Upgrade Guide

Abstract

This books explains how to upgrade from one major version of Volt Active Data to another.

V14

Upgrade Guide

V14

This document was generated on March 27, 2025.

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

This book helps existing customers upgrade the Volt Active Data software from one major version to another. New features and improvements that require changes to existing syntax or client applications are reserved for major releases of the product only (e.g. V8, V9, V10, and so on). To make the upgrade to these major versions smoother for existing users, this book describes any incompatible changes for each major version and the steps necessary when migrating to it from previous releases.

The best way to use this book is before you upgrade to a new major version, read the chapters for each of the intervening releases and take the appropriate actions. For example, if you are planning to upgrade from V8 to V11, you should read the chapters on migrating to V9, V10, and V11.

1. Structure of This Book

The first two chapters of this book describe the Volt Active Data release strategy and the upgrade process. The release strategy describes the schedule for specific releases and the role of the Long-Term Support release for each major version. The upgrade process gives you an overview of the steps to take when preparing to upgrade to a new major version of Volt Active Data. Subsequent chapters describe specific tasks for upgrading to each of the major releases. Finally, appendixes describe the old configuration file syntax used prior to version 14 and the original Client API, for those needing to maintain existing applications written against the older programming interface for VoltDB.

- Chapter 1, *Software Releases and Long-Term Support (LTS)*
- Chapter 2, *The Software Upgrade Process*
- Chapter 3, *Upgrading to Volt Active Data V8*
- Chapter 4, *Upgrading to Volt Active Data V9*
- Chapter 5, *Upgrading to Volt Active Data V10*
- Chapter 6, *Upgrading to Volt Active Data V11*
- Chapter 7, *Upgrading to Volt Active Data V12*
- Chapter 8, *Upgrading to Volt Active Data V13*
- Chapter 9, *Upgrading to Volt Active Data V14*
- Appendix A, *Configuration File (deployment.xml)*
- Appendix B, *Using the Original VoltDB Client Interface*

2. Related Documents

This book does not describe how to design or develop Volt Active Data databases. For a complete description of the development process for Volt and all of its features, please see the accompanying manual *Using VoltDB*. For new users, see the *VoltDB Tutorial*. These and other books describing Volt Active Data are available on the web from http://docs.voltactivedata.com/.

Chapter 1. Software Releases and Long-Term Support (LTS)

Volt Active Data provides best-in-class throughput of ACID transactions, enabling intelligent streaming applications and real-time decision making. To maintain its industry leadership and help customers meet current and future business goals, Volt is constantly improving and expanding the its capabilities by adding new features.

At the same time, it is important to provide the most stable software possible for production use. So Volt provides frequent product updates to ensure timely and proactive solutions to potential issues within the software. Balancing cutting edge features with bug fixes is the primary goal of the Volt Active Data release strategy and, specifically, its Long-Term Support (LTS) designation as described in the following section.

The Release Schedule The latest releases of Volt Active Data usually contain a combination of new features and bug fixes for known issues in previous releases. Releases fall into three categories: Major releases (V1.0, V2.0, V3.0 etc.) Major releases are normally reserved for significant new features. They are also where older features are deprecated or removed. In particular, any incompatible or behavioral changes that require user action to complete the upgrade process are reserved for major releases. Point releases (V3.1, V3.2, V3.3 etc.) Point releases are normally only applied to the latest version of the software or LTS releases. On the latest release branch, point releases contain both bug fixes and new features under development. On LTS branches, point releases contain only bug fixes. Long-Term Support releases (LTS) For each major release, after a few point releases to stabilize the product we announce one point release as the Long-Term Support (LTS) release. This means this version will receive all applicable bug fixes for the full term of its support period, which is three years from its initial release. These bug fixes will be applied as minor updates to the point release (for example, if V11.4 is the LTS release, bug fix updates will be V11.4.1, V11.4.2 etc.). Note that the LTS releases receive bug fixes only, they do not receive any additional new features. New features are applied to the latest release branch only. The goal of the LTS release is to clearly identify the most stable version available for any major release. If you encounter any issues with point releases prior to the LTS release, we will ask you to update to the LTS release to receive the fixes you need.

1.1. The Software Release Schedule

The latest releases of Volt Active Data usually contain a combination of new features and bug fixes for known issues in previous releases. Releases fall into three categories:

- **Major releases** (V1.0, V2.0, V3.0 etc.) Major releases are normally reserved for significant new features. They are also where older features are deprecated or removed. In particular, any incompatible or behavioral changes that require user action to complete the upgrade process are reserved for major releases.

- **Minor releases** (V3.1, V3.2, V3.3 etc.) Minor releases are normally only applied to the latest version of the software or LTS releases. On the latest release branch, minor releases contain both bug fixes and new features under development. On LTS branches, point updates to the minor release contain only bug fixes.

- **Long-Term Support releases** (LTS) For each major release, after a few minor releases to stabilize the product we announce one minor release as the Long-Term Support (LTS) release. This means this version will receive all applicable bug fixes for the full term of its support period, which is three years from its initial release. These bug fixes will be applied as point updates to the minor release (for example, if V11.4 is the LTS release, bug fix updates will be V11.4.1, V11.4.2 etc.). Note that, once named, the

LTS releases receive security and bug fixes only, they do not receive any additional new features. New features are applied to the latest release branch only.

The goal of the LTS release is to clearly identify the most stable version available for each major release. Installing the subsequent point releases ensures you have all the latest bug fixes and security patches for that LTS release. If you encounter any issues with minor releases prior to the LTS release, Volt Support will recommend updating to the LTS release to receive the fixes you need.

1.2. Staying Current

For customers who want to take advantage of the latest features of Volt Active Data, we recommend you install and test the latest version of the software. Your feedback is invaluable to the continual improvement of the product. In cases where your business application is dependent on features introduced in the latest major release, Volt provided timely updates to the latest version for bug fixes and functional improvements.

For customers focused on stability, we recommend working with the latest LTS release. Even though it may not be the most recent version, all LTS releases are maintained and patched as necessary to provide you with the most stable software platform possible. So it is important to keep up with the point releases of the LTS you are on to ensure the further stability of your systems. For planning purposes, the best practice is to install the latest LTS point release on a quarterly basis to receive the latest bug fixes and security patches available.

For customers who need both stability and cutting edge capabilities, we recommend using LTS releases in production and developing on the latest version. By doing planning, development and testing on the latest release, you are ensured you are ready to roll out the latest features into production as soon as a new LTS release is announced. As an added benefit, you have the opportunity to help influence new features to fit your business requirements before production starts.

Chapter 2. The Software Upgrade Process

Upgrading from one major version of Volt to another may involve changes to the supported features and/or configuration of the software. In addition, each cluster must reboot as part of the upgrade process. Which is why it is important you plan ahead before starting the upgrade process.

There are essentially two flavors of the upgrade process. You can upgrade an individual Volt cluster, which will require some minimal downtime while the cluster reboots. Or you can use cross datacenter replication (XDCR) to upgrade each cluster sequentially, so although each cluster must reboot, the service as a whole remains available throughout the upgrade process. The advantage of an XDCR upgrade is that your business systems do not incur any interruption.

Once you decide on your approach, there are three steps to the upgrade process:

1. Prepare for the upgrade by reviewing the applicable changes to the Volt software, making any necessary changes to your business applications and/or schema, saving a copy of your current configuration and any special settings, and backing up the database.
2. Test the upgrade on a development server to ensure your applications run as expected after the upgrade.
3. Perform the upgrade in the production environment.

Since the process is the same for upgrading a test environment or upgrading production systems, the process is only described once for each platform in the following sections. However, the process is different for Kubernetes compared to bare metal or servers hosted on generic virtual machines; so there are separate sections for each platform type.

Finally, it is a good idea to review your upgrade plans in advance with your Volt Customer Success representative. So we strongly encourage you to engage them during the preparation phase where they can help you evaluate your business solutions for any obsolete coding or features, as well as identifying new features or improvements that could provide significant benefits to your specific business case.

2.1. Preparing for the Upgrade

Before upgrading the Volt Active Data software to a new major version, you should:

1. **Read the upgrade notes** (later in the manual) for all major versions included in the update.
2. **Read the latest release notes** for the final version you are upgrading to.
3. **Update your applications and schema** accordingly.
4. **Test your changes** on updated development systems to verify they work as expected.
5. **Backup your database and schema** as a VoltDB snapshot.
6. **Make a record of your database and server configuration** to capture any customisations you have made to your implementation.

As the product evolves, features are extended to improve performance and enhance their capabilities. At the same time, old and obsolete features are sometimes removed from the product. These feature changes occur primarily at major releases. The later chapters in this book provide notes on such changes in each of the major versions of VoltDB. So before upgrading, be sure to read the chapters for each of the major versions involved in the upgrade. For example, if you are upgrading from V9 to V10, read the chapter on upgrading to V10. If you are upgrading from V9 to V12, read the chapters on upgrading to V10, V11, *and* V12.

There may be additional improvements found after the initial release of any major version. Which is why you should also read the latest release notes for any minor updates to your target version.

Once you know what product changes need to be addressed, compare the upgrade notes for the major versions with your current application source code and schema. Be sure to make appropriate revisions where necessary to avoid obsolete functions and to take advantage of new features where possible.

