkew is set to 1, and pods with the same labelSelector spread as 3/1/1: | zone1 | zone2 | zone3 | | P P P | P | P | If WhenUnsatisfiable is set to DoNotSchedule, incoming pod can only be scheduled to zone2(zone3) to become 3/2/1(3/1/2) as ActualSkew(2-1) on zone2(zone3) satisfies MaxSkew(1). In other words, the cluster can still be imbalanced, but scheduler won't make it *more* imbalanced. It's a required field.

true

labelSelector

object

LabelSelector is used to find matching pods. Pods that match this label selector are counted to determine the number of pods in their corresponding topology domain.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.topologySpreadConstraints[index].labelSelector   Parent

LabelSelector is used to find matching pods. Pods that match this label selector are counted to determine the number of pods in their corresponding topology domain.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.

false

PostgresCluster.spec.backups.pgbackrest.repoHost.topologySpreadConstraints[index].labelSelector.matchExpressions[index]   Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.restore   Parent

Defines details for performing an in-place restore using pgBackRest

Name

Type

Description

Required

enabled

boolean

Whether or not in-place pgBackRest restores are enabled for this PostgresCluster.

true

repoName

string

The name of the pgBackRest repo within the source PostgresCluster that contains the backups that should be utilized to perform a pgBackRest restore when initializing the data source for the new PostgresCluster.

true

affinity

	object	Scheduling constraints of the pgBackRest restore Job. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node	false
clusterName	string	The name of an existing PostgresCluster to use as the data source for the new PostgresCluster. Defaults to the name of the PostgresCluster being created if not provided.	false
clusterNamespace	string	The namespace of the cluster specified as the data source using the clusterName field. Defaults to the namespace of the PostgresCluster being created if not provided.	false
options	[]string	Command line options to include when running the pgBackRest restore command. https://pgbackrest.org/command.html#command-restore	false
priorityClassName	string	Priority class name for the pgBackRest restore Job pod. Changing this value causes PostgreSQL to restart. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/	false
resources	object	Resource requirements for the pgBackRest restore Job.	false
tolerations	[]object	Tolerations of the pgBackRest restore Job. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration	false

PostgresCluster.spec.backups.pgbackrest.restore.affinity Parent

Scheduling constraints of the pgBackRest restore Job. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node

Name	Type	Description	Required
nodeAffinity	object	Describes node affinity scheduling rules for the pod.	false
podAffinity	object	Describes pod affinity scheduling rules (e.g. co-locate this pod in the same node, zone, etc. as some other pod(s)).	

false

podAntiAffinity

object

Describes pod anti-affinity scheduling rules (e.g. avoid putting this pod in the same node, zone, etc. as some other pod(s)).

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.nodeAffinity Parent

Describes node affinity scheduling rules for the pod.

Name

Type

Description

Required

preferredDuringSchedulingIgnoredDuringExecution

[]object

The scheduler will prefer to schedule pods to nodes that satisfy the affinity expressions specified by this field, but it may choose a node that violates one or more of the expressions. The node that is most preferred is the one with the greatest sum of weights, i.e. for each node that meets all of the scheduling requirements (resource request, requiredDuringScheduling affinity expressions, etc.), compute a sum by iterating through the elements of this field and adding "weight" to the sum if the node matches the corresponding matchExpressions; the node(s) with the highest sum are the most preferred.

false

requiredDuringSchedulingIgnoredDuringExecution

object

If the affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to an update), the system may or may not try to eventually evict the pod from its node.

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index] Parent

An empty preferred scheduling term matches all objects with implicit weight 0 (i.e. it's a no-op). A null preferred scheduling term matches no objects (i.e. is also a no-op).

Name

Type

Description

Required

preference

object

A node selector term, associated with the corresponding weight.

true

weight

integer

Weight associated with matching the corresponding nodeSelectorTerm, in the range 1-100.

true

PostgresCluster.spec.backups.pgbackrest.restore.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].preference Parent

A node selector term, associated with the corresponding weight.

Name

Type

Description

Required

matchExpressions

[]object

A list of node selector requirements by node's labels.

false

matchFields

[]object

A list of node selector requirements by node's fields.

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].preference.m
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].preference.m
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution   Parent

If the affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to an update), the system may or may not try to eventually evict the pod from its node.

Name

Type

Description

Required

nodeSelectorTerms

[]object

Required. A list of node selector terms. The terms are ORed.

true

PostgresCluster.spec.backups.pgbackrest.restore.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms
Parent

A null or empty node selector term matches no objects. The requirements of them are ANDed. The TopologySelectorTerm type implements a subset of the NodeSelectorTerm.

Name

Type

Description

Required

matchExpressions

[]object

A list of node selector requirements by node's labels.

false

matchFields

[]object

A list of node selector requirements by node's fields.

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms
 Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.podAffinity Parent

Describes pod affinity scheduling rules (e.g. co-locate this pod in the same node, zone, etc. as some other pod(s)).

Name

Type

Description

Required

preferredDuringSchedulingIgnoredDuringExecution

[]object

The scheduler will prefer to schedule pods to nodes that satisfy the affinity expressions specified by this field, but it may choose a node that violates one or more of the expressions. The node that is most preferred is the one with the greatest sum of weights, i.e. for each node that meets all of the scheduling requirements (resource request, requiredDuringScheduling affinity expressions, etc.), compute a sum by iterating through the elements of this field and adding “weight” to the sum if the node has pods which matches the corresponding podAffinityTerm; the node(s) with the highest sum are the most preferred.

false

requiredDuringSchedulingIgnoredDuringExecution

[]object

If the affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to a pod label update), the system may or may not try to eventually evict the pod from its node. When there are multiple elements, the lists of nodes corresponding to each podAffinityTerm are intersected, i.e. all terms must be satisfied.

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index] Parent

The weights of all of the matched WeightedPodAffinityTerm fields are added per-node to find the most preferred node(s)

Name

Type

Description

Required

podAffinityTerm

object

Required. A pod affinity term, associated with the corresponding weight.

true

weight

integer

weight associated with matching the corresponding podAffinityTerm, in the range 1-100.

true

PostgresCluster.spec.backups.pgbackrest.restore.affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityT
Parent

Required. A pod affinity term, associated with the corresponding weight.

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means “this pod’s namespace”

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityT
Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is "key", the operator is "In", and the values array contains only "value". The requirements are ANDed.

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityT
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution[index] Parent

Defines a set of pods (namely those matching the labelSelector relative to the given namespace(s)) that this pod should be co-located (affinity) or not co-located (anti-affinity) with, where co-located is defined as running on a node whose value of the label with key matches that of any node on which a pod of the set of pods is running

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means “this pod’s namespace”

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSelector
Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSelector.
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.podAntiAffinity Parent

Describes pod anti-affinity scheduling rules (e.g. avoid putting this pod in the same node, zone, etc. as some other pod(s)).

Name

Type

Description

Required

preferredDuringSchedulingIgnoredDuringExecution

[]object

The scheduler will prefer to schedule pods to nodes that satisfy the anti-affinity expressions specified by this field, but it may choose a node that violates one or more of the expressions. The node that is most preferred is the one with the greatest sum of weights, i.e. for each node that meets all of the scheduling requirements (resource request, requiredDuringScheduling anti-affinity expressions, etc.), compute a sum by iterating through the elements of this field and adding "weight" to the sum if the node has pods which matches the corresponding podAffinityTerm; the node(s) with the highest sum are the most preferred.

false

requiredDuringSchedulingIgnoredDuringExecution

[]object

If the anti-affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the anti-affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to a pod label update), the system may or may not try to eventually evict the pod from its node. When there are multiple elements, the lists of nodes corresponding to each podAffinityTerm are intersected, i.e. all terms must be satisfied.

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index] Parent

The weights of all of the matched WeightedPodAffinityTerm fields are added per-node to find the most preferred node(s)

Name

Type

Description

Required

podAffinityTerm

object

Required. A pod affinity term, associated with the corresponding weight.

true

weight

integer

weight associated with matching the corresponding podAffinityTerm, in the range 1-100.

true

PostgresCluster.spec.backups.pgbackrest.restore.affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffin
Parent

Required. A pod affinity term, associated with the corresponding weight.

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means “this pod’s namespace”

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffi
Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffi
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution[index]
Parent

Defines a set of pods (namely those matching the labelSelector relative to the given namespace(s)) that this pod should be co-located (affinity) or not co-located (anti-affinity) with, where co-located is defined as running on a node whose value of the label with key matches that of any node on which a pod of the set of pods is running

Name	Type	Description	Required
topologyKey	string	This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.	true
labelSelector	object	A label query over a set of resources, in this case pods.	false
namespaces	[]string	namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means “this pod’s namespace”	false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSele
Parent

A label query over a set of resources, in this case pods.

Name	Type	Description	Required
matchExpressions	[]object	matchExpressions is a list of label selector requirements. The requirements are ANDed.	false
matchLabels	map[string]string	matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.	false

PostgresCluster.spec.backups.pgbackrest.restore.affinity.podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSele
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name	Type	Description	Required
key	string		

key is the label key that the selector applies to.

true

operator

string

operator represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.backups.pgbackrest.restore.resources Parent

Resource requirements for the pgBackRest restore Job.

Name

Type

Description

Required

limits

map[string]int or string

Limits describes the maximum amount of compute resources allowed. More info: https://kubernetes.io/docs/concepts/configuration/manage
compute-resources-container/

false

requests

map[string]int or string

Requests describes the minimum amount of compute resources required. If Requests is omitted for a container, it defaults to Limits if that is explicitly specified, otherwise to an implementation-defined value. More info: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

false

PostgresCluster.spec.backups.pgbackrest.restore.tolerations[index] Parent

The pod this Toleration is attached to tolerates any taint that matches the triple using the matching operator .

Name

Type

Description

Required

effect

string

Effect indicates the taint effect to match. Empty means match all taint effects. When specified, allowed values are NoSchedule, PreferNoSchedule and NoExecute.

false

key

string

Key is the taint key that the toleration applies to. Empty means match all taint keys. If the key is empty, operator must be Exists; this combination means to match all values and all keys.

false

operator

string

Operator represents a key's relationship to the value. Valid operators are Exists and Equal. Defaults to Equal. Exists is equivalent to wildcard for value, so that a pod can tolerate all taints of a particular category.

false

tolerationSeconds

integer

TolerationSeconds represents the period of time the toleration (which must be of effect NoExecute, otherwise this field is ignored) tolerates the taint. By default, it is not set, which means tolerate the taint forever (do not evict). Zero and negative values will be treated as 0 (evict immediately) by the system.

false

value

string

Value is the taint value the toleration matches to. If the operator is Exists, the value should be empty, otherwise just a regular string.

false

PostgresCluster.spec.backups.pgbackrest.sidecars   Parent

Configuration for pgBackRest sidecar containers

Name

Type

Description

Required

pgbackrest

object

Defines the configuration for the pgBackRest sidecar container

false

PostgresCluster.spec.backups.pgbackrest.sidecars.pgbackrest   Parent

Defines the configuration for the pgBackRest sidecar container

Name

Type

Description

Required

resources

object

Resource requirements for a sidecar container

false

PostgresCluster.spec.backups.pgbackrest.sidecars.pgbackrest.resources   Parent

Resource requirements for a sidecar container

Name

Type

Description

Required

limits

map[string]int or string

Limits describes the maximum amount of compute resources allowed. More info: https://kubernetes.io/docs/concepts/configuration/manage
compute-resources-container/

false

requests

map[string]int or string

Requests describes the minimum amount of compute resources required. If Requests is omitted for a container, it defaults to Limits if that is explicitly specified, otherwise to an implementation-defined value. More info: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

false

PostgresCluster.spec.instances[index]   Parent

Name

Type

Description

Required

dataVolumeClaimSpec

object

Defines a PersistentVolumeClaim for PostgreSQL data. More info: https://kubernetes.io/docs/concepts/storage/persistent-volumes

true

affinity

object

Scheduling constraints of a PostgreSQL pod. Changing this value causes PostgreSQL to restart. More info: https://kubernetes.io/docs/conce
eviction/assign-pod-node

false

metadata

object

Metadata contains metadata for PostgresCluster resources

false

name

string

Name that associates this set of PostgreSQL pods. This field is optional when only one instance set is defined. Each instance set in a cluster must have a unique name.

false

priorityClassName

string

Priority class name for the PostgreSQL pod. Changing this value causes PostgreSQL to restart. More info: https://kubernetes.io/docs/concep
eviction/pod-priority-preemption/

false

replicas

integer

Number of desired PostgreSQL pods.

false

resources

object

Compute resources of a PostgreSQL container.

false

sidecars

object

Configuration for instance sidecar containers

false

tolerations

[]object

Tolerations of a PostgreSQL pod. Changing this value causes PostgreSQL to restart. More info: https://kubernetes.io/docs/concepts/schedul
eviction/taint-and-toleration

false

topologySpreadConstraints

[]object

Topology spread constraints of a PostgreSQL pod. Changing this value causes PostgreSQL to restart. More info: https://kubernetes.io/docs/
topology-spread-constraints/

false

walVolumeClaimSpec

object

Defines a separate PersistentVolumeClaim for PostgreSQL's write-ahead log. More info: https://www.postgresql.org/docs/current/wal.html

false

PostgresCluster.spec.instances[index].dataVolumeClaimSpec   Parent

Defines a PersistentVolumeClaim for PostgreSQL data. More info: https://kubernetes.io/docs/concepts/storage/persistent-volumes

Name

Type

Description

Required

accessModes

[]string

AccessModes contains the desired access modes the volume should have. More info: https://kubernetes.io/docs/concepts/storage/persistent-volumes#access-modes-1

true

resources

object

Resources represents the minimum resources the volume should have. More info: https://kubernetes.io/docs/concepts/storage/persistent-volumes#resources

true

dataSource

object

This field can be used to specify either: * An existing VolumeSnapshot object (snapshot.storage.k8s.io/VolumeSnapshot) * An existing PVC (PersistentVolumeClaim) * An existing custom resource that implements data population (Alpha) In order to use custom resource types that implement data population, the AnyVolumeDataSource feature gate must be enabled. If the provisioner or an external controller can support the specified data source, it will create a new volume based on the contents of the specified data source.

false

selector

object

A label query over volumes to consider for binding.

false

storageClassName

string

Name of the StorageClass required by the claim. More info: https://kubernetes.io/docs/concepts/storage/persistent-volumes#class-1

false

volumeMode

string

volumeMode defines what type of volume is required by the claim. Value of Filesystem is implied when not included in claim spec.

false

volumeName

string

VolumeName is the binding reference to the PersistentVolume backing this claim.

false

PostgresCluster.spec.instances[index].dataVolumeClaimSpec.resources Parent

Resources represents the minimum resources the volume should have. More info: https://kubernetes.io/docs/concepts/storage/persistent-volumes#resources

Name

Type

Description

Required

requests

map[string]int or string

Requests describes the minimum amount of compute resources required. If Requests is omitted for a container, it defaults to Limits if that is explicitly specified, otherwise to an implementation-defined value. More info: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

true

limits

map[string]int or string

Limits describes the maximum amount of compute resources allowed. More info: https://kubernetes.io/docs/concepts/configuration/manage
compute-resources-container/

false

PostgresCluster.spec.instances[index].dataVolumeClaimSpec.dataSource Parent

This field can be used to specify either: * An existing VolumeSnapshot object (snapshot.storage.k8s.io/VolumeSnapshot) * An existing PVC (PersistentVolumeClaim) * An existing custom resource that implements data population (Alpha) In order to use custom resource types that implement data population, the AnyVolumeDataSource feature gate must be enabled. If the provisioner or an external controller can support the specified data source, it will create a new volume based on the contents of the specified data source.

