VoltDB Kubernetes Administrator's Guide

Abstract

This book explains how to create and manage VoltDB database clusters using Kubernetes.

V11.4

VoltDB Kubernetes Administrator's Guide

V11.4

This document was generated on September 27, 2022.

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

This book describes using Kubernetes and associated products to create and manage VoltDB databases and the clusters that host them. It is intended for database administrators and operators responsible for the ongoing management and maintenance of database infrastructure in a containerized environment.

This book is *not* a tutorial on Kubernetes or VoltDB. Please see “Related Documents” below for documents that can help you familiarize yourself with these topics.

1. Structure of This Book

This book is divided into 7 chapters and 1 appendix:

- Chapter 1, *Introduction*
- Chapter 2, *Configuring the VoltDB Database Cluster*
- Chapter 3, *Managing VoltDB Databases in Kubernetes*
- Chapter 4, *Starting and Stopping the Database*
- Chapter 5, *Updates and Upgrades*
- Chapter 6, *Monitoring VoltDB Databases in Kubernetes*
- Chapter 7, *Cross Datacenter Replication in Kubernetes*
- Chapter 8, *Managing XDCR Clusters in Kubernetes*
- Appendix A, *VoltDB Helm Properties*

2. Related Documents

This book assumes a working knowledge of Kubernetes, VoltDB, and the other technologies used in a containerized environment (specifically Docker and Helm). For information on developing and managing VoltDB databases, please see the manuals *Using VoltDB* and *VoltDB Administrator's Guide*. For new users, see the *VoltDB Tutorial*. For introductory information on the other products, please see their respective websites for appropriate documentation:

- Docker
- Helm
- Kubernetes

Finally, this book and all other documentation associated with VoltDB can be found on the web at http://docs.voltactivedata.com/.

Chapter 1. Introduction

Kubernetes is an environment for hosting virtualized applications and services run in containers. It is designed to automate the management of distributed applications, with a particular focus on microservices. VoltDB is not a microservice — there is coordination between the nodes of a VoltDB cluster that requires additional attention. So although it is possible to spin up a generic set of Kubernetes "pods" to run a VoltDB database, additional infrastructure is necessary to realize the full potential of Kubernetes and VoltDB working together.

VoltDB Enterprise Edition provides additional services to simplify, automate, and unlock the power of running VoltDB within Kubernetes environments. There are six key components to the VoltDB Kubernetes offering, three available as open-source applications for establishing the necessary hosting environment and three provided by VoltDB to Enterprise customers. The three open-source products required to run VoltDB in a Kubernetes environment are:

- **Kubernetes** itself
- **Docker**, for managing the container images
- **Helm**, for automating the creation and administration of VoltDB in Kubernetes

In addition to these base requirements, VoltDB provides the following three custom components:

- **Pre-packaged docker image** for running VoltDB cluster nodes
- **The VoltDB Operator**, a separate utility (and docker image) for orchestrating the startup and management of VoltDB clusters in Kubernetes
- **Helm charts** for initializing and communicating with Kubernetes, the VoltDB Operator and its associated VoltDB cluster

The remainder of this chapter provides an overview of how these components work together to support running virtualized VoltDB clusters in a Kubernetes environment, the requirements for the host and client systems, and instructions for preparing the host environment prior to running VoltDB. Subsequent chapters provide details on configuring and starting your VoltDB cluster as well as common administrative tasks such as:

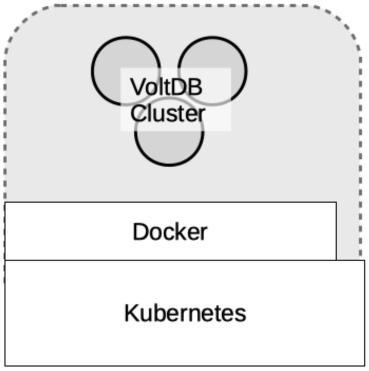
- Starting, stopping, and modifying the VoltDB cluster
- Managing the database schema and configuration
- Configuring and starting multiple clusters using cross datacenter replication (XDCR)

Finally, an appendix provides a full list of the Helm properties for configuring and controlling your VoltDB clusters.

1.1. Overview: Running VoltDB in Kubernetes

Kubernetes lets you create clusters of virtual machines, on which you run "pods". Each pod acts as a separate virtualized system or container. The containers are pre-defined collections of system and application components needed to run an application or service. Kubernetes provides the virtual machines, Docker defines the containers, and Kubernetes takes responsibility for starting and stopping the appropriate number of pods that your application needs.

So the basic architecture for running VoltDB is a VoltDB database running on multiple instances of a Docker container inside a Kubernetes cluster.

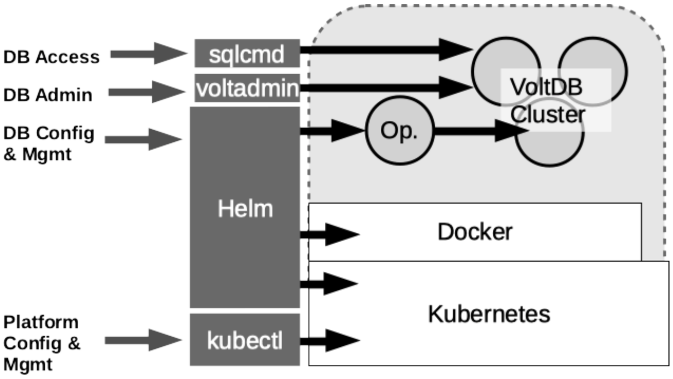


However, out of the box, VoltDB and Kubernetes do not "talk together" and so there is no agreement on when pods are started and stopped and whether a VoltDB node is active or not. To solve this problem, VoltDB provides an additional service, the VoltDB Operator that manages the interactions between the VoltDB cluster and the Kubernetes infrastructure. The Operator takes responsibility for initializing and starting the VoltDB server instances as appropriate, monitoring their health, and coordinating changes to the configuration.

To further simplify the process, VoltDB uses the open-source management product Helm to integrate Kubernetes, Docker, and VoltDB under a single interface. Helm uses "charts" to define complex management operations, such as configuring and starting the Kubernetes pods with the appropriate Docker images and then initializing and starting VoltDB on those pods. Simply by "installing" the appropriate Helm chart you can instantiate and run a VoltDB database cluster within Kubernetes using a single command.

Once the database is running, you can use standard VoltDB command line utilities to interact with and manage the database contents, such as modifying the schema or initiating manual snapshots. However, you will continue to use Helm to manage the server process and cluster on which the database runs, for activities such as stopping and starting the database. Figure 1.1, “Kubernetes/VoltDB Architecture” shows the overall architecture of using VoltDB, the VoltDB Operator, and Helm to automate running a VoltDB database within Kubernetes.

Figure 1.1. Kubernetes/VoltDB Architecture



1.2. Setting Up Your Kubernetes Environment

Before you can run VoltDB in a containerized environment, you must be sure your host systems and client are configured with the right software and permissions to support VoltDB. The following sections outline:

- What products are required on both the host environment and the local client you use to control Kubernetes and VoltDB
- How to configure the host environment and user accounts to run the VoltDB components
- How to configure your local client to control Kubernetes and the Helm charts
- How to set permissions in Kubernetes and Docker to allow access to the VoltDB components

1.2.1. Product Requirements

Before you start, you must make sure you have the correct software products and versions installed on both the host system and your local client. The host environment is the set of servers where Kubernetes is installed, whether they are systems you set up yourself or hosted by a third-party cloud service, such as the Google Cloud Platform or Microsoft Azure. The local client environment is the system, such as a desktop or laptop, you use to access the services.

The following are the software requirements for running VoltDB in Kubernetes.

Host Environment

- Kubernetes V1.19.x through V1.23.x

Client Environment

- Kubectl V1.19 or later[1]
- Helm V3.6.x or later

Optionally, you may want to install VoltDB on the client so you can use the **voltadmin** and **sqlcmd** command utilities to access the database remotely. If not, you can still use **kubectl** to create an interactive shell process on one of the server instances and run the utilities directly on the Kubernetes pods.

1.2.2. Configuring the Host Environment and Accounts

Once you have the necessary software installed, you must prepare the host environment to run VoltDB. This includes adding the appropriate Docker and chart repositories to Helm and configuring your host account with the permissions necessary to access those repositories.

First, you need accounts on the Kubernetes host environment and on the docker repository where the VoltDB images are stored, https://docker.io. To run the VoltDB Helm charts, your accounts must be set up with the following permissions:

- **Your Kubernetes host account** must have sufficient permissions to allocate persistent volumes and claims and create and manage pods.
- **Your Docker repository account** must have permission to access the VoltDB docker images. Access to the VoltDB docker images is assigned to VoltDB Enterprise customers on a per account basis. Contact VoltDB support for more information.

1.2.3. Configuring the Client

Next you must configure your client environment so you can communicate with and control Kubernetes and the Helm charts. First, install the Kubernetes and Helm command line interfaces, **kubectl** and **helm**. Next, configure the services to access the appropriate remote accounts and repositories.

[1] Kubectl on the client must be within one minor version of Kubernetes in the host environment. For example, if Kubernetes is at version 1.17, Kubectl can be 1.16, 1.17, or 1.18. See the Kubernetes version skew documentation for further information.

The primary setup task for kubectl is creating the appropriate context for accessing the Kubernetes host you will be using. This is usually done as part of the installation or with a Kubconfig file and the **kubectl config** command. Once you have a context defined, you can use the **kubectl cluster-info** command to verify that your client is configured correctly.

For helm, you must add a link to the VoltDB docker repository, using the **helm repo add** command:

```
$ helm repo add voltdb \
     https://voltdb-kubernetes-charts.storage.googleapis.com
```

The first argument to the command ("voltdb") is a short name for referencing the repository in future commands. You can specify whatever name you like. The second argument is the location of the repository itself and must be entered as shown above.

Note

Helm first looks in local folders for charts you specify, then in the repositories. So if the short name you use matches a local directory, they van conflict and cause errors. In that case, you may want to choose a different name, such as "voltkube", to avoid any ambiguity. Then the chart locations you use in Helm commands would be "voltkube/voltdb" rather than "voltdb/voltdb" as shown in the examples.

1.2.4. Granting Kubernetes Access to the Docker Repository

Finally, you need to tell Kubernetes to access the Docker repository using the credentials for your Docker account. There are several ways to do this. You can specify your credentials on the helm command line each time you install a new VoltDB cluster. You can save the credentials in a YAML file with other parameters you pass to helm. Or you can set the credentials in a Kuerbernetes secret using kubectl.

The advantage of using a secret to store the credentials is that you only need to define them once and they are not easily discovered by others, since they are encrypted. To create a Kubernetes secret you use the **kubectl create secret** command, specifying the type of secret (*docker-registry*) and the name of the secret (which must be *dockerio-registry*), plus the individual credential elements as arguments:

```
$ kubectl create secret docker-registry dockerio-registry \
   --docker-username=johndoe          \
   --docker-password='ThisIsASecret'  \
   --docker-email="jdoe@anybody.org
```

Once you add the secret, you do not need to specify them again. If, on the other hand, you prefer to specify the credentials when you issue the helm commands to initialize the VoltDB cluster, you can supply them as the following helm properties using the methods described in Chapter 2, *Configuring the VoltDB Database Cluster*:

- `global.image.credentials.username`
- `global.image.credentials.password`

Chapter 2. Configuring the VoltDB Database Cluster

Helm simplifies the process of starting a VoltDB database cluster within Kubernetes by coordinating all the different components involved, including Kubernetes, Docker, and VoltDB. By using the provided Helm charts, it is possible to start a default VoltDB cluster with a single command:

```
$ helm install mydb voltdb/voltdb \
   --set-file cluster.config.licenseXMLFile=license.xml
```

The name *mydb* specifies a name for the release you create, *voltdb/voltdb* specifies the Helm chart to install, and the `--set-file` argument specifies a new value for a property to customize the installation. In this case, `--set-file` specifies the location of the VoltDB license needed to start the database. The license is the only property you must specify; all other properties have default values that are used if not explicitly changed.

However, a default cluster of three nodes and no schema or configuration is not particularly useful. So VoltDB provides Helm properties to let you customize every aspect of the database and cluster configuration, including:

- Cluster configuration, including size of the cluster, available resources, and so on
- Network configuration, including the assignment of ports and external mappings
- Database initialization options, including administration username and password, schema, and class files
- Database configuration, including the settings normally found in the XML configuration file on non-Kubernetes installations

The following sections explain how to use those properties to make some of the most common customizations to your database. Appendix A, *VoltDB Helm Properties* provides a full list of the properties, including a brief description and the default value for each.

2.1. Using Helm Properties

First, it is useful to understand the different ways you can specify properties on the Helm command line. The following discussion is not intended as a complete description of Helm; only a summary to give you an idea of what they do and when to use them.