Once any changes are complete, it is a good idea to test your planned upgrade on a development system to ensure it works as expected.

When you are ready to start the upgrade, be sure to backup your database as a safety precaution. You can use the **voltadmin save** command to take a snapshot of the current database schema and contents.

You should also save a copy of the current database configuration. There are actually two parts to saving the configuration. While logged onto one of the database servers, you can save the current configuration file using the **voltdb get deployment** command[1], specifying the location of the database root directory:

```
$ voltdb get deployment -D ~/myvoltserver/ --output=config.xml
```

You should also make a note of any environment variables or command switches used when starting the database. In particular, make sure you capture the current settings for the following environment variables that can significantly impact the server process:

LOG4J_CONFIG_PATH
VOLTDB_GC_OPTS
VOLTDB_HEAPMAX
VOLTDB_OPTS

Once you complete the preceding steps, you are ready to perform the upgrade.

2.2. Upgrading Volt Databases on Managed Servers

For customers who manage their own servers, there are three options for upgrading the VoltDB software for an existing database:

1. Using the **voltadmin save** and **restore** commands before and after shutting down and upgrading the software.

2. Performing an *in-service upgrade* by stopping, upgrading, and rejoining nodes in a K-safe cluster one at a time, without having to shutdown the cluster as a whole.

3. Using two (or more) XDCR clusters to upgrade the clusters one at a time without a service interruption.

The following sections describe how to update the software itself and the three options for upgrading existing installations.

2.2.1. Upgrading the VoltDB Software

Updating the VoltDB software is very simple. However, you must make sure you perform this step at the right stage in the upgrade process, as described in the following sections. The product comes as a .tar.gz

[1] As of Volt Active Data V14, the configuration is saved in YAML format by default. However, earlier versions will generate the older XML format configuration files. This is not a problem because newer versions of the software still accept the older format configuration files for the **voltdb init** command.

file. When the time comes to upgrade the software, you unpack the tar file and move the resulting folder to replace your current installation. For example, if you have the VoltDB software installed as /var/voltdb, the software installation looks like the following, where you delete the previous version and replace it with the new one:

```
$ tar -zxvf voltdb-ent-n.n.n-xxxx.tar.gz -C /var
$ cd /var
$ rm -vr voltdb
$ mv voltdb-ent-n.n.n-xxxx voltdb
```

Remember, when upgrading an existing installation with a running database, you need to upgrade both the software *and* the database itself. Which means you must make sure you perform the update steps in the correct order. The following sections explain the different options for updating existing installations, including at what stage in the process you should replace the software.

2.2.2. Upgrading VoltDB Using Save and Restore

Upgrading the VoltDB software on a single database cluster is easy. All you need to do is perform an orderly shutdown saving a final snapshot, upgrade the VoltDB software on all servers in the cluster, then restart the database. The steps to perform this procedure are:

1. Shutdown the database and save a final snapshot (**voltadmin shutdown --save**).
2. Upgrade the VoltDB software on all cluster nodes (instructions).
3. Restart the database (**voltdb start**).

This process works for any recent (V6.8 or later) release of VoltDB.

2.2.3. Performing an In-Service Upgrade of a Single Cluster

Normally, when upgrading the VoltDB software, you must shutdown the cluster (for example, with the **voltadmin shutdown --save** command) and restart the entire cluster using the new software. Downtime can be avoided by performing an *in-service upgrade*. An in-service upgrade allows a K-safe cluster to be upgraded one node at a time, rather than the entire cluster all at once. This means the cluster, and the business processes it supports, remain available throughout the upgrade procedure.

The requirements for performing an in-service upgrade are:

- The cluster has the appropriate license for VoltDB that includes the In-Service Upgrade feature.
- The cluster must be K-safe. That is, the cluster has a K-safety factor of one or more. This is required so individual nodes can be stopped without crashing the cluster.
- The cluster must be running VoltDB V13.1.0 or later.
- The new version falls within the parameters allowed by in-service upgrades, as described in Section 2.2.3.1, “The Scope of In-Service Upgrades”.

To perform an in-service upgrade on bare metal servers, you upgrade the VoltDB software on each node consecutively. Specifically:

1. Stop one of the cluster nodes, using the **voltadmin stop node** command
2. Once the server process stops, replace the VoltDB software with the new version.

3. Restart the node using the **voltdb start** command, specifying one or more of the other nodes in the cluster as hosts.

4. Once the rejoin process is finished and the cluster is complete, repeat the process for the next node until all nodes are upgraded.

During the upgrade process, you can determine which nodes have been updated using the @SystemInformation system procedure with the OVERVIEW selector and looking for the VERSION keyword. For example, in the following command output, the first column is the host ID and the last column is the currently installed software version for that host. Once all hosts report using the upgraded software version, the upgrade is complete.

```
$ echo "exec @SystemInformation overview" | sqlcmd | grep VERSION
       2 VERSION                   13.1.2
       1 VERSION                   13.1.2
       0 VERSION                   13.1.3
```

Until the upgrade process is complete, all nodes in the cluster maintain the functionality of the lower version, even for those nodes that have already upgraded to the higher version software. Once the upgrade is complete and all nodes are running on the newer version, the cluster switches to operating with the higher version functionality. In other words, if the new software contains any new function or behavior, that feature will not be accessible until the entire in-service upgrade process is complete.

If the upgrade fails for any reason, or you choose to stop the upgrade midway, you can revert to the original version by reversing the process: removing a node that has been upgraded, replace the VoltDB software with the original version, rejoin the node and repeat for all nodes that were upgraded. Once the upgrade process is complete, the in-service upgrade is over. At which point, you can longer return to the previous version through an in-service upgrade and must perform a full cluster restart to downgrade.

2.2.3.1. The Scope of In-Service Upgrades

There are limits to which software versions can use in-service upgrades. The following table describes the rules for which releases can be upgraded with an in-service upgrade and which releases cannot.

✔ Patch Releases	You can upgrade between any two *patch releases*. That is, any two releases where only the third and final number of the version identifier changes. For example, upgrading from 13.1.1 to 13.1.4.
✔ Minor Releases	You can also use in-service upgrades to upgrade between two consecutive *minor releases*. That is where the second number in the version identifier differ. For example, you can upgrade from V13.2.0 to V13.3.0. You can also upgrade between any patch releases within those minor releases. For example, upgrading from V13.2.3 to V13.3.0. You *cannot* use in-service upgrades to upgrade more than one minor version at a time. In other words, you can upgrade from V13.2.0 to V13.3.0 but you cannot perform an in-service upgrade from V13.2.0 to V13.4.0. To transition across multiple minor releases your options are to perform consecutive in-service upgrades (for example, from V13.2.0 to V13.3.0, then from V13.3.0 to V13.4.0) or to perform a regular upgrade where all cluster nodes are upgrading at one time.
✘ Major Releases	You *cannot* use in-service upgrades between major versions of VoltDB. That is, where the first number in the version identifier is different. For example, you must perform a full cluster upgrade when migrating from V13.x.x to V14.0.0 or later.

2.2.4. Performing an Online Upgrade Using Multiple XDCR Clusters

It is also possible to upgrade the VoltDB software using cross data center replication (XDCR), by simply shutting down, upgrading, and then re-initalizing each cluster, one at a time. This process requires no downtime, assuming your client applications are already designed to switch between the active clusters.

Use of XDCR for upgrading the VoltDB software is easiest if you are already using XDCR because it does not require any additional hardware or reconfiguration. The following instructions assume that is the case. Of course, you could also create a new cluster and establish XDCR replication between the old and new clusters just for the purpose of upgrading VoltDB. The steps for the upgrade outlined in the following sections are the same. But first you must establish the cross data center replication between the two (or more) clusters. See the chapter on Database Replication in the *Using VoltDB* manual for instructions on completing this initial step.

Once you have two clusters actively replicating data with XCDCR (let's call them clusters A and B), the steps for upgrading the VoltDB software on the clusters is as follows:

1. Pause and shutdown cluster A (**voltadmin pause --wait** and **shutdown**).
2. Clear the DR state on cluster B (**voltadmin dr reset**).
3. Update the VoltDB software on cluster A.
4. Start a new database instance on A, making sure to use the old deployment file so the XDCR connections are configured properly (**voltdb init --force** and **voltdb start**).
5. Load the schema on Cluster A so replication starts.
6. Once the two clusters are synchronized, repeat steps 1 through 4 for cluster B.

Note that since you are upgrading the software, you must create a new instance after the upgrade (step #3). When upgrading the software, you cannot recover the database using just the **voltdb start** command; you must use **voltdb init --force** first to create a new instance and then reload the existing data from the running cluster B.

Also, be sure all data has been copied to the upgraded cluster A after step #4 and before proceeding to upgrade the second cluster. You can do this by checking the @Statistics system procedure selector DR-CONSUMER on cluster A. Once the DRCONSUMER statistics `State` column changes to "RECEIVE", you know the two clusters are properly synchronized and you can proceed to step #5.

2.3. Upgrading Volt in Kubernetes

For customers who run Volt Active Data in Kubernetes, the steps for upgrading the database software are:

1. Update your copy of the VoltDB Helm 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 2.1, “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 2.3.5, “Updating VoltDB for XDCR Clusters” explains the extra step necessary for XDCR clusters.

