NDB Operator Manual

NDB Operator for Kubernetes

NDB Operator Manual

This is the *NDB Operator Manual*, which provides information about installing and using NDB Operator 9.2 for Kubernetes. This Manual covers NDB Operator 9.2 releases through 9.2.0.

The information presented in this guide relating to MySQL NDB Cluster is current for recent releases up to and including NDB Cluster 9.2. For more information about NDB 9.2, see What is New in MySQL NDB Cluster 9.2.

For legal information, see the Legal Notices.

For help with using MySQL, please visit the MySQL Forums, where you can discuss your issues with other MySQL users.

Licensing information—MySQL NDB Operator for Kubernetes. If you are using MySQL NDB Operator with a *Commercial* release of MySQL NDB Cluster, see the MySQL NDB Operator 9.2 Commercial Release License Information User Manual for licensing information, including licensing information relating to third-party software that may be included in this Commercial release. If you are using MySQL NDB Operator with a *Community* release of MySQL NDB Cluster, see the MySQL NDB Operator 9.2 Community Release License Information User Manual for licensing information, including licensing information relating to third-party software that may be included in this Community release.

Document generated on: 2025-03-11 (revision: 81148)

Table of Contents

Chapter 1 Introduction to NDB Operator

MySQL NDB Cluster, the distributed version of MySQL, is implemented as a collection of processes or nodes of the following three types:

- Management nodes (`ndb_mgmd`): Provision of configuration data
- Data nodes (`ndbmtd`): User data storage
- SQL nodes (`mysqld`): SQL frontend to the data nodes.

NDB Operator provides services that allow for connections of the following two types:

- Connections to NDB Cluster management nodes by applications which use the MGM API such as `ndb_mgm` and `ndb_config`.
- Connections to NDB Cluster SQL nodes by MySQL client applications such as `mysql`.

NDB Operator currently does not provide a mechanism for NDB API applications to connect directly to the NDB Cluster data nodes. This means that NDB utility programs such as `ndb_select_all` and `ndb_restore` are not supported under NDB Operator.

See NDB Cluster Core Concepts, for more information about NDB Cluster node processes. See also NDB Cluster Programs.

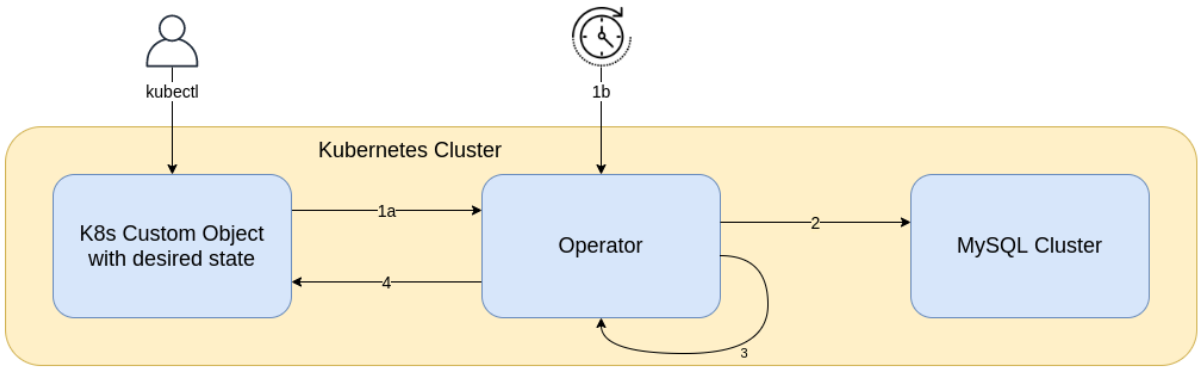
Deploying MySQL Cluster in a Kubernetes cluster requires scheduling these nodes onto different pods using multiple workload resources, as well as time and effort to choose and create the right type of workload resources in a configuration that takes full advantage of the redundancy and fault tolerance features found in both NDB Cluster and Kubernetes Cluster.

NDB Operator is the Kubernetes Operator for MySQL NDB Cluster, and is intended to simplify the task of deploying and managing MySQL Cluster in a Kubernetes Cluster. A Kubernetes operator is an application having operational knowledge of another application. It can be deployed within the Kubernetes Cluster after which it can begin monitoring the endpoints in which they are interested, and making changes to the application being managed. NDB Operator makes it possible to deploy, manage, and makes changes to an NDB Cluster with a minimum of human intervention.

To represent an NDB Cluster, we define an `Ndb` Custom Resource Definition (CRD) in the Kubernetes cluster. A custom object of kind Ndb can now be created in the Kubernetes cluster, representing the configuration of a desired NDB Cluster setup to be deployed. NDB Operator watches for any changes to such custom objects and, based on the changes to this object, deploys and maintains the NDB Cluster nodes in the Kubernetes cluster.

NDB Operator runs a reconciliation loop at regular intervals whenever there is a change to an `Ndb` custom resource, as shown in the following diagram:

Figure 1.1 NDB Operator reconciliation loop



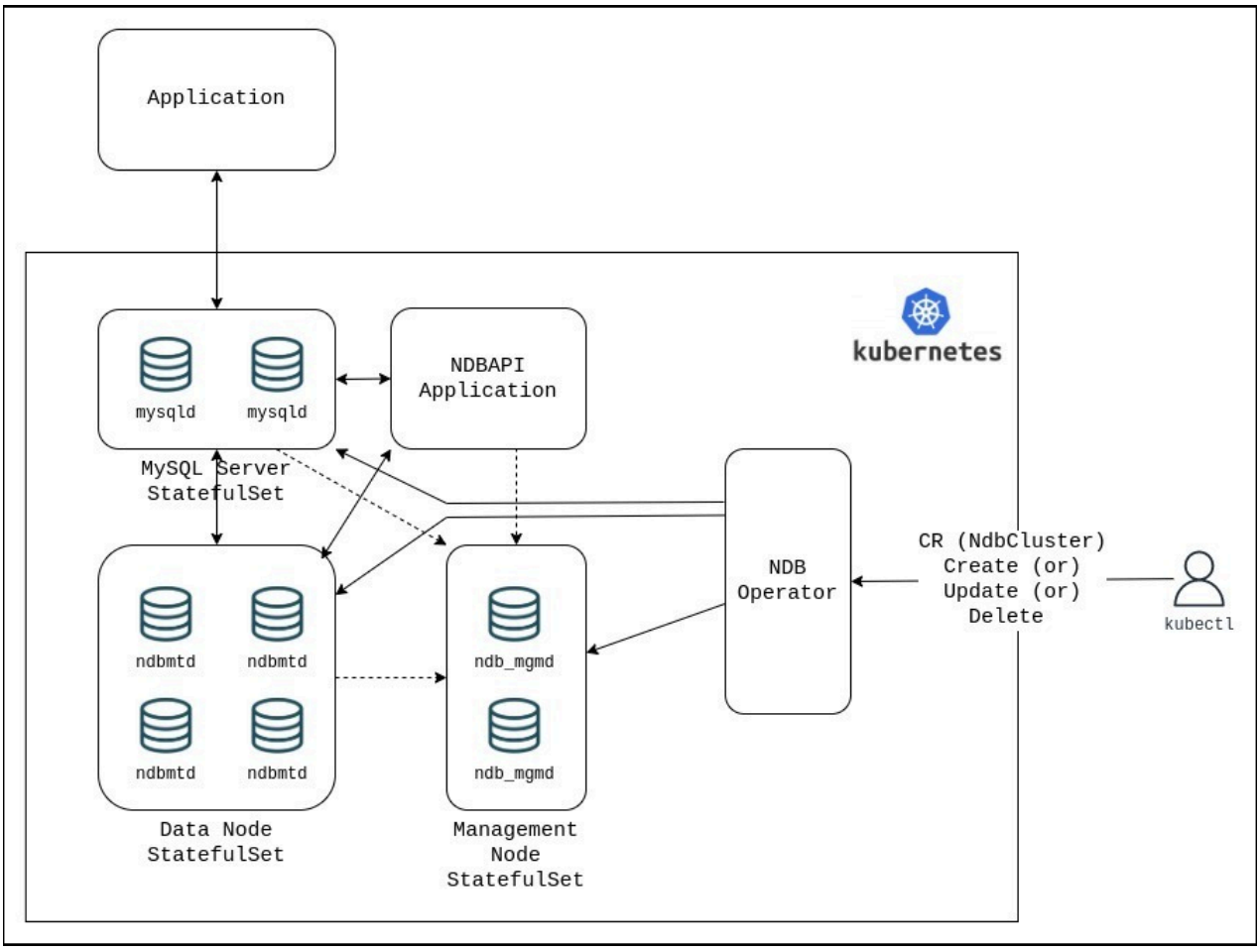
Reconciliation loop

1a. Events get sent when an object gets created, updated or deleted (or)
1b. Trigger reconciliation at regular intervals
2. Reconcile MySQL Cluster
   a. Observe the current state of MySQL Cluster
   b. Compare it with the desired state
   c. Make changes to MySQL Cluster to move it towards the desired state
3. Requeue if necessary - i.e. on error or if the current state is not the desired one yet, the loop is retried
4. Update the custom object's status to reflect the changes

Each time through the loop, the operator compares the desired state (the values from the custom resource in the Kubernetes cluster) with the current state (the actual state of the MySQL Cluster installation in the Kubernetes cluster). If the current and desired states do not match, the operator makes incremental changes to the MySQL Cluster installation to move it closer to the desired state; this is repeated if and as necessary over multiple iterations until the MySQL Cluster has the desired state. In this fashion, NDB Operator is able to monitor the NDB Cluster and to ensure that it is running as expected.

NDB Operator architecture:

Figure 1.2 NDB Operator architecture



Chapter 2 Installation of NDB Operator

Table of Contents

The NDB Operator and related resources can be installed in a Kubernetes cluster using either the helm chart or the included single YAML installation file. Both of these rely on an NDB Operator image being accessible to the Kubernetes cluster. It is also possible to build the NDB Operator image from source, then use that image to run NDB Operator in the Kubernetes cluster.

2.1 Requirements

NDB Operator requires a Kubernetes cluster of version 1.19.0 or greater. Older versions of Kubernetes are not supported; NDB Operator may not work as expected with these, if at all.

NDB Operator works with MySQL NDB Cluster 8.0.26 and later. Earlier releases of NDB Cluster are not supported.

For additional requirements for building NDB Operator from source, see Section 2.5, “Building an NDB Operator Image from Source”.

2.2 Obtaining NDB Operator

You can obtain the latest release of NDB Operator from the following locations:

- https://dev.mysql.com/downloads/ndb-operator/
- ArtifactHub (Helm charts; see Section 2.3, “Installing NDB Operator Using Helm”)
- Oracle Container Registry (Docker container image; see Section 2.4, “Installing NDB Operator Using the YAML File and kubectl”)
- https://github.com/mysql/mysql-ndb-operator (source code; see Section 2.5, “Building an NDB Operator Image from Source”)

For information about upgrading an existing NDB Operator installation, see Section 2.7, “Upgrading NDB Operator”

2.3 Installing NDB Operator Using Helm

You can install NDB Operator with the Helm package manager for Kubernetes using the Helm chart included in the NDB Operator distribution to create the necessary Custom Resource Definitions (CRDs) and to deploy NDB Operator (together with the web hook server) in a Kubernetes cluster. You can set a number of options in the Helm chart affecting the installation; these are described later in this section.

The remainder of this section assumes that Helm is available on the system. If Helm is not already present, see https://helm.sh/docs/intro/install/ for information about obtaining and installing it.

The NDB Operator Helm repository is hosted at `https://mysql.github.io/mysql-ndb-operator/`. To add the chart repository, execute the following commands:

```
> helm repo add ndb-operator-repo https://mysql.github.io/mysql-ndb-operator/
> helm repo update
```

To install the chart with the release name `ndbop`, use `helm install` as shown here:

```
> helm install --namespace=ndb-operator --create-namespace ndbop ndb-operator-repo/ndb-operator
```

To install NDB Operator from the source code, use the `helm install` command as shown here:

```
> helm install ndbop deploy/charts/ndb-operator
```

This creates the CRD and required resources, and deploys NDB Operator and the web hook server to the `default` namespace.