Helm offers three different ways to specify properties:

--set

The `--set` flag lets you specify individual property values on the command line. You can use `--set` multiple times or separate multiple property/value pairs with commas. For example, the following two commands are equivalent:

```
$ helm install mydb voltdb/voltdb \
   --set cluster.serviceSpec.clientPort=22222 \
   --set cluster.serviceSpec.adminPort=33333
$ helm install mydb voltdb/voltdb \
   --set cluster.serviceSpec.clientPort=22222,\
cluster.serviceSpec.adminPort=33333
```

The --set flag is useful for setting a few parameters that change frequently or for overriding parameters set earlier in the command line (such as in a YAML file).

--set-file

The `--set-file` flag lets you specify the contents of a file as the value for a property. For example, the following command sets the contents of the file `license.xml` as the license for starting the VoltDB cluster:

```
$ helm install mydb voltdb/voltdb \
   --set-file cluster.config.licenseXMLFile=license.xml
```

As with --set, You can use `--set-file` multiple times or separate multiple property/file pairs with commas. The `--set-file` flag is useful for setting parameters where the value is too complicated to set directly on the command line. For example, the contents of the VoltDB license file.

--values, -f

The --values flag lets you specify a file that contains multiple property definitions in YAML format. Whereas properties set on the command line with `--set` use dot notation to separate the property hierarchy, YAML puts each level of the hierarchy on a separate line, with indentation and followed by a colon. For example, the following YAML file (and `--values` flag set the same two properties show in the `--set` example above:

```
$ cat ports.yaml
cluster:
   serviceSpec:
      clientPort: 22222
      adminPort: 33333
$ helm install mydb voltdb/voltdb \
   --values ports.yaml
```

YAML files are extremely useful for setting multiple properties with values that do not change frequently. You can also use them to group properties (such as port settings or security) that work together to configure aspects of the database environment.

You can use any of the preceding techniques for specifying properties for the VoltDB Helm charts. In fact, you can use each method multiple times on the command line and mixed in any order. For example, the following example uses `--values` to set the database configuration and ports, `--set-file` to identify the license, and `--set` to specify the number of nodes requested:

```
$ helm install mydb voltdb/voltdb                       \
   --values dbconf.yaml,dbports.yaml                    \
   --set-file cluster.config.licenseXMLFile=license.xml \
   --set cluster.configSpec.replicas=5
```

2.2. Configuring the Cluster and Database

The two major differences between creating a VoltDB database cluster in Kubernetes and starting a cluster using traditional servers are:

- With Helm there is a single command (install) that performs both the initialization and the startup of the database.
- You specify the database configuration with properties rather than as an XML file.

In fact, all of the configuration — including the configuration of the virtual servers (or pods), the server processes, and the database — is accomplished using Helm properties. The following sections provide examples of some of the most common configuration settings when using Kubernetes. Appendix A, *VoltDB Helm Properties* gives a full list of all of the properties that are available for customization.

2.2.1. Configuring the Cluster

Many of the configuration options that are performed through hardware configuration, system commands or environment variables on traditional server platforms are now available through Helm properties. Most of these settings are listed in Section A.3, “Kubernetes Cluster Startup Options”.

Hardware Settings

Hardware settings, such as the number of processors and memory size, are defined as Kubernetes image resources through the Helm `cluster.clusterSpec.resources` property. Under `resources`, you can specify any of the YAML properties Kubernetes expects when configuring pods within a container. For example:

```
cluster:
  clusterSpec:
    resources:
      requests:
        cpu: 500m
        memory: 1000Mi
      limits:
        cpu: 500m
        memory: 1000Mi
```

System Settings

System settings that control process limits that are normally defined through environment variables can be set with the `cluster.clusterSpec.env` properties. For example, the following YAML increases the Java maximum heap size and disables the collection of JVM statistics:

```
cluster:
  clusterSpec:
    env:
      VOLTDB_HEAPMAX: 3072
      VOLTDB_OPTS: -XX:+PerfDisableSharedMem
```

One system setting that is *not* configurable through Kubernetes or Helm is whether the base platform has Transparent Huge Pages (THP) enabled or not. This is dependent of the memory management settings on the actual base hardware on which Kubernetes is hosted. Having THP enabled can cause problems with memory-intensive applications like VoltDB and it is strongly recommended that THP be disabled before starting your cluster. (See the section on Transparent Huge Pages in the *VoltDB Administrator's Guide* for an explanation of why this is an issue.)

If you are not managing the Kubernetes environment yourself or cannot get your provider to modify their environment, you will need to override VoltDB's warning about THP on startup by setting the cluster.clusterSpec.additionalArgs property to include the VoltDB start argument to disable the check for THP. For example:

```
cluster:
  clusterSpec:
    additionalStartArgs:
      - "--ignore=thp"
```

2.2.2. Configuring the Database

In addition to configuring the environment VoltDB runs in, there are many different characteristics of the database itself you can control. These include mapping network interfaces and ports, selecting and configuring database features, and identifying the database schema, class files, and security settings.

The network settings are defined through the `cluster.serviceSpec` properties, where you can choose the individual ports and choose whether to expose them through the networking service (`cluster.serviceSpec.type`) you can also select. For example, the following YAML file disables exposure of the admin port and assigns the externalized client port to 31313:

```
cluster:
  serviceSpec:
    type: NodePort
      adminPortEnabled: false
      clientPortEnabled: true
      clientNodePort: 31313
```

The majority of the database configuration options for VoltDB are traditionally defined in an XML configuration file. When using Kubernetes, these options are declared using YAML and Helm properties.

In general, the Helm properties follow the same structure as the XML configuration, beginninging with "cluster.config". So, for example, where the number of sites per host is defined in XML as :

```
<deployment>
   <cluster sitesperhost="{n}"/>
</deployment>
```

It is defined in Kubernetes as:

```
cluster:
  config:
    deployment:
      cluster:
        sitesperhost: {n}
```

The following sections give examples of defining common database configurations options using both XML and YAML. See Section A.6, “VoltDB Database Configuration Options” for a complete list of the Helm properties available for configuring the database.

2.2.2.1. Command Logging

Command logging provides durability of the database content across failures. You can control the level of durability as well as the length of time required to recover the database by configuring the type of command logging and size of the logs themselves. In Kubernetes this is done with the `cluster.config.deployment.commandlog` properties. The following examples show the equivalent configuration in both XML and YAML:

XML Configuration File	YAML Configuration File

`<commandlog enabled="true"`
`            synchronous="true"`
`            logsize="3072">`
`  <frequency time="300"`
`             transactions="1000"/>`
	`cluster:`
	`  config:`
	`    deployment:`
	`      commandlog:`
	`        enabled: true`

XML Configuration File	YAML Configuration File

`</commandlog>`	`        synchronous: true`
	`        logsize: 3072`
	`        frequency:`
	`          transactions 1000`

2.2.2.2. Export

Export simplifies the integration of the VoltDB database with external databases and systems. You use the export configuration to define external "targets" the database can write to. In Kubernetes you define export targets using the `cluster.config.deployment.export.configurations` property. Note that the `configurations` property can accept multiple configuration definitions. In YAML, you specify a list by prefixing each list element with a hyphen, even if there is only one element. The following examples show the equivalent configuration in both XML and YAML for configuring a file export connector:

XML Configuration File	YAML Configuration File

`<export>`
`  <configuration`
`      target="eventlog"`
`      type="file">`
`    <property`
`       name="type">csv</property>`
`    <property`
`       name="nonce">eventlog</property>`
`  </configuration>`
`</export>`
	`cluster:`
	`  config:`
	`    deployment:`
	`      export:`
	`        configurations:`
	`           - target: eventlog`
	`             type: file`
	`             properties:`
	`                type: csv`
	`                nonce: eventlog`

2.2.2.3. Security and User Accounts

There are a number of options for securing a VoltDB database, including basic usernames and passwords in addition to industry network solutions such as Kerberos and SSL. Basic security is enabled in the configuration with the `cluster.config.deployment.security.enabled` property. You must also use the property and its children to define the actual usernames, passwords, and assigned roles. Again, the `users` property expects a list of sub-elements so you must prefix each set of properties with a hyphen.

Finally, if you do enable basic security, you must also tell the VoltDB operator which account to use when accessing the database. To do that, you define the `cluster.config.auth` properties, as shown below, which must specify an account with the built-in *administrator* role. The following examples show the equivalent configurations in both XML and YAML, including the assignment of an account to the VoltDB Operator:

XML Configuration File	YAML Configuration File

`<security enabled="true"/>`
`<users>`
`  <user name="admin"`
`        password="superman"`
`        roles="administrator"/>`
`  <user name="mitty"`
`        password="thurber"`
`        roles="user"/>`
`</users>`
	`cluster:`
	`  config:`
	`    deployment:`
	`      security:`
	`        enabled: true`
	`      users:`
	`        - name: admin`
	`          password: superman`
	`          roles: administrator`

XML Configuration File	YAML Configuration File

	- name: mitty
	          password: thurber
	          roles: user
	    auth:
	      username: admin
	      password: superman

2.2.2.4. TLS/SSL

Another important aspect of security is securing and authenticating the ports used to access the database. The most common way to do this is by enabling TLS/SSL to encrypt data and authenticate the servers using user-created certificates. The process for creating the private keystore and truststore in Java is described in the section on "Configuring TLS/SSL on the VoltDB Server" in the *Using VoltDB* guide. This process is the same whether you are running the cluster directly on servers or in Kubernetes.

The one difference when enabling TLS/SSL for the cluster in Kubernetes is that you must also configure the operator with an appropriate truststore, in PEM format. The easiest way to do this is to configure the operator using the same truststore and password you use for the cluster itself. First, you will need to convert the truststore to PEM format using the Java **keytool**:

```
keytool -export          \
    -alias my.key -rfc   \
    -file mytrust.pem    \
    -keystore mykey.jks  \
    -storepass topsecret \
    -keypass topsecret
```

Once you have your keystore, truststore, and truststore in PEM format, you can configure the cluster and operator with the appropriate SSL properties. The following examples show the equivalent configurations in both XML and YAML (minus the actual truststore and keystore files).

XML Configuration File	YAML Configuration File

<ssl enabled="true"
     external="true"
     internal="true">
  <keystore path="mykey.jks"
     password="topsecret"/>
  <truststore path="mytrust.jks"
     password="topsecret"/>
</ssl>
	cluster:
	  config:
	    deployment:
	      ssl:
	        enabled: true
	        external: true
	        internal: true
	        keystore:
	          password: topsecret
	        truststore:
	          password: topsecret
	  clusterSpec:
	    ssl:
	      insecure: false

Using the preceding YAML file (calling it `ssl.yaml`), we can complete the SSL configuration by specifying the truststore and keystore files on the **helm** command line with the `--set-file` argument:

```
helm install mydb voltdb/voltdb                                                  \
  --values myconfig.yaml                                                         \
  --values ssl.yaml                                               \
  --set-file cluster.config.deployment.ssl.keystore.file=mykey.jks      \
  --set-file cluster.config.deployment.ssl.truststore.file=mytrust.jks  \
  --set-file cluster.clusterSpec.ssl.certificateFile=mytrust.pem
```

Three important notes concerning TLS/SSL configuration:

- If you enable SSL for the cluster's external interface and ports and you also want to enable Prometheus metrics, you must provide an appropriate SSL truststore and password for the metrics agent. See Section 6.1, "Using Prometheus to Monitor VoltDB" for more information on configuring the Prometheus agent in Kubernetes.

- If you do not require validation of the TLS certificate by the operator, you can avoid setting the truststore PEM for the operator and, instead, set the `cluster.clusterSpec.ssl.insecure` property to *true*.

- If you enable SSL for the cluster, you must repeat the specification of the truststore and keystore files every time you update the configuration. Using the `--reuse-values` argument on the `helm upgrade` command is *not* sufficient.

Finally, TLS/SSL certificates have an expiration date. It is important you replace the certificate before it expires (if `cluster.clusterSpec.ssl.insecure` is false, which is the default). If not, the operator will lose the ability to communicate with the cluster pods. See Section 5.3, "Updating TLS Security Certificates" for instructions on updating the TLS/SSL certificates in Kubernetes.

2.2.3. Configuring Logging

VoltDB uses Log4J for logging messages while the database is running. The chapter on "'Logging and Analyzing Activity in a VoltDB Database" in the *VoltDB Administrator's Guide* describes some of the ways you can customize the logging to meet your needs, including changing the logging level or adding appenders. Logging is also available in the Kubernetes environment and is configured using a Log4J XML configuration file. However, the default configuration and how you set the configuration when starting or updating the database in Kubernetes is different than as described in the *Administrator's Guide*.