2.3.1. 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
```

2.3.2. 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
$ ls *.tgz
voltdb-3.1.0.tgz
$ tar --strip-components=2 -xzf voltdb-3.1.0.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 use it to replace the CRD in Kubernetes:

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

2.3.3. Upgrading the VoltDB Operator and Software

Once you update the CRD, you are ready to upgrade VoltDB. You do this using the **helm upgrade** command and specifying the new software version you wish to use on the command line. What happens when you issue the **helm upgrade** command depends on whether you are performing a standard software upgrade or an in-service upgrade.

For a standard software upgrade, you simply issue the **helm upgrade** command specifying the software version in the `global.voltdbVersion` property. For example:

```
$ helm upgrade mydb voltdb/voltdb --reuse-values \
   --set global.voltdbVersion=13.2.1
```

When you issue the **helm upgrade** command, the operator saves a final snapshot, shuts down the cluster, restarts the cluster with the new version and restores the snapshot. For example, Example 2.1, “Process for Upgrading the VoltDB Software” summarizes all of the commands used to update a database release to VoltDB version *13.2.1*.

Example 2.1. Process for Upgrading the VoltDB Software

```
$    # Update the local copy of the charts
$ helm repo update
$    # Extract and replace the CRD
$ helm pull voltdb/voltdb
$ ls *.tgz
voltdb-3.1.0.tgz
$ tar --strip-components=2 -xzf voltdb-3.1.0.tgz  \
```

```
     voltdb/crds/voltdb.com_voltdbclusters_crd.yaml
$ kubectl replace -f voltdb.com_voltdbclusters_crd.yaml
$
$    # Upgrade the Operator and VoltDB software
$ helm upgrade mydb voltdb/voltdb --reuse-values \
   --set global.voltdbVersion=13.2.1
```

2.3.4. Using In-Service Upgrade to Update the VoltDB Software

Standard upgrades are convenient and can upgrade across multiple versions of the VoltDB software. However, they do require downtime while the cluster is shutdown and restarted. *In-Service Upgrades* avoid the need for downtime by upgrading the cluster nodes one at a time, while the database remains active and processing transactions.

To use in-service upgrades, you must have an appropriate software license (in-service upgrades are a separately licensed feature), the cluster must be K-safe (that is, have a K-safety factor of one or more), and the difference between the current software version and the version you are upgrading to must fall within the limits of in-service upgrades. The following sections describe:

- What versions can be upgraded using an in-service upgrade
- How to perform the in-service upgrade
- How to monitor the upgrade process
- How to rollback an in-service upgrade if the upgrade fails

2.3.4.1. The Scope of In-Service Upgrades

There are limits to which software versions can use in-service upgrades. The following table describes the rules for which releases can be upgraded with an in-service upgrade and which releases cannot.

✔ Patch Releases	You can upgrade between any two *patch releases*. That is, any two releases where only the third and final number of the version identifier changes. For example, upgrading from 13.1.1 to 13.1.4.
✔ Minor Releases	You can also use in-service upgrades to upgrade between two consecutive *minor releases*. That is where the second number in the version identifier differ. For example, you can upgrade from V13.2.0 to V13.3.0. You can also upgrade between any patch releases within those minor releases. For example, upgrading from V13.2.3 to V13.3.0. You *cannot* use in-service upgrades to upgrade more than one minor version at a time. In other words, you can upgrade from V13.2.0 to V13.3.0 but you cannot perform an in-service upgrade from V13.2.0 to V13.4.0. To transition across multiple minor releases your options are to perform consecutive in-service upgrades (for example, from V13.2.0 to V13.3.0, then from V13.3.0 to V13.4.0) or to perform a regular upgrade where all cluster nodes are upgrading at one time.
✘ Major Releases	You *cannot* use in-service upgrades between major versions of VoltDB. That is, where the first number in the version identifier is different. For example, you must perform a full cluster upgrade when migrating from V13.x.x to V14.0.0 or later.

2.3.4.2. How to Perform an In-Service Upgrade

If your cluster meets the requirements, you can use the in-service upgrade process to automate the software update and eliminate the downtime associated with standard upgrades. The procedure for performing an in-service upgrade is:

1. Set the property `cluster.clusterSpec.enableInServiceUpgrade` to true to allow the upgrade.

2. Set the property `global.voltdbVersion` to the software version you want to upgrade to.

For example, the following command performs an in-service upgrade from V13.1.2 to V13.2.0:

```
helm upgrade mydb voltdb/voltdb --reuse-values \
  --set cluster.clusterSpec.enableInServiceUpgrade=true \
  --set global.voltdbVersion=13.2.0
```

2.3.4.3. Monitoring the In-Service Upgrade Process

Once you initiate an in-service upgrade, the process proceeds by itself until completion. At a high level you can monitor the current status of the upgrade using the @SystemInformation system procedure with the OVERVIEW selector and looking for the VERSION keyword. For example, in the following command output, the first column is the host ID and the last column is the currently installed software version for that host. Once all hosts report using the upgraded software version, the upgrade is complete.

```
$ echo "exec @SystemInformation overview" | sqlcmd | grep VERSION
       2 VERSION                   13.1.2
       1 VERSION                   13.1.2
       0 VERSION                   13.1.3
```

During the upgrade, the Volt Operator reports various stages of the process as events to Kubernetes. So you can monitor the progression of the upgrade in more detail using the **kubectl get events** command. For example, the following is an abbreviated listing of events you might see during an in-service upgrade. (The messages often contain additional information concerning the pods or the software versions being upgraded from and to.)

```
$ kubectl get events -w
11m     Normal RollingUpgrade  mydb-voltdb-cluster  Gracefully terminating pod 2
11m     Normal RollingUpgrade  mydb-voltdb-cluster  Gracefully terminated pod 2
11m     Normal RollingUpgrade  mydb-voltdb-cluster  Recycling Gracefully terminated pod my
9m43s   Normal RollingUpgrade  mydb-voltdb-cluster  Recycled pod 2 has rejoined the cluste
9m42s   Normal RollingUpgrade  mydb-voltdb-cluster  Pod mydb-voltdb-cluster-2 is now READY
9m35s   Normal RollingUpgrade  mydb-voltdb-cluster  Gracefully terminating pod 1
 [ . . . ]
```

Once the upgrade is finished, the Operator reports this as well:

```
5m10s   Normal RollingUpgrade  mydb-voltdb-cluster  RollingUpgrade Done.
```

2.3.4.4. Recovering if an Upgrade Fails

The in-service upgrade process is automatic on Kubernetes — once you initiate the upgrade, the Volt Operator handles all of the activities until the upgrade is complete. However, if the upgrade fails for any reason — for example, if a node fails to rejoin the cluster — you can *rollback* the upgrade, returning the cluster to its original software version.

The Volt Operator detects an error during the upgrade whenever the VoltDB server process fails. The failure is reported as an appropriate series of events to Kubernetes:

```
12m   Warning RollingUpgrade  mydb-voltdb-cluster  Rolling Upgrade failed upgrading from..
12m   Normal  RollingUpgrade  mydb-voltdb-cluster  Please update the clusterSpec image bac
```

In addition to monitoring the events, you may wish to use the **kubectl** commands **get events**, **get pods**, and **logs** to determine exactly why the node is failing. The next step is to cancel the upgrade by initiating a rollback. You do this by resetting the image tag to the original version number.

Invoking the rollback is a manual task. However, once the rollback is initiated, the Operator automates the process of returning the cluster to its original state. Consider the previous example where you are upgrading from V13.1.2 to V13.2.0. Let us assume three nodes had upgraded but a fourth was refusing to join the cluster. You could initiate a rollback by resetting the `global.voltdbVersion` property to V13.1.2:

```
$ helm upgrade mydb voltdb/voltdb --reuse-values \
  --set global.voltdbVersion=13.1.2
```

Once you initiate the rollback, the Volt Operator stops the node currently being upgraded and restarts it using the original software version. After that process completes, the Operator goes through any node that had been upgraded, one at a time, downgrading them back to the original software. Once all nodes are reset and have rejoined the cluster, the rollback is complete.

Note that an in-service rollback can only occur if you initiate the rollback during the upgrade process. Once the in-service upgrade is complete and all nodes are running the new software version, resetting the image tag will force the cluster to perform a standard software downgrade, shutting down the cluster as a whole and restarting with the earlier version.

2.3.5. 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.mysql=myschema.sql  \
   --set-file cluster.config.classes.myjar=myclasses.jar
$ helm upgrade mydb voltdb/voltdb --reuse-values \
   --set global.voltdbVersion=12.3.1
```

Chapter 3. Upgrading to Volt Active Data V8

This chapter describes what new features are introduced in V8.0 and what upgrade tasks may be required of existing customers.