Name

Type

Description

Required

kind

string

Kind is the type of resource being referenced

true

name

string

Name is the name of resource being referenced

true

apiGroup

string

APIGroup is the group for the resource being referenced. If APIGroup is not specified, the specified Kind must be in the core API group. For any other third-party types, APIGroup is required.

false

PostgresCluster.spec.instances[index].dataVolumeClaimSpec.selector Parent

A label query over volumes to consider for binding.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is "key", the operator is "In", and the values array contains only "value". The requirements are ANDed.

false

PostgresCluster.spec.instances[index].dataVolumeClaimSpec.selector.matchExpressions[index] Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.instances[index].affinity Parent

Scheduling constraints of a PostgreSQL pod. Changing this value causes PostgreSQL to restart. More info: https://kubernetes.io/docs/conce
eviction/assign-pod-node

Name

Type

Description

Required

nodeAffinity

object

Describes node affinity scheduling rules for the pod.

false

podAffinity

object

Describes pod affinity scheduling rules (e.g. co-locate this pod in the same node, zone, etc. as some other pod(s)).

false

podAntiAffinity

object

Describes pod anti-affinity scheduling rules (e.g. avoid putting this pod in the same node, zone, etc. as some other pod(s)).

false

PostgresCluster.spec.instances[index].affinity.nodeAffinity   Parent

Describes node affinity scheduling rules for the pod.

Name

Type

Description

Required

preferredDuringSchedulingIgnoredDuringExecution

[]object

The scheduler will prefer to schedule pods to nodes that satisfy the affinity expressions specified by this field, but it may choose a node that violates one or more of the expressions. The node that is most preferred is the one with the greatest sum of weights, i.e. for each node that meets all of the scheduling requirements (resource request, requiredDuringScheduling affinity expressions, etc.), compute a sum by iterating through the elements of this field and adding "weight" to the sum if the node matches the corresponding matchExpressions; the node(s) with the highest sum are the most preferred.

false

requiredDuringSchedulingIgnoredDuringExecution

object

If the affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to an update), the system may or may not try to eventually evict the pod from its node.

false

PostgresCluster.spec.instances[index].affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index]   Parent

An empty preferred scheduling term matches all objects with implicit weight 0 (i.e. it's a no-op). A null preferred scheduling term matches no objects (i.e. is also a no-op).

Name

Type

Description

Required

preference

object

A node selector term, associated with the corresponding weight.

true

weight

integer

Weight associated with matching the corresponding nodeSelectorTerm, in the range 1-100.

true

PostgresCluster.spec.instances[index].affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].preference   Parent

A node selector term, associated with the corresponding weight.

Name	Type	Description	Required
matchExpressions	[]object	A list of node selector requirements by node’s labels.	false
matchFields	[]object	A list of node selector requirements by node’s fields.	false

PostgresCluster.spec.instances[index].affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].preference.matchExpre
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name	Type	Description	Required
key	string	The label key that the selector applies to.	true
operator	string	Represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.	true
values	[]string	An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.	false

PostgresCluster.spec.instances[index].affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].preference.matchFields
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name	Type	Description	Required
key	string	The label key that the selector applies to.	true
operator			

string

Represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.instances[index].affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution Parent

If the affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to an update), the system may or may not try to eventually evict the pod from its node.

Name

Type

Description

Required

nodeSelectorTerms

[]object

Required. A list of node selector terms. The terms are ORed.

true

PostgresCluster.spec.instances[index].affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms[index] Parent

A null or empty node selector term matches no objects. The requirements of them are ANDed. The TopologySelectorTerm type implements a subset of the NodeSelectorTerm.

Name

Type

Description

Required

matchExpressions

[]object

A list of node selector requirements by node's labels.

false

matchFields

[]object

A list of node selector requirements by node's fields.

false

PostgresCluster.spec.instances[index].affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms[index].mat
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.instances[index].affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms[index].mat
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.instances[index].affinity.podAffinity Parent

Describes pod affinity scheduling rules (e.g. co-locate this pod in the same node, zone, etc. as some other pod(s)).

Name

Type

Description

Required

preferredDuringSchedulingIgnoredDuringExecution

[]object

The scheduler will prefer to schedule pods to nodes that satisfy the affinity expressions specified by this field, but it may choose a node that violates one or more of the expressions. The node that is most preferred is the one with the greatest sum of weights, i.e. for each node that meets all of the scheduling requirements (resource request, requiredDuringScheduling affinity expressions, etc.), compute a sum by iterating through the elements of this field and adding "weight" to the sum if the node has pods which matches the corresponding podAffinityTerm; the node(s) with the highest sum are the most preferred.

false

requiredDuringSchedulingIgnoredDuringExecution

[]object

If the affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to a pod label update), the system may or may not try to eventually evict the pod from its node. When there are multiple elements, the lists of nodes corresponding to each podAffinityTerm are intersected, i.e. all terms must be satisfied.

false

PostgresCluster.spec.instances[index].affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index] Parent

The weights of all of the matched WeightedPodAffinityTerm fields are added per-node to find the most preferred node(s)

Name

Type

Description

Required

podAffinityTerm

object

Required. A pod affinity term, associated with the corresponding weight.

true

weight

integer

weight associated with matching the corresponding podAffinityTerm, in the range 1-100.

true

PostgresCluster.spec.instances[index].affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityTerm Parent

Required. A pod affinity term, associated with the corresponding weight.

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means "this pod's namespace"

false

PostgresCluster.spec.instances[index].affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityTerm.labelSe
Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is "key", the operator is "In", and the values array contains only "value". The requirements are ANDed.

false

PostgresCluster.spec.instances[index].affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityTerm.labelS
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.instances[index].affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution[index] Parent

Defines a set of pods (namely those matching the labelSelector relative to the given namespace(s)) that this pod should be co-located (affinity) or not co-located (anti-affinity) with, where co-located is defined as running on a node whose value of the label with key matches that of any node on which a pod of the set of pods is running

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means “this pod’s namespace”

false

PostgresCluster.spec.instances[index].affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSelector Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.

false

PostgresCluster.spec.instances[index].affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSelector.matchExpr
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.instances[index].affinity.podAntiAffinity Parent

Describes pod anti-affinity scheduling rules (e.g. avoid putting this pod in the same node, zone, etc. as some other pod(s)).

Name

Type

Description

Required

preferredDuringSchedulingIgnoredDuringExecution

[]object

The scheduler will prefer to schedule pods to nodes that satisfy the anti-affinity expressions specified by this field, but it may choose a node that violates one or more of the expressions. The node that is most preferred is the one with the greatest sum of weights, i.e. for each node that meets all of the scheduling requirements (resource request, requiredDuringScheduling anti-affinity expressions, etc.), compute a sum by iterating through the elements of this field and adding "weight" to the sum if the node has pods which matches the corresponding podAffinityTerm; the node(s) with the highest sum are the most preferred.

false

requiredDuringSchedulingIgnoredDuringExecution

[]object

If the anti-affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the anti-affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to a pod label update), the system may or may not try to eventually evict the pod from its node. When there are multiple elements, the lists of nodes corresponding to each podAffinityTerm are intersected, i.e. all terms must be satisfied.

false

PostgresCluster.spec.instances[index].affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index] Parent

The weights of all of the matched WeightedPodAffinityTerm fields are added per-node to find the most preferred node(s)

Name

Type

Description

Required

podAffinityTerm

object

Required. A pod affinity term, associated with the corresponding weight.

true

weight

integer

weight associated with matching the corresponding podAffinityTerm, in the range 1-100.

true

PostgresCluster.spec.instances[index].affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityTerm Parent

Required. A pod affinity term, associated with the corresponding weight.

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means “this pod’s namespace”

false

PostgresCluster.spec.instances[index].affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityTerm.la
Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.

false

PostgresCluster.spec.instances[index].affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityTerm.la
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.instances[index].affinity.podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution[index] Parent

Defines a set of pods (namely those matching the labelSelector relative to the given namespace(s)) that this pod should be co-located (affinity) or not co-located (anti-affinity) with, where co-located is defined as running on a node whose value of the label with key matches that of any node on which a pod of the set of pods is running

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means “this pod’s namespace”

false

PostgresCluster.spec.instances[index].affinity.podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSelector
Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.

false

PostgresCluster.spec.instances[index].affinity.podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSelector.match
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.instances[index].metadata Parent

Metadata contains metadata for PostgresCluster resources

Name

Type

Description

Required

annotations

map[string]string

false

labels

map[string]string

false

PostgresCluster.spec.instances[index].resources Parent

Compute resources of a PostgreSQL container.

Name

Type

Description

Required

limits

map[string]int or string

Limits describes the maximum amount of compute resources allowed. More info: https://kubernetes.io/docs/concepts/configuration/manage
compute-resources-container/

false

requests

map[string]int or string

Requests describes the minimum amount of compute resources required. If Requests is omitted for a container, it defaults to Limits if that is explicitly specified, otherwise to an implementation-defined value. More info: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

false

PostgresCluster.spec.instances[index].sidecars Parent

Configuration for instance sidecar containers

Name

Type

Description

Required

replicaCertCopy

object

Defines the configuration for the replica cert copy sidecar container

false

PostgresCluster.spec.instances[index].sidecars.replicaCertCopy Parent

Defines the configuration for the replica cert copy sidecar container

Name

Type

Description

Required

resources

object

Resource requirements for a sidecar container

false

PostgresCluster.spec.instances[index].sidecars.replicaCertCopy.resources Parent

Resource requirements for a sidecar container

Name

Type

Description

Required

limits

map[string]int or string

Limits describes the maximum amount of compute resources allowed. More info: https://kubernetes.io/docs/concepts/configuration/manage
compute-resources-container/

false

requests

map[string]int or string

Requests describes the minimum amount of compute resources required. If Requests is omitted for a container, it defaults to Limits if that is explicitly specified, otherwise to an implementation-defined value. More info: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

false

PostgresCluster.spec.instances[index].tolerations[index] Parent

The pod this Toleration is attached to tolerates any taint that matches the triple using the matching operator .

Name

Type

Description

Required

effect

string

Effect indicates the taint effect to match. Empty means match all taint effects. When specified, allowed values are NoSchedule, PreferNoSchedule and NoExecute.

false

key

string

Key is the taint key that the toleration applies to. Empty means match all taint keys. If the key is empty, operator must be Exists; this combination means to match all values and all keys.

false

operator

string

Operator represents a key's relationship to the value. Valid operators are Exists and Equal. Defaults to Equal. Exists is equivalent to wildcard for value, so that a pod can tolerate all taints of a particular category.

false

tolerationSeconds

integer

TolerationSeconds represents the period of time the toleration (which must be of effect NoExecute, otherwise this field is ignored) tolerates the taint. By default, it is not set, which means tolerate the taint forever (do not evict). Zero and negative values will be treated as 0 (evict immediately) by the system.

false

value

string

Value is the taint value the toleration matches to. If the operator is Exists, the value should be empty, otherwise just a regular string.

false

PostgresCluster.spec.instances[index].topologySpreadConstraints[index]   Parent

TopologySpreadConstraint specifies how to spread matching pods among the given topology.

Name

Type

Description

Required

maxSkew

integer

MaxSkew describes the degree to which pods may be unevenly distributed. When `whenUnsatisfiable=DoNotSchedule`, it is the maximum permitted difference between the number of matching pods in the target topology and the global minimum. For example, in a 3-zone cluster, MaxSkew is set to 1, and pods with the same labelSelector spread as 1/1/0: | zone1 | zone2 | zone3 | | P | P | | - if MaxSkew is 1, incoming pod can only be scheduled to zone3 to become 1/1/1; scheduling it onto zone1(zone2) would make the ActualSkew(2-0) on zone1(zone2) violate MaxSkew(1). - if MaxSkew is 2, incoming pod can be scheduled onto any zone. When `whenUnsatisfiable=ScheduleAnyway`, it is used to give higher precedence to topologies that satisfy it. It's a required field. Default value is 1 and 0 is not allowed.

true

topologyKey

string

TopologyKey is the key of node labels. Nodes that have a label with this key and identical values are considered to be in the same topology. We consider each as a "bucket", and try to put balanced number of pods into each bucket. It's a required field.

true

whenUnsatisfiable

string

WhenUnsatisfiable indicates how to deal with a pod if it doesn't satisfy the spread constraint. - DoNotSchedule (default) tells the scheduler not to schedule it. - ScheduleAnyway tells the scheduler to schedule the pod in any location, but giving higher precedence to topologies that would help reduce the skew. A constraint is considered "Unsatisfiable" for an incoming pod if and only if every possible node assigment for that pod would violate "MaxSkew" on some topology. For example, in a 3-zone cluster, MaxSkew is set to 1, and pods with the same labelSelector spread as 3/1/1: | zone1 | zone2 | zone3 | | P P P | P | P | If WhenUnsatisfiable is set to DoNotSchedule, incoming pod can only be scheduled to zone2(zone3) to become 3/2/1(3/1/2) as ActualSkew(2-1) on zone2(zone3) satisfies MaxSkew(1). In other words, the cluster can still be imbalanced, but scheduler won't make it *more* imbalanced. It's a required field.

true

labelSelector

object

LabelSelector is used to find matching pods. Pods that match this label selector are counted to determine the number of pods in their corresponding topology domain.

false

PostgresCluster.spec.instances[index].topologySpreadConstraints[index].labelSelector   Parent

LabelSelector is used to find matching pods. Pods that match this label selector are counted to determine the number of pods in their corresponding topology domain.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is "key", the operator is "In", and the values array contains only "value". The requirements are ANDed.

false

PostgresCluster.spec.instances[index].topologySpreadConstraints[index].labelSelector.matchExpressions[index]   Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.instances[index].walVolumeClaimSpec   Parent

Defines a separate PersistentVolumeClaim for PostgreSQL's write-ahead log. More info: https://www.postgresql.org/docs/current/wal.html