Before you attempt to customize the logging, you should familiarize yourself with the default settings. The easiest way to do this is to extract a copy of the default configuration from the Docker image you will be using. The following commands create a docker container without actually starting the image, extract the configuration file to a local file (`k8s-log4j.xml` in the example), then delete the container.

```
$ ID=$(docker create voltdb/voltdb-enterprise)
$ docker cp ${ID}:/opt/voltdb/tools/kubernetes/server-log4j.xml k8s-log4j.xml
$ docker rm $ID
```

Once you extract the default configuration and made the changes you want, you are ready to specify your new configuration on the Helm command to start the database. You do this by setting the `cluster.config.log4jcfgFile` property. For example:

```
$ helm install mydb voltdb/voltdb                      \
   --values myconfig.yaml                              \
   --set cluster.configSpec.replicas=5                 \
   --set-file cluster.config.licenseXMLFile=license.xml \
   --set-file cluster.config.log4jcfgFile=my-log4j.xml
```

Similarly, you can update the logging configuration on a running cluster by using the `--set-file` argument on the Helm upgrade command:

```
$ helm upgrade mydb voltdb/voltdb --reuse-values        \
   --set-file cluster.config.log4jcfgFile=my-log4j.xml
```

Chapter 3. Managing VoltDB Databases in Kubernetes

When running VoltDB in Kubernetes, you are implicitly managing two separate technologies: the database cluster — that consists of "nodes" and the server processes that run on them — and the collection of Kubernetes "pods" the database cluster runs on. There is a one-to-one relationship between VoltDB nodes and Kubernetes pods and it is important that these two technologies stay in sync.

The good news is that the VoltDB Operator and Helm manage the orchestration of Kubernetes and the VoltDB servers. If a database server goes down, Kubernetes recognizes that the corresponding pod is not "live" and spins up a replacement. On the other hand, if you *intentionally* stop the database without telling the Operator or Kubernetes, Kubernetes insists on trying to recreate it.

Therefore, whereas on traditional servers you use **voltadmin** and **sqlcmd** to manage both the cluster and the database content, it is important in a Kubernetes environment that you use the correct utilities for the separate functions:

- Use **kubectl** and **helm** to manage the cluster and the database configuration
- Use **voltadmin** and **sqlcmd** to manage the database contents.

The following sections explain how to access and use each of these utilities. Subsequent chapters explain how to perform common cluster and database management functions using these techniques.

3.1. Managing the Cluster Using kubectl and helm

The key advantage to using Kubernetes is that it automates common administrative tasks, such as making sure the cluster keeps running. This is because the VoltDB Operator and Helm charts manage the synchronization of VoltDB and Kubernetes for you. But it does mean you must use **helm** or **kubectl**, and *not* the equivalent **voltadmin** commands, to perform operations that affect Kubernetes, such as starting and stopping the database, resizing the cluster, changing the configuration, and so on.

When you start the database for the first time, you specify the VoltDB Helm chart and a set of properties that define how the cluster and database are configured. The result is a set of Kubernetes pods and VoltDB server processes known as a Helm "release".

To manage the cluster and database configuration you use the **helm upgrade** command to update the release and change the properties associated with the feature you want to control. For example, to change the frequency of periodic snapshots in the *mydb* release to 30 minutes, you specify the new value for the `cluster.config.deployment.snapshot.frequency` property, like so:

```
$ helm upgrade mydb voltdb/voltdb       \
   --reuse-values                       \
   --set cluster.config.deployment.snapshot.frequency=30m
```

Note

It is also possible to use the **kubectl patch** command to change release properties, specifying the new property value and action to take as a JSON string. However, the examples in this book use the **helm upgrade** equivalent wherever possible as the helm command tends to be easier to read and remember.

One caveat to using the **helm upgrade** command is that it not only upgrades the release, it checks to see if there is a new version of the original chart (in this example, *voltdb/voltdb*) and upgrades that too. Problems could occur if there are changes to the original chart between when you first start the cluster and when you need to stop or resize it.

The public charts are not changed very frequently. But if your database is in production for an extended period of time it could be an issue. Fortunately, there is a solution. To avoid any unexpected changes, you can tell Helm to use a specific version of the chart — the version you started with.

First, use the **helm list** command to list all of the releases (that is, database instances) you have installed. In the listing it will include both the name and version of the chart in use. For example:

```
$ helm list
NAME  NAMESPACE  REVISION  UPDATED              STATUS    CHART         APP VERSION
mydb  default    1         2020-08-12 12:45:30  deployed  voltdb-1.0.0  10.0.0
```

You can then specify the specific chart version when your upgrade the release, thereby avoiding any unexpected side effects:

```
$ helm upgrade mydb voltdb/voltdb       \
   --reuse-values                       \
   --set cluster.clusterSpec.replicas=7 \
   --version=1.0.0
```

3.2. Managing the Database Using voltadmin and sqlcmd

You manage the database using the VoltDB command line utilities **voltadmin** and **sqlcmd**, the same way you would in a traditional server environment. The one difference is that before you can issue VoltDB commands, you need to decide how to access the database cluster itself. There are two types of access available to you:

- Interactive access for issuing **sqlcmd** or **voltadmin** commands to manage the database
- Programmatic access, through the client or admin port, for invoking stored procedures

3.2.1. Accessing the Database Interactively

Kubernetes provides several ways to access the pods running your services. You can run commands on individual pods interactively through the **kubectl exec** command. You can use the same command to access the command shell for the pod by running bash. Or you can use port forwarding to open ports from the pod to your current environment.

In all three cases, you need to know the name of the pod you wish to access. When you start a VoltDB cluster with Helm, the pods are created with templated names based on the Helm release name and a sequential number. So if you named your three node cluster *mydb*, the pods would be called mydb-voltdb-cluster-0, mydb-voltdb-cluster-1, and mydb-voltdb-cluster-2. If you are not sure of the names, you can use the **kubectl get pods** command to see a list:

```
$ kubectl get pods
NAME                                        READY   STATUS    RESTARTS   AGE
mydb-voltdb-cluster-0                       1/1     Running   0          26m
mydb-voltdb-cluster-1                       1/1     Running   0          26m
```

```
mydb-voltdb-operator-6bbb96b575-8z75x   1/1     Running   0          26m
```

Having chosen a pod to use, running VoltDB commands interactively with **kubectl exec** is useful for issuing individual commands. After the command executes, **kubectl** returns you to your local shell. For example, you can check the status of the cluster using the **voltadmin status** command:

```
$ kubectl exec -it mydb-voltdb-cluster-0 -- voltadmin status
Cluster 0, version 10.0, hostcount 2, kfactor 0
 2 live host, 0 missing host, 0 live client, uptime 0 days 00:41:34.293
-----------------------------------------------
  HostId       Host Name
       0mydb-voltdb-cluster-0
       1mydb-voltdb-cluster-1
```

You can even use **kubectl exec** to start an interactive **sqlcmd** session, which stays active until you exit **sqlcmd**:

```
$ kubectl exec -it mydb-voltdb-cluster-0 -- sqlcmd
SQL Command :: localhost:21212
1> exit
$
```

Or you can pipe a file of SQL statements to **sqlcmd** as part of the command:

```
$ kubectl exec -it mydb-voltdb-cluster-0 -- sqlcmd < myschema.sql
```

However, **kubectl exec** commands execute in the context of the pod. So you cannot do things like load JAR files that are in your local directory. If you need to load schema and stored procedures, it is easier to use port forwarding, where ports on the pod are forwarded to the equivalent ports on localhost for your local machine, so you can run applications and utilities (such as **sqlcmd**, **voltdb**, and **voltadmin**) locally.

The **kubectl port-forward** command initiates port forwarding, which is active until you stop the command process. So you need a second process to utilize the linked ports. In the following example the user runs the voter sample application locally on a database in a Kubernetes cluster. To do this, one session enables port forwarding on the client and http ports and the second session loads the stored procedures, schema, and then runs the client application:

Session #1

```
$ kubectl port-forward mydb-voltdb-cluster-0 21212 8080
```

Session #2

```
$ cd ~/voltdb/examples/voter
$ sqlcmd
SQL Command :: localhost:21212
1> load classes voter-procs.jar;
2> file ddl.sql;
3> exit
$ ./run.sh client
```

Port forwarding is useful for ad hoc activities such as:

- Loading schema and stored procedures to a running database
- Monitoring VoltDB with the web-based VoltDB Management Center (by forwarding port 8080)

- Quick test runs of client applications

Port forwarding is *not* good for running production applications or any ongoing activities, due to its inherent lack of security or robustness as a network solution.

3.2.2. Accessing the Database Programmatically

The approaches for connecting to the database interactively do not work for access by applications, because interactive access focuses on connecting to one node of the database. Applications are encouraged to create connections to *all* nodes of the database to distribute the workload and avoid bottle necks. In fact, the Java client for VoltDB has special settings to automatically connect to all available nodes (topology awareness) and direct partitioned procedures to the appropriate host (client affinity).

Kubernetes provides a number of services to make pods accessible beyond the Kubernetes cluster they run in; services such as cluster IPs, node ports, and load balancers. These services usually change the address and/or port number seen outside the cluster. And there are still other layers of networking and firewalls to traverse before these open ports are accessible outside of Kubernetes itself. This complexity, plus the fact that these services result in port numbers and external network addresses that do not match what the database itself thinks it is running on, make accessing the database from external applications impractical.

The recommended way to access a VoltDB database running in Kubernetes programmatically is to run your application as its own service within the same Kubernetes cluster as the database. This way you can take advantage of the existing VoltDB service names, such as *mydb-voltdb-cluster-client*, to connect to the database. You can then enable topology awareness in the Java cient and let the client make the appropriate connections to the current VoltDB host IPs.

For example, if your database Helm release is called *mydb* and is running in the namespace *mydata*, the Java application code to initiate access to the database might look like the following:

```
org.voltdb.client.Client client = null;

ClientConfig config = new ClientConfig("","");
config.setTopologyChangeAware(true);

client = ClientFactory.createClient(config);
client.createConnection("mydb-voltdb-cluster-client.mydata.svc.cluster.local");
```

Chapter 4. Starting and Stopping the Database

The key to managing VoltDB clusters in Kubernetes is to let the Helm charts do the work for you. You can use **helm** commands to perform all basic database management activities. This chapter explains how to use helm commands to:

- Start the cluster for the first time
- Stop and restart the cluster
- Resize the cluster
- Pause and resume

Subsequent chapters explain how to modify the database and cluster configuration of a running database as well as upgrade the VoltDB software itself.

4.1. Starting the Cluster for the First Time

As described in Chapter 2, *Configuring the VoltDB Database Cluster* you can customize every aspect of the database and the cluster using Helm properties and the configuration can be a simple or as complex as you choose. But once you have determined the configuration options you want to use, actually initializing and starting the database cluster is a single command, **helm install**. For example:

```
$ helm install mydb voltdb/voltdb                        \
   --values myconfig.yaml                                \
   --set-file cluster.config.licenseXMLFile=license.xml  \
   --set cluster.clusterSpec.replicas=5
```

4.2. Stopping and Restarting the Cluster

Once the cluster is running (what Helm calls a "release"), you can adjust the cluster to stop it, restart it, or resize it, by "upgrading" the release chart, specifying the new value for the number of nodes you want. You upgrade the release using much the same command, except rather than repeating the configuration, you can use the `--reuse-values` flag. So, for example, to stop the cluster, you simply set the number of replicas to zero, reusing all other parameters:

```
$ helm upgrade mydb voltdb/voltdb       \
   --reuse-values                       \
   --set cluster.clusterSpec.replicas=0
```

To restart the cluster after you stop it, you reset the replica count to five, or whatever you set it to when you initially defined and started it:

```
$ helm upgrade mydb voltdb/voltdb       \
   --reuse-values                       \
   --set cluster.clusterSpec.replicas=5
```

4.3. Resizing the Cluster with Elastic Scaling

To resize the cluster by adding nodes you simply upgrade the release specifying the new number of nodes you want. Of course, the new value must meet the requirements for elastically expanding the cluster, as set out in the discussion of adding nodes to the cluster in the *VoltDB Administrator's Guide*. So, for example, to increase the cluster size by two nodes, you can set the replica count to seven:

```
$ helm upgrade mydb voltdb/voltdb       \
   --reuse-values                       \
   --set cluster.clusterSpec.replicas=7
```

4.4. Pausing and Resuming the Cluster

To pause the database — that is stop client activity through the client port when performing certain administrative functions — you set the property cluster.clusterSpec.maintenanceMode to true. For example, the following commands pause and then resume the database associated with release *mydb*:

```
$ helm upgrade mydb voltdb/voltdb       \
   --reuse-values                       \
   --set cluster.clusterSpec.maintenanceMode=true

$ helm upgrade mydb voltdb/voltdb       \
   --reuse-values                       \
   --set cluster.clusterSpec.maintenanceMode=false
```

Chapter 5. Updates and Upgrades

Once the database is up and running, Kubernetes works to keep it running in the configuration you specified. However, you may need to change that configuration as your database requirements evolve. Changes may be as simple as adding, deleting, or modifying database tables or procedures. Or you may want to modify the configuration of the database, adding new users, or even expanding the cluster by adding nodes.

The following sections describe some common update scenarios and how to perform them in a Kubernetes environment, including:

- Modifying the database schema
- Modifying the database or cluster configuration
- Upgrading the VoltDB software and Helm charts

5.1. Updating the Database Schema

Once the VoltDB database starts, you are ready to manage the database contents. Using Kubernetes does not change *how* you manage the database content. However, it does require a few extra steps to ensure you have access to the database, as described in Section 3.2.1, “Accessing the Database Interactively”.