3.1. What's New in Volt V8.0

Volt 8 is a major release incorporating features from recent point releases plus new capabilities. The major new features in V8 include:

- **More Network Security** — Volt now provides SSL/TLS encryption as an option on all inter-node and inter-cluster communication, including internal, external, and DR ports. See the chapter on "Security" in the *Using VoltDB* manual for details.
- **User-Defined Functions** — It is now possible to define and declare your own functions for use in SQL statements. User-defined functions are written in Java and declared using the CREATE FUNCTION statement. See the chapter on "Creating Custom SQL Functions" in the *Volt Guide to Performance and Customization* manual for details.
- **Common Table Expressions** — Volt SQL queries can now include common table expressions, using the WITH clause. Common table expressions help organize complex SQL queries and make them easier to read. Volt also supports recursive common table expressions, making it possible to evaluate complex tree and graph structures within a single statement. See the description of the SELECT statement in the *Using VoltDB* manual for details.
- **Kafka Enhancements** — Volt now supports the latest releases of Apache Kafka, by default. The Kafka export connector continues to support all Kafka versions starting with 0.8.2. For import, the Kafka import connector and the kafkaloader command line utility now support Kafka 0.10.2 and later, up through and including the recently released version 1.0.0. For customers still using earlier versions of Kafka, Kafka 8 support is available as a configurable option for both the import connector and a legacy kafkaloader8 command line tool.
- **Python V3 API** — Volt now supports the use of Python V3.x for developing client applications. The Volt Python client library (available from GitHub) supports both Python 2.7 and 3.x.

3.2. Special Considerations for Existing Customers

Most of the new features and capabilities in Volt V8.0 do not impact existing applications. However, there are a few changes that do require minor changes to the configuration when upgrading from earlier versions. Existing customers should take note of of the following changes:

- **Change to default Kafka versions for import**

  For the kafka import connector and the **kafkaloader** command line utility, the default Kafka version has changed from 0.8.2 to 0.10.2 or later. For customers already using Kafka 0.10.2 or later, there are no changes needed to their configuration, scripts, or applications. For customers who wish to continue using the older Kafka version 0.8.2, they will need to add the attribute `version="8"` to the import connector configuration and/or use the command line utility **kafkaloader8** instead of the default **kafkaloader**.

- **The "elastic" attribute removed from <cluster>**

An artifact of an old feature provided for backwards compatibility, the `elastic` attribute of the `<cluster>` element in the configuration file was disabled and deprecated several years ago. It has now been removed. Although it is unlikely any still exist, configuration files that do include this attribute will now fail to parse. Simply remove the attribute and try again.

- **The <consistency> element removed from the configuration file**

  The `<consistency>` element was recently deprecated, since "fast" read consistency no longer provides any significant performance improvement over "safe" mode but does introduce potential risks during failure scenarios. It has now been removed from the allowable configuration file syntax. If you included `<consistency>` in your configuration file, please remove it before starting a Volt 8.0 cluster.

- **Old commands for starting Volt are no longer supported**

  Volt 6.6 introduced two new integrated commands, **init** and **start**, for starting Volt servers. At that time the old commands (`add`, `create`, `recover`, and `rejoin`) were deprecated. The old commands have now been removed from the product. If you still use the older commands, please update your scripts to use the new commands as described in the chapter "Starting the Database" in the *Using VoltDB* manual.

Chapter 4. Upgrading to Volt Active Data V9

This chapter describes what new features are introduced in V9.0 and what upgrade tasks may be required of existing customers.

4.1. What's New in Volt V9.0

Volt 9.0 is a major release incorporating features from previous point releases plus new capabilities. The major new features in V9.0 include:

- **Automated Deletion of Old Data** — Volt 8.4 introduced a new feature, USING TTL ("time to live"), that lets you define when records expire and can be deleted. This feature simplifies application design by automatically removing old data from the database based on settings you define in the table schema. With Volt 9.0, this feature is extended to include the migration of deleted data to other systems for archival purposes, as described next.

- **New Export Capabilities**— The code that supports export of data to external systems has been rewritten to provide flexibility, improve reliability, reduce system resource utilization, and support new and future product features. The new export system reinforces the durability of data queued to the export connectors across unexpected system and network failures and allows export to be extended to add new capabilities.

  The first two new capabilities are:

  - **ALTER STREAM** — The ability to modify an existing stream. You can use the new ALTER STREAM statement to modify the schema of the stream or the target for export without interrupting any already queued export data. See the description of ALTER STREAM in the *Using VoltDB* manual for details.

  - **Automated Data Migration** — You can now automate the export of data from Volt database tables to other systems as part of the data aging process. For tables declared with the USING TTL clause you can now add a MIGRATE TO TARGET clause. With MIGRATE TO TARGET, data that exceeds its "time to live" is queued to the specified export connector. Once the data is exported and acknowledged by the external system, it is then deleted from the Volt database. This automated process not only automates the archiving of old data it ensures that the data stays in the Volt database until it is confirmed as received by the export target. See the description of the CREATE TABLE statement in the *Using VoltDB* manual for more information about the USING TTL and MIGRATE TO TARGET options.

  Two important aspects of the new export infrastructure are the effect on the overall export process:

  - Export now starts when the stream is defined, not when the target is defined. Previously, stream data was not queued for export until a valid export connector was configured and connected. This meant data written to a stream might be dropped rather than queued for export if the connector was not configured correctly. Starting with Volt 9.0, data written to streams declared with the EXPORT TO TARGET clause are queued for export whether the target is configured or not. Similarly, the queued data is removed as soon as the stream itself is removed with the DROP STREAM statement.

    This change makes the queuing of export data much more reliable, easier to understand, and easier to control. Data is queued for export as soon as the stream is defined and purged as soon as the stream is dropped. The new ALTER STREAM statement further lets you modify the stream definition without having to clear any existing export queues.

    - Export is now an enterprise feature. The Volt Community Edition provides access to two streams per database, so users have access to basic export functionality. But for unlimited access to export and migration features, the Enterprise Edition is required. (See Special Considerations for more information about upgrading community databases that use export.)

- **"Live" Schema Updates with Database Replication** — Previously, database replication (DR) required the schema of the cooperating databases to match for all DR tables. So updating the schema required a pause while all of the affected databases were updated. Starting with 9.0, this limitation has been loosened. DR continues even if the schema are different. So it is possible to update the schema without interrupting ongoing transactions.

  Of course, it is not possible for Volt to resolve individual transactions if the schema differ. So if a DR consumer (either a replica in passive DR or an XDCR cluster in active replication) receives a binary log where the schema of the affected table(s) does not match, DR will stall and wait for the schema to be updated to match the incoming data. Therefore, care must be taken when updating the schema to ensure that no transactions that are affected by the schema change are processed during the interval when the clusters' schema do not match. See the sections on updating DR schema for passive and active DR for more information.

- **Simplified JSON interface** — A new version of the Volt JSON API, 2.0, is now available. The original JSON interface provides complete information about the schema for the data being returned, including separate entries for the data, the column names, and datatypes. The 2.0 API returns a much more compact result set with each row represented by an associative array with each element consisting of the column name and value.

  The 1.0 API is useful if you do not know what data is returned and want to deconstruct the results in detail. The 2.0 API is more useful for rapidly fetching and using known results. Both versions of the API accept the same parameters, as described in the section on using the JSON API in the *Using VoltDB* manual. So the following calls return the same data except at different levels of detail.

  ```
  http://myserver:8080/api/1.0/?Procedure=@Statistics&Parameters=["TABLE",0]
  http://myserver:8080/api/2.0/?Procedure=@Statistics&Parameters=["TABLE",0]
  ```

- **Support for Java 11** — Volt now supports both Java 8 and Java 11.

4.2. Special Considerations for Existing Customers

Many of the new features and capabilities in Volt V9.0 do not impact existing applications. However, there are a few changes that may require action for users upgrading from earlier versions. Existing customers should take note of of the following changes:

- **Change to how stream data is queued for export**

  Previously, if you defined a stream with the EXPORT TO TARGET clause but no matching target was configured, any data inserted into the stream was dropped. No data was queued until the specified target was both configured and successfully connected. With Volt 9.0 export queuing has been simplified: data is queued as soon as the stream is declared and the queue is deleted as soon as the stream is dropped.

  This means that if you declare a stream and it is being written to, but you do not configure the associated target, the data inserted into the stream will be queued in the `export_overflow` directory, consuming disk space that would not have been used in earlier versions of the product.

- **Change to the reporting of streams in @Statistics**

In Volt 8.4, a new @Statistics selector, EXPORT, was introduced to provide improved visibility and more detail concerning the export lifecycle. At that time, export streams continued to be reported under the TABLE selector, so as not to disrupt existing scripts or procedures users might have that rely on that information. With Volt 9.0, export streams have been removed from the TABLE selector results. Now the TABLE selector only reports on tables and streams that are not associated with an export target. Information on streams declared with the EXPORT TO TARGET clause is provided under the EXPORT selector.

- **Managing export queues during outages**

  In most cases, Volt manages the export queues even in unusual cases where nodes go down in a K-safe cluster. If at any time the node managing an export partition finds a gap in the export queue (due to the current server having been down when that data was written to the stream), the system queries the other servers to find one with the missing data and export continues. In the rare case where Volt cannot find the missing records in any of the current server export queues, the export connector for that queue will stop, waiting for a server that has the data.

  Even if servers stop and rejoin frequently, the data will eventually be found on a rejoining node. However, in the unusual case that, say, failed nodes are replaced by new, initialized servers, it is possible that the gap in the queue cannot be resolved. Previously, Volt would eventually (once the cluster was at a full complement of servers) skip the missing data and restart the connector at the next available export record. Starting with Volt 9.0, export will not skip over gaps automatically. The queue will stop and warnings will be logged to the console and reported via SNMP. You must issue the **voltadmin export release** command to resume processing of the export connector at the next available record.