Name

Type

Description

Required

accessModes

[]string

AccessModes contains the desired access modes the volume should have. More info: https://kubernetes.io/docs/concepts/storage/persistent-volumes#access-modes-1

true

resources

object

Resources represents the minimum resources the volume should have. More info: https://kubernetes.io/docs/concepts/storage/persistent-volumes#resources

true

dataSource

object

This field can be used to specify either: * An existing VolumeSnapshot object (snapshot.storage.k8s.io/VolumeSnapshot) * An existing PVC (PersistentVolumeClaim) * An existing custom resource that implements data population (Alpha) In order to use custom resource types that implement data population, the AnyVolumeDataSource feature gate must be enabled. If the provisioner or an external controller can support the specified data source, it will create a new volume based on the contents of the specified data source.

false

selector

object

A label query over volumes to consider for binding.

false

storageClassName

string

Name of the StorageClass required by the claim. More info: https://kubernetes.io/docs/concepts/storage/persistent-volumes#class-1

false

volumeMode

string

volumeMode defines what type of volume is required by the claim. Value of Filesystem is implied when not included in claim spec.

false

volumeName

string

VolumeName is the binding reference to the PersistentVolume backing this claim.

false

PostgresCluster.spec.instances[index].walVolumeClaimSpec.resources   Parent

Resources represents the minimum resources the volume should have. More info: https://kubernetes.io/docs/concepts/storage/persistent-volumes#resources

Name

Type

Description

Required

requests

map[string]int or string

Requests describes the minimum amount of compute resources required. If Requests is omitted for a container, it defaults to Limits if that is explicitly specified, otherwise to an implementation-defined value. More info: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

true

limits

map[string]int or string

Limits describes the maximum amount of compute resources allowed. More info: https://kubernetes.io/docs/concepts/configuration/manage
compute-resources-container/

false

PostgresCluster.spec.instances[index].walVolumeClaimSpec.dataSource Parent

This field can be used to specify either: * An existing VolumeSnapshot object (snapshot.storage.k8s.io/VolumeSnapshot) * An existing PVC (PersistentVolumeClaim) * An existing custom resource that implements data population (Alpha) In order to use custom resource types that implement data population, the AnyVolumeDataSource feature gate must be enabled. If the provisioner or an external controller can support the specified data source, it will create a new volume based on the contents of the specified data source.

Name

Type

Description

Required

kind

string

Kind is the type of resource being referenced

true

name

string

Name is the name of resource being referenced

true

apiGroup

string

APIGroup is the group for the resource being referenced. If APIGroup is not specified, the specified Kind must be in the core API group. For any other third-party types, APIGroup is required.

false

PostgresCluster.spec.instances[index].walVolumeClaimSpec.selector Parent

A label query over volumes to consider for binding.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.

false

PostgresCluster.spec.instances[index].walVolumeClaimSpec.selector.matchExpressions[index] Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

		Description	Required
key	string	key is the label key that the selector applies to.	true
operator	string	operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.	true
values	[]string	values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.	false

PostgresCluster.spec.customReplicationTLSSecret Parent

The secret containing the replication client certificates and keys for secure connections to the PostgreSQL server. It will need to contain the client TLS certificate, TLS key and the Certificate Authority certificate with the data keys set to tls.crt, tls.key and ca.crt, respectively. NOTE: If CustomReplicationClientTLSSecret is provided, CustomTLSSecret MUST be provided and the ca.crt provided must be the same.

Name	Type	Description	Required
items	[]object	If unspecified, each key-value pair in the Data field of the referenced Secret will be projected into the volume as a file whose name is the key and content is the value. If specified, the listed keys will be projected into the specified paths, and unlisted keys will not be present. If a key is specified which is not present in the Secret, the volume setup will error unless it is marked optional. Paths must be relative and may not contain the ‘..’ path or start with ‘..’.	false
name	string	Name of the referent. More info: https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#names TODO: Add other useful fields. apiVersion, kind, uid?	false
optional	boolean	Specify whether the Secret or its key must be defined	false

PostgresCluster.spec.customReplicationTLSSecret.items[index] Parent

Maps a string key to a path within a volume.

Name	Type	Description	Required

key	string	The key to project.	true
path	string	The relative path of the file to map the key to. May not be an absolute path. May not contain the path element ‘..’. May not start with the string ‘..’.	true
mode	integer	Optional: mode bits used to set permissions on this file. Must be an octal value between 0000 and 0777 or a decimal value between 0 and 511. YAML accepts both octal and decimal values, JSON requires decimal values for mode bits. If not specified, the volume defaultMode will be used. This might be in conflict with other options that affect the file mode, like fsGroup, and the result can be other mode bits set.	false

PostgresCluster.spec.customTLSSecret   Parent

The secret containing the Certificates and Keys to encrypt PostgreSQL traffic will need to contain the server TLS certificate, TLS key and the Certificate Authority certificate with the data keys set to tls.crt, tls.key and ca.crt, respectively. It will then be mounted as a volume projection to the ‘/pgconf/tls’ directory. For more information on Kubernetes secret projections, please see https://k8s.io/docs/concepts/configuration/secret/#projection-of-secret-keys-to-specific-paths NOTE: If CustomTLSSecret is provided, CustomReplicationClientTLSSecret MUST be provided and the ca.crt provided must be the same.

Name	Type	Description	Required
items	[]object	If unspecified, each key-value pair in the Data field of the referenced Secret will be projected into the volume as a file whose name is the key and content is the value. If specified, the listed keys will be projected into the specified paths, and unlisted keys will not be present. If a key is specified which is not present in the Secret, the volume setup will error unless it is marked optional. Paths must be relative and may not contain the ‘..’ path or start with ‘..’.	false
name	string	Name of the referent. More info: https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#names TODO: Add other useful fields. apiVersion, kind, uid?	false
optional	boolean	Specify whether the Secret or its key must be defined	false

PostgresCluster.spec.customTLSSecret.items[index]   Parent

Maps a string key to a path within a volume.

Name	Type	Description	Required

key	string	The key to project.	true
path	string	The relative path of the file to map the key to. May not be an absolute path. May not contain the path element '..'. May not start with the string '..'.	true
mode	integer	Optional: mode bits used to set permissions on this file. Must be an octal value between 0000 and 0777 or a decimal value between 0 and 511. YAML accepts both octal and decimal values, JSON requires decimal values for mode bits. If not specified, the volume defaultMode will be used. This might be in conflict with other options that affect the file mode, like fsGroup, and the result can be other mode bits set.	false

PostgresCluster.spec.dataSource   Parent

Specifies a data source for bootstrapping the PostgreSQL cluster.

Name	Type	Description	Required
postgresCluster	object	Defines a pgBackRest data source that can be used to pre-populate the PostgreSQL data directory for a new PostgreSQL cluster using a pgBackRest restore.	false
volumes	object	Defines any existing volumes to reuse for this PostgresCluster.	false

PostgresCluster.spec.dataSource.postgresCluster   Parent

Defines a pgBackRest data source that can be used to pre-populate the PostgreSQL data directory for a new PostgreSQL cluster using a pgBackRest restore.

Name	Type	Description	Required
repoName	string	The name of the pgBackRest repo within the source PostgresCluster that contains the backups that should be utilized to perform a pgBackRest restore when initializing the data source for the new PostgresCluster.	true
affinity	object	Scheduling constraints of the pgBackRest restore Job. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node	

false

clusterName

string

The name of an existing PostgresCluster to use as the data source for the new PostgresCluster. Defaults to the name of the PostgresCluster being created if not provided.

false

clusterNamespace

string

The namespace of the cluster specified as the data source using the clusterName field. Defaults to the namespace of the PostgresCluster being created if not provided.

false

options

[]string

Command line options to include when running the pgBackRest restore command. https://pgbackrest.org/command.html#command-restore

false

priorityClassName

string

Priority class name for the pgBackRest restore Job pod. Changing this value causes PostgreSQL to restart. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/

false

resources

object

Resource requirements for the pgBackRest restore Job.

false

tolerations

[]object

Tolerations of the pgBackRest restore Job. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration

false

PostgresCluster.spec.dataSource.postgresCluster.affinity   Parent

Scheduling constraints of the pgBackRest restore Job. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node

Name

Type

Description

Required

nodeAffinity

object

Describes node affinity scheduling rules for the pod.

false

podAffinity

object

Describes pod affinity scheduling rules (e.g. co-locate this pod in the same node, zone, etc. as some other pod(s)).

false

podAntiAffinity

object

Describes pod anti-affinity scheduling rules (e.g. avoid putting this pod in the same node, zone, etc. as some other pod(s)).

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.nodeAffinity   Parent

Describes node affinity scheduling rules for the pod.

Name

Type

Description

Required

preferredDuringSchedulingIgnoredDuringExecution

[]object

The scheduler will prefer to schedule pods to nodes that satisfy the affinity expressions specified by this field, but it may choose a node that violates one or more of the expressions. The node that is most preferred is the one with the greatest sum of weights, i.e. for each node that meets all of the scheduling requirements (resource request, requiredDuringScheduling affinity expressions, etc.), compute a sum by iterating through the elements of this field and adding "weight" to the sum if the node matches the corresponding matchExpressions; the node(s) with the highest sum are the most preferred.

false

requiredDuringSchedulingIgnoredDuringExecution

object

If the affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to an update), the system may or may not try to eventually evict the pod from its node.

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index]   Parent

An empty preferred scheduling term matches all objects with implicit weight 0 (i.e. it's a no-op). A null preferred scheduling term matches no objects (i.e. is also a no-op).

Name

Type

Description

Required

preference

object

A node selector term, associated with the corresponding weight.

true

weight

integer

Weight associated with matching the corresponding nodeSelectorTerm, in the range 1-100.

true

PostgresCluster.spec.dataSource.postgresCluster.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].preference Parent

A node selector term, associated with the corresponding weight.

Name

Type

Description

Required

matchExpressions

[]object

A list of node selector requirements by node's labels.

false

matchFields

[]object

A list of node selector requirements by node's fields.

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].preference.r
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].preference.r
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution Parent

If the affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to an update), the system may or may not try to eventually evict the pod from its node.

Name

Type

Description

Required

nodeSelectorTerms

[]object

Required. A list of node selector terms. The terms are ORed.

true

PostgresCluster.spec.dataSource.postgresCluster.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms
Parent

A null or empty node selector term matches no objects. The requirements of them are ANDed. The TopologySelectorTerm type implements a subset of the NodeSelectorTerm.

Name

Type

Description

Required

matchExpressions

[]object

A list of node selector requirements by node's labels.

false

matchFields

[]object

A list of node selector requirements by node's fields.

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.podAffinity Parent

Describes pod affinity scheduling rules (e.g. co-locate this pod in the same node, zone, etc. as some other pod(s)).

Name

Type

Description

Required

preferredDuringSchedulingIgnoredDuringExecution

[]object

The scheduler will prefer to schedule pods to nodes that satisfy the affinity expressions specified by this field, but it may choose a node that violates one or more of the expressions. The node that is most preferred is the one with the greatest sum of weights, i.e. for each node that meets all of the scheduling requirements (resource request, requiredDuringScheduling affinity expressions, etc.), compute a sum by iterating through the elements of this field and adding "weight" to the sum if the node has pods which matches the corresponding podAffinityTerm; the node(s) with the highest sum are the most preferred.

false

requiredDuringSchedulingIgnoredDuringExecution

[]object

If the affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to a pod label update), the system may or may not try to eventually evict the pod from its node. When there are multiple elements, the lists of nodes corresponding to each podAffinityTerm are intersected, i.e. all terms must be satisfied.

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index]   Parent

The weights of all of the matched WeightedPodAffinityTerm fields are added per-node to find the most preferred node(s)

Name

Type

Description

Required

podAffinityTerm

object

Required. A pod affinity term, associated with the corresponding weight.

true

weight

integer

weight associated with matching the corresponding podAffinityTerm, in the range 1-100.

true

PostgresCluster.spec.dataSource.postgresCluster.affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityT
Parent

Required. A pod affinity term, associated with the corresponding weight.

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means “this pod’s namespace”

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityT
Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityT
 Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution[index] Parent

Defines a set of pods (namely those matching the labelSelector relative to the given namespace(s)) that this pod should be co-located (affinity) or not co-located (anti-affinity) with, where co-located is defined as running on a node whose value of the label with key matches that of any node on which a pod of the set of pods is running

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means “this pod’s namespace”

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSelector
Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSelector
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.podAntiAffinity Parent

Describes pod anti-affinity scheduling rules (e.g. avoid putting this pod in the same node, zone, etc. as some other pod(s)).

Name

Type

Description

Required

preferredDuringSchedulingIgnoredDuringExecution

[]object

The scheduler will prefer to schedule pods to nodes that satisfy the anti-affinity expressions specified by this field, but it may choose a node that violates one or more of the expressions. The node that is most preferred is the one with the greatest sum of weights, i.e. for each node that meets all of the scheduling requirements (resource request, requiredDuringScheduling anti-affinity expressions, etc.), compute a sum by iterating through the elements of this field and adding “weight” to the sum if the node has pods which matches the corresponding podAffinityTerm; the node(s) with the highest sum are the most preferred.

false

requiredDuringSchedulingIgnoredDuringExecution

[]object

If the anti-affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the anti-affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to a pod label update), the system may or may not try to eventually evict the pod from its node. When there are multiple elements, the lists of nodes corresponding to each podAffinityTerm are intersected, i.e. all terms must be satisfied.

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index] Parent

The weights of all of the matched WeightedPodAffinityTerm fields are added per-node to find the most preferred node(s)

Name

Type

Description

Required

podAffinityTerm

object

Required. A pod affinity term, associated with the corresponding weight.

true

weight

integer

weight associated with matching the corresponding podAffinityTerm, in the range 1-100.

true

PostgresCluster.spec.dataSource.postgresCluster.affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffi
Parent

Required. A pod affinity term, associated with the corresponding weight.

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means “this pod’s namespace”

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffi
Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffi
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution[index]
Parent

Defines a set of pods (namely those matching the labelSelector relative to the given namespace(s)) that this pod should be co-located (affinity) or not co-located (anti-affinity) with, where co-located is defined as running on a node whose value of the label with key matches that of any node on which a pod of the set of pods is running

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means “this pod’s namespace”

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSele
Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.

false

PostgresCluster.spec.dataSource.postgresCluster.affinity.podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSele
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.dataSource.postgresCluster.resources   Parent

Resource requirements for the pgBackRest restore Job.

Name

Type

Description

Required

limits

map[string]int or string

Limits describes the maximum amount of compute resources allowed. More info: https://kubernetes.io/docs/concepts/configuration/manage
compute-resources-container/

false

requests

map[string]int or string

Requests describes the minimum amount of compute resources required. If Requests is omitted for a container, it defaults to Limits if that is explicitly specified, otherwise to an implementation-defined value. More info: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

false

PostgresCluster.spec.dataSource.postgresCluster.tolerations[index]   Parent

The pod this Toleration is attached to tolerates any taint that matches the triple using the matching operator .