First you need to identify the pods using the **kubectl get pods** command. You can then access the pods, individually, using the **kubectl exec** command, specifying the pod you want to access and the command you want to run. For example, to run sqlcmd on the first pod, use the following command:

```
$ kubectl exec -it mydb-voltdb-cluster-0 -- sqlcmd
SQL Command :: localhost:21212
1>
```

You can execute a local batch file of sqlcmd commands remotely by piping the file into the utility. For example:

```
$ cat schema.sql
 CREATE TABLE HELLOWORLD (
    HELLO VARCHAR(15), WORLD VARCHAR(15),
    DIALECT VARCHAR(15) NOT NULL
 );
 PARTITION TABLE HELLOWORLD ON COLUMN DIALECT;
$ kubectl exec -it mydb-voltdb-cluster-0 -- sqlcmd < schema.sql
Command succeeded.
Command succeeded.
$
```

Changing the database schema does not require synchronization with Helm or Kubernetes necessarily. However, if you specified the schema and/or procedure classes when you initially create the Helm release, it may be a good idea to keep those properties updated in case you need to re-initialize the database. (for example, when re-establishing a XDCR connection that was broken due to conflicts.) This can be done by updating the `cluster.config.schemas` and/or `cluster.config.classes` properties. For example:

```
$ helm upgrade mydb voltdb/voltdb       \
--reuse-values                      \
--set-file cluster.config.schemas schema.sql \
--set-file cluster.config.classes procs.jar
```

5.2. Updating the Database Configuration

You can also change the configuration options for the database or the cluster while the database is running. In Kubernetes, you do this by updating the release properties rather than with the **voltadmin update** command.

How you update the configuration properties is the same for all properties: you use the **helm upgrade** command to update the individual properties. However, what actions result from the update depend on the type of properties you want to modify:

- Dynamic database configuration properties that can be modified "on the fly" without restarting the database
- Static database configuration properties that require the database be restarted before they are applied
- Cluster configuration properties that alter the operation of the cluster and associated Kubernetes pods

The following sections describe these three circumstances in detail.

5.2.1. Changing Database Properties on the Running Database

There are a number of database configuration options that can be changed while the database is running. Those options include:

- Security settings, including user accounts

  ```
  cluster.config.deployment.security.enabled
  cluster.config.deployment.users
  ```

- Import and export settings

  ```
  cluster.config.deployment.export.configurations
  cluster.config.deployment.import.configurations
  ```

- Database replication settings (except the DR cluster ID)

  ```
  cluster.config.deployment.dr.role
  cluster.config.deployment.dr.connection
  ```

- Automated snapshots

  ```
  cluster.config.deployment.snapshot.*
  ```

- System settings:

  ```
  cluster.config.deployment.heartbeat.timeout
  cluster.config.deployment.systemsettings.flushinterval.*
  cluster.config.deployment.systemsettings.query.timeout
  ```

```
cluster.config.deployment.systemsettings.resourcemonitor.*
```

For example, the following helm upgrade command changes the heartbeat timeout to 30 seconds:

```
$ helm upgrade mydb voltdb/voltdb       \
   --reuse-values                       \
   --set cluster.config.deployment.heartbeat.timeout=30
```

When dynamic configuration properties are modified, the VoltDB Operator updates the running database configuration as soon as it is notified of the change.

5.2.2. Changing Database Properties That Require a Restart

Many database configuration properties are static — they cannot be changed without restarting the database. Normally, this requires manually performing a **voltadmin shutdown --save**, reinitializing and restarting the database cluster, then restoring the final snapshot. For example, command logging cannot be turned on or off while the database is running; similarly, the number of sites per host cannot be altered on the fly.

However, you *can* change these properties using the **helm upgrade** command and the VoltDB Operator will make the changes, but *not* while the database is running. Instead, the Operator recognizes the changes to the configuration, marks the database as requiring a restart, and then schedules a shutdown snapshot, reinitialization, and restart of the database for later.

For example, you cannot change the number of sites per host while the database is running. But the Operator does let you change the property in Kubernetes:

```
$ helm upgrade mydb voltdb/voltdb       \
   --reuse-values                       \
   --set cluster.config.deployment.siteperhost=12
```

No action is taken immediately, since the change will require a restart and is likely to interrupt ongoing transactions. Instead, the Operator waits until you are ready to restart the cluster, which you signify by changing another property, `cluster.clusterSpec.allowRestartDuringUpdate`, to true:

```
$ helm upgrade mydb voltdb/voltdb       \
   --reuse-values                       \
   --set cluster.clusterSpec.allowRestartDuringUpdate=true
```

If you are sure you are ready to restart the cluster when you change the configuration property, you can set the two properties at the same time so that the change takes immediate effect:

```
$ helm upgrade mydb voltdb/voltdb                      \
   --reuse-values                                      \
   --set cluster.config.deployment.siteperhost=12  \
   --set cluster.clusterSpec.allowRestartDuringUpdate=true
```

Once `allowRestartDuringUpdate` is set to true, the Operator initiates the restart process, saving, shutting down, reinitializing, restarting and restoring the database automatically. Note that once the database is restarted, it is a good idea to reset `allowRestartDuringUpdate` to false to avoid future configuration changes triggering immediate restarts:

```
$ helm upgrade mydb voltdb/voltdb       \
   --reuse-values                       \
--set cluster.clusterSpec.allowRestartDuringUpdate=false
```

Warning

There are certain database configuration changes that cannot be made either on the fly or with a restart. In particular, do *not* attempt to change properties associated with directory paths or SSL configuration. Changing any of these properties will leave your database in an unstable state.

5.2.3. Changing Cluster Properties

There are properties associated with the environment that the VoltDB database runs on that you can also modify with the **helm upgrade** command. Most notably, you can increase the size of the cluster, using elastic scaling, by changing the `cluster.clusterSpec.replicas` property, as described in Section 4.3, “Resizing the Cluster with Elastic Scaling”.

Some properties affect the computing environment, such as environment variables and number of nodes. Others control the network ports assigned or features specific to Kubernetes, such as liveness and readiness. All these properties can be modified. However, they each have separate scopes that affect when the changes will go into affect.

Of particular note, pod-specific properties will not take affect until each pod restarts. If this is not a high availability cluster (that is, K=0), the Operator will wait until you to change the property `cluster.clusterSpec.allowRestartDuringUpdate` to true before restarting the cluster and applying the changes. The same applies for any cluster-wide properties.

However, for a K-safe cluster, the Operator can apply pod-specific changes without any downtime by performing a *rolling upgrade*. That is, stopping and replacing each pod in sequence. So for high availability clusters, the Operator will start applying pod-specific changes automatically via a rolling restart regardless of the `cluster.clusterSpec.allowRestartDuringUpdate` setting.

5.3. Updating TLS Security Certificates

If you are using TLS/SSL to encrypt data (either internally, externally, or both), you will need to update those certificates *before* they expire to ensure minimal disruption to normal operation. To update the TLS key stores, truststores, and credentials, you must not only update the appropriate properties, you must stop and restart both the cluster and the Volt Operator in the correct order.

The following instructions describe the process for updating TLS/SSL certificates. Before starting the renewal process, make sure you have the appropriately updated security files. The section on "Configuring TLS/SSL on the VoltDB Server" in the *Using VoltDB* manual and Section 2.2.2.4, “TLS/SSL” explain how to generate the necessary files when using self-signed certificates, as shown in the following examples.The procedure is similar when using certificates from a third-party authority.

1. First, stop the database cluster by setting the replication count to zero. Notice you must repeat the property settings for *the current TLS certificate files* when issuing the **helm upgrade** command::

   ```
   $ helm upgrade mydb voltddb/voltdb --reuse-values                        \
      --set-file cluster.config.deployment.ssl.keystore.file=oldkey.jks     \
      --set-file cluster.config.deployment.ssl.truststore.file=oldtrust.jks \
      --set-file cluster.clusterSpec.ssl.certificateFile=oldtrust.pem       \
      --set cluster.clusterSpec.replicas=0
   ```

2. Next, stop the Volt Operator. You do this by setting the property `operator.enabled` to false:

```
$ helm upgrade mydb voltddb/voltdb --reuse-values                          \
   --set-file cluster.config.deployment.ssl.keystore.file=oldkey.jks     \
   --set-file cluster.config.deployment.ssl.truststore.file=oldtrust.jks \
   --set-file cluster.clusterSpec.ssl.certificateFile=oldtrust.pem       \
   --set operator.enabled=false
```

3. Once the pods for both the cluster nodes and operator have stopped, you are ready to update the helm properties to point to the new security credentials, while restarting the operator:

```
$ helm upgrade mydb voltddb/voltdb --reuse-values                          \
   --set-file cluster.config.deployment.ssl.keystore.file=newkey.jks     \
   --set-file cluster.config.deployment.ssl.truststore.file=newtrust.jks \
   --set-file cluster.clusterSpec.ssl.certificateFile=newtrust.pem       \
   --set operator.enabled=true
```

4. Finally, you are ready to restart the database and return to normal operation:

```
$ helm upgrade mydb voltddb/voltdb --reuse-values                          \
   --set-file cluster.config.deployment.ssl.keystore.file=newkey.jks     \
   --set-file cluster.config.deployment.ssl.truststore.file=newtrust.jks \
   --set-file cluster.clusterSpec.ssl.certificateFile=newtrust.pem       \
   --set cluster.clusterSpec.replicas=5
```

5.4. Upgrading the VoltDB Software and Helm Charts

When new versions of the VoltDB software are released they are accompanied by new versions of the Helm charts that support them. By default when you "install" a "release" of VoltDB with Helm, you get the latest version of the VoltDB software at that time. Your release will stay on its initial version of VoltDB as long as you don't update the charts and VoltDB Operator in use.

You can upgrade an existing database instance to a recent version using a combination of **kubectl** and **helm** commands to update the charts, the operator, and the VoltDB software. The steps to upgrade the VoltDB software in Kubernetes are:

1. Update your copy of the VoltDB repository.

2. Update the custom resource definition (CRD) for the VoltDB Operator.

3. Upgrade the VoltDB Operator and software.

The following sections explain how to perform each step of this process, including a full example of the entire process in Example 5.1, “Process for Upgrading the VoltDB Software” However, when upgrading an XDCR cluster, there is an additional step required to ensure the cluster's schema is maintained during the upgrade process. Section 5.4.4, “Updating VoltDB for XDCR Clusters” explains the extra step necessary for XDCR clusters.

Note

To use the **helm upgrade** command to upgrade the VoltDB software, the starting version of VoltDB must be 10.1 or higher. See the *VoltDB Release Notes* for instructions when using Helm to upgrade earlier versions of VoltDB.

5.4.1. Updating Your Helm Repository

The first step when upgrading VoltDB is to make sure your local copy of the VoltDB Helm repository is up to date. You do this using the **helm repo update** command:

```
$ helm repo update
```

Once you update your local copy of the charts, you can determine which version — of both the charts and the software — you want to use by listing all available versions. You do this with the **helm search repo** command.

```
$ helm search repo voltdb/voltdb --versions
NAME            CHART VERSION  APP VERSION  DESCRIPTION
voltdb/voltdb  1.3.0          10.2.0       A Helm chart for VoltDB
voltdb/voltdb  1.2.1          10.1.3       A Helm chart for VoltDB
voltdb/voltdb  1.2.0          10.1.2       A Helm chart for VoltDB
voltdb/voltdb  1.1.0          10.1.0       A Helm chart for VoltDB
voltdb/voltdb  1.0.2          10.0.0       A Helm chart for VoltDB
```

The display shows the available versions, including for each release a version number for the chart and one for the VoltDB software (app version). Make a note of the pair of version numbers who want to use because you will need them both to complete the following steps of the process. All of the examples in this document use the chart version *1.2.1* and the software version *10.1.3* for the purposes of demonstration.

5.4.2. Updating the Custom Resource Definition (CRD)

The second step is to update the custom resource definition (CRD) for the VoltDB Operator. This allows the Operator to be upgraded to the latest version.

To update the CRD, you must first save a copy of the latest chart, then extract the CRD from the resulting tar file. The **helm pull** command saves the chart as a gzipped tar file and the **tar** command lets you extract the CRD. For example:

```
$ helm pull voltdb/voltdb --version 1.2.1
$ tar --strip-components=2 -xzf voltdb-1.2.1.tgz  \
     voltdb/crds/voltdb.com_voltdbclusters_crd.yaml
```

Note that the file name of the resulting tar file includes the chart version number. Once you have extracted the CRD as a YAML file, you can apply it to Kubernetes:

```
$ kubectl apply -f voltdb.com_voltdbclusters_crd.yaml
```

5.4.3. Upgrading the VoltDB Operator and Software

Once you update the CRD, you are ready to upgrade VoltDB, including both the Operator and the server software. You do this using the **helm upgrade** command and specifying the version numbers for both items on the command line. As soon as you make this change, the Operator will pause the database, take a final snapshot, shutdown the database and then restart with the new version, restoring the snapshot in the process. For example:

```
$ helm upgrade mydb voltdb/voltdb --reuse-values \
   --set operator.image.tag=1.2.1 \
   --set cluster.clusterSpec.image.tag=10.1.3
```

Example 5.1, “Process for Upgrading the VoltDB Software” summarizes all of the commands needed to update a database release to VoltDB version *10.1.3*.

