Deploying and Managing VoltDB Databases With Kubernetes

V8.3

October, 2018

1. Introduction

This document explains how to use Kubernetes and Docker to run and manage VoltDB databases in a virtualized environment. This is *not* a tutorial on Kubernetes, Docker, or VoltDB. You should be familiar with all three technologies before using this document.

1.1. Overview of the Kubernetes Deployment Process

Kubernetes is an orchestration tool for managing applications and services run in containers. The key advantage of Kubernetes is that it provides automated management functions for distributed applications such as VoltDB.

Kubernetes is very flexible and customizable; there are many different ways you could use it to host a VoltDB database. This document and the scripts it describes demonstrate one recommended way to use VoltDB with Kubernetes that has been tested and verified.

The process uses a build script within the installed VoltDB kit (`tools/kubernetes/build_image.sh`) to build a Docker image that includes everything needed to start and run VoltDB including:

- The VoltDB kit itself
- Database assets such as the schema, stored procedures, configuration file, and VoltDB license
- Special scripts to start the VoltDB instance and, if it is not already present, save the database root directory to persistent storage within Kubernetes

You specify the assets to include in a configuration file. By default, the name of the configuration file is used for all output files, for the Docker image, and the Kubernetes cluster itself. It is strongly recommended that you include the database schema and stored procedure classes as part of the configuration. If not, the database will start without any schema and you will need to manually load the schema after it starts.

The scripts ensure your assets are included in the Docker image, copied to the appropriate locations within the Docker containers, and that the voltdbroot directory is saved to persistent storage by Kubernetes. The use of persistent storage means the root directory is preserved, even when individual pods or the cluster as a whole goes down. This supports VoltDB availability features, such as command logging and automated snapshots, that save and restore the database contents across sessions.

1.2. Requirements

Before starting the process described in this document, make sure your development system and account meet the following requirements:

- Installed:
  - VoltDB 8.3 or later
  - Kubernetes command line (kubectl) 1.10.4 or later

  - Docker[1]

- Copies of:
  - Database configuration file and license (if using the Enterprise Edition)
  - Optionally, the database schema and JAR file containing stored procedure classes

- Access to:
  - A private Docker registry
  - The target Kubernetes hosting environment

A private Docker repository is recommended since the Docker image contains both your schema and your VoltDB license. If you are using the community edition and do not consider your schema proprietary, use of a public repository is an option.

In addition to the preceding requirements for your development system, your Kubernetes hosting environment must also have access to the Docker repository you are using (or another repository the compiled Docker image will be copied to) and have the ability to instantiate persistent storage volumes that will be used to retain the VoltDB root directories and their contents when pods are deleted and recreated.

2. Special Considerations for Running VoltDB in a Virtualized Environment

VoltDB, like all memory-intensive applications, is sensitive to base system settings that control access to assets such as memory, CPU, and networking. Memory configuration options such as swapping and Transparent Huge Pages (THP) can dramatically impact the performance of a VoltDB database. In extreme cases, THP can result in unnecessary allocations of memory to the point where the server process runs out of memory. This is why VoltDB recommends turning off THP.

The VoltDB Administrator's Guide contains a chapter on how to prepare a server for optimal use with VoltDB, including disabling THP. However, in a virtualized environment — and particularly when using containers — you do not have direct access to the base system's core settings from within the virtual machine (VM) or container. Therefore, to ensure proper and consistent operation of VoltDB in such environments, you must make the necessary adjustments on the base system itself.

If you are running VMs on hardware you manage, this can be done by configuring the servers yourself. If, on the other hand, you are using a hosted environment, you should consider other alternatives. You can negotiate with the hosting provider to ensure the servers are configured correctly. Alternately, you can use privileged mode VMs (essentially, dedicated servers reserved for your account) and Kubernetes features to modify the base system before any containers are run.