Name

Type

Description

Required

effect

string

Effect indicates the taint effect to match. Empty means match all taint effects. When specified, allowed values are NoSchedule, PreferNoSchedule and NoExecute.

false

key

string

Key is the taint key that the toleration applies to. Empty means match all taint keys. If the key is empty, operator must be Exists; this combination means to match all values and all keys.

false

operator

string

Operator represents a key’s relationship to the value. Valid operators are Exists and Equal. Defaults to Equal. Exists is equivalent to wildcard for value, so that a pod can tolerate all taints of a particular category.

false

tolerationSeconds	integer	TolerationSeconds represents the period of time the toleration (which must be of effect NoExecute, otherwise this field is ignored) tolerates the taint. By default, it is not set, which means tolerate the taint forever (do not evict). Zero and negative values will be treated as 0 (evict immediately) by the system.	false
value	string	Value is the taint value the toleration matches to. If the operator is Exists, the value should be empty, otherwise just a regular string.	false

PostgresCluster.spec.dataSource.volumes   Parent

Defines any existing volumes to reuse for this PostgresCluster.

Name	Type	Description	Required
pgBackRestVolume	object	Defines the existing pgBackRest repo volume and directory to use in the current PostgresCluster.	false
pgDataVolume	object	Defines the existing pgData volume and directory to use in the current PostgresCluster.	false
pgWALVolume	object	Defines the existing pg_wal volume and directory to use in the current PostgresCluster. Note that a defined pg_wal volume MUST be accompanied by a pgData volume.	false

PostgresCluster.spec.dataSource.volumes.pgBackRestVolume   Parent

Defines the existing pgBackRest repo volume and directory to use in the current PostgresCluster.

Name	Type	Description	Required
pvcName	string	The existing PVC name.	true
directory	string	The existing directory. When not set, a move Job is not created for the associated volume.	false

PostgresCluster.spec.dataSource.volumes.pgDataVolume   Parent

Defines the existing pgData volume and directory to use in the current PostgresCluster.

Name	Type	Description	Required
pvcName	string	The existing PVC name.	true
directory	string	The existing directory. When not set, a move Job is not created for the associated volume.	false

PostgresCluster.spec.dataSource.volumes.pgWALVolume Parent

Defines the existing pg_wal volume and directory to use in the current PostgresCluster. Note that a defined pg_wal volume MUST be accompanied by a pgData volume.

Name	Type	Description	Required
pvcName	string	The existing PVC name.	true
directory	string	The existing directory. When not set, a move Job is not created for the associated volume.	false

PostgresCluster.spec.databaseInitSQL Parent

DatabaseInitSQL defines a ConfigMap containing custom SQL that will be run after the cluster is initialized. This ConfigMap must be in the same namespace as the cluster.

Name	Type	Description	Required
key	string	Key is the ConfigMap data key that points to a SQL string	true
name	string	Name is the name of a ConfigMap	true

PostgresCluster.spec.imagePullSecrets[index] Parent

LocalObjectReference contains enough information to let you locate the referenced object inside the same namespace.

Name

Type

Description

Required

name

string

Name of the referent. More info: https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#names TODO: Add other useful fields. apiVersion, kind, uid?

false

PostgresCluster.spec.metadata Parent

Metadata contains metadata for PostgresCluster resources

Name

Type

Description

Required

annotations

map[string]string

false

labels

map[string]string

false

PostgresCluster.spec.monitoring Parent

The specification of monitoring tools that connect to PostgreSQL

Name

Type

Description

Required

pgmonitor

object

PGMonitorSpec defines the desired state of the pgMonitor tool suite

false

PostgresCluster.spec.monitoring.pgmonitor Parent

PGMonitorSpec defines the desired state of the pgMonitor tool suite

Name

Type

Description

Required

exporter

object

false

PostgresCluster.spec.monitoring.pgmonitor.exporter Parent

Name

Type

Description

Required

configuration

[]object

Projected volumes containing custom PostgreSQL Exporter configuration. Currently supports the customization of PostgreSQL Exporter queries. If a “queries.yaml” file is detected in any volume projected using this field, it will be loaded using the “extend.query-path” flag: https://github.com/prometheus-community/postgres_exporter#flags Changing the values of field causes PostgreSQL and the exporter to restart.

false

image

string

The image name to use for crunchy-postgres-exporter containers. The image may also be set using the RELATED_IMAGE_PGEXPORTER environment variable.

false

resources

object

Changing this value causes PostgreSQL and the exporter to restart. More info: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers

false

PostgresCluster.spec.monitoring.pgmonitor.exporter.configuration[index]   Parent

Projection that may be projected along with other supported volume types

Name

Type

Description

Required

configMap

object

information about the configMap data to project

false

downwardAPI

object

information about the downwardAPI data to project

false

secret

object

information about the secret data to project

false

serviceAccountToken

object

information about the serviceAccountToken data to project

false

PostgresCluster.spec.monitoring.pgmonitor.exporter.configuration[index].configMap   Parent

information about the configMap data to project

Name

Type

Description

Required

items

[]object

If unspecified, each key-value pair in the Data field of the referenced ConfigMap will be projected into the volume as a file whose name is the key and content is the value. If specified, the listed keys will be projected into the specified paths, and unlisted keys will not be present. If a key is specified which is not present in the ConfigMap, the volume setup will error unless it is marked optional. Paths must be relative and may not contain the '..' path or start with '..'.

false

name

string

Name of the referent. More info: https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#names TODO: Add other useful fields. apiVersion, kind, uid?

false

optional

boolean

Specify whether the ConfigMap or its keys must be defined

false

PostgresCluster.spec.monitoring.pgmonitor.exporter.configuration[index].configMap.items[index]   Parent

Maps a string key to a path within a volume.

Name

Type

Description

Required

key

string

The key to project.

true

path

string

The relative path of the file to map the key to. May not be an absolute path. May not contain the path element '..'. May not start with the string '..'.

true

mode

integer

Optional: mode bits used to set permissions on this file. Must be an octal value between 0000 and 0777 or a decimal value between 0 and 511. YAML accepts both octal and decimal values, JSON requires decimal values for mode bits. If not specified, the volume defaultMode will be used. This might be in conflict with other options that affect the file mode, like fsGroup, and the result can be other mode bits set.

false

PostgresCluster.spec.monitoring.pgmonitor.exporter.configuration[index].downwardAPI   Parent

information about the downwardAPI data to project

Name

Type

Description

Required

items

[]object

Items is a list of DownwardAPIVolume file

false

PostgresCluster.spec.monitoring.pgmonitor.exporter.configuration[index].downwardAPI.items[index] Parent

DownwardAPIVolumeFile represents information to create the file containing the pod field

Name

Type

Description

Required

path

string

Required: Path is the relative path name of the file to be created. Must not be absolute or contain the '..' path. Must be utf-8 encoded. The first item of the relative path must not start with '..'

true

fieldRef

object

Required: Selects a field of the pod: only annotations, labels, name and namespace are supported.

false

mode

integer

Optional: mode bits used to set permissions on this file, must be an octal value between 0000 and 0777 or a decimal value between 0 and 511. YAML accepts both octal and decimal values, JSON requires decimal values for mode bits. If not specified, the volume defaultMode will be used. This might be in conflict with other options that affect the file mode, like fsGroup, and the result can be other mode bits set.

false

resourceFieldRef

object

Selects a resource of the container: only resources limits and requests (limits.cpu, limits.memory, requests.cpu and requests.memory) are currently supported.

false

PostgresCluster.spec.monitoring.pgmonitor.exporter.configuration[index].downwardAPI.items[index].fieldRef Parent

Required: Selects a field of the pod: only annotations, labels, name and namespace are supported.

Name

Type

Description

Required

fieldPath

string

Path of the field to select in the specified API version.

true

apiVersion

string

Version of the schema the FieldPath is written in terms of, defaults to "v1".

false

PostgresCluster.spec.monitoring.pgmonitor.exporter.configuration[index].downwardAPI.items[index].resourceFieldRef Parent

Selects a resource of the container: only resources limits and requests (limits.cpu, limits.memory, requests.cpu and requests.memory) are currently supported.

Name	Type	Description	Required
resource	string	Required: resource to select	true
containerName	string	Container name: required for volumes, optional for env vars	false
divisor	int or string	Specifies the output format of the exposed resources, defaults to “1”	false

PostgresCluster.spec.monitoring.pgmonitor.exporter.configuration[index].secret   Parent

information about the secret data to project

Name	Type	Description	Required
items	[]object	If unspecified, each key-value pair in the Data field of the referenced Secret will be projected into the volume as a file whose name is the key and content is the value. If specified, the listed keys will be projected into the specified paths, and unlisted keys will not be present. If a key is specified which is not present in the Secret, the volume setup will error unless it is marked optional. Paths must be relative and may not contain the ‘..’ path or start with ‘..’.	false
name	string	Name of the referent. More info: https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#names TODO: Add other useful fields. apiVersion, kind, uid?	false
optional	boolean	Specify whether the Secret or its key must be defined	false

PostgresCluster.spec.monitoring.pgmonitor.exporter.configuration[index].secret.items[index]   Parent

Maps a string key to a path within a volume.

Name	Type	Description	Required
key			

string

The key to project.

true

path

string

The relative path of the file to map the key to. May not be an absolute path. May not contain the path element ‘..’. May not start with the string ‘..’.

true

mode

integer

Optional: mode bits used to set permissions on this file. Must be an octal value between 0000 and 0777 or a decimal value between 0 and 511. YAML accepts both octal and decimal values, JSON requires decimal values for mode bits. If not specified, the volume defaultMode will be used. This might be in conflict with other options that affect the file mode, like fsGroup, and the result can be other mode bits set.

false

PostgresCluster.spec.monitoring.pgmonitor.exporter.configuration[index].serviceAccountToken Parent

information about the serviceAccountToken data to project

Name

Type

Description

Required

path

string

Path is the path relative to the mount point of the file to project the token into.

true

audience

string

Audience is the intended audience of the token. A recipient of a token must identify itself with an identifier specified in the audience of the token, and otherwise should reject the token. The audience defaults to the identifier of the apiserver.

false

expirationSeconds

integer

ExpirationSeconds is the requested duration of validity of the service account token. As the token approaches expiration, the kubelet volume plugin will proactively rotate the service account token. The kubelet will start trying to rotate the token if the token is older than 80 percent of its time to live or if the token is older than 24 hours.Defaults to 1 hour and must be at least 10 minutes.

false

PostgresCluster.spec.monitoring.pgmonitor.exporter.resources Parent

Changing this value causes PostgreSQL and the exporter to restart. More info: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers

Name

Type

Description

Required

limits

map[string]int or string

Limits describes the maximum amount of compute resources allowed. More info: https://kubernetes.io/docs/concepts/configuration/manage
compute-resources-container/

false

requests

map[string]int or string

Requests describes the minimum amount of compute resources required. If Requests is omitted for a container, it defaults to Limits if that is explicitly specified, otherwise to an implementation-defined value. More info: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

false

PostgresCluster.spec.patroni Parent

Name

Type

Description

Required

dynamicConfiguration

object

false

leaderLeaseDurationSeconds

integer

TTL of the cluster leader lock. "Think of it as the length of time before initiation of the automatic failover process."

false

port

integer

The port on which Patroni should listen.

false

syncPeriodSeconds

integer

The interval for refreshing the leader lock and applying dynamicConfiguration. Must be less than leaderLeaseDurationSeconds.

false

PostgresCluster.spec.proxy Parent

The specification of a proxy that connects to PostgreSQL.

Name

Type

Description

Required

pgBouncer

object

Defines a PgBouncer proxy and connection pooler.

true

PostgresCluster.spec.proxy.pgBouncer Parent

Defines a PgBouncer proxy and connection pooler.

Name

Type

Description

Required

affinity

object

Scheduling constraints of a PgBouncer pod. Changing this value causes PgBouncer to restart. More info: https://kubernetes.io/docs/concept
eviction/assign-pod-node

false

config

object

Configuration settings for the PgBouncer process. Changes to any of these values will be automatically reloaded without validation. Be careful, as you may put PgBouncer into an unusable state. More info: https://www.pgbouncer.org/usage.html#reload

false

customTLSSecret

object

A secret projection containing a certificate and key with which to encrypt connections to PgBouncer. The “tls.crt”, “tls.key”, and “ca.crt” paths must be PEM-encoded certificates and keys. Changing this value causes PgBouncer to restart. More info: https://kubernetes.io/docs/concepts/configuration/secret/#projection-of-secret-keys-to-specific-paths

false

image

string

Name of a container image that can run PgBouncer 1.15 or newer. Changing this value causes PgBouncer to restart. The image may also be set using the RELATED_IMAGE_PGBOUNCER environment variable. More info: https://kubernetes.io/docs/concepts/containers/image

false

metadata

object

Metadata contains metadata for PostgresCluster resources

false

port

integer

Port on which PgBouncer should listen for client connections. Changing this value causes PgBouncer to restart.

false

priorityClassName

string

Priority class name for the pgBouncer pod. Changing this value causes PostgreSQL to restart. More info: https://kubernetes.io/docs/concept
eviction/pod-priority-preemption/

false

replicas

integer

Number of desired PgBouncer pods.

false

resources

object

Compute resources of a PgBouncer container. Changing this value causes PgBouncer to restart. More info: https://kubernetes.io/docs/conce
resources-containers

false

service

object

Specification of the service that exposes PgBouncer.

false

sidecars

object

Configuration for pgBouncer sidecar containers

false

tolerations

[]object

Tolerations of a PgBouncer pod. Changing this value causes PgBouncer to restart. More info: https://kubernetes.io/docs/concepts/schedulin
eviction/taint-and-toleration

false

topologySpreadConstraints

[]object

Topology spread constraints of a PgBouncer pod. Changing this value causes PgBouncer to restart. More info: https://kubernetes.io/docs/co
topology-spread-constraints/

false

PostgresCluster.spec.proxy.pgBouncer.affinity Parent

Scheduling constraints of a PgBouncer pod. Changing this value causes PgBouncer to restart. More info: https://kubernetes.io/docs/concept
eviction/assign-pod-node

Name

Type

Description

Required

nodeAffinity

object

Describes node affinity scheduling rules for the pod.

false

podAffinity

object

Describes pod affinity scheduling rules (e.g. co-locate this pod in the same node, zone, etc. as some other pod(s)).

false

podAntiAffinity

object

Describes pod anti-affinity scheduling rules (e.g. avoid putting this pod in the same node, zone, etc. as some other pod(s)).

false

PostgresCluster.spec.proxy.pgBouncer.affinity.nodeAffinity Parent

Describes node affinity scheduling rules for the pod.