Use the command's `--namespace` option to deploy the operator to a specific namespace. Here, we deploy to the `ndb-operator` namespace:

```
> helm install --namespace=ndb-operator --create-namespace ndbop deploy/charts/ndb-operator
```

The NDB Operator Helm chart contains the configurable parameters described in the following list:

- `clusterScoped`: Scope of the Ndb Operator.

  If this is set to `true` (the default), the operator has cluster scope, and watches for changes to any `NdbCluster` resource across all namespaces. If it is `false`, the operator has namespace scope, and watches for changes only in the namespace to which it applies.

- `image`: The operator image name to be deployed by the Helm chart. By default, the Helm chart deploys the image from `mysql/ndb-operator:latest`.

  If you want to host the NDB Operator image in a private registry and use it from there, the image location can be set in this parameter.

- `imagePullPolicy`: NDB Operator image pull policy. Permitted values are `Always`, `Never`, and `IfNotPresent`; the default is `IfNotPresent`.

- `imagePullSecretName`: Secret to be used when pulling the NDB Operator image from a private repository.

  This is used only if the `image` parameter specifies an Operator image hosted in a private registry. Otherwise, it is ignored. There is no default value.

These options can be set using the `--set` argument of the Helm `install` command. For example:

```
> helm install --set imagePullPolicy=Always ndbop deploy/charts/ndb-operator
```

2.4 Installing NDB Operator Using the YAML File and kubectl

You can deploy the NDB Operator and the other related resources using the install manifest available in the source code, like this:

```
> kubectl apply -f deploy/manifests/ndb-operator.yaml
```

The command just shown deploys NDB Operator in the `ndb-operator` namespace.

Alternatively, you can also apply the remote copy of the manifest, without having to clone the entire repository, in a manner similar to what is shown here:

```
> kubectl apply -f https://raw.githubusercontent.com/mysql/mysql-ndb-operator/main/deploy/manifests/ndb-op
```

To configure NDB Operator parameters such as the image name when installing NDB Operator using this method, it is necessary to download the manifest file, modify the local copy, then apply this local copy to the Kubernetes cluster.

2.5 Building an NDB Operator Image from Source

This section describes how to compile, install, and test NDB Operator from source. If you do not already have the source code, see Section 2.2, “Obtaining NDB Operator”'.

The following prerequisites must be installed on the system to build and test NDB Operator:

Golang 1.16 or newer to compile the operator.

Docker to build the NDB Operator and other container images. Docker is also used to run the E2E testcases.

Minikube or KinD to deploy and test the NDB Operator. Kubernetes 1.19 or later is also required.

The `Makefile` included with the source code contains all targets needed to build the operator; this can done using the command shown here:

```
> make build
```

The Custom Resource Definition and other manifest files are regenerated by the `build` target.

By default, NDB Operator is built in release mode. To build it in debug mode for use in development instead, set the `WITH_DEBUG` environment variable to `1` when invoking the build, like this:

```
> WITH_DEBUG=1 make build
```

To generate the NDB Operator Docker image, run the command shown here:

```
> make operator-image
```

Once the image is built, it must be made accessible to the Kubernetes Cluster. For KinD, you can use `kind load` *`docker_image`* command to load the image. For Minikube, it depends on the container runtime used; see Pushing images into minikube clusters. See Chapter 3, *Deploying NDB Cluster with NDB Operator*, for information about setting up NDB Operator and deploying an NDB Cluster using the operator.

When using Minikube, the default memory allocation might not be adequate for running NDB Operator; you can increase it using `minikube config set` *`memory_limit`*. To deploy the example at `docs/examples/example-ndb.yaml`, Minikube requires at least 5GB memory; you can set this as shown here:

```
> minikube config set memory 5GB
```

Making changes to the NdbCluster type. If any change is made to the `NdbCluster` type in `pkg/apis/ndbcontroller/v1/types.go`, you must renegerate the client set, informers, and listers. You can do this by executing the following command in `pkg/generated`:

```
> make generate
```

Warning

Do not attempt to modify the `NdbCluster` type while an upgrade of NDB Operator is in progress. See Section 2.7, “Upgrading NDB Operator”, for more information.

Testing NDB Operator. The NDB Operator project comes with unit tests that are developed using the go testing package and a more elaborate End-To-End (E2E) test suite that is built on top of the Ginkgo/Gomega testing framework.

Most of the unit tests are colocated with the packages, and test the methods within those packages. They sometimes use a simulated Kubernetes client to verify whether requests sent by NDB Operator match expectations.

The E2E testcases are a collection of integration tests that make changes to an `NdbCluster` resource object and verify whether the NDB Cluster configuration controlled by the object changes accordingly. The testcases use a E2E framework developed internally and built on top of the Ginkgo testing framework. The tests should be run using `e2e-tests/run-e2e-test.go`; options for this testing tool can be found in `e2e-tests/README.md`. The tests can be run either in an existing Kubernetes cluster or a new one using KinD.

To generate the E2E testing image, run `make e2e-tests` on the command line.

To run the E2E tests in an existing Kubernetes cluster, pass the path to the `kubeconfig` file for this cluster to the `run-e2e-test.go` tool, like this:

```
> go run e2e-tests/run-e2e-test.go -kubeconfig=path/to/file
```

Both the `mysql/ndb-operator` and the `e2e-tests` images must be accessible to the Kubernetes Cluster.