Example 5.1. Process for Upgrading the VoltDB Software

```
$    # Update the local copy of the charts
$ helm repo update
$ helm search repo voltdb/voltdb --versions
NAME           CHART VERSION  APP VERSION  DESCRIPTION
voltdb/voltdb  1.2.1          10.1.3       A Helm chart for VoltDB
voltdb/voltdb  1.2.0          10.1.2       A Helm chart for VoltDB
voltdb/voltdb  1.1.0          10.1.0       A Helm chart for VoltDB
voltdb/voltdb  1.0.2          10.0.0       A Helm chart for VoltDB
$
$
$    # Extract and update the CRD
$ helm pull voltdb/voltdb --version 1.2.1
$ tar --strip-components=2 -xzf voltdb-1.2.1.tgz  \
     voltdb/crds/voltdb.com_voltdbclusters_crd.yaml
$ kubectl apply -f voltdb.com_voltdbclusters_crd.yaml
$
$
$    # Upgrade the Operator and VoltDB software
$ helm upgrade mydb voltdb/voltdb --reuse-values \
   --set operator.image.tag=1.2.1 \
   --set cluster.clusterSpec.image.tag=10.1.3
```

5.4.4. Updating VoltDB for XDCR Clusters

When upgrading an XDCR cluster, there is one extra step you must pay attention to. Normally, during the upgrade, VoltDB saves and restores a snapshot between versions and so all data and schema information is maintained. When upgrading an XDCR cluster, the data and schema is deleted, since the cluster will need to reload the data from another cluster in the XDCR relationship once the upgrade is complete.

Loading the data is automatic. But loading the schema depends on the schema being stored properly before the upgrade begins.

If the schema was loaded through the YAML properties `cluster.config.schemas` and `cluster.config.classes` originally and has not changed, the schema and classes will be restored automatically. However, if the schema was loaded manually or has been changed since it was originally loaded, you must make sure a current copy of the schema and classes is available after the upgrade. There are two ways to do this.

For both methods, the first step is to save a copy of the schema and the classes. You can do this using the **voltdb get schema** and **voltdb get classes** commands. For example, using Kubernetes port forwarding you can save a copy of the schema and class JAR file to your local working directory:

```
$ kubectl port-forward mydb-voltdb-cluster-0 21212 &
$ voltdb get schema -o myschema.sql
$ voltdb get classes -o myclasses.jar
```

Once you have copies of the current schema and class files, you can either set them as the default schema and classes for your database release before you upgrade the software or you can set them in the same command as you upgrade the software. For example, the following commands set the default schema and classes first, then upgrade the Operator and server software. Alternately, you could put the two `--set-file` and two `--set` arguments in a single command.

```
$ helm upgrade mydb voltdb/voltdb --reuse-values \
   --set-file cluster.config.schemas=myschema.sql  \
   --set-file cluster.config.classes=myclasses.jar
$ helm upgrade mydb voltdb/voltdb --reuse-values \
   --set operator.image.tag=1.2.1 \
   --set cluster.clusterSpec.image.tag=10.1.3
```

Chapter 6. Monitoring VoltDB Databases in Kubernetes

Once the database is running, you need to monitor the system to ensure reliable uptime and performance. Variations in usage, workload, or the operational environment can affect the dynamics of the data application, which may need corresponding adjustments to the schema, procedures, or hardware configuration. VoltDB provides system procedures (such as @Statistics) and the web-based Volt Management Center to help monitor current performance. But to provide persistent, historical intelligence concerning application performance it is best to use a dedicated metrics data store, such as Prometheus.

Prometheus is a metrics monitoring and alerting system that provides ongoing data collection and persistent storage for applications and other resources. By providing an open source industry standard for collecting and storing metrics, Prometheus allows you to:

- Offload monitoring from the database platform itself
- Combine metrics from VoltDB with other applications within your business ecosystem
- Query and visualize historical information about your database activity and performance (through tools such as Grafana)

Section 6.1, “Using Prometheus to Monitor VoltDB” explains how to configure your VoltDB database so the information you need is gathered and made available through Prometheus and compatible graphic consoles such as Grafana.

6.1. Using Prometheus to Monitor VoltDB

When monitoring VoltDB with Prometheus on Kubernetes, the Operator starts a separate pod to run the Prometheus agent, with one agent providing metrics for the entire cluster. The agent takes data from the database (through the system procedures @Statistics and @SystemInformation) and makes them available as metrics in the format that Prometheus requires. In other words, the agent stands between the VoltDB cluster and the Prometheus server, doing the data collection and translation needed.

Before you start, you must make sure Prometheus is installed on your Kubernetes cluster, ideally within the same namespace as your VoltDB instance. Once the Prometheus stack is available, you can enable and configure the Prometheus agent using the `metrics...` properties in Helm. First, set the `metrics.enabled` property to true. You can set this property when initially configuring the database or using **helm upgrade** after the database is running. For example, the following command enables the Prometheus agent with default settings while initializing the *mydb* database cluster:

```
$ helm install mydb voltdb/voltdb                        \
   --set-file cluster.config.licenseXMLFile=license.xml  \
   --set cluster.clusterSpec.replicas=5                  \
   --set metrics.enabled=true
```

Once enabled, the agent starts in a separate pod using the database name (in this case "mydb"), the string "voltdb-metrics", and an arbitrary suffix as the pod name. The agent then uses port number 8484 to report its metrics. By default, the connection to the Prometheus server is handled automatically so you do not need to add the agent as a target manually to the Prometheus configuration. Unless you wish to report your metrics to a Prometheus server outside the current Kubernetes environment.

If the database has security enabled, you will also need to set the `metrics.username` and `metrics.password` properties specifying a valid account on the database cluster so the agent can access the system procedures. Similarly, if the database cluster is configured to use SSL/TLS, you must configure the Prometheus agent to use SSL/TLS as well, using the SSL properties such as `metrics.ssl.enabled` and `metrics.ssl.truststore.file`, making sure the agent uses a truststore that can validate against the database's certificate. For example, the following Helm command enables SSL for both the database and the Prometheus agent, using the same user-generated certificate for both:

```
$ helm install mydb voltdb/voltdb                                          \
   --set-file cluster.config.licenseXMLFile=license.xml                    \
   --set cluster.clusterSpec.replicas=5                                    \
   --set cluster.config.deployment.ssl.enabled=true                        \
   --set cluster.config.deployment.ssl.keystore.password=topsecret         \
   --set cluster.config.deployment.ssl.truststore.password=topsecret       \
   --set-file cluster.config.deployment.ssl.keystore.file=mycert.jks       \
   --set-file cluster.config.deployment.ssl.truststore.file=mytrust.jks    \
   --set-file cluster.clusterSpec.ssl.certificateFile=mytrust.pem          \
   --set metrics.enabled=true                                              \
   --set metrics.ssl.enabled=true                                          \
   --set metrics.ssl.truststore.password=topsecret                         \
   --set-file metrics.ssl.truststore.file=mytrust.jks
```

Additional properties are available for configuring other aspects of the agent, such as filtering which statistics or system information to include as metrics. For example, the following YAML file enables the Prometheus agent and excludes the *import* and *task* statistics:

```
metrics:
    enabled: true
    skipstats: import,task
```

See Section A.7, “Metrics Configuration Options” for a full list of metrics properties.

Chapter 7. Cross Datacenter Replication in Kubernetes

Previous chapters describe how to run a single VoltDB cluster within Kubernetes. Of course, you can run multiple independent VoltDB databases in Kubernetes. You do this by starting each cluster in separate regions, under different namespaces within the same Kubernetes cluster, or running a single instance of the VoltDB Operator managing multiple clusters in the same namespace. However, some business applications require the same database running in multiple locations — whether for data redundancy, disaster recovery, or geographic distribution. In VoltDB this is done through *Cross Datacenter Replication*, or XDCR.

Important

Please note that in addition to the guidance specific to Kubernetes provided in this chapter, the following rules apply to XDCR in *any* operating environment:

- **You must have command logging enabled** for three or more clusters.
- **You can only join (or rejoin) one cluster at a time** to the XDCR environment.

Command logging is always recommended when using XDCR to ensure durability. Using XDCR without command logging on two clusters, it is possible for transactions processed on one cluster to be lost if the cluster crashes before the binary log is sent to the other cluster. However, for three or more clusters, command logging is *required*. Without command logging, not only can XDCR transactions be lost, but *it is likely the databases will diverge without warning*, if a cluster crashes after sending a binary log to one cooperating cluster but not to the other.

7.1. Requirements for XDCR in Kubernetes

Once established, XDCR in Kubernetes works the same way it does in any other network environment, as described in the chapter on Database Replication in the *Using VoltDB* guide. The key difference when using XDCR in Kubernetes is how you establish the initial connection between the clusters. Unlike traditional servers with known IP addresses, in Kubernetes network addresses are assigned on the fly and are not normally accessible outside individual namespaces or regions. Therefore, you must do additional work to create the appropriate network relationships. Specifically, you must:

- **Establish a network mesh** between the Kubernetes clusters containing the VoltDB databases so that the nodes of each VoltDB cluster can identify and resolve the IP addresses and ports of all the nodes from the other VoltDB clusters.
- **Configure the VoltDB clusters**, including properties that identify the type of mesh involved and mesh-specific annotations that determine what network addresses and ports to use.

The following sections describe the different approaches to establishing a network mesh and how to configure the clusters in each case.

7.2. Choosing How to Establish a Network Mesh

For XDCR to work, each cluster must be able to identify and connect to the nodes of the other cluster. Establishing the XDCR relationship occurs in two distinct phases:

1. **Network Discovery** — First, the clusters connect over the replication port (port 5555, by default). The initial connection confirms that the configurations are compatible, that the schema of the two clusters match for all DR tables, and that there is data in only one of the clusters.
2. **Replication** — Once the clusters agree on the schema, each cluster sends a list of node IP addresses and ports to the other cluster and multiple connections are made, node-to-node, between the two clusters. If there is existing data, a synchronization snapshot is sent between the clusters and then replication begins.

For the network discovery phase, each cluster must have a clearly identifiable network address that the other cluster can specify as part of its XDCR configuration. For the replication phase, each cluster must have externally reachable network addresses for each node in the cluster that it can advertise during the discovery phase and that the other cluster uses to make the necessary connections for replication.

Since, by default, the ports on a Kubernetes pod are not externally accessible, you must use additional services to make the VoltDB nodes accessible. Three such options are:

- **Kubernetes Load Balancers** — One way to establish a network mesh is to use the built-in load balancer service within Kubernetes. Load balancers provide a defined, persistent external interface for internal pods. The advantage of using load balancers is that they are a native component of Kubernetes and are easy to configure. The disadvantage is that if you are running your VoltDB clusters in a hosted environment, load balancers tend to be far more expensive than regular pods and creating a separate load balancer for each node in the cluster to handle the replication phase can be prohibitively expensive unless you are managing your own infrastructure.
- **Kubernetes Node Ports** — An alternative to load balancers is using node ports. Node ports, like load balancers, are native services of Kubernetes and provide an externally accessible interface for the internal pods. However, unlike load balancers where the addresses are persistent over time, node ports take on the addresses of the underlying Kubernetes nodes and therefore can change as Kubernetes nodes are recycled. Therefore node ports are not appropriate for the Network Discovery phase. On the other hand, they can be a cheaper alternative to load balancers for the replication phase, since the cluster can advertise the current set of node port addresses as pods come and go.
- **Network Mesh Services** — These additional services, such as Consul, create a network mesh between Kubernetes clusters and regions. They essentially act as a virtual private network (VPN) within Kubernetes so the VoltDB clusters can interoperate as if they were local to each other. The advantage of using network mesh services is that configuring the VoltDB clusters is simpler, since all of the network topology is handled separately. The deficit is that this requires yet another service to set up. And the configuration of these services can be quite complex, requiring a deep understanding of — and access to — the networking layer in Kubernetes.

Which networking solution you use is up to you. You can even mix and match the alternatives — using, for example, a single load balancer per cluster for the Network Discovery phase and individual node ports for each VoltDB cluster node during the replication phase.

You define the type of network mesh to use and how to connect using YAML properties when you configure your clusters. In general, the Helm properties starting with `cluster.config.deployment.dr`, such as `id` and `role`, are generic properties common to all XDCR implementations. Helm properties starting with `cluster.serviceSpec` define the type of network mesh to use and annotations specific to the network type.

The following sections explain how to configure XDCR using Helm properties, with individual sections discussing the differences necessary for various networking options, including:

- Common XDCR Properties
- Configuring XDCR in Local Namespaces

- Configuring XDCR Using Load Balancers
- Configuring XDCR Using Node Ports for Replication
- Configuring XDCR Using Network Services

7.3. Common XDCR Properties

No matter what approach you choose for establishing the network mesh, you must first configure the clusters as members of the XDCR quorum the same way you do on bare metal. That is, you must assign:

- A unique DR ID for each cluster between 0 and 127
- The cluster role (XDCR)
- At least one node from the other cluster as the point of connection for the Network Discovery phase

On traditional servers these properties are defined in an XML configuration file. On Kubernetes, you specify the configuration using YAML properties. The following table shows two equivalent XDCR configurations in the different formats.