Name

Type

Description

Required

preferredDuringSchedulingIgnoredDuringExecution

[]object

The scheduler will prefer to schedule pods to nodes that satisfy the affinity expressions specified by this field, but it may choose a node that violates one or more of the expressions. The node that is most preferred is the one with the greatest sum of weights, i.e. for each node that meets all of the scheduling requirements (resource request, requiredDuringScheduling affinity expressions, etc.), compute a sum by iterating through the elements of this field and adding “weight” to the sum if the node matches the corresponding matchExpressions; the node(s) with the highest sum are the most preferred.

false

requiredDuringSchedulingIgnoredDuringExecution

object

If the affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to an update), the system may or may not try to eventually evict the pod from its node.

false

PostgresCluster.spec.proxy.pgBouncer.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index]   Parent

An empty preferred scheduling term matches all objects with implicit weight 0 (i.e. it's a no-op). A null preferred scheduling term matches no objects (i.e. is also a no-op).

Name

Type

Description

Required

preference

object

A node selector term, associated with the corresponding weight.

true

weight

integer

Weight associated with matching the corresponding nodeSelectorTerm, in the range 1-100.

true

PostgresCluster.spec.proxy.pgBouncer.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].preference   Parent

A node selector term, associated with the corresponding weight.

Name

Type

Description

Required

matchExpressions

[]object

A list of node selector requirements by node's labels.

false

matchFields

[]object

A list of node selector requirements by node's fields.

false

PostgresCluster.spec.proxy.pgBouncer.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].preference.matchExpr
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.proxy.pgBouncer.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].preference.matchFiel
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.proxy.pgBouncer.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution Parent

If the affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to an update), the system may or may not try to eventually evict the pod from its node.

Name

Type

Description

Required

nodeSelectorTerms

[]object

Required. A list of node selector terms. The terms are ORed.

true

PostgresCluster.spec.proxy.pgBouncer.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms[index]
Parent

A null or empty node selector term matches no objects. The requirements of them are ANDed. The TopologySelectorTerm type implements a subset of the NodeSelectorTerm.

Name

Type

Description

Required

matchExpressions

[]object

A list of node selector requirements by node's labels.

false

matchFields

[]object

A list of node selector requirements by node's fields.

false

PostgresCluster.spec.proxy.pgBouncer.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms[index].ma
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.proxy.pgBouncer.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms[index].ma
Parent

A node selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

The label key that the selector applies to.

true

operator

string

Represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists, DoesNotExist. Gt, and Lt.

true

values

[]string

An array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. If the operator is Gt or Lt, the values array must have a single element, which will be interpreted as an integer. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.proxy.pgBouncer.affinity.podAffinity   Parent

Describes pod affinity scheduling rules (e.g. co-locate this pod in the same node, zone, etc. as some other pod(s)).

Name

Type

Description

Required

preferredDuringSchedulingIgnoredDuringExecution

[]object

The scheduler will prefer to schedule pods to nodes that satisfy the affinity expressions specified by this field, but it may choose a node that violates one or more of the expressions. The node that is most preferred is the one with the greatest sum of weights, i.e. for each node that meets all of the scheduling requirements (resource request, requiredDuringScheduling affinity expressions, etc.), compute a sum by iterating through the elements of this field and adding “weight” to the sum if the node has pods which matches the corresponding podAffinityTerm; the node(s) with the highest sum are the most preferred.

false

requiredDuringSchedulingIgnoredDuringExecution

[]object

If the affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to a pod label update), the system may or may not try to eventually evict the pod from its node. When there are multiple elements, the lists of nodes corresponding to each podAffinityTerm are intersected, i.e. all terms must be satisfied.

false

PostgresCluster.spec.proxy.pgBouncer.affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index]   Parent

The weights of all of the matched WeightedPodAffinityTerm fields are added per-node to find the most preferred node(s)

Name

Type

Description

Required

podAffinityTerm

object

Required. A pod affinity term, associated with the corresponding weight.

true

weight

integer

weight associated with matching the corresponding podAffinityTerm, in the range 1-100.

true

PostgresCluster.spec.proxy.pgBouncer.affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityTerm
Parent

Required. A pod affinity term, associated with the corresponding weight.

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means "this pod's namespace"

false

PostgresCluster.spec.proxy.pgBouncer.affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityTerm.labelS
Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is "key", the operator is "In", and the values array contains only "value". The requirements are ANDed.

false

PostgresCluster.spec.proxy.pgBouncer.affinity.podAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityTerm.labelS
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.proxy.pgBouncer.affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution[index] Parent

Defines a set of pods (namely those matching the labelSelector relative to the given namespace(s)) that this pod should be co-located (affinity) or not co-located (anti-affinity) with, where co-located is defined as running on a node whose value of the label with key matches that of any node on which a pod of the set of pods is running

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means "this pod's namespace"

false

PostgresCluster.spec.proxy.pgBouncer.affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSelector Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.

false

PostgresCluster.spec.proxy.pgBouncer.affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSelector.matchExp
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name	Type	Description	Required
key	string	key is the label key that the selector applies to.	true
operator	string	operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.	true
values	[]string	values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.	false

PostgresCluster.spec.proxy.pgBouncer.affinity.podAntiAffinity Parent

Describes pod anti-affinity scheduling rules (e.g. avoid putting this pod in the same node, zone, etc. as some other pod(s)).

Name	Type	Description	Required
preferredDuringSchedulingIgnoredDuringExecution	[]object	The scheduler will prefer to schedule pods to nodes that satisfy the anti-affinity expressions specified by this field, but it may choose a node that violates one or more of the expressions. The node that is most preferred is the one with the greatest sum of weights, i.e. for each node that meets all of the scheduling requirements (resource request, requiredDuringScheduling anti-affinity expressions, etc.), compute a sum by iterating through the elements of this field and adding “weight” to the sum if the node has pods which matches the corresponding podAffinityTerm; the node(s) with the highest sum are the most preferred.	false
requiredDuringSchedulingIgnoredDuringExecution	[]object	If the anti-affinity requirements specified by this field are not met at scheduling time, the pod will not be scheduled onto the node. If the anti-affinity requirements specified by this field cease to be met at some point during pod execution (e.g. due to a pod label update), the system may or may not try to eventually evict the pod from its node. When there are multiple elements, the lists of nodes corresponding to each podAffinityTerm are intersected, i.e. all terms must be satisfied.	false

PostgresCluster.spec.proxy.pgBouncer.affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index] Parent

The weights of all of the matched WeightedPodAffinityTerm fields are added per-node to find the most preferred node(s)

Name

Type

Description

Required

podAffinityTerm

object

Required. A pod affinity term, associated with the corresponding weight.

true

weight

integer

weight associated with matching the corresponding podAffinityTerm, in the range 1-100.

true

PostgresCluster.spec.proxy.pgBouncer.affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityTerm
Parent

Required. A pod affinity term, associated with the corresponding weight.

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means "this pod's namespace"

false

PostgresCluster.spec.proxy.pgBouncer.affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityTerm.l
Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.

false

PostgresCluster.spec.proxy.pgBouncer.affinity.podAntiAffinity.preferredDuringSchedulingIgnoredDuringExecution[index].podAffinityTerm.l
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.proxy.pgBouncer.affinity.podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution[index] Parent

Defines a set of pods (namely those matching the labelSelector relative to the given namespace(s)) that this pod should be co-located (affinity) or not co-located (anti-affinity) with, where co-located is defined as running on a node whose value of the label with key matches that of any node on which a pod of the set of pods is running

Name

Type

Description

Required

topologyKey

string

This pod should be co-located (affinity) or not co-located (anti-affinity) with the pods matching the labelSelector in the specified namespaces, where co-located is defined as running on a node whose value of the label with key topologyKey matches that of any node on which any of the selected pods is running. Empty topologyKey is not allowed.

true

labelSelector

object

A label query over a set of resources, in this case pods.

false

namespaces

[]string

namespaces specifies which namespaces the labelSelector applies to (matches against); null or empty list means “this pod’s namespace”

false

PostgresCluster.spec.proxy.pgBouncer.affinity.podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSelector
Parent

A label query over a set of resources, in this case pods.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels

map[string]string

matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is "key", the operator is "In", and the values array contains only "value". The requirements are ANDed.

false

PostgresCluster.spec.proxy.pgBouncer.affinity.podAntiAffinity.requiredDuringSchedulingIgnoredDuringExecution[index].labelSelector.match
Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name

Type

Description

Required

key

string

key is the label key that the selector applies to.

true

operator

string

operator represents a key's relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.

true

values

[]string

values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.

false

PostgresCluster.spec.proxy.pgBouncer.config Parent

Configuration settings for the PgBouncer process. Changes to any of these values will be automatically reloaded without validation. Be careful, as you may put PgBouncer into an unusable state. More info: https://www.pgbouncer.org/usage.html#reload

Name

Type

Description

Required

databases

map[string]string

PgBouncer database definitions. The key is the database requested by a client while the value is a libpq-styled connection string. The special key "*" acts as a fallback. When this field is empty, PgBouncer is configured with a single "*" entry that connects to the primary PostgreSQL instance. More info: https://www.pgbouncer.org/config.html#section-databases

false

files

[]object

Files to mount under “/etc/pgbouncer”. When specified, settings in the “pgbouncer.ini” file are loaded before all others. From there, other files may be included by absolute path. Changing these references causes PgBouncer to restart, but changes to the file contents are automatically reloaded. More info: https://www.pgbouncer.org/config.html#include-directive

false

global

map[string]string

Settings that apply to the entire PgBouncer process. More info: https://www.pgbouncer.org/config.html

false

users

map[string]string

Connection settings specific to particular users. More info: https://www.pgbouncer.org/config.html#section-users

false

PostgresCluster.spec.proxy.pgBouncer.config.files[index]   Parent

Projection that may be projected along with other supported volume types

Name

Type

Description

Required

configMap

object

information about the configMap data to project

false

downwardAPI

object

information about the downwardAPI data to project

false

secret

object

information about the secret data to project

false

serviceAccountToken

object

information about the serviceAccountToken data to project

false

PostgresCluster.spec.proxy.pgBouncer.config.files[index].configMap   Parent

information about the configMap data to project

Name

Type

Description

Required

items

[]object

If unspecified, each key-value pair in the Data field of the referenced ConfigMap will be projected into the volume as a file whose name is the key and content is the value. If specified, the listed keys will be projected into the specified paths, and unlisted keys will not be present. If a key is specified which is not present in the ConfigMap, the volume setup will error unless it is marked optional. Paths must be relative and may not contain the '..' path or start with '..'.

false

name

string

Name of the referent. More info: https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#names TODO: Add other useful fields. apiVersion, kind, uid?

false

optional

boolean

Specify whether the ConfigMap or its keys must be defined

false

PostgresCluster.spec.proxy.pgBouncer.config.files[index].configMap.items[index]   Parent

Maps a string key to a path within a volume.

Name

Type

Description

Required

key

string

The key to project.

true

path

string

The relative path of the file to map the key to. May not be an absolute path. May not contain the path element '..'. May not start with the string '..'.

true

mode

integer

Optional: mode bits used to set permissions on this file. Must be an octal value between 0000 and 0777 or a decimal value between 0 and 511. YAML accepts both octal and decimal values, JSON requires decimal values for mode bits. If not specified, the volume defaultMode will be used. This might be in conflict with other options that affect the file mode, like fsGroup, and the result can be other mode bits set.

false

PostgresCluster.spec.proxy.pgBouncer.config.files[index].downwardAPI   Parent

information about the downwardAPI data to project

Name

Type

Description

Required

items

[]object

Items is a list of DownwardAPIVolume file

false

PostgresCluster.spec.proxy.pgBouncer.config.files[index].downwardAPI.items[index]   Parent

DownwardAPIVolumeFile represents information to create the file containing the pod field

Name	Type	Description	Required
path	string	Required: Path is the relative path name of the file to be created. Must not be absolute or contain the '..' path. Must be utf-8 encoded. The first item of the relative path must not start with '..'	true
fieldRef	object	Required: Selects a field of the pod: only annotations, labels, name and namespace are supported.	false
mode	integer	Optional: mode bits used to set permissions on this file, must be an octal value between 0000 and 0777 or a decimal value between 0 and 511. YAML accepts both octal and decimal values, JSON requires decimal values for mode bits. If not specified, the volume defaultMode will be used. This might be in conflict with other options that affect the file mode, like fsGroup, and the result can be other mode bits set.	false
resourceFieldRef	object	Selects a resource of the container: only resources limits and requests (limits.cpu, limits.memory, requests.cpu and requests.memory) are currently supported.	false

PostgresCluster.spec.proxy.pgBouncer.config.files[index].downwardAPI.items[index].fieldRef   Parent

Required: Selects a field of the pod: only annotations, labels, name and namespace are supported.

Name	Type	Description	Required
fieldPath	string	Path of the field to select in the specified API version.	true
apiVersion	string	Version of the schema the FieldPath is written in terms of, defaults to "v1".	false

PostgresCluster.spec.proxy.pgBouncer.config.files[index].downwardAPI.items[index].resourceFieldRef   Parent

Selects a resource of the container: only resources limits and requests (limits.cpu, limits.memory, requests.cpu and requests.memory) are currently supported.

Name

Type

Description

Required

resource

string

Required: resource to select

true

containerName

string

Container name: required for volumes, optional for env vars

false

divisor

int or string

Specifies the output format of the exposed resources, defaults to “1”

false

PostgresCluster.spec.proxy.pgBouncer.config.files[index].secret   Parent

information about the secret data to project

Name

Type

Description

Required

items

[]object

If unspecified, each key-value pair in the Data field of the referenced Secret will be projected into the volume as a file whose name is the key and content is the value. If specified, the listed keys will be projected into the specified paths, and unlisted keys will not be present. If a key is specified which is not present in the Secret, the volume setup will error unless it is marked optional. Paths must be relative and may not contain the ‘..’ path or start with ‘..’.

false

name

string

Name of the referent. More info: https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#names TODO: Add other useful fields. apiVersion, kind, uid?

false

optional

boolean

Specify whether the Secret or its key must be defined

false

PostgresCluster.spec.proxy.pgBouncer.config.files[index].secret.items[index]   Parent

Maps a string key to a path within a volume.