The test tool can also start its own KinD cluster and then run the E2E tests in it. A Docker instance with both the `mysql/ndb-operator` and `e2e-tests` images must be accessible from the terminal in which the test is being run. The KinD cluster is started inside Docker as part of this process. You can do this running either one of the commands shown here:

```
> go run e2e-tests/run-e2e-test.go -use-kind

> make e2e-kind
```

2.6 Post-Installation

Once you have installed NDB Operator using either of the methods described previously in this chapter, you can verify the installation by executing the following command:

```
> kubectl get pods -n ndb-operator -l 'app in (ndb-operator,ndb-operator-webhook-server)'
NAME                                        READY   STATUS    RESTARTS   AGE
ndb-operator-555b7b65-7fmv8                 1/1     Running   0          13s
ndb-operator-webhook-server-d67c97d54-zdhhp   1/1     Running   0          13s
```

The pod `ndb-operator-555b7b65-7fmv8` runs NDB Operator; the other pod (`ndb-operator-webhook-server-d67c97d54-zdhhp`) runs a server that acts as an admission controller for the `NdbCluster` resource, which NDB Operator is ready to handle once both pods are ready.

2.7 Upgrading NDB Operator

NDB Operator can be upgraded from one release to another without affecting any existing NDB cluster, subject only to the constraint that you should not attempt to modify the `NdbCluster` CRD (see Section 5.1, “NdbCluster Resource”) while the upgrade is in progress.

Upgrade using Helm chart. The procedure outlined following assumes that the installed Helm chart is named `ndbop`, and that the chart is installed in the default namespace.

To update the local Helm repository and upgrade NDB Operator to the latest version, execute the following helm commands:

```
> helm repo update

> helm upgrade --namespace=ndb-operator ndbop ndb-operator-repo/ndb-operator
```

If the namespace in which NDB Operator is installed is other than `ndbop`, you can substitute its name as the argument to the `--namespace` option to specify the correct namespace.

Upgrade from source. To upgrade NDB Operator from the source code, use Helm's `upgrade` command as shown here:

```
> helm upgrade ndbop deploy/charts/ndb-operator
```

Both `helm install` and `helm upgrade` can use the `--set` option to modify the configurable parameters specified in Section 2.3, “Installing NDB Operator Using Helm”.

Upgrade using YAML file and kubectl. You can upgrade NDB Operator using the same installation commands shown in Section 2.4, “Installing NDB Operator Using the YAML File and kubectl”. In this case, you must make sure that the copy of the manifest YAML file is updated before executing `kubectl apply`.

2.8 Uninstalling NDB Operator

The method for uninstalling NDB Operator depends on the installation used to install it in Kubernetes. If it was installed using Helm, NDB Operator installed aas release `ndbop` can be uninstalled from the Kubernetes cluster by using the following command:

```
> helm uninstall ndbop
```

`helm uninstall` removes only the NDB Operator and webhook server, but does not remove the `NdbCluster` CRD, which you can delete from the Kubernetes cluster as shown here:

```
> kubectl delete customresourcedefinitions ndbclusters.mysql.oracle.com
```

If NDB Operator was installed using the YAML manifest file, you can uninstall it by removing the file, like this:

```
> kubectl delete -f deploy/manifests/ndb-operator.yaml
```

Uninstalling only the NDB Operator does not affect any existing NDB Clusters running inside the Kubernetes cluster. Deleting the `NdbCluster` custom resource definition stops and deletes all running MySQL Cluster pods.

Chapter 3 Deploying NDB Cluster with NDB Operator

Table of Contents

Deployment of a new NDB Cluster in a Kubernetes cluster consists of three steps, each of which is discussed in this chapter:

1. Determining the characteristics of the NDB Cluster; for example, the numbers of data nodes and SQL nodes are directly configurable.
2. Creation of an `NdbCluster` object using the desired configuration in Kubernetes.
3. Verifying the installation by checking the NDB Cluster's status and (if necessary) logs.

Removal of an installed NDB Cluster is also covered here.

3.1 Setting the NDB Cluster's Configuration

NDB Operator relies on a custom resource definition (CRD) named `NdbCluster` to obtain the MySQL Cluster configuration data that it needs to start. Whenever a user creates, modifies, or deletes a Kubernetes object of type `NdbCluster`, NDB Operator receives a change event, and updates the NDB Cluster running in Kubernetes Cluster accordingly. (See Chapter 1, *Introduction to NDB Operator*, for a description of this mechanism.)

The `NdbCluster` CRD defines a Kubernetes resource type which can be used to specify the configuration of an NDB Cluster. See Section 5.1, “NdbCluster Resource”, for more information.

The `docs/examples` directory in the NDB Operator source tree contains several examples, including `example-ndb.yaml`. This file contains an `NdbCluster` specification having the characteristics shown here, specified using YAML format:

```
apiVersion: mysql.oracle.com/v1
kind: NdbCluster
metadata:
  name: example-ndb
spec:
  redundancyLevel: 2
  dataNode:
    nodeCount: 2
  mysqlNode:
    nodeCount: 2
```

`spec.dataNode.nodeCount` sets the number of data nodes.

`spec.redundancyLevel` specifies the number of replicas as well as the number of management nodes (`ndb_mgmd` processes). Since this is greater than 1, the NDB Cluster is created with two management nodes.

Note

The number of management nodes is not directly configurable; it is determined solely by the value of `redundancyLevel`.

`spec.dataNode.nodeCount` determines the number of data nodes in the NDB Cluster.

`spec.mysqld.nodeCount` determines the number of MySQL Servers attached to the NDB Cluster as SQL nodes, providing an SQL front end to the NDB Cluster data nodes.

3.2 Creating an NdbCluster Object

To create an `NdbCluster` object in the Kubernetes Cluster, issue the following command:

```
> kubectl apply -f docs/examples/example-ndb.yaml
ndbcluster.mysql.oracle.com/example-ndb created
```

Once this object has been created in the Kubernetes Cluster, the NDB Operator—which watches for changes to any `NdbCluster` object—detects the action and begins to set up the NDB Cluster inside the Kubernetes Cluster.

NDB Cluster nodes are run inside multiple pods, which can be viewed by issuing the command shown here:

```
> kubectl get pods -w
NAME                                     READY   STATUS    RESTARTS   AGE
example-ndb-mgmd-0                       1/1     Running   0          8m44s
example-ndb-mgmd-1                       1/1     Running   0          48s
example-ndb-mysqld-599bcfbd45-qzrsr      0/1     Running   0          7s
example-ndb-mysqld-599bcfbd45-r7g2x      0/1     Running   0          7s
example-ndb-ndbd-0                       1/1     Running   0          8m44s
example-ndb-ndbd-1                       1/1     Running   0          8m44s
ndb-operator-555b7b65-2rssd              1/1     Running   0          48m
ndb-operator-webhook-server-d67c97d54-m5d42   1/1     Running   0          48m
```

The pods whose names begin with `example-ndb-` are those running NDB Cluster nodes. The type of node can be seen in each name.