XML Configuration File	YAML Configuration File

`<dr id="1" role="xdcr" >`
`   <connection`
`      source="brooklyn.mycorp.lan" />`
`</dr>`
	`cluster:`
	`  config:`
	`    deployment:`
	`      dr:`
	`        id: 1`
	`        role: xdcr`
	`        connection:`
	`          enabled: true`
	`          source: \`
	`  "brooklyn-voltdb-cluster-dr:5555"`

7.4. Configuring XDCR in Local Namespaces

The easiest way to configure XDCR clusters is when the VoltDB clusters are within the same Kubernetes namespace or cluster. In this case, the cluster IP addresses are all locally visible and so do not need any additional network setup. The first step is to enable the DR service using the `cluster.serviceSpec.dr.enabled` property:

```
cluster:
  serviceSpec:
    dr:
      enabled: true
```

Next, you must provide the address of a replication port from one node of the remote cluster as the `source` property.

In Kubernetes the cluster nodes are assigned unique host names based on the initial Helm release name (that is, the name you assigned the cluster when you installed it). The VoltDB Operator also creates services that abstract the individual server addresses and provide a single entry point for specific ports on the database cluster. The two services of interest are DR and client, which will direct traffic to the corresponding port (5555 or 21212 by default) on an arbitrary node of the cluster. If the two database instances are within the same Kubernetes cluster, you can use the DR service to make the initial connection between the database systems, as shown in the following YAML configuration file.

If the databases are running in different namespaces, you will need to specify the fully qualified service name as the connection source in the configuration, which includes the namespace. So, for example, if the *manhattan* database is in namespace *ny1* and *brooklyn* is in *ny2*, the YAML configuration files related to XDCR for the two clusters would be the following.

Manhattan Cluster

```
cluster:
  config:
    deployment:
      dr:
        id: 1
        role: xdcr
        connection:
          enabled: true
          source: "brooklyn-voltdb-cluster-dr.ny2.svc.cluster.local:5555"
```

Brooklyn Cluster

```
cluster:
  config:
    deployment:
      dr:
        id: 2
        role: xdcr
        connection:
          enabled: true
          source: "manhattan-voltdb-cluster-dr.ny1.svc.cluster.local:5555"
```

7.5. Configuring XDCR Using Load Balancers

Kubernetes load balancers are an alternative for making VoltDB clusters accessible outside the Kubernetes cluster or region they are in. In this case you are not using load balancers for their traditional role, balancing the load between multiple pods. Instead, the load balancers are solely used to provide externally accessible IP addresses.

There are two approaches to using load balancers. The first approach is to assign a load balancer for each node of the cluster. Since the nodes are externally reachable through persistent IP addresses on their corresponding load balancer, the load balancers can be used for both the network discovery and replication phases. The second approach is to use only one load balancer for the entire cluster to provide network discovery, and use virtual network peering, available from your hosting provider, for replication.

Many hosting platforms, such as Google Cloud or AWS, provide proprietary mechanisms for performing network peering between regions or data centers. Each of these solutions has its own unique set up and configuration, separate from the configuration of VoltDB and the VoltDB Operator . As a result, using a network peering service is not as simple as the use of load balancers for replication. However, they can be significantly more cost effective when paired with a single load balancer for network discovery.

There is also the choice of assigning the IP addresses for the load balancers dynamically, or having them selected from a range of static addresses. Dynamic assignment is simpler, since you do not need to arrange with your hosting provider for pre-assigned IPs or hostnames. However, dynamic addresses also mean you do not know what the addresses are *until the cluster starts*. This means the remote XDCR cluster cannot assign the `source` property until after the cluster starts with its associated load balancers and you can determine the IP addresses assigned to them.

7.5.1. Separate Load Balancers For Each Node (cluster.serviceSpec.perpod)

First you must assign the DR `id` and `role` as Helm properties. If the remote cluster is using static addresses, you can specify one of its nodes as the `source`, as in the following example. If you are using dynamic load balancers, leave the `source` property blank and use the **helm upgrade --set** command once the clusters are running to assign a resulting node address for the remote cluster.

```
cluster:
  config:
    deployment:
      dr:
        id: 1
        role: xdcr
        connection:
          enabled: true
          source: "chicago-dc-2"  # Remote cluster
```

Then in the `cluster.serviceSpec` section, you enable `perpod` by setting its `type` to *LoadBalancer*. You will also want to set the `dr.enabled` property to *true* so the per pod load balancers are used for network discovery as well as replication.

For dynamically assigned addresses, set the `publicIPFromService` to *true*:

```
cluster:
  serviceSpec:
    perpod:
      type: LoadBalancer
      publicIPFromService: true
      dr:
        enabled: true
```

For static IP addresses, use the `staticIPs` property to specify the addresses to assign when creating the load balancers and, again, set `dr.enabled` to true.

```
cluster:
  serviceSpec:
    perpod:
      enabled: true
      type: LoadBalancer
      staticIPs:
      - 12.34.56.78
      - 12.34.56.79
      - 12.34.56.80
      dr:
        enabled: true
```

7.5.2. Single Load Balancer For Discovery with Virtual Networking Peering (cluster.serviceSpec.dr)

To reduce the number of resources needed to connect XDCR clusters in different regions, you can use a single load balancer for network discovery and use virtual network peering services from your hosting provider for connecting the two clusters during replication. How you set up and configure your network peering is specific to each provider. See your provider's documentation for additional information. This section describes how to set up a single Kubernetes load balancer for network discovery once you have your network peering established.

First you must assign the DR `id` and `role` as Helm properties and, if known in advance, the `source` for the remote cluster:

```
cluster:
  config:
    deployment:
      dr:
        id: 1
        role: xdcr
        connection:
          enabled: true
          source: "chicago-dc-2"  # Remote cluster
```

Then in the `cluster.serviceSpec` section, you enable the `dr` service (rather than `perpod`) and set its `type` to *LoadBalancer*. You may also need to provide additional annotations that help configure the service. These annotations are specific to the host environment you are using. So, for example, the following configuration provides annotations for AWS and the Google Cloud:

```
cluster:
  serviceSpec:
    dr:
      enabled: true
      type: LoadBalancer
      annotations:
        # Google Cloud
        networking.gke.io/load-balancer-type: "Internal"
        networking.gke.io/internal-load-balancer-allow-global-access: "true"

        # AWS
        service.beta.kubernetes.io/aws-load-balancer-internal: "true"
        service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
```

7.6. Configuring XDCR Using Node Ports for Replication

Kubernetes node ports are another option for providing external access to the VoltDB cluster for replication. Node ports are similar to load balancers in that they provide an externally accessible network address for individual ports. Node ports are different in that the addresses are transitory — the address and/or port number will change as pods come and go. So node ports are less practical for the Network Discovery phase. However, they can be a cheap alternative for providing external access during the replication phase, since the cluster can advertise the new addresses as its topology changes.

It is also possible to mix and match solutions. So a single load balancer can be used to provide the Network Discovery service for a cluster, while node ports provide per pod network addresses for the replication phase, as described next.

Again, you start by assigning the DR `id` and `role` as Helm properties and, if known in advance, the `source` for the remote cluster:

```
cluster:
  config:
    deployment:
      dr:
        id: 1
        role: xdcr
        connection:
          enabled: true
          source: "chicago-dc-2"  # Remote cluster
```

You then define the load balancer for Network Discovery by setting the values of the `cluster.serviceSpec.dr` properties `enabled` to *true* and `type` to *LoadBalancer*.

```
cluster:
  serviceSpec:
    dr:
      enabled: true
      type: LoadBalancer
```

Finally, define the replication phase as using node ports by configuring `cluster.serviceSpec.perpod` properties `type` to *NodePort* and `dr.enabled` to *true*. You can also use the `dr.startReplicationNodePort` property to specify the starting port number for the externally accessible ports assigned to the node ports.

```
cluster:
  serviceSpec:
    perpod:
      type: NodePort
      dr:
        enabled: true
        startReplicationNodePort: 33111
```

7.7. Configuring XDCR Using Network Services

The goal of network services, such as Consul, is to make Kubernetes pods in different clusters or regions appear as if they were local to each other. This makes configuring XDCR within VoltDB itself easier; in most cases it is almost identical to how you configure clusters within local namespaces. However, how you configure the network service itself is very dependent on which service you are using and the hosting environment in which you are operating.

Using Consul as an example, Consul provides a "sidecar" — an additional process running in the same pod as the VoltDB process — that makes remote pods and clusters appear to be local to the pod itself. So rather than providing a remote IP address and port as the source for XDCR Network Discovery, you specify a local port. For example:

```
cluster:
  config:
    deployment:
      dr:
        id: 1
        role: xdcr
        connection:
          enabled: true
          source: "localhost:4444"
```

What port you specify and how you configure and start Consul and the Consul sidecar, is specific to the Consul product and your implementation of it. The same is true when using other third-party networking services. You may also need to provide additional annotations within the Helm configuration to complete the network setup, depending upon which network service you use. For example:

```
cluster:
  clusterSpec:
    additionalAnnotations:
      "consul.hashicorp.com/connect-service": "chicago-voltdb-cluster"
      "consul.hashicorp.com/connect-service-upstreams": "chicago-voltdb-cluster:55
```

See the product documentation for the specific service for further information.

Chapter 8. Managing XDCR Clusters in Kubernetes

Once you have configured your XDCR clusters and your network environment, you are ready to start the clusters. You begin by starting two of the clusters. (Remember, only one of the clusters can have data in the DR tables before the XDCR communication begins.) Once the schema of the DR tables in two databases match, synchronization starts. After the initial two databases are synchronized, you can start additional XDCR clusters, one at a time.

There are several management procedures that help keep the clusters in sync, especially when shutting down or removing clusters from the XDCR environment. In other environments, these procedures use **voltadmin** commands, such as **shutdown**, **dr drop** and **dr reset**. In Kubernetes, you execute these procedures through the VoltDB Operator using Helm properties. Activities include:

- Removing a cluster temporarily
- Removing a cluster permanently
- Resetting XDCR when a cluster is lost
- Rejoining a cluster that was removed

8.1. Removing a Cluster Temporarily

If you want to remove a cluster from the XDCR environment temporarily, you simply shutdown the cluster normally, by setting the number of replicas to zero. This way, when the cluster restarts, the command logs will take care of recovering all of the data and re-establishing the XDCR "conversations" with the other clusters:

```
--set cluster.clusterSpec.replicas=0
```

8.2. Removing a Cluster Permanently

If you want to remove a cluster from the XDCR environment permanently, you want to make sure it sends all of its completed transactions to the other clusters before it shuts down. You do this by setting the DR role to "none" to perform an orderly shutdown:

```
--set cluster.config.deployment.dr.role="none"
--set cluster.clusterSpec.replicas=0
```

Of course, you do not have to shut the cluster down. You can simply remove it from the XDCR environment. Note that if you do so, the data in the current cluster will diverge from those clusters still participating in XDCR. So only do this if you are sure you want to maintain a detached copy of the data:

```
--set cluster.config.deployment.dr.role="none"
```

Finally, if you cannot perform an orderly removal from XDCR — for example, if one of the other clusters is offline or if sending the outstanding transactions will take too long and you are willing to lose that data — you can set the property `cluster.clusterSpec.dr.forceDrop` to "TRUE" to force the cluster to drop out of the XDCR mesh without finalizing its XDCR transfers. Once the cluster has been removed, it is advisable to reset this property to "FALSE" so future procedures revert to the orderly approach of flushing the queues.

```
--set cluster.clusterSpec.dr.forceDrop=TRUE
--set cluster.config.deployment.dr.role="none"
--set cluster.clusterSpec.replicas=0
 . . .
--set cluster.clusterSpec.dr.forceDrop=FALSE
```

8.3. Resetting XDCR When a Cluster Leaves Unexpectedly

Normally, when a cluster is removed from XDCR in an orderly fashion, the other clusters are notified that the cluster has left the mesh. However, if a cluster leaves unexpectedly — for example, if it crashes or is shutdown and deleted without setting its role to "none" to notify the other clusters — the XDCR network still thinks the cluster is a member and may return. As a result, the remaining clusters continue to save DR logs for the missing member, using up unnecessary processing cycles and disk space. You need to reset the XDCR network mesh to correct this situation.

To reset the mesh you notify the remaining clusters that the missing cluster is no longer a member. You do this be adding the DR ID of the missing cluster to the `cluster.clusterSpec.dr.excludeClusters` property. The property value is an array of DR IDs. For example, if the DR ID (`cluster.config.deployment.dr.id`) of the lost cluster is "3", you set the property to "{3}":

```
--set cluster.clusterSpec.dr.excludeClusters='{3}'
```

You must set this property for *all* of the clusters remaining in the XDCR environment. If later, you want to add the missing cluster (or another cluster with the same DR ID) back into the XDCR mesh, you will need to reset this property. For example:

```
--set cluster.clusterSpec.dr.excludeClusters=null
```

8.4. Rejoining an XDCR Cluster That Was Previously Removed

If a cluster is removed from the XDCR cluster permanently, by resetting the DR role, or through exclusion by the other clusters, it is still possible to rejoin that cluster to the XDCR network. To do that you must reinitialize the cluster and, if it was forcibly excluded, remove the exclusion from the current members of the network. (Note, the following procedure is *not* necessary if the cluster was removed temporarily by setting the number of replicas to zero.)