Name

Type

Description

Required

key

string

The key to project.

true

path

string

The relative path of the file to map the key to. May not be an absolute path. May not contain the path element '..'. May not start with the string '..'.

true

mode

integer

Optional: mode bits used to set permissions on this file. Must be an octal value between 0000 and 0777 or a decimal value between 0 and 511. YAML accepts both octal and decimal values, JSON requires decimal values for mode bits. If not specified, the volume defaultMode will be used. This might be in conflict with other options that affect the file mode, like fsGroup, and the result can be other mode bits set.

false

PostgresCluster.spec.proxy.pgBouncer.config.files[index].serviceAccountToken   Parent

information about the serviceAccountToken data to project

Name

Type

Description

Required

path

string

Path is the path relative to the mount point of the file to project the token into.

true

audience

string

Audience is the intended audience of the token. A recipient of a token must identify itself with an identifier specified in the audience of the token, and otherwise should reject the token. The audience defaults to the identifier of the apiserver.

false

expirationSeconds

integer

ExpirationSeconds is the requested duration of validity of the service account token. As the token approaches expiration, the kubelet volume plugin will proactively rotate the service account token. The kubelet will start trying to rotate the token if the token is older than 80 percent of its time to live or if the token is older than 24 hours.Defaults to 1 hour and must be at least 10 minutes.

false

PostgresCluster.spec.proxy.pgBouncer.customTLSSecret   Parent

A secret projection containing a certificate and key with which to encrypt connections to PgBouncer. The "tls.crt", "tls.key", and "ca.crt" paths must be PEM-encoded certificates and keys. Changing this value causes PgBouncer to restart. More info: https://kubernetes.io/docs/concepts/configuration/secret/#projection-of-secret-keys-to-specific-paths

Name

Type

Description

Required

items

[]object

If unspecified, each key-value pair in the Data field of the referenced Secret will be projected into the volume as a file whose name is the key and content is the value. If specified, the listed keys will be projected into the specified paths, and unlisted keys will not be present. If a key is specified which is not present in the Secret, the volume setup will error unless it is marked optional. Paths must be relative and may not contain the '..' path or start with '..'.

false

name

string

Name of the referent. More info: https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#names TODO: Add other useful fields. apiVersion, kind, uid?

false

optional

boolean

Specify whether the Secret or its key must be defined

false

PostgresCluster.spec.proxy.pgBouncer.customTLSSecret.items[index]   Parent

Maps a string key to a path within a volume.

Name

Type

Description

Required

key

string

The key to project.

true

path

string

The relative path of the file to map the key to. May not be an absolute path. May not contain the path element '..'. May not start with the string '..'.

true

mode

integer

Optional: mode bits used to set permissions on this file. Must be an octal value between 0000 and 0777 or a decimal value between 0 and 511. YAML accepts both octal and decimal values, JSON requires decimal values for mode bits. If not specified, the volume defaultMode will be used. This might be in conflict with other options that affect the file mode, like fsGroup, and the result can be other mode bits set.

false

PostgresCluster.spec.proxy.pgBouncer.metadata   Parent

Metadata contains metadata for PostgresCluster resources

Name

Type

Description

Required

annotations

map[string]string

false

labels

map[string]string

false

PostgresCluster.spec.proxy.pgBouncer.resources   Parent

Compute resources of a PgBouncer container. Changing this value causes PgBouncer to restart. More info: https://kubernetes.io/docs/conce
resources-containers

Name	Type	Description	Required
limits	map[string]int or string	Limits describes the maximum amount of compute resources allowed. More info: https://kubernetes.io/docs/concepts/configuration/manage compute-resources-container/	false
requests	map[string]int or string	Requests describes the minimum amount of compute resources required. If Requests is omitted for a container, it defaults to Limits if that is explicitly specified, otherwise to an implementation-defined value. More info: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/	false

PostgresCluster.spec.proxy.pgBouncer.service Parent

Specification of the service that exposes PgBouncer.

Name	Type	Description	Required
type	enum	More info: https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services-service-types	true

PostgresCluster.spec.proxy.pgBouncer.sidecars Parent

Configuration for pgBouncer sidecar containers

Name	Type	Description	Required
pgbouncerConfig	object	Defines the configuration for the pgBouncer config sidecar container	false

PostgresCluster.spec.proxy.pgBouncer.sidecars.pgbouncerConfig Parent

Defines the configuration for the pgBouncer config sidecar container

Name	Type	Description	Required
resources	object	Resource requirements for a sidecar container	false

PostgresCluster.spec.proxy.pgBouncer.sidecars.pgbouncerConfig.resources   Parent

Resource requirements for a sidecar container

Name	Type	Description	Required
limits	map[string]int or string	Limits describes the maximum amount of compute resources allowed. More info: https://kubernetes.io/docs/concepts/configuration/manage compute-resources-container/	false
requests	map[string]int or string	Requests describes the minimum amount of compute resources required. If Requests is omitted for a container, it defaults to Limits if that is explicitly specified, otherwise to an implementation-defined value. More info: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/	false

PostgresCluster.spec.proxy.pgBouncer.tolerations[index]   Parent

The pod this Toleration is attached to tolerates any taint that matches the triple using the matching operator .

Name	Type	Description	Required
effect	string	Effect indicates the taint effect to match. Empty means match all taint effects. When specified, allowed values are NoSchedule, PreferNoSchedule and NoExecute.	false
key	string	Key is the taint key that the toleration applies to. Empty means match all taint keys. If the key is empty, operator must be Exists; this combination means to match all values and all keys.	false
operator	string	Operator represents a key's relationship to the value. Valid operators are Exists and Equal. Defaults to Equal. Exists is equivalent to wildcard for value, so that a pod can tolerate all taints of a particular category.	false
tolerationSeconds	integer	TolerationSeconds represents the period of time the toleration (which must be of effect NoExecute, otherwise this field is ignored) tolerates the taint. By default, it is not set, which means tolerate the taint forever (do not evict). Zero and negative values will be treated as 0 (evict immediately) by the system.	false
value	string		

Value is the taint value the toleration matches to. If the operator is Exists, the value should be empty, otherwise just a regular string.

false

PostgresCluster.spec.proxy.pgBouncer.topologySpreadConstraints[index]   Parent

TopologySpreadConstraint specifies how to spread matching pods among the given topology.

Name

Type

Description

Required

maxSkew

integer

MaxSkew describes the degree to which pods may be unevenly distributed. When `whenUnsatisfiable=DoNotSchedule`, it is the maximum permitted difference between the number of matching pods in the target topology and the global minimum. For example, in a 3-zone cluster, MaxSkew is set to 1, and pods with the same labelSelector spread as 1/1/0: | zone1 | zone2 | zone3 | | P | P | | - if MaxSkew is 1, incoming pod can only be scheduled to zone3 to become 1/1/1; scheduling it onto zone1(zone2) would make the ActualSkew(2-0) on zone1(zone2) violate MaxSkew(1). - if MaxSkew is 2, incoming pod can be scheduled onto any zone. When `whenUnsatisfiable=ScheduleAnyway`, it is used to give higher precedence to topologies that satisfy it. It's a required field. Default value is 1 and 0 is not allowed.

true

topologyKey

string

TopologyKey is the key of node labels. Nodes that have a label with this key and identical values are considered to be in the same topology. We consider each as a "bucket", and try to put balanced number of pods into each bucket. It's a required field.

true

whenUnsatisfiable

string

WhenUnsatisfiable indicates how to deal with a pod if it doesn't satisfy the spread constraint. - DoNotSchedule (default) tells the scheduler not to schedule it. - ScheduleAnyway tells the scheduler to schedule the pod in any location, but giving higher precedence to topologies that would help reduce the skew. A constraint is considered "Unsatisfiable" for an incoming pod if and only if every possible node assigment for that pod would violate "MaxSkew" on some topology. For example, in a 3-zone cluster, MaxSkew is set to 1, and pods with the same labelSelector spread as 3/1/1: | zone1 | zone2 | zone3 | | P P P | P | P | If WhenUnsatisfiable is set to DoNotSchedule, incoming pod can only be scheduled to zone2(zone3) to become 3/2/1(3/1/2) as ActualSkew(2-1) on zone2(zone3) satisfies MaxSkew(1). In other words, the cluster can still be imbalanced, but scheduler won't make it *more* imbalanced. It's a required field.

true

labelSelector

object

LabelSelector is used to find matching pods. Pods that match this label selector are counted to determine the number of pods in their corresponding topology domain.

false

PostgresCluster.spec.proxy.pgBouncer.topologySpreadConstraints[index].labelSelector   Parent

LabelSelector is used to find matching pods. Pods that match this label selector are counted to determine the number of pods in their corresponding topology domain.

Name

Type

Description

Required

matchExpressions

[]object

matchExpressions is a list of label selector requirements. The requirements are ANDed.

false

matchLabels	map[string]string	matchLabels is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “key”, the operator is “In”, and the values array contains only “value”. The requirements are ANDed.	false

PostgresCluster.spec.proxy.pgBouncer.topologySpreadConstraints[index].labelSelector.matchExpressions[index]   Parent

A label selector requirement is a selector that contains values, a key, and an operator that relates the key and values.

Name	Type	Description	Required
key	string	key is the label key that the selector applies to.	true
operator	string	operator represents a key’s relationship to a set of values. Valid operators are In, NotIn, Exists and DoesNotExist.	true
values	[]string	values is an array of string values. If the operator is In or NotIn, the values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty. This array is replaced during a strategic merge patch.	false

PostgresCluster.spec.service   Parent

Specification of the service that exposes the PostgreSQL primary instance.

Name	Type	Description	Required
type	enum	More info: https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services-service-types	true

PostgresCluster.spec.standby   Parent

Run this cluster as a read-only copy of an existing cluster or archive.

Name	Type	Description	Required
repoName	string	The name of the pgBackRest repository to follow for WAL files.	true

enabled

boolean

Whether or not the PostgreSQL cluster should be read-only. When this is true, WAL files are applied from the pgBackRest repository.

false

PostgresCluster.spec.users[index]   Parent

Name

Type

Description

Required

name

string

The name of this PostgreSQL user. The value may contain only lowercase letters, numbers, and hyphen so that it fits into Kubernetes metadata.

true

databases

[]string

Databases to which this user can connect and create objects. Removing a database from this list does NOT revoke access. This field is ignored for the "postgres" user.

false

options

string

ALTER ROLE options except for PASSWORD. This field is ignored for the "postgres" user. More info: https://www.postgresql.org/docs/cur
attributes.html

false

PostgresCluster.status   Parent

PostgresClusterStatus defines the observed state of PostgresCluster

Name

Type

Description

Required

conditions

[]object

conditions represent the observations of postgrescluster's current state. Known .status.conditions.type are: "PersistentVolumeResizing", "ProxyAvailable"

false

databaseInitSQL

string

DatabaseInitSQL state of custom database initialization in the cluster

false

databaseRevision

string

Identifies the databases that have been installed into PostgreSQL.

false

instances

[]object

Current state of PostgreSQL instances.

false

monitoring

object

Current state of PostgreSQL cluster monitoring tool configuration

false

observedGeneration

integer

observedGeneration represents the .metadata.generation on which the status was based.

false

patroni

object

false

pgbackrest

object

Status information for pgBackRest

false

proxy

object

Current state of the PostgreSQL proxy.

false

startupInstance

string

The instance that should be started first when bootstrapping and/or starting a PostgresCluster.

false

startupInstanceSet

string

The instance set associated with the startupInstance

false

usersRevision

string

Identifies the users that have been installed into PostgreSQL.

false

PostgresCluster.status.conditions[index]   Parent

Condition contains details for one aspect of the current state of this API Resource. — This struct is intended for direct use as an array at the field path .status.conditions. For example, type FooStatus struct{ // Represents the observations of a foo's current state. // Known .status.conditions.type are: "Available", "Progressing", and "Degraded" // +patchMergeKey=type // +patchStrategy=merge // +listType=map // +listMapKey=type Conditions []metav1.Condition `json:"conditions,omitempty" patchStrategy:"merge" patchMergeKey:"type" protobuf:"bytes,1,rep,name=conditions"` // other fields }

Name

Type

Description

Required

lastTransitionTime

string

lastTransitionTime is the last time the condition transitioned from one status to another. This should be when the underlying condition changed. If that is not known, then using the time when the API field changed is acceptable.

true

message

string

message is a human readable message indicating details about the transition. This may be an empty string.

true

reason

string

reason contains a programmatic identifier indicating the reason for the condition's last transition. Producers of specific condition types may define expected values and meanings for this field, and whether the values are considered a guaranteed API. The value should be a CamelCase string. This field may not be empty.

true

status

enum

status of the condition, one of True, False, Unknown.

true

type

string

type of condition in CamelCase or in foo.example.com/CamelCase. — Many .condition.type values are consistent across resources like Available, but because arbitrary conditions can be useful (see .node.status.conditions), the ability to deconflict is important. The regex it matches is (dns1123SubdomainFmt/)?(qualifiedNameFmt)

true

observedGeneration

integer

observedGeneration represents the .metadata.generation that the condition was set based upon. For instance, if .metadata.generation is currently 12, but the .status.conditions[x].observedGeneration is 9, the condition is out of date with respect to the current state of the instance.

false

PostgresCluster.status.instances[index]   Parent

Name

Type

Description

Required

name

string

true

readyReplicas

integer

Total number of ready pods.

false

replicas

integer

Total number of non-terminated pods.

false

updatedReplicas

integer

Total number of non-terminated pods that have the desired specification.

false

PostgresCluster.status.monitoring   Parent

Current state of PostgreSQL cluster monitoring tool configuration

Name

Type

Description

Required

exporterConfiguration

string

false

PostgresCluster.status.patroni   Parent

Name

Type

Description

Required

systemIdentifier

string

The PostgreSQL system identifier reported by Patroni.

false

PostgresCluster.status.pgbackrest   Parent

Status information for pgBackRest

Name

Type

Description

Required

manualBackup

object

Status information for manual backups

false

repoHost

object

Status information for the pgBackRest dedicated repository host

false

repos

[]object

Status information for pgBackRest repositories

false

restore

object

Status information for in-place restores

false

scheduledBackups

[]object	Status information for scheduled backups	false

PostgresCluster.status.pgbackrest.manualBackup Parent

Status information for manual backups

Name	Type	Description	Required
finished	boolean	Specifies whether or not the Job is finished executing (does not indicate success or failure).	true
id	string	A unique identifier for the manual backup as provided using the “pgbackrest-backup” annotation when initiating a backup.	true
active	integer	The number of actively running manual backup Pods.	false
completionTime	string	Represents the time the manual backup Job was determined by the Job controller to be completed. This field is only set if the backup completed successfully. Additionally, it is represented in RFC3339 form and is in UTC.	false
failed	integer	The number of Pods for the manual backup Job that reached the “Failed” phase.	false
startTime	string	Represents the time the manual backup Job was acknowledged by the Job controller. It is represented in RFC3339 form and is in UTC.	false
succeeded	integer	The number of Pods for the manual backup Job that reached the “Succeeded” phase.	false