The NDB Cluster is ready for transactions once all pods are ready.

3.3 Checking NDB Cluster Status and Logs

The status of a Kubernetes `NdbCluster` object can be seen by running the command shown here:

```
> ndb-operator(main)$ kubectl get ndbcluster example-ndb
NAME          REDUNDANCY LEVEL
example-ndb   2
```

You can view NDB Operator's logs using the following command, which includes the name of the NDB Operator pod obtained using `kubectl get pods` previously:

```
> kubectl logs -f ndb-operator-555b7b65-2rssd
```

3.4 Removing NDB Cluster

To stop and delete NDB Cluster pods from Kubernetes, run the following command:

```
> kubectl delete -f docs/examples/example-ndb.yaml
ndbcluster.mysql.oracle.com/example-ndb deleted
```

Alternatively, you can use the name, like this:

```
> kubectl delete ndbcluster example-ndb
```

Either of these commands stops all NDB Cluster nodes and removes all related pods from the Kubernetes cluster. Any data is lost unless the NDB Cluster was deployed using a persistent volume (see `dataNodePVCSpec` in Section 5.5, “NdbClusterSpec Resource”, for more information about this).

Chapter 4 Performing Common Tasks with NDB Operator

Table of Contents

This chapter provides information about performing some common tasks using NDB Operator.

4.1 Accessing the NDB Cluster

To provide access to the NDB Cluster by applications, NDB Operator creates two load balancer services on top of the pods running NDB Cluster nodes. These services are listed here:

- Management server load balancer (*`ndb_cluster_name`*`-mgmd-ext`): Provides access to the NDB Cluster management servers
- MySQL server (SQL node) load balancer (*`ndb_cluster_name`*`-mysqld-ext`): Provides access to the NDB Cluster SQL nodes

An application running either inside or outside the Kubernetes cluster can make use of one or both of these services to connect to an NDB Cluster.

Each MySQL server is set up with a root account and a random password. The password is base-64 encoded and stored in a Kubernetes secret whose name has the format *`ndb_cluster_name`*`-mysqld-root-password`. The password can be retrieved using a command like the one shown here:

```
> base64 -d <<< \
  $(kubectl get secret example-ndb-mysqld-root-password \
  -o jsonpath={.data.password})
```

You can also set a custom password. Create a Kubernetes secret containing the password.

After this, set the name of the secret to the value of the `rootPasswordSecretName` field of the `mysqld` spec (see Section 5.9, “NdbMysqldSpec Resource”).

4.1.1 Access From Within Kubernetes

An application running inside the Kubernetes Cluster can use these extracted information and can access the data in MySQL Cluster without any additional setup.

NDB Cluster tools such as `ndb_mgm` can use the management node service name as a connection string to connect to the management servers. MySQL clients such as `mysql` can use the `mysqld-ext` service and the password stored in `rootPasswordSecretName` to connect to the SQL nodes.

It is first necessary to log in to an NDB Cluster pod using `kubectl`, like this:

```
> kubectl exec -it example-ndb-mysqld-599bcfbd45-hq8ll
```

Now you can execute `ndb_mgm` to start the NDB management client, using the mnagement node service name as the connection string:

```
> ndb_mgm -c example-ndb-mgmd-ext
```

From within `ndb_mgm` you can issue all of the usual management client commands, as shown here:

```
-- NDB Cluster -- Management Client --
ndb_mgm> SHOW
Connected to Management Server at: example-ndb-mgmd-ext:1186
Cluster Configuration
---------------------
[ndbd(NDB)] 2 node(s)
id=3    @172.17.0.6  (mysql-9.2.0 ndb-9.2.0, Nodegroup: 0, *)
id=4    @172.17.0.7  (mysql-9.2.0 ndb-9.2.0, Nodegroup: 0)

[ndb_mgmd(MGM)] 2 node(s)
id=1    @172.17.0.5  (mysql-9.2.0 ndb-9.2.0)
id=2    @172.17.0.8  (mysql-9.2.0 ndb-9.2.0)

[mysqld(API)]   5 node(s)
id=145  @172.17.0.10  (mysql-9.2.0 ndb-9.2.0)
id=146  @172.17.0.9  (mysql-9.2.0 ndb-9.2.0)
id=147 (not connected, accepting connect from any host)
id=148 (not connected, accepting connect from any host)
id=149 (not connected, accepting connect from any host)

ndb_mgm> ALL STATUS
Node 5: started (mysql-9.2.0 ndb-9.2.0)
Node 6: started (mysql-9.2.0 ndb-9.2.0)

ndb_mgm> EXIT
```

To connect to MySQL with the `mysql` client, use the service name `example-ndb-mysqld-ext` for the host name, like this:

```
> mysql -h example-ndb-mysqld-ext -uroot -p

Enter password: ********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 500
Server version: 9.2.0-cluster MySQL Cluster Community Server - GPL

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> SELECT * FROM INFORMATION_SCHEMA.ENGINES
    -> WHERE ENGINE LIKE "ndb%"\G
*************************** 1. row ***************************
      ENGINE: ndbinfo
     SUPPORT: YES
     COMMENT: MySQL Cluster system information storage engine
TRANSACTIONS: NO
          XA: NO
  SAVEPOINTS: NO
*************************** 2. row ***************************
      ENGINE: ndbcluster
     SUPPORT: YES
     COMMENT: Clustered, fault-tolerant tables
TRANSACTIONS: YES
          XA: NO
  SAVEPOINTS: NO
2 rows in set (0.00 sec)
```

4.1.2 Access From Outside Kubernetes

NDB management and SQL applications running outside the Kubernetes cluster can access the NDB Cluster running inside using the management server and SQL node services described previously; these services are of type `LoadBalancer`. A Kubernetes cluster provider such as `minikube` provisions load balancers for these services. To expose the services outside the Kubernetes cluster if you are running `minikube`, execute the command shown here:

```
> minikube tunnel
```

Using `kubectl get service`, and passing it the service name or label, you can retrieve the IP address needed by applications to connect to the NDB Cluster. For the example management node service, this can be done using either of the commands shown here:

```
# Retrieve management load balancer service IP address using service name
> kubectl get service "example-ndb-mgmd-ext" \
      -o jsonpath={.status.loadBalancer.ingress[0].ip}

# Retrieve management load balancer service IP address using service label
> kubectl get service \
      -l "mysql.oracle.com/resource-type=mgmd-service-ext" \
      -o jsonpath={.items[0].status.loadBalancer.ingress[0].ip}
```

With the service IP address just extracted, NDB management clients like `ndb_mgm` can connect to the NDB Cluster management servers by way of the management node service, and perform management tasks.