The following are some resources that can help you make the appropriate decision about your configuration choices:

- Stack Overflow discussion of using Kubernetes' init-containers and DaemonSets to disable Transparent Huge Pages

- Paul Done's blog post on configuring Kubernetes and hosted VMs for running MongoDB

3. Deploying VoltDB in Kubernetes

Use the following steps to deploy a VoltDB cluster in Kubernetes:

[1]The required version of Docker is dependent on the version of Kubernetes and the choice of base OS to be used. See the Kubernetes release notes for details.

1. If not already installed, install VoltDB.

2. Copy and customize the template configuration file found in the `tools/kubernetes` folder where VoltDB is installed. The name of the configuration file will be used to identify the cluster and all output files. So, for example, if you want to create a cluster called `boston`, you might start with the following commands:

   ```
   $ cd /opt/voltdb/tools/kubernetes
   $ cp config_template.cfg boston.cfg
   ```

   The edited configuration file identifies the location of your Docker repository, database assets, license file, and so on. It also specifies the size of the cluster. The edited file might look like this:

   ```
   # DOCKER IMAGE REPOSITORY -- the image will be pushed to this repo
   REP=grc.io/mydb-12345678901234
   # DOCKER IMAGE TAG - default is the cluster name
   #IMAGE_TAG=
   NODECOUNT=3
   MEMORY_SIZE=
   CPU_COUNT=
   PVOLUME_SIZE=1Gi
   LICENSE_FILE=~/license.xml
   DEPLOYMENT_FILE=~/boston/configuration.xml
   SCHEMA_FILE=~/boston/mydb.sql
   CLASSES_JAR=~/boston/mydb.jar
   ```

   See the next section, Section 3.1, “Customizing the Configuration File”, for more information about the specific settings available in the configuration file.

3. Run the build script `build_image.sh` to build, tag, and push your Docker image to the Docker repository based on your modified configuration file. The build script also creates a custom YAML file for starting the cluster in Kubernetes. The Docker image, the cluster in Kubernetes, and the YAML file all take their name from the name of the customized configuration file you specify as input:

   ```
   $ ./build_image.sh boston.cfg
   ```

4. Start the cluster using the **kubectl** command line and YAML file created in Step #3:

   ```
   $ kubectl create -f boston.yaml
   ```

Note that the cluster nodes, or pods, derive their host names from the tag name or, by default, the configuration file name. So to run multiple VoltDB clusters in the same Kubernetes cluster, you simply need to define and start each cluster using a distinct configuration file name so the host names are distinct.

3.1. Customizing the Configuration File

The configuration file includes variables for setting the most common attributes needed for running VoltDB. Table 1, “Configuration File Options” describes each of the variables in more detail. Unless otherwise noted, all variables are required.

Table 1. Configuration File Options

Option	Description
REP	The Docker repository where the resulting docker image is stored.

Option	Description
IMAGE_TAG	The Docker image tag where the image is stored. Optional. The default tag name is the name of the configuration file.
NODECOUNT	The number of nodes, or pods, in the cluster.
MEMORY_SIZE	The amount of memory to request from Kubernetes for each pod. You can specify the size as a value and a unit (such as "Gi" for gigabyte). The default unit is bytes. If you do not specify a memory size, the default size is four gigabytes.
CPU_COUNT	The number of CPU units to request from Kubernetes. If you do not specify the number of CPU units, the default is two CPU. You can set the CPU count as a fixed number or a level of CPU resource. See the Kubernetes documentation for details.
PVOLUME_SIZE	The amount of storage to request from Kubernetes for the persistent storage volumes for each pod. The script sets a default size of one gigabyte.
LICENSE_FILE	The location of the license file. The default is the `voltdb` subfolder in the VoltDB installation. Required when using the VoltDB Enterprise Edition.
DEPLOYMENT_FILE	The location of the VoltDB configuration file.
SCHEMA_FILE	The location of a schema definition file (DDL statements) to be loaded when the database is started for the first time. Optional.
CLASSES_JAR	The location of a JAR file of classes (usually stored procedures) to load when the database is started for the first time. Optional.
EXTENSION_DIR	The location of extensions (JAR files) to include when the server process starts. The default is the extensions subfolder of the current installation.
BUNDLES_DIR	The location of OSGi (Open Service Gateway Initiative) bundles accessible to the server process. OSGi bundles are usually used for custom import connectors. The default is the bundles subfolder of the current installation.
LOG4J_CUSTOM _FILE	The location of a custom Log4J configuration file. The default in no configuration file. That is, the default VoltDB Log4J configuration.