PostgresCluster.status.pgbackrest.repoHost Parent

Status information for the pgBackRest dedicated repository host

Name	Type	Description	Required

apiVersion

string

APIVersion defines the versioned schema of this representation of an object. Servers should convert recognized schemas to the latest internal value, and may reject unrecognized values. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources

false

kind

string

Kind is a string value representing the REST resource this object represents. Servers may infer this from the endpoint the client submits requests to. Cannot be updated. In CamelCase. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds

false

ready

boolean

Whether or not the pgBackRest repository host is ready for use

false

PostgresCluster.status.pgbackrest.repos[index]   Parent

RepoStatus the status of a pgBackRest repository

Name

Type

Description

Required

name

string

The name of the pgBackRest repository

true

bound

boolean

Whether or not the pgBackRest repository PersistentVolumeClaim is bound to a volume

false

replicaCreateBackupComplete

boolean

ReplicaCreateBackupReady indicates whether a backup exists in the repository as needed to bootstrap replicas.

false

repoOptionsHash

string

A hash of the required fields in the spec for defining an Azure, GCS or S3 repository, Utilizd to detect changes to these fields and then execute pgBackRest stanza-create commands accordingly.

false

stanzaCreated

boolean

Specifies whether or not a stanza has been successfully created for the repository

false

volume

string

The name of the volume the containing the pgBackRest repository

false

PostgresCluster.status.pgbackrest.restore   Parent

Status information for in-place restores

Name	Type	Description	Required
finished	boolean	Specifies whether or not the Job is finished executing (does not indicate success or failure).	true
id	string	A unique identifier for the manual backup as provided using the “pgbackrest-backup” annotation when initiating a backup.	true
active	integer	The number of actively running manual backup Pods.	false
completionTime	string	Represents the time the manual backup Job was determined by the Job controller to be completed. This field is only set if the backup completed successfully. Additionally, it is represented in RFC3339 form and is in UTC.	false
failed	integer	The number of Pods for the manual backup Job that reached the “Failed” phase.	false
startTime	string	Represents the time the manual backup Job was acknowledged by the Job controller. It is represented in RFC3339 form and is in UTC.	false
succeeded	integer	The number of Pods for the manual backup Job that reached the “Succeeded” phase.	false

PostgresCluster.status.pgbackrest.scheduledBackups[index]   Parent

Name	Type	Description	Required
active	integer		

The number of actively running manual backup Pods.

false

completionTime

string

Represents the time the manual backup Job was determined by the Job controller to be completed. This field is only set if the backup completed successfully. Additionally, it is represented in RFC3339 form and is in UTC.

false

cronJobName

string

The name of the associated pgBackRest scheduled backup CronJob

false

failed

integer

The number of Pods for the manual backup Job that reached the "Failed" phase.

false

repo

string

The name of the associated pgBackRest repository

false

startTime

string

Represents the time the manual backup Job was acknowledged by the Job controller. It is represented in RFC3339 form and is in UTC.

false

succeeded

integer

The number of Pods for the manual backup Job that reached the "Succeeded" phase.

false

type

string

The pgBackRest backup type for this Job

false

PostgresCluster.status.proxy Parent

Current state of the PostgreSQL proxy.

Name

Type

Description

Required

pgBouncer

object

false

PostgresCluster.status.proxy.pgBouncer Parent

Name

Type

Description

Required

postgresRevision

string

Identifies the revision of PgBouncer assets that have been installed into PostgreSQL.

false

readyReplicas

integer

Total number of ready pods.

false

replicas

integer

Total number of non-terminated pods.

false

Kubernetes Compatibility

PGO, the Postgres Operator from Crunchy Data, is tested on the following platforms:

- Kubernetes 1.19+
- OpenShift 4.6+
- Rancher
- Google Kubernetes Engine (GKE), including Anthos
- Amazon EKS
- Microsoft AKS
- VMware Tanzu

Components Compatibility

The following table defines the compatibility between PGO and the various component containers needed to deploy PostgreSQL clusters using PGO.

The listed versions of Postgres show the latest minor release (e.g. {{< param postgresVersion13 >}}) of each major version (e.g. {{< param postgresVersion >}}). Older minor releases may still be compatible with PGO. We generally recommend to run the latest minor release for the same reasons that the PostgreSQL community provides.

Note that for the 5.0.3 release and beyond, the Postgres containers were renamed to `crunchy-postgres` and `crunchy-postgres-gis`.

Component	Version	PGO Version Min.	PGO Version Max.
`crunchy-pgbackrest`	2.36	5.0.4	{{< param operatorVersion >}}
`crunchy-pgbackrest`	2.35	5.0.3	5.0.3
`crunchy-pgbackrest`	2.33	5.0.0	5.0.2
`crunchy-pgbouncer`	1.16.1	5.0.4	{{< param operatorVersion >}}
`crunchy-pgbouncer`	1.15	5.0.0	{{< param operatorVersion >}}
`crunchy-postgres`	{{< param postgresVersion14 >}}	5.0.3	{{< param operatorVersion >}}
`crunchy-postgres`	{{< param postgresVersion13 >}}	5.0.3	{{< param operatorVersion >}}
`crunchy-postgres`	{{< param postgresVersion12 >}}	5.0.3	{{< param operatorVersion >}}
`crunchy-postgres`	{{< param postgresVersion11 >}}	5.0.3	{{< param operatorVersion >}}
`crunchy-postgres`	{{< param postgresVersion10 >}}	5.0.3	{{< param operatorVersion >}}
`crunchy-postgres-gis`	{{< param postgresVersion14 >}} -3.1	5.0.3	{{< param operatorVersion >}}
`crunchy-postgres-gis`	{{< param postgresVersion13 >}}-3.1	5.0.3	{{< param operatorVersion >}}
`crunchy-postgres-gis`	{{< param postgresVersion13 >}}-3.0	5.0.3	{{< param operatorVersion >}}

Component	Version	PGO Version Min.	PGO Version Max.
`crunchy-postgres-gis`	{{< param postgresVersion12 >}}-3.0	5.0.3	{{< param operatorVersion >}}
`crunchy-postgres-gis`	{{< param postgresVersion12 >}}-2.5	5.0.3	{{< param operatorVersion >}}
`crunchy-postgres-gis`	{{< param postgresVersion11 >}}-2.5	5.0.3	{{< param operatorVersion >}}
`crunchy-postgres-gis`	{{< param postgresVersion11 >}}-2.4	5.0.3	{{< param operatorVersion >}}
`crunchy-postgres-gis`	{{< param postgresVersion10 >}}-2.4	5.0.3	{{< param operatorVersion >}}
`crunchy-postgres-gis`	{{< param postgresVersion10 >}}-2.3	5.0.3	{{< param operatorVersion >}}

The latest Postgres containers include Patroni 2.1.1.

The following are the Postgres containers available for version 5.0.2 of PGO and older:

Component	Version	PGO Version Min.	PGO Version Max.
`crunchy-postgres-ha`	13.4	5.0.0	5.0.2
`crunchy-postgres-ha`	12.8	5.0.0	5.0.2
`crunchy-postgres-ha`	11.13	5.0.0	5.0.2
`crunchy-postgres-ha`	10.18	5.0.0	5.0.2
`crunchy-postgres-gis-ha`	13.4-3.1	5.0.0	5.0.2
`crunchy-postgres-gis-ha`	13.4-3.0	5.0.0	5.0.2
`crunchy-postgres-gis-ha`	12.8-3.0	5.0.0	5.0.2
`crunchy-postgres-gis-ha`	12.8-2.5	5.0.0	5.0.2
`crunchy-postgres-gis-ha`	11.13-2.5	5.0.0	5.0.2
`crunchy-postgres-gis-ha`	11.13-2.4	5.0.0	5.0.2
`crunchy-postgres-gis-ha`	10.18-2.4	5.0.0	5.0.2
`crunchy-postgres-gis-ha`	10.18-2.3	5.0.0	5.0.2

Container Tags

The container tags follow one of two patterns:

- `<baseImage>-<softwareVersion>-<buildVersion>`
- `<baseImage>-<softwareVersion>-<pgoVersion>-<buildVersion>` (Customer Portal only)

For example, if pulling from the customer portal, the following would both be valid tags to reference the pgBouncer container:

- `ubi8-1.15-3`
- `ubi8-1.15-5.0.3-0`
- `centos8-1.15-3`
- `centos8-1.15-5.0.3-0`

The developer portal provides CentOS based images. For example, pgBouncer would use this tag:

- `centos8-1.15-3`

PostGIS enabled containers have both the Postgres and PostGIS software versions included. For example, Postgres 14 with Postgis 3.1 would use the following tags:

- `ubi8-14.0-3.1-0`
- `ubi8-14.0-3.1-5.0.3-0`
- `centos8-14.0-3.1-0`
- `centos8-14.0-3.1-5.0.3-0`

Extensions Compatibility

The following table defines the compatibility between Postgres extensions and versions of Postgres they are available in. The "Postgres version" corresponds with the major version of a Postgres container.

The table also lists the initial PGO version that the version of the extension is available in.

Extension	Version	Postgres Versions	Initial PGO Version
`pgAudit`	1.6.1	14	5.0.4
`pgAudit`	1.6.0	14	5.0.3
`pgAudit`	1.5.0	13	5.0.0
`pgAudit`	1.4.1	12	5.0.0
`pgAudit`	1.3.2	11	5.0.0
`pgAudit`	1.2.2	10	5.0.0
`pgAudit Analyze`	1.0.8	14, 13, 12, 11, 10	5.0.3
`pgAudit Analyze`	1.0.7	13, 12, 11, 10	5.0.0
`pg_cron`	1.3.1	14, 13, 12, 11, 10	5.0.0
`pg_partman`	4.6.0	14, 13, 12, 11, 10	5.0.4
`pg_partman`	4.5.1	13, 12, 11, 10	5.0.0
`pgnodemx`	1.2.0	14, 13, 12, 11, 10	5.0.4
`pgnodemx`	1.0.5	14, 13, 12, 11, 10	5.0.3
`pgnodemx`	1.0.4	13, 12, 11, 10	5.0.0
`set_user`	3.0.0	14, 13, 12, 11, 10	5.0.3
`set_user`	2.0.1	13, 12, 11, 10	5.0.2
`set_user`	2.0.0	13, 12, 11, 10	5.0.0
`TimescaleDB`	2.5.0	14, 13, 12	5.0.3
`TimescaleDB`	2.4.2	13, 12	5.0.3
`TimescaleDB`	2.4.0	13, 12	5.0.2
`TimescaleDB`	2.3.1	11	5.0.1
`TimescaleDB`	2.2.0	13, 12, 11	5.0.0
`wal2json`	2.4	14, 13, 12, 11, 10	5.0.3
`wal2json`	2.3	13, 12, 11, 10	5.0.0

Geospatial Extensions

The following extensions are available in the geospatially aware containers (`crunchy-postgres-gis`):

Extension	Version	Postgres Versions	Initial PGO Version
`PostGIS`	3.1	14, 13	5.0.0
`PostGIS`	3.0	13, 12	5.0.0
`PostGIS`	2.5	12, 11	5.0.0
`PostGIS`	2.4	11, 10	5.0.0
`PostGIS`	2.3	10	5.0.0
`pgrouting`	3.1.3	13	5.0.0
`pgrouting`	3.0.5	13, 12	5.0.0
`pgrouting`	2.6.3	12, 11, 10	5.0.0

Crunchy Data announces the release of Crunchy Postgres for Kubernetes 5.0.4.

Crunchy Postgres for Kubernetes is powered by PGO, the open source Postgres Operator from Crunchy Data. PGO is released in conjunction with the Crunchy Container Suite.

Crunchy Postgres for Kubernetes 5.0.4 includes the following software versions upgrades:

- PostgreSQL versions 14.1, 13.5, 12.9, 11.14, and 10.19 are now available.
- PostGIS version 3.1.4 is now available.
- pgBackRest is now at version 2.36.
- PgBouncer is now at version 1.16.
- The pgAudit extension is now at version 1.6.1.
- The pgnodemx extension is now at version 1.2.0.
- The pg_partman extension is now at version 4.6.0.
- The TimescaleDB extension is now at version 2.5.0.

Read more about how you can [get started]({{< relref "quickstart/_index.md" >}}) with Crunchy Postgres for Kubernetes. We recommend forking the Postgres Operator examples repo.

Features

- The JDBC connection string for the Postgres database and a PgBouncer instance is now available in the User Secret using `jdbc-uri` and `pgbouncer-jdbc-uri` respectively.
- Editing the `password` field of a User Secret now [changes a password]({{< relref "architecture/user-management.md" >}}#custom-passwords), instead of having to create a verifier.

Changes

- PostGIS is now automatically enabled when using the `crunchy-postgres-gis` container.
- The Downward API is mounted to the `database` containers.
- pgnodemx can now be enabled and used without having to enable monitoring.
- The description of the `name` field for an instance set now states that a name is only optional when a single instance set is defined.

Fixes

- Fix issue when performing a restore with PostgreSQL 14. Specifically, if there are mismatched PostgreSQL configuration parameters, PGO will resume replay and let PostgreSQL crash so PGO can ultimately fix it, vs. the restore pausing indefinitely.
- The pgBackRest Pod no longer automatically mounts the default Service Account. Reported by (@Shrivastava-Varsha).
- The Jobs that move data between volumes now have the correct Security Context set.
- The UBI 8 `crunchy-upgrade` container contains all recent PostgreSQL versions that can be upgraded.
- Ensure controller references are used for all objects that need them, instead of owner references.
- It is no longer necessary to have external WAL volumes enabled in order to upgrade a PGO v4 cluster to PGO v5 using the "Migrate From Backups" or "Migrate Using a Standby Cluster" upgrade methods.

Crunchy Data announces the release of Crunchy Postgres for Kubernetes 5.0.3.

Crunchy Postgres for Kubernetes is powered by PGO, the open source Postgres Operator from Crunchy Data. PGO is released in conjunction with the Crunchy Container Suite.

Crunchy Postgres for Kubernetes 5.0.3 includes the following software versions upgrades:

- PostgreSQL 14 is now available.
- pgBackRest is updated to version 2.35.
- Patroni is updated to version 2.1.1.
- The pgAudit extension is now at version 1.6.0.
- The pgAudit Analyze extension is now at version 1.0.8.
- The pgnodemx extension is now at version 1.0.5.
- The set_user extension is now at version 3.0.0.
- The wal2json extension is now at version 2.4.

Read more about how you can [get started]({{< relref "quickstart/_index.md" >}}) with Crunchy Postgres for Kubernetes. We recommend forking the Postgres Operator examples repo.