Similarly, you can obtain the IP address for the example SQL node load balancer service using either of these `kubectl get service` commands:

```
# Retrieve SQL node load balancer service IP address using service name
> kubectl get service "example-ndb-mysqld-ext" \
      -o jsonpath={.status.loadBalancer.ingress[0].ip}

# Retrieve SQL node load balancer service IP address using service name
> kubectl get service \
      -l "mysql.oracle.com/resource-type=mysqld-service-ext" \
      -o jsonpath={.items[0].status.loadBalancer.ingress[0].ip}
```

With the IP address obtained by either of these two commands, MySQL clients such as `mysql` can connect to the NDB Cluster SQL nodes and execute SQL statements.

4.2 Updating the NDB Cluster Configuration

Making changes in the configuration of an NDB Cluster running in Kubernetes, requires editing the `NdbCluster` YAML spec file, then applying the updated file to the Kubernetes Cluster.

Consider the example `example-ndb`, installed from the file `docs/examples/example-ndb.yaml` (see Section 3.1, “Setting the NDB Cluster's Configuration”), which provides two SQL nodes. To increase this number to five, update `spec.mysqld.nodeCount` in this file to the preferred value. The updated file should look like what is shown here (with the edited line in highlighted text):

```
apiVersion: mysql.oracle.com/v1
kind: NdbCluster
metadata:
  name: example-ndb
spec:
  redundancyLevel: 2
  dataNode:
    nodeCount: 2
  mysqlNode:
```

```
    nodeCount: 5
```

Now you can apply the updated YAML file to the Kubernetes Cluster, like this:

```
> kubectl apply -f docs/examples/example-ndb.yaml
ndbclusters.mysql.oracle.com/example-ndb configured
```

Once the change has been applied, NDB Operator picks up the changes and begins applying them to the NDB Cluster. In this particular case, it updates the management node configuration files, performs a rolling restart (see Performing a Rolling Restart of an NDB Cluster), and starts additional `mysqld` processes (SQL nodes).

The status and readiness of the NDB Cluster nodes can be observed as when they were first deployed.

4.3 Using NdbClusterPodSpec

The NDB Cluster Custom Resource Definition provides the `NdbClusterPodSpec` structure for defining specifications of pods for individual management, data, and SQL nodes using their respective `.spec.managementNode.ndbPodSpec`, `.spec.dataNode.ndbPodSpec`, and `.spec.mysqlNode.ndbPodSpec` fields. Values set for these `NdbPodSpec` fields are copied into their respective `StatefulSet` definitions. This chapter explains how these `NdbClusterPodSpec` fields can be used for assigning pods to specific worker nodes, defining affinity rules, and specifying pod resource requirements.

Assigning NDB node pods to worker pods. You can specify the label of the worker node on which a given NDB Cluster node should be scheduled by specifying it in the `nodeSelector` field.

See Assigning Pods to Nodes for more information.

Affinity and anti-affinity. Affinity and anti-affinity rules for NDB Cluster nodes can be defined using the `affinity` field. The Kubernetes Cluster uses these rules to decide where to schedule an NDB Cluster pod. These rules are applied after filtering out the available worker node pool based on any specified `nodeSelector` labels.

NDB Operator defines default anti-affinity rules for each of the three MySQL Cluster node types (`ndb_mgmd`, `ndbmtd`, and `mysqld`) to prevent them from being scheduled on the same worker node whenever possible. Such a rule is always defined as `preferredDuringSchedulingIgnoredDuringExecution`, so that it is satisfied by the Kubernetes scheduler only if there are sufficient resources. For example, if four data nodes must be scheduled on four worker nodes, each data node is scheduled on a separate worker node, but if six data nodes must be scheduled on four worker nodes, the first four data nodes are scheduled on four separate worker nodes while the fifth and sixth data nodes must be scheduled on worker nodes where a data node is already running. Default anti-affinity rules can be overridden by specifying the desired anti-affinity rules in the `affinity` field.

For more information, see Affinity and anti-affinity.

Specify Resource Requirements. Pod resource requirements can be specified using the resources field. These requirements are copied into the container definitions.

NDB Operator defines default memory requirements for data nodes based on the configuration of the NDB Cluster. The minimum calculated by NDB Operator for this purpose is an estimate, and can be overridden by specifying an alternative using the data node's `ndbPodSpec`.

For more information, see Resource Management for Pods and Containers.

Chapter 5 NDB Operator CRD Reference

Table of Contents

This chapter provides information about the Custom Resource Definitions (CRDs) used by NDB Operator 9.2.

5.1 NdbCluster Resource

`NdbCluster` is the Schema for the `Ndb` CRD API. It contains the fields listed and described here:

- `apiVersion` (string): This is always `mysql.oracle.com/v1`.
- `kind` (string): This is always `NdbCluster`.
- `spec` (`NdbClusterSpec`): Specifies the desired state of the NDB Cluster.
- `status` (`NdbClusterStatus`): Contains status information regarding the `Ndb` resource and the NDB Cluster managed by it.

5.2 NdbClusterCondition Resource

`NdbClusterCondition` describes the state of an NDB Cluster installation at a given point in time. Used by `NdbClusterStatus`.