- **Limits on streams in Community Edition**

  Export is now an Enterprise Edition feature and the Community Edition is limited to two streams per database. This means if you try to restore a database from a previous version with more than two streams using the Volt 9.0 Community Edition, the restore will fail. If this happens when upgrading, you can initialize a new database root, load the schema without the additional streams, then manually restore the data.

- **Support for CentOS and RHEL 6 removed**

  The officially supported platforms for Volt have been updated. CentOS and RHEL version 6 are no longer officially supported. The current list of supported platforms include CentOS and RHEL version 7.0 or later, Ubuntu versions 14.04, 16.04, and18.04, and Macintosh OS X 10.9 or later.

- **Support for export and import to Kafka 0.8.2 removed**

  The Kafka import and export connectors now require Kafka version 0.10.2 or later.

- **Logging name change from JOIN to ELASTIC**

  The log4J logger reporting on elastic changes to Volt clusters has been renamed from JOIN to ELASTIC.

Chapter 5. Upgrading to Volt Active Data V10

This chapter describes what new features are introduced in V10.0. V10.0 does *not* require any changes to client applications or databases prior to upgrading.

5.1. What's New in Volt V10.0

Volt 10.0 is a major release incorporating features from recent updates plus new capabilities. The major new features in V10.0 include:

- **New Volt Operator for Kubernetes** — Volt now offers a complete solution for running Volt databases in a Kubernetes cloud environment. Volt V10.0 provides managed control of the database startup process, a new Volt Operator for coordinating cluster activities, and Helm charts for managing the relationship between Kubernetes, Volt and the Operator. The Volt Kubernetes solution is available to Enterprise customers and includes support for all Volt functionality, including cross data center replication (XDCR). See the *Volt Kubernetes Administrator's Guide* for more information.

- **New Prometheus agent for Volt** — For customers who use Prometheus to monitor their systems, Volt now provides a Prometheus agent that can collect statistics from a running cluster and make them available to the Prometheus engine. The Prometheus agent is available as a Kubernetes container or as a separate process that can either run on one of the Volt servers or remotely and makes itself available through port 1234 by default. See the README file in the `/tools/monitoring/prometheus` folder in the directory where you install Volt for details.

- **Enhancements to Export** — Recent updates to export provide significant improvements to reliability and performance. The key advantages of the new export subsystem are:

  - **Better throughput** — Initial performance tests demonstrate significantly better throughput on export queues using the new subsystem over previous versions of Volt.

  - **Adjustable thread pools** — The new subsystem let's you set the thread pool size for export as a whole or to define thread pools for individual connectors.

  - **Fewer duplicate rows** — When cluster nodes fail and rejoin the cluster, the export subsystem resubmits certain rows to ensure they are delivered. The new subsystem keeps better track of the acknowledged rows and does not need to send as many duplicates to maintain the same level of durability.

- **Improved license management** — Starting with Volt V10.0, specifying the product license has moved from the **voltdb start** command to the **voltdb init** command. In other words, you only have to specify the license once, when you initialize the database root directory, rather than every time you start the database. When you do specify the license on the **init** command, it is stored in the root directory the same way the configuration is.

  The same rules apply about the default location of the license as before. So if you store your license in your current working directory, your home directory, or the `/voltdb` subfolder where Volt is installed, you do not need to include the **--license** argument when initializing the database. Also note that the **--license** argument on the **voltdb start** command is now deprecated but still operational. So if you have scripts to start Volt that include **--license** on the **start** command, they will continue to work. However, we recommend you change to the new syntax whenever convenient because support for **voltdb start --license** may be removed in some future major release.

- **Support for RHEL and CentOS V8** — After internal testing and validation, RHEL and CentOS V8 are now supported platforms for production use of Volt.

5.2. Special Considerations for Existing Customers

Volt 10.0 contains no incompatible changes with Volt V9. All existing Volt V9.x databases and client applications can be upgraded to V10.0 as is.

Chapter 6. Upgrading to Volt Active Data V11

This chapter describes what new features are introduced in V11.0 and what upgrade tasks may be required of existing customers.

6.1. What's New in Volt V11.0

Volt 11 is a major release incorporating features from recent updates plus new capabilities. The major new features in V11 include:

- **Volt Topics** — Volt Topics provide the intelligent streaming of Volt's existing import and export capabilities, but with the flexibility of Kafka-like streams. Topics allow for both inbound and outbound streaming to multiple client producers and consumers. They also use the existing Kafka interface to simplify integration into existing infrastructure. But most importantly, they allow for intelligent processing and manipulation of the data as it passes through the pipeline.

  Volt topics, which were released as a beta feature in V10.2, are now ready for production use. See the chapter on Streaming Data in the *Using VoltDB* manual for more information.

- **Support for Python 3.6** — The Volt command line tools have been upgraded to use Python version 3. Python 3 is commonly available on modern operating systems and so simplifies the process of configuring platforms for Volt.

- **New Kubernetes capabilities** — Upgrades to the Volt Operator for Kubernetes provide two important new features:

  - **Multi-cluster XDCR** — It is now possible to create a cross datacenter replication (XDCR) network with three or more clusters. Additional Helm properties help simplify the management and maintenance of the XDCR clusters. See the chapter on Database Replication in the *Volt Kubernetes Administrator's Guide* for details.

  - **Volt Topic Support** — The Operator now provides the properties necessary to configure and start Volt topics in clusters running in Kubernetes.

- **Volt Java Client improvements**— The Volt Java client interface has been updated with the following improvements:

  - **Topology Awareness** — Previously, there were two options for handling topology changes on the server, `setReconnectOnConnectionLoss()` and `setTopologyChangeAware()`, which were mutually exclusive. This limitation has been removed and `setTopologyChangeAware()` has been enhanced to include reconnection when the last connection is lost, further improving connectivity and resilience.

  - **Non-Blocking Asynchronous Calls** — Normally, the asynchronous callProcedure method returns an error if the client cannot queue the call because of backpressure. However, it is still possible for the call to block in certain cases. It is now possible to avoid blocking entirely by setting the `setNonblockingAsync()` configuration option on the client. See the Javadoc for details.

  - **Connection timeouts** — Handling of connection timeouts by the client has been improved. Now, if the client is able to detect a timeout before it is sent to the server, the client aborts the transaction and returns the procedure status GRACEFUL_FAILURE, with a status string of "Procedure call not queued: timed out waiting for host connection."

- **Security updates** — The Volt Management Center (VMC) web-based console for Volt has been updated to the latest versions of the jQuery libraries to address security vulnerabilities. The current library versions for VMC are JQuery v3.5.1 and jQuery-UI v1.12.1.

6.2. Special Considerations for Existing Customers

Most of the new features and capabilities in Volt V11.0 do not impact existing applications. However, there are a few changes that require action for users upgrading from earlier versions. Also several deprecated features have now been removed. Existing customers should take note of of the following changes:

- **Supported platforms**

  Ubuntu 16.04 is no longer a supported production platform for Volt. The currently supported operating systems for running production Volt databases are:

  - CentOS and Red Hat (RHEL) V7.0 and later or V8.0 and later
  - Ubuntu 18.04 and 20.04

- **Python 3.6 is now required**

  The system requirements for Volt have been changed from Python 2.7 to Python 3.6.

- **Support for LIMIT PARTITION ROWS has been removed**

  The LIMIT PARTITION ROWS clause of the CREATE TABLE statement has been removed. The USING TTL clause and scheduled tasks are the recommended replacements for this feature.

- **The deprecated @SnapshotStatus system procedure has been removed**

  The `@SnapshotStatus` system procedure has been deleted. The @Statistics system procedure with the SNAPSHOTSUMMARY selector is the recommended replacement.

Chapter 7. Upgrading to Volt Active Data V12

This chapter describes what new features are introduced in V12.0 and what upgrade tasks may be required of existing customers.

7.1. What's New in Volt V12.0

Volt 12 is a major release that includes several key features. First and foremost, the memory management for the database table data has been rewritten and optimized to reduce certain impediments in the previous implementation. Although this change is primarily internal and transparent to you as a customer, it does have two direct benefits in terms of eliminating development and operational roadblocks:

- **No Large Compaction Events** — As tuples are inserted and deleted, small gaps of unused memory are created within the larger allocated blocks. Previously, if the total amount of allocated but unused memory hit a specific high watermark, the database would compact all of the table memory before continuing. As effective as this mechanism was, it could result in unpredictable latency spikes in the ongoing workload.

  Now, defragmentation of data storage is performed incrementally on a per table and per partition basis. Since the compaction transactions are much smaller, and also partitioned, they have little or no impact on the ongoing business workload. In addition, you as the database administrator have control over how large those periodic compaction events are and and how often they occur. See the chapter on memory management in the *Volt Performance and Customization Guide* for more information about the new memory management algorithm.

- **No Hash Mismatches Due to Row Order** — In the past, developers had to be careful not to introduce non-deterministic behavior into their stored procedures by performing unordered queries. The issue was that, when using K-Safety, different copies of a partition could return results in a different order if you did not include the appropriate ORDER BY clause.

  A key aspect of the memory management scheme introduced in V12.0 is that all copies of a partition now always return a query's results in one, deterministic order, even if the query itself is not sorted. This means that queries without an appropriate ORDER BY clause will not cause a hash mismatch.