First, if the cluster was forcibly removed by exclusion, you must remove the exclusion from the current members of the XDCR network by clearing the `cluster.clusterSpec.dr.excludeClusters` property (removing the missing cluster's ID from the array):

```
--set cluster.clusterSpec.dr.excludeClusters=null
```

Then you must restart the cluster you want to rejoin, reinitializing the cluster's contents with the `cluster.clusterSpec.initForce` property and setting the appropriate properties (such as the DR role and connection properties):

```
--set cluster.clusterSpec.initForce=TRUE
--set cluster.config.deployment.dr.role="xdcr"
--set cluster.clusterSpec.replicas=3
```

Once the cluster rejoins the XDCR network and synchronizes with the current members, be sure to reset the `cluster.clusterSpec.initForce` property to false.

Appendix A. VoltDB Helm Properties

You communicate with the VoltDB Operator, and Kubernetes itself, through the Helm charts that VoltDB provides. You can also specify additional Helm properties that customize what the Helm charts do. The properties are hierarchical in nature and can be specified on the Helm command line either as one or more YAML files or as individual arguments. For example, you can specify multiple properties in a YAML file then reference the file as part of your command using the `--values` or `-f` argument, like so:

```
$ helm install mydb voltdb/voltdb --values myoptions.yaml
```

Or you can specify the properties individually in dot notation on the command line using the --set flag, like so:

```
$ helm install mydb voltdb/voltdb \
     --set cluster.clusterSpec.replicas=5 \
     --set cluster.config.deployment.cluster.kfactor=2 \
     --set cluster.config.deployment.cluster.sitesperhost=12
```

For arrays and lists, you can specify the values in dot notation by enclosing the list in braces and then quoting the command as required by the shell you are using. For example:

```
$ helm upgrade mydb voltdb/voltdb -reuse-values
     --set cluster.clusterSpec.excludeClusters='{1,3}'
```

In YAML, you specify each element of the property on a separate line, following each parent element with a colon, indenting each level appropriately, and following the last element with the value of the property . On the command line you specify the property with the elements separated by periods and the value following an equals sign. So in the preceding `install` example, the matching YAML file for the command line properties would look like this:

```
cluster:
  clusterSpec:
     replicas: 5
  config:
     deployment:
        cluster:
           kfactor: 2
           sitesperhost: 12
```

Many of the properties have default values; the following tables specify the default values where applicable. You do not need to specify values for all of the properties. In fact, you can start a generic VoltDB database specifying only the license file. Otherwise, you need only specify those properties you want to customize.

Finally, the properties are processed in order and can be overridden. So if you specify different values for the same property in two YAML files and as a command line argument, the latter YAML file setting overrides the first and the command line option overrides them both.

A.1. How to Use the Properties

The following sections detail all of the pertinent Helm properties that you can specify when creating or modifying the VoltDB Operator and its associated cluster. The properties are divided into categories and each category identified by the root elements common to all properties in that category:

- Top-Level Kubernetes Options

- Kubernetes Cluster Startup Options
- Network Options
- VoltDB Database Startup Options
- VoltDB Database Configuration Options

For the sake of brevity and readability, the properties in the tables are listed by only the unique elements of the property after the root. However, when specifying a property in YAML or on the command line, you must specify all elements of the full property name, including both the root and the unique elements.

A.2. Top-Level Kubernetes Options

The following properties affect how Helm interacts with the Kubernetes infrastructure.

Table A.1. Top-Level Options

Parameter	Description	Default
cluster.enabled	Create VoltDB Cluster	true
cluster.serviceAccount.create	If true, create & use service account for VoltDB cluster node containers	true
cluster.serviceAccount.name	If not set and create is true, a name is generated using the fullname template	""

A.3. Kubernetes Cluster Startup Options

The following properties affect the size and structure of the Kubernetes cluster that gets started, as well as the startup attributes of the VoltDB cluster running on those pods.

Table A.2. Options Starting with cluster.clusterSpec...

Parameter	Description	Default
.replicas	Pod (VoltDB Node) replica count, scaling to 0 will shutdown the cluster gracefully	3
.maxPodUnavailable	Maximum pods unavailable in Pod Disruption Budget	kfactor
.maintenanceMode	VoltDB Cluster maintenance mode (pause all nodes)	false
.takeSnapshotOnShutdown	Takes a snapshot when cluster is shut down by scaling to 0. One of: NoCommandLogging (default), Always, Never. NoCommandLogging means 'snapshot only if command logging is disabled'.	""
.initForce	Always init --force on VoltDB node start/restart. WARNING:	false

Parameter	Description	Default
	This will destroy VoltDB data on PVCs except snapshots.	
.deletePVC	Delete and cleanup generated PVCs when VoltDBCluster is deleted, requires finalizers to be enabled (on by default)	false
.allowRestartDuringUpdate	Allow VoltDB cluster restarts if necessary to apply user-requested configuration changes. May include automatic save and restore of database.	false
.stoppedNodes	User-specified list of stopped nodes based on StatefulSet	[ ]
.additionalXDCRReadiness	Add additional readiness checks using XDCR to ensure both clusters are healthy (WARNING: May cause app downtime)	false
.persistentVolume.size	Persistent Volume size per Pod (VoltDB Node)	1Gi
.persistentVolume .storageClassName	Storage Class name to use, otherwise use default	""
.persistentVolume.hostpath .enabled	Use HostPath volume for local storage of VoltDB. This node storage is often ephemeral and will not use PVC storage classes if enabled.	false
.persistentVolume.hostpath.path	HostPath mount point, defaults to /data/voltdb/ if not specified.	""
.ssl.certificateFile	PEM encoded certificate chain used by the VoltDB operator when SSL/TLS is enabled	""
.ssl.insecure	If true, skip certificate verification by the VoltDB operator when SSL/TLS is enabled	false
.storageConfigs	Optional storage configs for provisioning additional persistent volume claims automatically	[ ]
.additionalVolumes	Additional list of volumes that can be mounted by node containers	[ ]
.additionalVolumeMounts	Pod volumes to mount into the container's filesystem, cannot be modified once set	[ ]
.image.registry	Image registry	docker.io
.image.repository	Image repository	voltdb/voltdb-enterprise
.image.tag	Image tag	10.0.0
.image.pullPolicy	Image pull policy	Always

Parameter	Description	Default
.additionalStartArgs	Additional VoltDB start command args for the pod container	[ ]
.priorityClassName	Pod priority defined by an existing PriorityClass	""
.additionalAnnotations	Additional custom Pod annotations	{ }
.additionalLabels	Additional custom Pod labels	{ }
.resources	CPU/Memory resource requests/limits	{ }
.nodeSelector	Node labels for pod assignment	{ }
.tolerations	Pod tolerations for Node assignment	[ ]
.affinity	Node affinity	{ }
.podSecurityContext	Pod security context	{"runAsNonRoot":true, "runAsUser":1001, "fsGroup":1001}
.securityContext	Container security context. WARNING: Changing user or group ID may prevent VoltDB from operating.	{"privileged":false, "runAsNonRoot":true, "runAsUser":1001, "runAsGroup":1001, "readOnlyRootFilesystem":true}
.clusterInit.initSecretRefName	Name of pre-created Kubernetes secret containing init configuration (deployment.xml, license.xml and log4j.xml), ignores init configuration if set	""
.clusterInit .schemaConfigMapRefName	Name of pre-created Kubernetes configmap containing schema configuration	""
.clusterInit .classesConfigMapRefName	Name of pre-created Kubernetes configmap containing schema configuration	""
.podTerminationGracePeriodSeconds	Duration in seconds the Pod needs to terminate gracefully. Defaults to 30 seconds if not specified.	30
.livenessProbe.enabled	Enable/disable livenessProbe	true
.livenessProbe .initialDelaySeconds	Delay before liveness probe is initiated	20
.livenessProbe.periodSeconds	How often to perform the probe	10
.livenessProbe.timeoutSeconds	When the probe times out	1
.livenessProbe.failureThreshold	Minimum consecutive failures for the probe	10
.livenessProbe.successThreshold	Minimum consecutive successes for the probe	1

Parameter	Description	Default
.readinessProbe.enabled	Enable/disable readinessProbe	true
.readinessProbe .initialDelaySeconds	Delay before readiness probe is initiated	30
.readinessProbe.periodSeconds	How often to perform the probe	17
.readinessProbe.timeoutSeconds	When the probe times out	2
.readinessProbe.failureThreshold	Minimum consecutive failures for the probe	6
.readinessProbe .successThreshold	Minimum consecutive successes for the probe	1
.startupProbe.enabled	Enable/disable startupProbe, feature flag must also be enabled at a cluster level (enabled by default in 1.18)	true
.startupProbe .initialDelaySeconds	Delay before startup probe is initiated	45
.startupProbe.periodSeconds	How often to perform the probe	10
.startupProbe.timeoutSeconds	When the probe times out	1
.startupProbe.failureThreshold	Minimum consecutive failures for the probe	18
.startupProbe.successThreshold	Minimum consecutive successes for the probe	1
.env.VOLTDB_OPTS	VoltDB cluster additional java runtime options (VOLTDB_OPTS)	""
.env.VOLTDB_GC_OPTS	VoltDB cluster java runtime garbage collector options (VOLTDB_GC_OPTS)	""
.env.VOLTDB_HEAPMAX	VoltDB cluster heap size, integer number of megabytes (VOLTDB_HEAPMAX)	""
.env.VOLTDB_HEAPCOMMIT	Commit VoltDB cluster heap at startup, true/false (VOLTDB_HEAPCOMMIT)	""
.env .VOLTDB_K8S_LOG_CONFIG	VoltDB log4jcfg file path	""
.customEnv	Key-value map of additional envvars to set in all VoltDB node containers	{ }
.dr.forceDrop	Indicate if you want to drop cluster from XDCR without producer drain.	false
.dr.excludeClusters	User-specified list of clusters not part of XDCR	[ ]

A.4. Network Options

The following properties specify what ports to use and the port-mapping protocol.

Table A.3. Options Starting with cluster.serviceSpec...

Parameter	Description	Default
.type	VoltDB service type (options ClusterIP, NodePort, and LoadBalancer)	ClusterIP
.externalTrafficPolicy	VoltDB service external traffic policy (options Cluster, Local)	Cluster
.vmcPort	Volt Management Center web interface Service port	8080
.vmcNodePort	Port to expose Volt Management Center service on each node, type NodePort only	31080
.vmcSecurePort	Volt Management Center secure web interface Service port	8443
.vmcSecureNodePort	Port to expose Volt Management Center secure service on each node, type NodePort only	31443
.adminPortEnabled	Enable exposing admin port with the VoltDB Service	true
.adminPort	VoltDB Admin exposed Service port	21211
.adminNodePort	Port to expose VoltDB Admin service on each node, type NodePort only	31211
.clientPortEnabled	Enable exposing client port with the VoltDB Service	true
.clientPort	VoltDB Client exposed service port	21212
.clientNodePort	Port to expose VoltDB Client service on each node, type NodePort only	31212
.loadBalancerIP	VoltDB Load Balancer IP	""
.loadBalancerSourceRanges	VoltDB Load Balancer Source Ranges	[ ]
.externalIPs	List of IP addresses at which the VoltDB service is available	[ ]
.http.sessionAffinity	SessionAffinity override for the HTTP service	ClientIP
.http.sessionAffinityConfig .clientIP.timeoutSeconds	Timeout override for http.sessionAffinity=ClientIP	10800

Parameter	Description	Default
.dr.type	VoltDB DR service type, valid options are ClusterIP (default), LoadBalancer, or NodePort	""
.dr.annotations	Additional custom Service annotations	{ }
.dr.availableIPs[ ]	(OBSOLETE as of 1.6.0)	[ ]
.dr.staticIP	Single static IP for DR service use when creating LoadBalancers single DR service	``
.dr.enabled	Create single DR service for DR	false
.dr.externalTrafficPolicy	VoltDB DR service external traffic policy	""
.dr.replicationPort	VoltDB DR replication exposed Service port	5555
.dr.replicationNodePort	VoltDB DR port to expose VoltDB replication service on each node, type NodePort only. If -1 is specified port 5555 will be used.	31555
.dr.servicePerPod	(OBSOLETE as of 1.6.0)	false
.dr.publicIPFromService	Operator will wait to get the public IP address from the service status set by Kubernetes	false
.dr.override	Allows per-pod-service overrides of serviceSpec	[ ]
.dr.override[ ].podIndex	(OBSOLETE as of 1.6.0)	""
.dr.override[ ].annotations	(OBSOLETE as of 1.6.0)	""
.dr.override[ ].publicIP	(OBSOLETE as of 1.6.0)	""
.dr.override[ ].spec	(OBSOLETE as of 1.6.0)	{ }
.dr.override[ ].spec.type	(OBSOLETE as of 1.6.0)	""
.dr.override[ ].spec.loadBalancerIP	(OBSOLETE as of 1.6.0)	""
.dr.override[ ].spec.externalIPs	(OBSOLETE as of 1.6.0)	[ ]
.perpod.type	VoltDB service type, valid options are ClusterIP (default), LoadBalancer, or NodePort	""
.perpod.publicIPFromService	Operator will wait to get the public IP address from the service status set by Kubernetes	false
.perpod.staticIPs[ ]	Available IPs and IP-ranges to use when creating LoadBalancers on a per-pod basis	[ ]
.perpod.dr.enabled	Enable DR services on a per-pod basis	false

Parameter	Description	Default
.perpod.dr.replicationPort	VoltDB DR port to expose VoltDB replication service on each node, type NodePort only	5555
.perpod.dr .startReplicationNodePort	VoltDB DR node port to use for VoltDB replication service on each node, type NodePort only. Start port indicates starting port and each pod gets subsequent port. If specified -1 port 5555 will be used	32555
.perpod.dr.externalTrafficPolicy	VoltDB DR service external traffic policy for per pod DR services.	""

A.5. VoltDB Database Startup Options

The following properties affect how Helm interacts with the VoltDB cluster adn specific initialization options, such as the initial schema and procedure classes.