3.2. Further Customization Options

Of course, if you need to perform additional customizations beyond what the configuration file supports, you are free to customize the configuration file, the build script, the custom Dockerfile (in `tools/kubernetes/docker/Dockerfile`), or the VoltDB installation itself to meet your needs. For example, you can edit the Dockerfile to modify the resulting Docker image or change the YAML template file (`voltdb-statefulset.yaml`) to customize the resulting Kubernetes cluster.

Warning

You are free to modify the files in the `tools/kubernetes/` folder and its subfolders. However, do not *move, rename, or delete* any files other than the custom configuration files you create for your application. If you do, the build process will not function properly.

One customization you might consider is changing the base OS used in the Docker configuration file. By default, VoltDB uses Ubuntu 16.04 as the base OS. You can edit the Dockerfile, in `tools/kubernetes/docker/Dockerfile`, to specify a different base OS if you choose.

4. Managing VoltDB with Kubernetes

The key advantage of Kubernetes is that once you define and start your cluster (or *stateful set* in Kubernetes terminology) Kubernetes takes responsibility for maintaining the cluster state. If a pod (that is, a node in your cluster) goes down, Kubernetes will automatically recreate it. In a K-safe VoltDB cluster, this means the node will rejoin the cluster. If the cluster as a whole goes down, Kubernetes will restart all of the pods and the database will recover using whatever features you configured; such as replaying command logs or automatically restoring automated snapshots.

Many of the basic operations for managing a VoltDB database are the same whether running on a physical server or running within Kubernetes. Schema changes, backups, monitoring can all be done either by creating a connection to the pods (using a command such as **kubectl exec -it**) or creating proxies to a local machine from the internal Kubernetes network (through commands such as **kubectl port-forward**). However, in certain cases there are differences. The following sections describe some common VoltDB management tasks and how to perform them within a Kubernetes environment. Section 6, “Setting Up Database Replication with Kubernetes” discusses special considerations when using Kubernetes to run cluster using passive database replication (DR) or cross datacenter replication (XDCR).

4.1. Starting the Database For the First Time

As mentioned before, the main purpose of using Kubernetes is to allow Kubernetes to start and maintain the database cluster without intervention. To start the cluster for the first time, you use the YAML file created by the build script as described in Section 3, “Deploying VoltDB in Kubernetes”:

```
$ kubectrl create -f boston.yaml
```

If this is the first time the database is run, persistent storage volumes are created for each pod in the cluster and a initialized database root directory is copied to the persistent storage. After the first start, any subsequent restarts will pick up the database state from the persistent storage.

4.2. Monitoring the Database

You can use any number of techniques to monitor VoltDB databases, whether they are being managed by Kubernetes or not. For example, you can use SNMP traps to notify operators about error conditions or apply monitoring tools such as Nagios to integrate VoltDB monitoring with the monitoring of other systems. In addition, Kubernetes adds two monitoring "probes" of its own.

To automate the management of your applications, Kubernetes performs periodic health checks on the pods. The two key Kubernetes health checks, or probes, are *liveness* and *readiness*.

- **Liveness** tells Kubernetes whether the pod is still viable and the application is still running. If a pod's liveness probe returns false, Kubernetes will restart the pod in an attempt to return it to normal operation.
- **Readiness** is an indicator of whether the application itself is operating properly. Kubernetes does *not* automatically restart pods that are not ready. But it does show the readiness state on the dashboard and will not include the pod in certain activities (such as load balancing) if it is not ready.

By default, the VoltDB liveness probe tests to see that the internal port is open, the port the cluster uses to communicate internally. This indicates that the server process is up and running. It does *not* indicate the operational state on the database itself.

The default readiness probe tests to see if the client port is open, the port client applications use to interact with the database. This indicates that the database itself is running and any startup tasks, such as restoring snapshots or command logs, have been completed.