  Mind you, including an ORDER BY clause is still recommended so you can depend on the order in which the results are returned. Although VoltDB now returns results in a deterministic order, you do not know what that order will be. Also, although the new deterministic row order helps, there are other practices (such as calling system-specific time functions) that still cause hash mismatches and must be avoided. See the section on stored procedures and determinism in the *Using VoltDB* manual for a reminder of what to watch out for.

Other new features introduced in V12.0 and recent releases include:

- **Support for Ubuntu 22.04 and the Rocky OS** — Volt Active Data has added Ubuntu 22.04 and Rocky OS as supported platforms for VoltDB.

- **Support for storing TLS/SSL credentials in Kubernetes secrets** — When enabling TLS/SSL in Kubernetes, you can now store your TLS/SSL credentials (including the keystore, truststore, and passwords) in a Kubernetes secret. This avoids having to specify passwords on the Helm command line and simplifies the commands needed to start and update database instances. See the section on configuring TLS/SSL in the *Volt Kubernetes Administrator's Guide* for details.

- **Expiration dates for user accounts** — You can now specify an expiration date for user accounts in the database configuration file. Once the specified date is past, the associated account can no longer access the database, until the configuration for the user account is updated. The expiration date is optional. See the section on defining users and roles in the *Using VoltDB* manual for details.

- **New LAG() windowing function** — The LAG() function accesses previous rows from the window results using an offset. See the section on windowing functions in the SELECT reference page for more information.

- **Dedicated pod for VMC and HTTP API in Kubernetes** — By default, Volt in Kubernetes now creates a separate pod for the Volt Management Center (VMC) and HTTP API. This provides a single service name for accessing these resources, as well as a single instance for the entire cluster (rather than separate instances for each host). The new pod is available from the service name `{release-name}-voltdb-vmc` where *{release-name}* is the name of the Helm release for the database cluster.

7.2. Special Considerations for Existing Customers

Most of the new features and capabilities in VoltDB V12.0 do not impact existing applications. However, there are a few changes that require action for users upgrading from earlier versions. Also several deprecated features have now been removed. Existing customers should take note of of the following changes:

- **A license is required on the voltdb init command**

  Starting with V12, the **voltdb init** command must find and load a license file or the initialization of the database root directory will fail. The license file can either be specified explicitly using the `-l` or `--license` flag or it can be found in one of the three default locations (the current working directory, the user's home directory, or the `voltdb` folder where VoltDB is installed). It is still possible to specify a license file on the **voltdb start** command — in case you need to change or update the license after initialization — but a license must be specified on the **voltdb init** command first.

- **The utility kafkaloader10 is now deprecated**

  To support different versions of the Kafka API, two versions of the kafkaloader utility were provided in the past: kafkaloader and kafkaloader10. Now that support for older versions of Kafka has been dropped, the legacy loader, kafkaloader10, has been deprecated and will be removed in a future release.

- **Old deprecated methods removed from the Java client API**

  Several obsolete methods in the Java client API that were previously deprecated have now been removed. Those methods were `setClientAffinity`, `setSendReadsToReplicas`, and `setReconnectOnConnectionLoss`.

- **The voltadmin plan_upgrade command has been removed**

  The procedure for upgrading the VoltDB software using limited hardware is no longer supported. The associated command, **plan_upgrade**, has been removed from the **voltadmin** utility.

- **@Statistics DRCONSUMER column renamed**

  The results of the @Statistics system procedure DRCONSUMER selector have been altered slightly. Specifically, the last column of the third results table has been renamed to be more descriptive from LAST_FAILURE to LAST_FAILURE_CODE.

Chapter 8. Upgrading to Volt Active Data V13

This chapter describes what new features are introduced in V13.0 and what upgrade tasks may be required of existing customers.

8.1. What's New in Volt V13

Volt Active Data V13 is a major release that includes a number of structural enhancements as well as the general availability for production use of features previewed in earlier releases. New features and enhancements in V13.0 include:

- **New Prometheus metrics system** — Volt V13 announces the general production release of a new metrics system, where every server in the cluster reports its own data through a Prometheus-compliant endpoint. You enable Prometheus metrics in the configuration file when initializing the database by adding the <metrics> element to the Volt configuration file:

  ```
  <deployment>
   <cluster kfactor="1"/>
   <metrics enabled="true"/>
  </deployment>
  ```

  Once enabled, each Volt server reports server-specific information through the metrics port, which defaults to 11781. The new metrics system replaces the standalone Prometheus agent, which has now been deprecated. See the sections on integrating with Prometheus in the *Volt Administrator's Guide* and *Volt Kubernetes Administrator's Guide*.

- **New Grafana dashboards** — To match the new metrics system, Volt provides sample Grafana dashboards to help visualize your database's performance and status. There are matching dashboards designed for use with Kubernetes and bare metal:

  - Kubernetes dashboards: https://github.com/VoltDB/volt-monitoring/tree/main/dashboards/Volt-K8s-13.x/new-metrics

  - Bare metal dashboards: https://github.com/VoltDB/volt-monitoring/tree/main/dashboards/Volt-V13.x/new-metrics

- **Support for alternate character sets** — Volt provides full support for international characters through its use of UTF-8 for storing and displaying textual data. However, there are other character encodings that may be in use by customers, including both older, established encodings such as Shift_JIS and newer encodings like the Chinese GB18030-2022 standard. For customers using alternate character encodings as the default in their client environment, Volt now provides automatic conversion of character encodings both interactively and for file input. The **sqlcmd** utility automatically converts to and from the terminal session's localized character set for input and display. When reading and writing files, you can use the `--charset` qualifier to specify the character encoding of the file, both with the **sqlcmd** and **csvloader** utilities.

- **Improved command line interface for the sqlmd utility** — In addition to full character set support, the **sqlcmd** utility has a more complete and consistent set of command line qualifiers. The new `--output-file` qualifier captures the output of SQL statements; it does not echo the commands or informational messages. When used with the `--output-format=csv` qualifier, `--output-file` now generates valid CSV files without extraneous lines. Similarly, the `--file` qualifier lets you process a

file containing any valid **sqlcmd** statements or directives. And if the file contains only data definition language (DDL) statements, you can use the `--batch` qualifier to process all of the statement as a single batch significantly reducing the time required to update the schema. See the **sqlcmd** reference page for these and other improvements to the **sqlcmd** utility.

- **Beta support for ARM architecture** — Beta software kits for running the VoltDB server software natively on ARM64 architecture are now available. If you are interested in participating in the ARM64 beta program, please contact your Volt Customer Success Manager.
- **Support for Kubernetes 1.27** — Volt Active Data now supports version 1.27 of the Kubernetes platform. See the *Kubernetes Compatibility Chart* for details on what versions of Kubernetes are supported by each version of VoltDB.

8.2. Special Considerations for Existing Customers

The new features and capabilities in VoltDB V13.0 do not impact existing applications. However, there are a few changes that require action for users upgrading from earlier versions. Existing customers should take note of the following changes:

- **Ubuntu 18.04 and CentOS/RHEL V7 are no longer supported**

  Starting with V13, Ubuntu 18.04, which has reached end of life, and CentOS and Red Hat V7 are no longer supported as production platforms for Volt Active Data. Although the VoltDB software may continue to run, we strongly recommend upgrading to either Ubuntu 22.04 or Rocky Linux or REHL V8.6 or later for production use.

- **The standalone Prometheus agent is deprecated**

  Support for Prometheus data scraping was originally provided by a separate Prometheus agent (run in a separate pod on Kubernetes). That original cluster-wide agent has been replaced with a new metrics system that provides Prometheus endpoints on all of the cluster nodes, each reporting its own data. As a result, the standalone Prometheus agent has been deprecated and will be removed in a future release.

Chapter 9. Upgrading to Volt Active Data V14

This chapter describes what new features are introduced in V14.0 and what upgrade tasks may be required of existing customers.

9.1. What's New in Volt Active Data V14

Volt Active Data V14 is a major release that includes a redesign and enhancement to how you configure the database, as well as general cleanup of old and obsolete functions. The key new features and enhancements in V14.0 include:

- **Redesign of Database Configuration** — Traditionally, VoltDB has been configured using a single XML file. Once the database was running, you could update the configuration using the **voltadmin update** command and passing it an updated XML file. But that file had to be complete; any options configured on initialization had to also be set for the update or else the update would fail or options would be reset to the default. You could not simply specify the one or two options to change.

  VoltDB now uses YAML properties for defining the configuration. The use of YAML has multiple advantages:

  - **Ease-of-Use:** YAML is a simple indented text format that is easier to read and edit than XML.

  - **Modularity:** You are no longer restricted to a single configuration file. You can specify multiple YAML files when you initialize the database and the contents are merged. This allows you to group and manage configuration options by category, such as security, export, directory paths, etc. For example, you could have a single file for configuring two XDCR databases identically and have separate YAML files for the XDCR settings, which require a unique ID per cluster:

    ```
    $ voltdb initialize -C common.yaml,xdcr_cluster1.yaml
    ```

  - **Getting and Setting Individual Properties:** You can now get and set individual configuration properties on a running database using the **voltadmin get** and **set** commands, rather than having to relist all of the original properties. For example:

    ```
    $ voltadmin set deployment.snapshot.frequency=2d
    ```

    You can also set multiple properties in a single **set** command using a list of dot notation settings or a YAML file of just those properties you want to change. Of course, you can still use the **voltadmin update** command, providing a complete set of properties in either YAML or XML.