Table A.4. Options Starting with cluster.config...

Parameter	Description	Default
.auth.username	User added for operator VoltDB API communication when hash security is enabled	voltdb-operator
.auth.password	Password added for operator VoltDB API communication when hash security is enabled	""
.schemas	Map of optional schema files containing data definition statements	{ }
.classes	Map of optional jar files container stored procedures	{ }
.licenseXMLFile	VoltDB Enterprise license.xml	{ }
.log4jcfgFile	Custom Log4j configuration file	{ }

A.6. VoltDB Database Configuration Options

The following properties define the VoltDB database configuration.

Table A.5. Options Starting with cluster.config.deployment...

Parameter	Description	Default
.cluster.kfactor	K-factor to use for database durability and data safety replication	1
.cluster.sitesperhost	SitesPerHost for VoltDB Cluster	8
.heartbeat.timeout	Internal VoltDB cluster verification of presence of other nodes (seconds)	90

Parameter	Description	Default
.partitiondetection.enabled	Controls detection of network partitioning	true
.commandlog.enabled	Command logging for database durability (recommended)	true
.commandlog.logsize	Command logging allocated disk space (MB)	1024
.commandlog.synchronous	Transactions do not complete until logged to disk	false
.commandlog.frequency.time	How often the command log is written, by time (milliseconds)	200
.commandlog.frequency .transactions	How often the command log is written, by transaction command	2147483647
.dr.id	Unique cluster id, 0-127	0
.dr.role	Role for this cluster, currently the only accepted value is 'xdcr'	xdcr
.dr.conflictretention	Automatic pruning of xdcr conflict logs; value is integer followed by one of m/h/d, for minutes/hours/days	""
.dr.connection.enabled	Specifies whether disaster recovery is enabled	false
.dr.connection.source	If role is replica or xdcr: list of host names or IP addresses of remote node(s)	""
.dr.connection.preferredSource	Cluster ID of preferred source	""
.dr.connection.ssl	Certificate file path for DR consumer (replica or xdcr mode), defaults to truststore.file location /etc/voltdb/ssl/certificate.txt when SSL is enabled, otherwise it must be specified	""
.dr.consumerlimit.maxsize	Enable DR consumer flow control either maxsize or maxbuffers must be specified maxsize can be specified as 50m, 1g or just number for bytes	""
.dr.consumerlimit.maxbuffers	Enable DR consumer flow control either maxsize or maxbuffers must be specified	""
.export.configurations	List of export configurations	[ ]
.import.configurations	List of import configurations	[ ]
.avro.namespace	Avro namespace	""
.avro.registry	Avro registry URL	""
.avro.prefix	Avro configuration prefix	""

Parameter	Description	Default
.avro.properties	Avro configuration properties	{ }
.topics.threadpool	Kafka topics threadpool to use	``
.topics.enabled	Kafka topics enabled or not	true
.topics.broker	Kafka topics broker configuration	``
.topics.broker.properties	Kafka topics broker configuration properties	[ ]
.topics.topic	List of topics	[ ]
.topics.topic.name	topic name	``
.topics.topic.procedure	Procedure to invoke upon getting message	``
.topics.topic.format	Format of topic message	``
.topics.topic.retention	Topic retention policy	``
.topics.topic.opaque	Is this an opaque topic	false
.topics.topic.allow	List of roles allowed to access the topic	``
.topics.topic.priority	Priority for topics requests (if priority scheduling is enabled)	4
.topics.topic.properties	Topic configuration properties	[ ]
.httpd.enabled	Determines if HTTP API daemon is enabled	true
.httpd.jsonapi.enabled	Determines if jSON over HTTP API is enabled	true
.paths.commandlog.path	Directory path for command log	/pvc/voltdb/voltdbroot/command_log
.paths.commandlogsnapshot.path	Directory path for command log snapshot	/pvc/voltdb/voltdbroot/command_log_snapshot
.paths.droverflow.path	Directory path for disaster recovery overflow	/pvc/voltdb/voltdbroot/dr_overflow
.paths.exportcursor.path	Directory path for export cursors	/pvc/voltdb/voltdbroot/export_cursor
.paths.exportoverflow.path	Directory path for export overflow	/pvc/voltdb/voltdbroot/export_overflow
.paths.largequeryswap.path	Directory path for large query swapping	/pvc/voltdb/voltdbroot/large_query_swap
.paths.snapshots.path	Directory path for snapshots. Must not be located in a read-only root directory of mounted storage (as init --force will rename existing snapshot folder). Use a subdirectory.	/pvc/voltdb/voltdbroot/snapshots
.security.enabled	Controls whether user-based authentication and authorization are used	false

Parameter	Description	Default
.security.provider	Allows use of external Kerberos provider; one of: hash, kerberos	hash
.snapshot.enabled	Enable/disable periodic automatic snapshots	true
.snapshot.frequency	Frequency of automatic snapshots (in s,m,h)	24h
.snapshot.prefix	Unique prefix for snapshot files	AUTOSNAP
.snapshot.retain	Number of snapshots to retain	2
.snmp.enabled	Enables or disables use of SNMP	false
.snmp.target	Host name or IP address, and optional port (default 162), for SNMP server	""
.snmp.authkey	SNMPv3 authentication key if protocol is not NoAuth	voltdbauthkey
.snmp.authprotocol	SNMPv3 authentication protocol. One of: SHA, MD5, NoAuth	SHA
.snmp.community	Name of SNMP community	public
.snmp.privacykey	SNMPv3 privacy key if protocol is not NoPriv	voltdbprivacykey
.snmp.privacyprotocol	SNMPv3 privacy protocol. One of: AES, DES, 3DES, AES192, AES256, NoPriv	AES
.snmp.username	Username for SNMPv3 authentication; else SNMPv2c is used	""
.ssl.enabled	Enables TLS/SSL security for the HTTP port (default 8080, 8443)	false
.ssl.external	Extends TLS/SSL security to all external ports (default admin 21211, client 21212)	false
.ssl.internal	Extends TLS/SSL security to the internal port (default 3021)	false
.ssl.dr	Extends TLS/SSL security to the DR port (5555)	false
.ssl.keystore.file	Keystore file to mount at the keystore path	""
.ssl.keystore.password	Password for VoltDB keystore	""
.ssl.truststore.file	Truststore file to mount at the truststore path	""
.ssl.truststore.password	Password for VoltDB truststore	""
.systemsettings.elastic.duration	Target value for the length of time each rebalance transaction will take (milliseconds)	50

Parameter	**Description**	**Default**
.systemsettings.elastic.throughput	Target value for rate of data processing by rebalance transactions (MB)	2
.systemsettings.flushinterval .minimum	Interval between checking for need to flush (milliseconds)	1000
.systemsettings.flushinterval.dr .interval	Interval for flushing DR data (milliseconds)	1000
.systemsettings.flushinterval .export.interval	Interval for flushing export data (milliseconds)	4000
.systemsettings.procedure .copyparameters	if set, mutable array parameters should be copied before processing	true
.systemsettings.procedure.loginfo	Threshold for long-running task detection (milliseconds)	10000
.systemsettings.query.timeout	Timeout on SQL queries (milliseconds)	10000
.systemsettings.priorities.enabled	Enables priority scheduling of requests by VoltDB cluster (true/false)	false
.systemsettings.priorities .maxwait	Modifies priority scheduling by setting a limit on time waiting while higher priority requests execute (millisecs)	1000
.systemsettings.priorities.batch	Modifies priority scheduling algorithm to execute multiple requests before rescheduling	25
.systemsettings.priorities.dr .priority	Priority for DR requests (1-8, 1 is highest priority)	5
.systemsettings.priorities .snapshot.priority	Priority for snapshot requests (1-8, 1 is highest priority)	6
.systemsettings.resourcemonitor .frequency	Resource Monitor interval between resource checks (seconds)	60
.systemsettings.resourcemonitor .memorylimit.size	Limit on memory use (in GB or as percentage)	80%
.systemsettings.resourcemonitor .memorylimit.alert	Alert level for memory use (in GB or as percentage)	70%
.systemsettings.resourcemonitor .disklimit.commandlog.size	Resource Monitor disk limit on disk use (in GB or percentage, empty is unlimited)	""
.systemsettings.resourcemonitor .disklimit.commandlog.alert	Resource Monitor alert level for disk use (in GB or as percentage, empty is unlimited)	""
.systemsettings.resourcemonitor .disklimit.commandlogsnapshot .size	Resource Monitor disk limit on disk use (in GB or percentage, empty is unlimited)	""

Parameter	Description	Default
.systemsettings.resourcemonitor .disklimit.commandlogsnapshot .alert	Resource Monitor alert level for disk use (in GB or as percentage, empty is unlimited)	""
.systemsettings.resourcemonitor .disklimit.droverflow.size	Resource Monitor disk limit on disk use (in GB or percentage, empty is unlimited)	""
.systemsettings.resourcemonitor .disklimit.droverflow.alert	Resource Monitor alert level for disk use (in GB or as percentage, empty is unlimited)	""
.systemsettings.resourcemonitor .disklimit.exportoverflow.size	Resource Monitor disk limit on disk use (in GB or percentage, empty is unlimited)	""
.systemsettings.resourcemonitor .disklimit.exportoverflow.alert	Resource Monitor alert level for disk use (in GB or as percentage, empty is unlimited)	""
.systemsettings.resourcemonitor .disklimit.snapshots.size	Resource Monitor disk limit on disk use (in GB or percentage, empty is unlimited)	""
.systemsettings.resourcemonitor .disklimit.snapshots.alert	Resource Monitor alert level for disk use (in GB or as percentage, empty is unlimited)	""
.systemsettings.resourcemonitor .disklimit.topicsdata.size	Resource Monitor disk limit on disk use (in GB or percentage, empty is unlimited)	""
.systemsettings.resourcemonitor .disklimit.topicsdata.alert	Resource Monitor alert level for disk use (in GB or as percentage, empty is unlimited)	""
.systemsettings.snapshot.priority	Priority for snapshot work (really a delay factor; see under systemsettings for scheduling priority)	6
.systemsettings.temptables .maxsize	Limit the size of temporary database tables (MB)	100
.systemsettings.clockskew .interval	Interval of the scheduled clock skew collection (minutes). 0 is allowed and it disables collection. Interval cannot be less than 0 and if set below such value it will be reset to default.	60
.users	Define a list of VoltDB users to be added to the deployment	[ ]

A.7. Metrics Configuration Options

The following properties start and configure the VoltDB Prometheus agent, used for reporting metrics to the Prometheus server.

Table A.6. Options Starting with metrics...

Parameter	Description	Default
.enabled	Create VoltDB metrics exporter agent to manage clusters	false
.image.registry	Image registry	docker.io
.image.repository	Image repository	voltdb/voltdb-prometheus
.image.tag	Image tag	latest
.image.pullPolicy	Image pull policy	Always
.manageSecrets	Variable names for existing metric secrets	
.user	Metric service user id used to access VoltDB. This value and the password value must exist in VoltDB. Superseded by credSecretName when provided. Required when security is enabled in VoltDB.	""
.password	Metric service password used to access VoltDB. Superseded by credSecretName when provided. Required when security is enabled in VoltDB.	""
.ssl	SSL configuration settings for metrics.	
.manageSecrets.credSecretName	Variable name for the secret containing metric username and password. This overrides user and password values below.	
.ssl.enabled	Enable SSL for metrics	
.ssl.keystore.password	Password for the keystore file	
.ssl.keystore.file	Contents of the keystore file	
.ssl.truststore.password	Password for the trust store file	
.ssl.truststore.file	Contents of the trust store file	
.port	VoltDB API port, uses 21211 if not specified	""
.stats	Comma-separated list of stats to collect from the VoltDB database	""
.skipstats	Stats to ignore when collecting all stats, cannot be used in conjunction with stats	""
.delta	If 'true', collects 'delta' stats where available, rather than cumulative stats	""
.resources	CPU/Memory resource requests/limits	{ }