It should be noted that by default Kubernetes does not allow a pod to expose network ports until the application is "ready". However, VoltDB needs to expose ports to establish the cluster mesh before it can be ready for normal operation. So, VoltDB overrides the default readiness requirement in this respect. This allows the cluster to form, recover existing data, then open the client port before the practical definition of readiness is reported.

4.3. Performing an Orderly Shutdown

After you start the cluster for the first time Kubernetes takes responsibility for keeping the database up and running. As a result, if you use the normal procedure for stopping a database, which is the **voltadmin shutdown** command, the database stops and immediately restarts again.

For those cases where you actually want to perform an orderly shutdown and stop the database, what you need to do is stop the Kubernetes stateful set. You do this with the following process:

1. Pause the database, using the **voltadmin pause --wait** command.

2. Wait for the pause operation to finish, to ensure all changes have been saved to the command logs in persistent storage. If you are not using command logs, you will want to take a final snapshot as well, using the **voltadmin save --blocking** command. Do *not* shutdown the database.

3. Delete the stateful set using the **kubectl delete** command, specifying the YAML definition of the stateful set. For example:

   ```
   $ kubectl delete -f boston.yaml
   ```

4.4. Manually Restarting the Database

If you use the preceding process for shutting down the cluster, the cluster state — including the schema, data, and configuration — is saved to persistent storage. To restart the database, you simply need to recreate the Kubernetes stateful set using the **kubectl create** command:

```
$ kubectl create -f boston.yaml
```

Kubernetes restarts the database cluster, which automatically picks up the previous database state from persistent storage. If you want to restart the cluster from scratch, without any schema or data, you must delete the persistent storage associated with the pods prior to issuing the **kubectl create** command. However, extreme caution must be exercised when doing this, since deleting the persistent storage is the equivalent of issuing a **voltdb init --force** command: all existing data will be deleted when you do this.

4.5. Restarting Individual Nodes

There are times when you want to "reboot" individual nodes of a K-safe cluster. This procedure can be helpful to clear up certain unusual conditions or stalled processes.

To restart individual nodes in Kubernetes, you use the same process as in other environments: you use the **voltadmin stop** command, specifying the name of the node you want to reset. For example:

```
$ voltadmin stop boston-1
```

Note that as soon as the VoltDB process stops for the specified node, the pod will stop and Kubernetes will automatically recreate it, causing the node to rejoin the VoltDB cluster. You do not need to manually rejoin the node.

4.6. Updating the Database Schema

You update the database schema — to add, modify, or deletes tables and so on — using the same commands as normal. For example, you can connect to one of the cluster nodes and use **sqlcmd** to execute DML statements. Similarly, you can update stored procedure definitions and class files. These changes are stored in the database root directory along with the command logs and/or snapshots, which are maintained by the persistent storage volumes. So changes persist across database sessions.

Note, however, as in any production environment you should synchronize any changes to the database schema with the original source files. So, for example, be sure to create and upload a new Docker image containing the schema updates whenever you make changes.

4.7. Updating the Database Configuration

Changes to the configuration, like changes to the schema, are persisted in the database root directories. Any changes to the configuration that you can make to a running cluster using the **voltadmin update** command or the VoltDB Management Center (VMC) web interface are retained across stops and starts of both pods and the database as a whole. As long as the persistent storage volumes are retained, the database context is retained. Again, be sure to rebuild and reload your Docker image with any changes you make to the configuration file.

4.8. Upgrading the VoltDB Software

Upgrading the VoltDB software is different in a Kubernetes environment than on a dedicated server, since the VoltDB kit is built into the Docker image. How you upgrade in Kubernetes depends on the features you are using.

4.8.1. Upgrading a Standalone Cluster

To update a single cluster, use the following process:

1. Install the new VoltDB software kit on your development system.
2. Copy the customized configuration and YAML file from the old installation to the new one.
3. Follow the steps in Section 3, “Deploying VoltDB in Kubernetes” for building and posting the Docker image to the repository. At this point, the tagged image in the repository contains the new VoltDB version.
4. Connect to the Kubernetes cluster and use the **voltadmin shutdown --save** command.