- `type` (`NdbClusterConditionType`): Type of `NdbCluster` condition.
- `status` (`Kubernetes core/v1.ConditionStatus`): Status of the condition: one of `True`, `False`, or `Unknown`.
- `lastTransitionTime` (`Kubernetes meta/v1.Time`): Last time the condition transitioned from one status to another.
- `reason` (string): The reason for the condition's last transition.
- `message` (string): A human-readable message indicating details about the transition.

5.3 NdbClusterConditionType

`NdbClusterConditionType` defines the `type` of an `NdbClusterCondition`.

If set, this is a constant `NdbClusterUpToDate`, whose value is the string “UpToDate”.

5.4 NdbClusterPodSpec Resource

`NdbClusterPodSpec` contains a subset of `PodSpec` fields which, when set, are copied into to the `podSpec` of the relevant MySQL Cluster node workload definitions. Used by `NdbDataNodeSpec`, `NdbManagementNodeSpec`, and `NdbMysqldSpec`.

- `resources` (core/v1.ResourceRequirements): (*optional*) Total compute resources required by this pod. Cannot be changed.
- `nodeSelector` (map[string]string): (*optional*) A selector which must be true for the pod to fit on a node; that is, a selector which must match a node's labels for the pod to be scheduled on that node.

  For more information, see nodeSelector.
- `affinity` (Kubernetes core/v1.Affinity): (*optional*) If specified, these are the pod's scheduling constraints.
- `schedulerName` (string): (*optional*) If given, the pod is dispatched by the specified scheduler; otherwise, the pod is dispatched by the default scheduler.
- `tolerations[]` (core/v1.Toleration): (*optional*) If specified, the pod's tolerations.

5.5 NdbClusterSpec Resource

`NdbClusterSpec` defines the desired state of an NDB Cluster. It is used by `NdbCluster`.

`NdbClusterSpec` contains the fields named and described in the following list:

- `redundancyLevel` (integer): The number of data replicas or copies of data stored by NDB Cluster. Supported values are 1, 2, 3, and 4.

  A redundancy level of 1 provides no fault tolerance in case of node failure, and is not recommended. With a redundancy level of 2 or higher, the cluster can continue to serve client requests even in the event of node failures; this is the default value, recommended for most deployments. A redundancy level of 3 (or 4) provides additional protection, but is usually not necessary.

  One management server is created when the redundancy level is set to 1. For a value of 2 or higher, two management servers are used.

  Important

  Once an NDB Cluster has been created, it is possible but quite difficult and time-consuming to change this value. Unless you are sure that you need a different value for the redundancy level, it is recommended that you use the default (2).

  For more information, see the description of the `NoOfReplicas` data node configuration parameter, as well as NDB Cluster Nodes, Node Groups, Fragment Replicas, and Partitions, in the NDB 9.2 documentation.
- `managementNode` (`NdbManagementNodeSpec`): (*optional*) Specifies the configuration of the management node running in MySQL Cluster.
- `dataNode` (string): (*optional*) Configuration parameters to pass to the data nodes. Consists of one or more lines using the format *`paramName: paramValue`*.

  Example:

  ```
  spec:
  ```

```
...
  dataNode:
    # Specified in paramName: paramValue format
    DataMemory: 100M
    MaxNoOfTables: 1024
    MaxNoOfConcurrentOperations: 409600
    Arbitration: WaitExternal
```

NDB Cluster Data Node Configuration Parameters, provides a quick reference to NDB Cluster data node configuration parameters.

- `mysqlNode` (`NdbMysqldSpec`): (*optional*) Specification for any MySQL Servers run as NDB Cluster SQL nodes.

  NDB Operator requires at least one MySQL Server running in the NDB Cluster for internal operations. If none is specified, the operator by default adds one MySQL Server to the specification.

- `freeAPISlots` (integer): (*optional*) The number of extra API sections declared in the NDB Cluster configuration in addition to any declared implicitly by NDB Operator for MySQL servers. Any NDB API application can connect to the NDB Cluster using one of these free slots.

- `tdeSecretName` (string): (*optional*) The name of the secret that holds the encryption key or password required for transparent data encryption (TDE) in NDB Cluster. If a value is provided, NDB Operator enables TDE and uses the password stored in the secret as the file system password for all data nodes in the cluster. If no value is provided, TDE is not enabled.

- `image` (string): (*optional*) The name of the MySQL NDB Cluster image to be used. If not specified, this defaults to `mysql/mysql-cluster:latest`.

  Important

  The minimum version of NDB Cluster supported by NDB Operator is NDB 8.0.26.

- `imagePullPolicy` (Kubernetes core/v1.PullPolicy): (*optional*) Describes a policy for if and when to pull the MySQL NDB Cluster container image.

- `imagePullSecretName` (string): (*optional*) Specifies the name of the secret that holds the credentials required for pulling the MySQL Cluster image.

5.6 NdbClusterStatus Resource

`NdbClusterStatus` represents the status of an `Ndb` resource. Used by `NdbCluster`.

- `processedGeneration` (integer): Holds the latest generation of the `Ndb` resource whose specs have been successfully applied to the NDB Cluster running inside Kubernetes.

- `readyManagementNodes` (string): The status of the NDB Cluster management nodes.

- `readyDataNodes` (string): The status of the NDB Cluster data nodes.

- `readyMySQLServers` (string): The status of the MySQL servers attached to the NDB Cluster as SQL nodes.

- `conditions` (`[]NdbClusterCondition`): The latest available observations of the NDB Cluster's current state.

- `generatedRootPasswordSecretName` (string): This is the name of the secret generated by the operator, used as the MySQL server root account password. This will be set to null if a secret has already been provided to the operator using `spec.mysqlNode.rootPasswordSecretName`.

5.7 NdbDataNodeSpec Resource

`NdbDataNodeSpec` specifies a data node in an NDB Cluster. Used by `NdbClusterSpec`.