Features

- The Postgres containers are renamed. `crunchy-postgres-ha` is now `crunchy-postgres`, and `crunchy-postgres-gis-ha` is now `crunchy-postgres-gis`.
- Some network filesystems are sensitive to Linux user and group permissions. Process GIDs can now be configured through `PostgresCluster.spec.supplementalGroups` for when your PVs don't advertise their GID requirements.
- A replica service is now automatically reconciled for access to Postgres replicas within a cluster.
- The Postgres primary service and PgBouncer service can now each be configured to have either a `ClusterIP`, `NodePort` or `LoadBalancer` service type. Suggested by Bryan A. S. (@bryanasdev000).
- Pod Topology Spread Constraints can now be specified for Postgres instances, the pgBackRest dedicated repository host as well as PgBouncer. Suggested by Annette Clewett.
- Default topology spread constraints are included to ensure PGO always attempts to deploy a high availability cluster architecture.
- PGO can now execute a custom SQL script when initializing a Postgres cluster.
- Custom resource requests and limits are now configurable for all `init` containers, therefore ensuring the desired Quality of Service (QoS) class can be assigned to the various Pods comprising a cluster.
- Custom resource requests and limits are now configurable for all Jobs created for a `PostgresCluster`.
- A Pod Priority Class is configurable for the Pods created for a `PostgresCluster`.
- An `imagePullPolicy` can now be configured for Pods created for a `PostgresCluster`.
- Existing `PGDATA`, Write-Ahead Log (WAL) and pgBackRest repository volumes can now be migrated from PGO v4 to PGO v5 by specifying a `volumes` data source when creating a `PostgresCluster`.
- There is now a [migration guide available for moving Postgres clusters between PGO v4 to PGO v5]({{< relref "guides/v4tov5.md" >}}).
- The pgAudit extension is now enabled by default in all clusters.
- There is now additional validation for PVC definitions within the `PostgresCluster` spec to ensure successful PVC reconciliation.
- Postgres server certificates are now automatically reloaded when they change.

Changes

- The supplemental group `65534` is no longer applied by default. Upgrading the operator will perform a rolling update on all `PostgresCluster` custom resources to remove it.

If you need this GID for your network filesystem, you should perform the following steps when upgrading:

1. Before deploying the new operator, deploy the new CRD. You can get the new CRD from the Postgres Operator Examples repository and executing the following command: `console     $ kubectl apply -k kustomize/install`

2. Add the group to your existing `PostgresCluster` custom resource: "'console $ kubectl edit postgrescluster/hippo

   kind: PostgresCluster … spec: supplementalGroups:

   - 65534 … "'

   or

   `console $ kubectl patch postgrescluster/hippo --type=merge --patch='{"spec":{"supplementalGroups":[65534]}}'`

   or

   by modifying `spec.supplementalGroups` in your manifest.

3. Deploy the new operator. If you are using an up-to-date version of the manifest, you can run: `console     $ kubectl apply -k kustomize/install`

- A dedicated pgBackRest repository host is now only deployed if a `volume` repository is configured. This means that deployments that use only cloud-based (`s3`, `gcs`, `azure`) repos will no longer see a dedicated repository host, nor will `SSHD` run in within that Postgres cluster. As a result of this change, the `spec.backups.pgbackrest.repoHost.dedicated` section is removed from the `PostgresCluster` spec, and all settings within it are consolidated under the `spec.backups.pgbackrest.repoHost` section. When upgrading please update the `PostgresCluster` spec to ensure any settings from section `spec.backups.pgbackrest.repoHost.dedicated` are moved into section `spec.backups.pgbackrest.repoHost`.
- PgBouncer now uses SCRAM when authenticating into Postgres.
- Generated Postgres certificates include the FQDN and other local names of the primary Postgres service. To regenerate the certificate of an existing cluster, delete the `tls.key` field from its certificate secret. Suggested by @ackerr01.

Fixes

- Validation for the PostgresCluster spec is updated to ensure at least one repo is always defined for section `spec.backups.pgbackrest.re`
- A restore will now complete successfully If `max_connections` and/or `max_worker_processes` is configured to a value higher than the default when backing up the Postgres database. Reported by Tiberiu Patrascu (@tpatrascu).
- The installation documentation now properly defines how to set the `PGO_TARGET_NAMESPACE` environment variable for a single namespace installation.
- Ensure the full allocation of shared memory is available to Postgres containers. Reported by Yuyang Zhang (@helloqiu).
- OpenShift auto-detection logic now looks for the presence of the `SecurityContextConstraints` API to avoid false positives when APIs with an `openshift.io` Group suffix are installed in non-OpenShift clusters. Reported by Jean-Daniel.

Crunchy Data announces the release of Crunchy Postgres for Kubernetes 5.0.2.

Crunchy Postgres for Kubernetes is powered by PGO, the open source Postgres Operator from Crunchy Data. PGO is released in conjunction with the Crunchy Container Suite.

Crunchy Postgres for Kubernetes 5.0.2 includes the following software versions upgrades:

- PostgreSQL is updated to 13.4, 12.8, 11.13, and 10.18.
- PL/Tcl is now included in the PostGIS (`crunchy-postgres-gis-ha`) container.
- The TimescaleDB extension is now at version 2.4.0.
- The set_user extension is now at version 2.0.1.

Read more about how you can [get started]({{< relref "quickstart/_index.md" >}}) with Crunchy Postgres for Kubernetes. We recommend forking the Postgres Operator examples repo.

Crunchy Data announces the release of Crunchy Postgres for Kubernetes 5.0.1.

Crunchy Postgres for Kubernetes is powered by PGO, the open source Postgres Operator from Crunchy Data. PGO is released in conjunction with the Crunchy Container Suite.

Crunchy Postgres for Kubernetes 5.0.1 includes the following software versions upgrades:

- Patroni is now at 2.1.0.
- PL/Tcl is now included in the PostGIS (`crunchy-postgres-gis-ha`) container.

Read more about how you can [get started]({{< relref "quickstart/_index.md" >}}) with Crunchy Postgres for Kubernetes. We recommend forking the Postgres Operator examples repo.

Features

- Custom affinity rules and tolerations can now be added to pgBackRest restore Jobs.
- OLM bundles can now be generated for PGO 5.

Changes

- The `replicas` value for an instance set must now be greater than `0`, and at least one instance set must now be defined for a `PostgresCluster`. This is to prevent the cluster from being scaled down to `0` instances, since doing so results in the inability to scale the cluster back up.
- Refreshed the PostgresCluster CRD documentation using the latest version of `crdoc` (`v0.3.0`).
- The PGO test suite now includes a test to validate image pull secrets.
- Related Image functionality has been implemented for the OLM installer as required to support offline deployments.
- The name of the PGO Deployment and ServiceAccount has been changed to `pgo` for all installers, allowing both PGO v4.x and PGO v5.x to be run in the same namespace. If you are using Kustomize to install PGO and are upgrading from PGO 5.0.0, please see the [Upgrade Guide]({{< relref "../installation/upgrade.md" >}}) for addtional steps that must be completed as a result of this change in order to ensure a successful upgrade.
- PGO now automatically detects whether or not it is running in an OpenShift environment.
- Postgres users and databases can be specified in `PostgresCluster.spec.users`. The credentials stored in the `{cluster}-pguser` Secret are still valid, but they are no longer reconciled. References to that Secret should be replaced with `{cluster}-pguser-{cluster}`. Once all references are updated, the old `{cluster}-pguser` Secret can be deleted.
- The built-in `postgres` superuser can now be managed the same way as other users. Specifying it in `PostgresCluster.spec.users` will give it a password, allowing it to connect over the network.
- PostgreSQL data and pgBackRest repo volumes are now reconciled using labels.

Fixes

- It is now possible to customize `shared_preload_libraries` when monitoring is enabled.
- Fixed a typo in the description of the `openshift` field in the PostgresCluster CRD.
- When a new cluster is created using an existing PostgresCluster as its dataSource, the original primary for that cluster will now properly initialize as a replica following a switchover. This is fixed with the upgrade to Patroni 2.1.0).
- A consistent `startupInstance` name is now set in the PostgresCluster status when bootstrapping a new cluster using an existing PostgresCluster as its data source.
- It is now possible to properly customize the `pg_hba.conf` configuration file.

Crunchy Data announces the release of the PGO, the open source Postgres Operator, 5.0.0 on June 30, 2021.

To get started with PGO 5.0.0, we invite you to read through the [quickstart]({{< relref "quickstart/_index.md" >}}). We also encourage you to work through the [PGO tutorial]({{< relref "tutorial/_index.md" >}}).

PGO 5.0.0 is a major release of the Postgres Operator. The focus of this release was to take the features from the previous versions of PGO, add in some new features, and allow you to deploy Kubernetes native Postgres through a fully declarative, GitOps style workflow. As with previous versions, PGO 5.0 makes it easy to deploy production ready, cloud native Postgres.

Postgres clusters are now fully managed through a custom resource called [`postgrescluster.postgres-operator.crunchydata.com`]({{< relref "references/crd.md" >}}). You can also view the various attributes of the custom resource using `kubectl explain postgrescluster.postgres-operator.crunchydata.com` or `kubectl explain postgrescluster`. The custom resource can be edited at any time, and all of the changes are rolled out in a minimally disruptive way.

There are a set of examples for how to use Kustomize and Helm with PGO 5.0. This example set will grow and we encourage you to contribute to it.

PGO 5.0 continues to support the Postgres architecture that was built up in previous releases. This means that Postgres clusters are deployed without a single-point-of-failure and can continue operating even if PGO is unavailable. PGO 5.0 includes support for Postgres high availability, backup management, disaster recovery, monitoring, full customizability, database cloning, connection pooling, security, running with locked down container settings, and more.

PGO 5.0 also continuously monitors your environment to ensure all of the components you want deployed are available. For example, if PGO detects that your connection pooler is missing, it will recreate it as you specified in the custom resource. PGO 5.0 can watch for Postgres clusters in all Kubernetes namespaces or be isolated to individual namespaces.

As PGO 5.0 is a major release, it is not backwards compatible with PGO 4.x. However, you can run PGO 4.x and PGO 5.0 in the same Kubernetes cluster, which allows you to migrate Postgres clusters from 4.x to 5.0.

Changes

Beyond being fully declarative, PGO 5.0 has some notable changes that you should be aware of. These include:

- The minimum Kubernetes version is now 1.18. The minimum OpenShift version is 4.5. This release drops support for OpenShift 3.11.
- We recommend running the latest bug fix releases of Kubernetes.
- The removal of the `pgo` client. This may be reintroduced in a later release, but all actions on a Postgres cluster can be accomplished using `kubectl`, `oc`, or your preferred Kubernetes management tool (e.g. ArgoCD).
- A fully defined `status` subresource is now available within the `postgrescluster` custom resource that provides direct insight into the current status of a PostgreSQL cluster.
- Native Kubernetes eventing is now utilized to generate and record events related to the creation and management of PostgreSQL clusters.
- Postgres instances now use Kubernetes Statefulsets.
- Scheduled backups now use Kubernetes CronJobs.
- Connections to Postgres require TLS. You can bring your own TLS infrastructure, otherwise PGO provides it for you.
- Custom configurations for all components can be set directly on the `postgrescluster` custom resource.

Features

In addition to supporting the PGO 4.x feature set, the PGO 5.0.0 adds the following new features:

- Postgres minor version (bug fix) updates can be applied without having to update PGO. You only need to update the `image` attribute in the custom resource.
- Adds support for Azure Blob Storage for storing backups. This is in addition to using Kubernetes storage, Amazon S3 (or S3-equivalents like MinIO), and Google Cloud Storage (GCS).
- Allows for backups to be stored in up to four different locations simultaneously.
- Backup locations can be changed during the lifetime of a Postgres cluster, e.g. moving from "posix" to "s3".

Project FAQ

What is The PGO Project?

The PGO Project is the open source project associated with the development of PGO, the Postgres Operator for Kubernetes from Crunchy Data.

PGO is a Kubernetes Operator, providing a declarative solution for managing your PostgreSQL clusters. Within a few moments, you can have a Postgres cluster complete with high availability, disaster recovery, and monitoring, all over secure TLS communications.

PGO is the upstream project from which Crunchy PostgreSQL for Kubernetes is derived. You can find more information on Crunchy PostgreSQL for Kubernetes here.

What's the difference between PGO and Crunchy PostgreSQL for Kubernetes?

PGO is the Postgres Operator from Crunchy Data. It developed pursuant to the PGO Project and is designed to be a frequently released, fast-moving project where all new development happens.

Crunchy PostgreSQL for Kubernetes is produced by taking selected releases of PGO, combining them with Crunchy Certified PostgreSQL and PostgreSQL containers certified by Crunchy Data, maintained for commercial support, and made available to customers as the Crunchy PostgreSQL for Kubernetes offering.

Where can I find support for PGO?

The community can help answer questions about PGO via the PGO mailing list.

Information regarding support for PGO is available in the [Support]({{< relref "support/_index.md" >}}) section of the PGO documentation, which you can find [here]({{< relref "support/_index.md" >}}).

For additional information regarding commercial support and Crunchy PostgreSQL for Kubernetes, you can contact Crunchy Data.

Under which open source license is PGO source code available?

The PGO source code is available under the Apache License 2.0.

Where are the release tags for PGO v5?

With PGO v5, we've made some changes to our overall process. Instead of providing quarterly release tags as we did with PGO v4, we're focused on ongoing active development in the v5 primary development branch (`master`, which will become `main`). Consistent with our practices in v4, previews of stable releases with the release tags are made available in the Crunchy Data Developer Portal.

These changes allow for more rapid feature development and releases in the upstream PGO project, while providing Crunchy Postgres for Kubernetes users with stable releases for production use.

To the extent you have constraints specific to your use, please feel free to reach out on info@crunchydata.com to discuss how we can address those specifically.

How can I get involved with the PGO Project?

PGO is developed by the PGO Project. The PGO Project that welcomes community engagement and contribution.

The PGO source code and community issue trackers are hosted at GitHub.

For community questions and support, please sign up for the PGO mailing list.

For information regarding contribution, please review the contributor guide here.

Please register for the Crunchy Data Developer Portal mailing list to receive updates regarding Crunchy PostgreSQL for Kubernetes releases and the Crunchy Data newsletter for general updates from Crunchy Data.

Where do I report a PGO bug?

The PGO Project uses GitHub for its issue tracking. You can file your issue here.

How often is PGO released?

The PGO team currently plans to release new builds approximately every few weeks. The PGO team will flag certain builds as "stable" at their discretion. Note that the term "stable" does not imply fitness for production usage or any kind of warranty whatsoever.

There are a few options available for community support of the PGO: the Postgres Operator:

- **If you believe you have found a bug** or have a detailed feature request: please open an issue on GitHub. The Postgres Operator community and the Crunchy Data team behind the PGO is generally active in responding to issues.
- **For general questions or community support**: please join the PostgreSQL Operator community mailing list at https://groups.google.com/a/crunchydata.com/forum/#!forum/postgres-operator/join,

In all cases, please be sure to provide as many details as possible in regards to your issue, including:

- Your Platform (e.g. Kubernetes vX.YY.Z)
- Operator Version (e.g. {{< param centosBase >}}-{{< param operatorVersion >}})
- A detailed description of the issue, as well as steps you took that lead up to the issue
- Any relevant logs
- Any additional information you can provide that you may find helpful

For production and commercial support of the PostgreSQL Operator, please contact Crunchy Data at info@crunchydata.com for information regarding an Enterprise Support Subscription.