Step #4 creates a final snapshot in the VoltDB root directory. It then shuts down the cluster. At which point Kubernetes will recreate the cluster and, as it comes back up, it will restore the schema and data from the final snapshot into the new Docker image.

4.8.2. Upgrading Clusters Using Cross Datacenter Replication (XDCR)

In a XDCR environment, upgrading the clusters is achieved with the following steps for each cluster, one at a time:

1. Build a new Docker image with the current database assets (license, configuration, schema, etc) and the new VoltDB software.
2. Use **kubectl** to identify the specific persistent storage volumes associated with the cluster being upgraded.

3. Shutdown the cluster as described in Section 4.3, “Performing an Orderly Shutdown”.

4. Drop the cluster from the XDCR environment by connecting to another cluster in the environment and issuing the **voltadmin dr reset --cluster={cluster-ID}** command. Specify the cluster ID of the cluster being upgraded.

5. Delete the persistent storage identified in Step #2 as associated with the cluster. This ensures that the new cluster will start initialized with no data.

6. Restart the cluster as described in Section 4.4, “Manually Restarting the Database”.

When the cluster restarts, it connects to the other XDCR clusters. Since it was dropped from the environment, it will join as a new cluster and receive a snapshot containing a copy of all of the current data. Once the snapshot is completed, the cluster is a full working member of the XDCR environment and you can proceed to upgrade the next cluster.

5. Connecting to a VoltDB Database in Kubernetes

There are many different ways to connect to a service within Kubernetes. The simplest approach is to run the client application within the same Kubernetes subnet as the VoltDB cluster itself, for example as a Kubernetes headless service. In this configuration your client application can refer to the VoltDB cluster servers by their hostnames alone.

If your client is running outside of the Kubernetes cluster, you must use a different approach to connecting to the database cluster. The key is to find a persistent IP address that both the client and server agree on as the public Internet interface for the server.

You can open a port on the underlying hardware (NodePort) to the server or establish a separate load balancing service that is accessible externally. But, quite frankly, none of the generic options available in Kubernetes provide reliable performance guarantees without significant customization.

There are essentially two approaches to connecting to a VoltDB database running within a Kubernetes environment:

- Opening up an externally accessible port and IP address to which applications connect

- Tunneling in to the Kubernetes internal subnet and connecting to the internal IP addresses

Opening up an IP address and port externally to the individual pods in a VoltDB usually involves ephemeral addresses that do not match the addresses the cluster itself knows about or requires significant privileges and special functions within the hosting environment. In almost all cases this means performance optimizations in the client — such as topology awareness and client affinity — cannot operate properly and performance will suffer consequently.

For client affinity and topology awareness to operate properly, the client application needs to have access to IP addresses for each of the nodes in the VoltDB cluster and the cluster needs to be able to advertise those addresses either directly or through the externalinterface option. One reliable way to achieve this is to either run the client application within the same Kubernetes cluster or provide a secure tunnel into the Kubernetes subnet. This approach has several advantages:

- It allows the client to connect to the cluster nodes by host name

- The host names are persistent and definable

- Client features such as topology awareness and client affinity operate properly

This is also true for database replication (DR) and XDCR) where the consumer needs to be able to identify appropriate producer nodes for each partition. Using a private tunnel or other mechanism to have all clusters within a single XDCR relationship running within the same logical subnet significantly reduces the complexity of establishing the necessary network relationship between clusters.

6. Setting Up Database Replication with Kubernetes

The previous discussion of defining and starting VoltDB in Kubernetes focuses on individual clusters. VoltDB also supports database replication (DR), where data from one database is replicated to another (master/replica), or cross datacenter replication (XDCR) where two or more database clusters are actively processing read and write transactions and sharing the results.

In most cases DR and XDCR are used to connect geographically disparate clusters. In other words, clusters in different Kubernetes instances, regions, or possibly between Kubernetes and completely different hosting environments. In these cases, special consideration needs to be taken about how the distributed clusters establish their connections.