- `config` (map[string]* Kubernetes util/intstr.IntOrStringConfig): (*optional*) A map of default NDB data node configuration parameters (see NDB Cluster Data Node Configuration Parameters).
- `ndbPodSpec` (`NdbClusterPodSpec`): (*optional*) A subset of `PodSpec` fields which, when set, are copied into to the `podSpec` of the data node's `StatefulSet` definition.
- `nodeCount` (integer): The total number of data nodes in a MySQL NDB Cluster; this must be an integer multiple of `redundancyLevel`. A maximum of 144 data nodes is supported.
- `pvcSpec` (Kubernetes core/v1.PersistentVolumeClaimSpec): (*optional*) The `PersistentVolumeClaimSpec` to be used as the `VolumeClaimTemplate` of the data node `StatefulSet`. A PVC is created for each data node by the `StatefulSet` controller and is loaded into the data node pod and the container.

5.8 NdbManagementNodeSpec Resource

`NdbManagementNodeSpec` specifies a management server node in an NDB Cluster. Used by `NdbClusterSpec`.

- `config` (map[string]* Kubernetes util/intstr.IntOrStringConfig): (*optional*) A map of default MySQL Cluster management node configuration parameters (see NDB Cluster Management Node Configuration Parameters).
- `ndbPodSpec` (NdbClusterPodSpec): (*optional*) A subset of `PodSpec` fields which, when set, are copied into to the `podSpec` of the management node's `StatefulSet` definition.
- `enableLoadBalancer` (bool): (*optional*) Exposes the management servers externally using the Kubernetes cloud provider's load balancer. By default, the operator creates a service of type `ClusterIP` to expose the management server pods internally within the Kubernetes cluster. If `enableLoadBalancer` is set to `true`, a service of type `LoadBalancer` is created instead, exposing the management servers outside the Kubernetes cluster.

5.9 NdbMysqldSpec Resource

`NdbMysqldSpec` is the specification for any MySQL Servers to be run as NDB Cluster SQL nodes. Used by `NdbClusterSpec`.

- `nodeCount` (integer): The number of SQL nodes (that is, MySQL servers or instances of `mysqld`) running in the NDB Cluster.
- `maxNodeCount` (integer): (*optional*) The NDB Cluster's MySQL servers scale up to this number without forcing a configuration update. If this is unspecified, NDB Operator includes API sections for additional MySQL Servers in the configuration file.
- `connectionPoolSize` (integer): (*optional*) This is the number of connections a single MySQL Server should use to connect to the MySQL Cluster data nodes. See the description of the `mysqld --ndb-cluster-connection-pool` option for further information.
- `rootPasswordSecretName` (string): (*optional*) The name of the secret that holds the password for the MySQL root accounts. The secret should contain a `password` key that holds the password. If unspecified, a secret is created by the operator with a generated name in the format *`ndb_resource_name`*`-mysqld-root-password`.

- `rootHost` (string): (*optional*) Names the host or hosts from which the root user can connect to the MySQL server. If unspecified, the root user can connect from any host that can access the MySQL server.

- `myCnf` (string): (*optional*) `mysqld` configuration options to pass to the SQL nodes when they are started.

  Example:

  ```
  myCnf: |
    [mysqld]
    max-user-connections=42
    ndb-extra-logging=10
  ```

  The format used for the configuration string is similar to that used in a MySQL `my.cnf` file. See Option File Syntax, for more information. MySQL Server Options and Variables for NDB Cluster, provides a reference of MySQL Server configuration options specific to NDB Cluster.

- `enableLoadBalancer` (`bool`): (*optional*) Exposes the MySQL servers externally using the Kubernetes cloud provider's load balancer. By default, the operator creates a ClusterIP type service to expose the MySQL server pods internally within the Kubernetes cluster. If `EnableLoadBalancer` is set to true, a `LoadBalancer` service is created instead, exposing the MySQL servers outside the Kubernetes cluster.

- `ndbPodSpec` (`NdbClusterPodSpec`): (*optional*) A subset of `PodSpec` fields which, when set, are copied into to the podSpec of the MySQL server's `StatefulSet`.

- `initScripts` (`map[string][]string`): (*optional*) A map of `configMap` names from the same namespace, and, optionally, an array of keys which store the SQL scripts to be executed during MySQL server initialization. If key names are omitted, contents of all the keys are treated as initialization SQL scripts. All scripts are mounted into the MySQL pods and executed by `configMap` name and key name, in alphabetical order.

- `pvcSpec` (`Kubernetes core/v1.PersistentVolumeClaimSpec`:

  (*optional*) The `PersistentVolumeClaimSpec` used as the `VolumeClaimTemplate` of the MySQL server `StatefulSet`. A PVC is created for each MySQL server by the `StatefulSet` controller and loaded into the MySQL server pod and the container.

See also Defining SQL and Other API Nodes in an NDB Cluster.

Chapter 6 Contributing to NDB Operator

Table of Contents

We appreciate any feedback we receive from our users. The most useful forms of feedback are bug reports, enhancements requests, and contributions of code.

The rest of this chapter contains information about filing bug reports and enhancement requests, as well as how to submit code that you wish to contribute.

6.1 Reporting Issues and Requesting Enhancements

To report issues with NDB Operator, please file a bug report in the MySQL Bug System, using the category *MySQL Cluster: NDB Operator*.

Bug reports should provide as much information as possible, including the following:

- Complete steps to reproduce the issue
- Any information about the Kubernetes environment that may be specific to the bug
- Specific version of NDB Operator
- Specific version of the NDB Cluster being used
- Sample code to help reproduce the issue, if possible.

You can also request enhancements to NDB Operator using the MySQL Bug System; these should also be filed under *MySQL Cluster: NDB Operator*.

Before filing bug reports or enhancement requests, please make sure that you are not submitting a duplicate of an issue which has already been filed. You can view existing issues here.

To report issues with or request enhancements to this documentation, please file a report in the MySQL Bug System using the category *MySQL Cluster: Documentation*.

6.2 Contributing Code

We are happy to consider your contributions of code. For you to contribute, it is necessary that you complete the following steps:

- Sign the Oracle Contributor Agreement
- Develop your pull request
- Validate your pull request by including tests that sufficiently cover the functionality you are adding
- Verify that the entire test suite passes with your code applied
- Submit your pull request. You can do so using GitHub, or directly to MySQL at Oracle using bugs.mysql.com