  See the section on understanding YAML syntax in the *Using VoltDB* manual for more information on how to use YAML effectively.

  Finally, although XML format for configuring databases is now deprecated in favor of YAML — and we encourage the use of YAML — XML is still supported and continues to work as in previous releases for both the **voltdb init** and **voltadmin update** commands. So your existing scripts for starting and/or updating a database do not have to change at this time.

- **New voltadmin get and set Commands** — As mentioned above, there are two new **voltadmin** commands: **get** and **set**. The voltadmin get command lets you retrieve part or all of the current configuration settings in YAML. If you specify "deployment" as the argument, you get all of the settings. Or you can

specify a single property or group of properties by specifying the desired settings in dot notation. For example, you can get just the current K-safety factor with the **voltadmin get deployment.cluster.k-factor** command or all export settings with **voltadmin get deployment.export**. See the appendix of YAML properties in the *Using VoltDB* manual for details.

Similarly, the **voltadmin set** command lets you modify individual properties. You can either specify an individual property using dot notion (such as **voltadmin set deployment.snapshot.enabled=true**) or a YAML file for setting multiple properties at once (for example, **voltadmin set --file=newusers.yaml**)

- **Updated results for @SystemInformation OVERVIEW** — The return results for the @SystemInformation system procedure OVERVIEW selector have been updated and a new field added to make it clearer when a cluster is at full K-safety or not. Originally, the field CLUSTERSAFETY could be misleading because it only reported on whether a hash mismatch had forced the cluster into reduced K-safety. Its value did not change if one or more nodes had failed for other reasons. To make it less misleading, CLUSTERSAFETY now reports FULL or REDUCED depending on whether the cluster is fully functional or nodes are missing for any reason. A new field, REDUCEDSAFETY, reports on whether K-safety has been intentionally reduced due to a hash mismatch.

- **Updated Platform Support** — Volt Active Data now supports Kubernetes up through version 1.30, Ubuntu version 24.04, and Java versions 17 and 21.

- **Removing Obsolete Functionality** — Over the life cycle of version 13, a number of older features were deprecated and replaced by improved and enhanced implementations. WIth the release of V14, these deprecated items are being removed from the product. These include:

  - Embedded Volt Management Center (VMC) and HTTP JSON API

  - Standalone Prometheus agent

9.2. Special Considerations for Existing Customers

The new features in VoltDB V14.0 add capabilities without altering the behavior of existing applications. However, there are a few changes that require action for users upgrading from earlier versions. Existing customers should take note of the following changes:

- **Deprecated Features Removed**

  Several obsolete technologies that have already been deprecated have now be removed from the product. Specifically:

  - **Embedded VMC and HTTP JSON API** — The embedded Volt Management Console (VMC) and HTTP programming interface has been deprecated and replaced with a VMC service that must be started separately on bare metal. (In Kubernetes, the new VMC service is started by default.) If you have been depending on VMC or the HTTP API starting automatically on bare metal, you will need to install and start the VMC service manually. See the *Volt Administrator's Guide* for information on running the VMC service.

  - **Standalone Prometheus agent** — The standalone Prometheus agent is no longer included in the Volt Active Data software kit. Instead, Prometheus metrics are available directly from the VoltDB servers, with each server reporting its own performance metrics. To use the new metrics, be sure to enable metrics in the database configuration, as described in the *Volt Administrator's Guide*. For example:

    ```
    deployment:
      metrics:
        enabled: true
    ```

- **Updated System Requirements**

  The operating system and software requirements for Volt Active Data have been updated to add support for new releases and to remove products that have reached end of life. Specifically, the new requirements are:

  - **Operating System:**
    - Red Hat (RHEL) and Rocky version 8.8 or later, including subsequent releases. Version 8.6 is no longer supported.
    - Ubuntu 20.04, 22.04, and 24.04 and subsequent releases. Support for Ubuntu 24.04 has been added.
    - Macintosh OS X 13.0 and later (for development only). Support for versions 11 and 12 has been dropped.
  - **Java for VoltDB server:** Java 17 or 21. Java 11 is no longer supported for running the Volt server. (Note that Java 8, 11, 17, and 21 are all supported for Java clients.)
  - **Cloud Computing:** Kubernetes versions 1.25 through 1.30 are tested and supported, along with subsequent releases.

Appendix A. Configuration File (deployment.xml)

Vote Active Data V14 introduced the use of YAML for configuring the database. Use of XML for configuration is now deprecated. However, to ensure a smooth transition for existing databases, XML configuration files are still accepted as input for the **voltadmin init** and **update** commands. This appendix describes the syntax for XML configuration files.

A.1. Understanding XML Syntax

XML files consist of a series of nested *elements* identified by beginning and ending "tags". The beginning tag is the element name enclosed in angle brackets and the ending tag is the same except that the element name is preceded by a slash. For example:

```
<deployment>
   <cluster>
   </cluster>
</deployment>
```

Elements can be nested. In the preceding example `cluster` is a child of the element `deployment`.

Elements can also have *attributes* that are specified within the starting tag by the attribute name, an equals sign, and its value enclosed in single or double quotes. In the following example the `kfactor` and `sitesperhost` attributes of the cluster element are assigned values of "1" and "12", respectively.

```
<deployment>
   <cluster kfactor="1" sitesperhost="12">
   </cluster>
</deployment>
```

Finally, as a shorthand, elements that do not contain any children can be entered without an ending tag by adding the slash to the end of the initial tag. In the following example, the `cluster` and `heartbeat` tags use this form of shorthand:

```
<deployment>
   <cluster kfactor="1" sitesperhost="12"/>
   <heartbeat timeout="10"/>
</deployment>
```

For complete information about the XML standard and XML syntax, see the official XML site at http://www.w3.org/XML/.

A.2. The Structure of the XML Configuration File

The configuration file starts with the XML declaration. After the XML declaration, the root element of the configuration file is the deployment element. The remainder of the XML document consists of elements that are children of the deployment element.

Figure A.1, “Configuration XML Structure” shows the structure of the configuration file. The indentation indicates the hierarchical parent-child relationships of the elements and an ellipsis (...) shows where an element may appear multiple times.

Figure A.1. Configuration XML Structure

```
<deployment>
    <cluster/>
    <paths>
        <commandlog/>
        <commandlogsnapshot/>
        <exportoverflow/>
        <snapshots/>
        <voltdbroot/>
    </paths>
    <commandlog>
        <frequency/>
    </commandlog>
    <dr>
         <connection/>
         <schemachange/>
         <consumerlimit>
            <maxbuffers/>
            <maxsize/>
         </consumerlimit>
    </dr>
    <export>
        <configuration>
            <property/>...
        </configuration>...
    </export>
    <heartbeat/>
    <httpd/>
    <import>
        <configuration>
            <property/>...
        </configuration>...
    </import>
    <metrics/>
    <partition-detection/>
    <security>
      <ldap>
        <group/>...
        <ssl>
           <truststore/>
        </ssl>
      </ldap>
    <security/>
    <snapshot/>
    <ssl>
        <keystore/>
        <truststore/>
    </ssl>
    <snmp/>
    <systemsettings>
        <clockskew/>
        <compaction/>
        <elastic/>
```

```
        <flushinterval>
            <dr/>
            <export/>
        </flushinterval>
        <procedure/>
        <query/>
        <resourcemonitor>
            <disklimit>
                <feature/>...
            </disklimit>
            <memorylimit/>
        </resourcemonitor>
        <snapshot/>
        <temptables/>
    </systemsettings>
    <topics>
        <broker>
            <property/>...
        </broker>
        <topic/>...
    </topics>
    <users>
        <user/>...
    </users>
</deployment>
```

Table A.1, “XML Configuration File Elements and Attributes” provides further detail on the elements, including their relationships (as child or parent) and the allowable attributes for each.

Table A.1. XML Configuration File Elements and Attributes

Element	Child of	Parent of	Attributes
deployment*	(root element)	avro, cluster, commandlog, dr, export, heartbeat, httpd, import, partition-detection, paths, security, snapshot, snmp, ssl, systemsettings, topics, users	
avro	deployment		registry={url}* namespace={text} prefix={text}
cluster*	deployment		kfactor={int} sitesperhost={int}
heartbeat	deployment		timeout={int}*
partition-detection	deployment		enabled={true|false}
commandlog	deployment	frequency	enabled={true|false} logsize={int} synchronous={true|false}
frequency	commandlog		time={int}

Element	Child of	Parent of	Attributes
			transactions={int}
dr	deployment	connection, consumerlimit, schemachange	id={int}* role={master|replica|xdcr}
connection	dr		source={server[,...]}* connectiontimeout= {int} enabled={true|false} preferred-source={int} receivetimeout= {int} ssl=[file-path]
consumerlimit	dr	maxbuffers, maxsize	
maxbuffers	consumerlimit		
maxsize	consumerlimit		
schemachange	dr		enabled={true|false}
export	deployment	configuration	
configuration*	export	property	target={text}* enabled={true|false} exportconnectorclass={class-name} type={file|http|jdbc|kafka|custom}
property	configuration		name={text}*
import	deployment	configuration	
configuration*	import	property	type={kafka|custom}* enabled={true|false} format={csv|tsv} module={text} priority={int} reconnect={int}{s|m|h}
property	configuration		name={text}
httpd	deployment		enabled={true|false}
metrics	deployment		enabled={true|false} interval={int}{s|m|h} maxbuffersize={int}
paths	deployment	commandlog, commandlogsnapshot, droverflow, exportoverflow, snapshots, voltdbroot	
commandlog	paths		path={directory-path}*
commandlogsnapshot	paths		path={directory-path}*
droverflow	paths		path={directory-path}*
exportoverflow	paths		path={directory-path}*
snapshots	paths		path={directory-path}*
voltdbroot	paths		path={directory-path}*