6.1. Requirements for DR in Kubernetes

To establish DR between two VoltDB clusters, the nodes of the consumer cluster must be able to create a network connection to the IP addresses of all the producer cluster nodes. In passive DR, this means the replica cluster must be able to resolve the IP addresses of the master cluster. XDCR clusters act as both consumer and producer, so all clusters must be able to resolve the IP addresses of other clusters in the XDCR relationship.

When setting up DR between clusters in different Kubernetes clusters or between Kubernetes and another production environment, it is important that your network topology meets this requirement; for example, by using a virtual private cloud (VPC) or virtual Local area network (VLAN). Which of these features is available to you and how you instantiate them is dependent on your Kubernetes provider and your agreement with them.

A second requirement is that there must be at least one persistent network address associated with the producer cluster that the consumer can use to "discover" the producer's IP addresses. Establishing the DR connection at runtime occurs in two steps:

1. First, a consumer cluster contacts the producer through a network address specified in the VoltDB configuration file (using the `<dr>` and `<connection>` tags). The producer responds with the list of IP addresses and port numbers for its nodes.

2. The first cluster then drops the initial connection and opens connections (one per partition) to nodes on the producer cluster using the list of IP addresses received in step #1.

The IP address used in step #1 needs to be static so it can be defined in the configuration file before the clusters start. On the other hand, the IP addresses Kubernetes assigns to individual pods used in step #2 are temporal and can change.

Fortunately, the addresses used in the configuration file to find the producer cluster and the addresses returned by the producer at runtime do not have to be the same. One way to bridge this difference is to use a Kubernetes load balancer to connect a static IP address to the producer cluster. The consumer cluster can configure its DR connection using the static IP address of the load balancer and then resolve the actual IP addresses returned at runtime using a virtual network.

Finally, it is important to note that performance of the network dramatically impacts the performance of data replication and VoltDB itself. For best performance, you must use networking resources that can support the peak bandwidth requirements of the replication stream. Hosting services often provide networking features with different performance tiers. Inadequate network bandwidth can result in bottlenecks, backlogs, and ultimately, if not addressed, the DR connection itself may be broken. So use of high bandwidth cross-zone or cross-region networking is recommended when supporting DR.

6.2. One Approach for Resolving DR Connections

As mentioned before, what approach you use to resolve IP addresses of clusters in different regions is up to you and can depend on what capabilities are available in your specific environment. The following is one practical approach.

Virtual private clouds (VPC) allow Kubernetes clusters in disparate regions to share a network topology. This allows a VoltDB cluster in one region to resolve the IP addresses of a cluster in another region, as required for DR and XDCR. Several hosting platforms support the use of VPCs within their hosted environment.

However, the IP addresses of the pods themselves are ephemeral. So the clusters do not know in advance (before the initial DR connection) what the IP addresses of the other clusters are. So using a VPC, the consumer cluster cannot reliably use the internal addresses to establish the initial connection (in the configuration file).

There is a way to work around this limitation. It so happens that the IP address used by the consumer to make the initial connection does not need to be the same as any of the addresses received in response from the producer. Therefore, you can use a load balancer in the producer's Kubernetes cluster to expose the DR port (port 5555 by default) to a persistent external IP address. The consumer cluster can use this external address in the `<connection>` definition of the configuration file to establish the initial DR connection.

For example, say there are two VoltDB clusters, *boston* and *dallas*, hosted in two regions (*region-1* and r*egion-2*, respectively) by the same Kubernetes provider. A VPC is set up between the Kubernetes clusters in the two regions. The DR port of the VoltDB clusters are also exposed externally to the IP addresses *myhost.region-1.mycompany.boston* and *myhost.region-2.mycompany.dallas* through the use of a load balancer. The configuration files for the two clusters can then be defined as follows to set up and maintain an XDCR relationship between the two clusters.

boston

```
<dr id="1" role="xdcr">
   <connection source="myhost.region-2.mycompany.dallas" />
</dr>
```

dallas

```
<dr id="2" role="xdcr">
   <connection source="myhost.region-1.mycompany.boston" />
</dr>
```