Element	Child of	Parent of	Attributes
security	deployment	ldap	enabled={true|false} provider={hash|kerberos|ldap}
ldap	security	group, ssl	server={url}* rootdn={text}* user={text}* password={text}* timeout={int}
group	ldap		name={text}* role={text}*
ssl	ldap	truststore	
truststore	ssl		path={file-path}* password={text}*
snapshot	deployment		enabled={true|false} frequency={int}{s|m|h} prefix={text} retain={int}
ssl	deployment	keystore, truststore	enabled={true|false} external={true|false} internal={true|false}
keystore*	ssl		path={file-path}* password={text}*
truststore	ssl		path={file-path}* password={text}
snmp	deployment		target={IP-address}* authkey={text} authprotocol={SHA|MD5|NoAuth} community={text} enabled={true|false} privacykey={text} privacyprotocol={text} username={text}
systemsettings	deployment	clockskew, compaction, elastic, flushinterval, priorities, procedure, query, resourcemonitor, snapshot, temptables	
clockskew	systemsettings		interval={int}
compaction	systemsettings		interval={int} maxcount={int}
elastic	systemsettings		duration={int} throughput={int}
flushinterval	systemsettings	dr, export	minimum={int}
dr	flushinterval		interval={int}
export	flushinterval		interval={int}
priorities	systemsettings	dr, snapshot	batchsize={int}

Element	Child of	Parent of	Attributes
			enabled={true|false} maxwait={int}
dr	priorities		priority={int}
snapshot	priorities		priority={int}
procedure	systemsettings		loginfo={int} copyparameters={true|false}
query	systemsettings		timeout={int}*
resourcemonitor	systemsettings	disklimit, memorylimit	frequency={int}
disklimit	resourcemonitor	feature	
feature	disklimit		name={text}* size={int[%]}* alert={int[%]}
memorylimit	resourcemonitor		size={int[%]}* alert={int[%]} compact={true|false}
snapshot	systemsettings		priority={int}*
temptables	systemsettings		maxsize={int}*
threadpools	deployment	pool	
pool	threadpools		name={text}* size={text}*
topics	deployment	broker, topic	enabled={true|false} threadpool={text}
broker	topics	property	
topic	topics	property	name={text}* allow={role-name[,..]} format={avro|csv|json} opaque={true|false} priority={int} procedure={text} retention={text}
property	broker,topic		name={text}
users	deployment	user	
user	users		name={text}* password={text}* expires={date} roles={role-name[,..]}

*Required

Appendix B. Using the Original VoltDB Client Interface

Volt Active Data has two Java client programming interfaces: Client and Client2. Client2 is a modern API using the best features and design patterns of the Java programming language. Client is the original client API. Because of the improvements made to Client2, Client2 is recommended for all new application development and is featured in the documentation. However, the original Client API is still supported and in use in a number of existing customer applications.

Which is why this appendix is provided for anyone needing to support applications that use the original Client API. The following sections explain how to use the original VoltDB Java client interface for runtime access to VoltDB databases and functions. Please see the *Using VoltDB* guide for information on writing new applications using the Client2 API.

B.1. Connecting to the VoltDB Database

The first task for the calling program is to create a connection to the VoltDB database. You do this with the following steps:

```
org.voltdb.client.Client client = null;
ClientConfig config = null;
try {
       config = new ClientConfig("advent","xyzzy");        ❶

       client = ClientFactory.createClient(config);        ❷

       client.createConnection("myserver.xyz.net");        ❸
} catch (java.io.IOException e) {
       e.printStackTrace();
       System.exit(-1);
}
```

❶ Define the configuration for your connections. In its simplest form, the `ClientConfig` class specifies the username and password to use. It is not absolutely necessary to create a client configuration object. For example, if security is not enabled (and therefore a username and password are not needed) a configuration object is not required. But it is a good practice to define the client configuration to ensure the same credentials are used for all connections against a single client. It is also possible to define additional characteristics of the client connections as part of the configuration, such as the timeout period for procedure invocations or a status listener. (See Section B.5, “Handling Errors”.)

❷ Create an instance of the VoltDB `Client` class.

❸ Call the `createConnection()` method. After you instantiate your client object, the argument to `createConnection()` specifies the database node to connect to. You can specify the server node as a hostname (as in the preceding example) or as an IP address. You can also add a second argument if you want to connect to a port other than the default. For example, the following `createConnection()` call attempts to connect to the admin port, 21211:

```
client.createConnection("myserver.xyz.net",21211);
```

If security is enabled and the username and password in the `ClientConfig()` call do not match a user defined in the configuration file, the call to `createConnection()` will throw an exception.

When you are done with the connection, you should make sure your application calls the `close()` method to clean up any memory allocated for the connection. See Section B.4, “Closing the Connection”.

B.1.1. Connecting to Multiple Servers

You can create the connection to any of the nodes in the database cluster and your stored procedure will be routed appropriately. In fact, you can create connections to multiple nodes on the server and your subsequent requests will be distributed to the various connections. For example, the following Java code creates the client object and then connects to all three nodes of the cluster. In this case, security is not enabled so no client configuration is needed:

```
try {
       client = ClientFactory.createClient();
       client.createConnection("server1.xyz.net");
       client.createConnection("server2.xyz.net");
       client.createConnection("server3.xyz.net");
} catch (java.io.IOException e) {
       e.printStackTrace();
       System.exit(-1);
}
```

Creating multiple connections has three major benefits:

- Multiple connections distribute the stored procedure requests around the cluster, avoiding a bottleneck where all requests are queued through a single host. This is particularly important when using asynchronous procedure calls or multiple clients.
- For Java applications, the VoltDB Java client library uses client affinity. That is, the client knows which server to send each request to based on the partitioning, thereby eliminating unnecessary network hops.
- Finally, if a server fails for any reason, when using K-safety the client can continue to submit requests through connections to the remaining nodes. This avoids a single point of failure between client and database cluster.

B.1.2. Using the Auto-Connecting Client

An easier way to create connections to all of the database servers is to use the "smart" or topology-aware client. By setting the Java client to be aware of the cluster topology, you only need to connect to one server and the client automatically connects to all of the servers in the cluster.

An additional advantage of the smart client is that it will automatically reconnect whenever the topology changes. That is, if a server fails and then rejoins the cluster, or new nodes are added to the cluster, the client will automatically create connections to the newly available servers.

You enable auto-connecting when you initialize the client object by setting the configuration option before creating the client object. For example:

```
org.voltdb.client.Client client = null;
ClientConfig config = new ClientConfig("","");
config.setTopologyChangeAware(true);
 try {
       client = ClientFactory.createClient(config);
       client.createConnection("server1.xyz.net");
             . . .
```

When `setTopologyChangeAware()` is set to true, the client library will automatically connect to all servers in the cluster and adjust its connections any time the cluster topology changes.

B.2. Invoking Stored Procedures

After your client creates the connection to the database, it is ready to call the stored procedures. You invoke a stored procedure using the `callProcedure()` method, passing the procedure name and variables as arguments. For example:

```
VoltTable[] results;

try { results = client.callProcedure("LookupFlight",                ❶
                                     origin,
                                     dest,
                                     departtime).getResults();      ❷
} catch (Exception e) {                                             ❸
     e.printStackTrace();
     System.exit(-1);
}
```

❶ The `callProcedure()` method takes the procedure name and the procedure's variables as arguments. The `LookupFlight()` stored procedure requires three variables: the originating airport, the destination, and the departure time.

❷ Once a synchronous call completes, you can evaluate the results of the stored procedure. The `callProcedure()` method returns a `ClientResponse` object, which includes information about the success or failure of the stored procedure. To retrieve the actual return values you use the `getResults()` method.

❸ Note that since `callProcedure()` can throw an exception (such as `VoltAbortException`) it is a good practice to perform error handling and catch known exceptions.

B.3. Invoking Stored Procedures Asynchronously

Calling stored procedures synchronously simplifies the program logic because your client application waits for the procedure to complete before continuing. However, for high performance applications looking to maximize throughput, it is better to queue stored procedure invocations asynchronously.

Asynchronous Invocation

To invoke stored procedures asynchronously, use the `callProcedure()` method with an additional first argument, a callback that will be notified when the procedure completes (or an error occurs). For example, to invoke a `NewCustomer()` stored procedure asynchronously, the call to `callProcedure()` might look like the following:

```
client.callProcedure(new MyCallback(),
                     "NewCustomer",
                     firstname,
                     lastname,
                     custID};
```

The following are other important points to note when making asynchronous invocations of stored procedures:

- Asynchronous calls to `callProcedure()` return control to the calling application as soon as the procedure call is queued.

- If the database server queue is full, `callProcedure()` will block until it is able to queue the procedure call. This is a condition known as backpressure. This situation does not normally happen unless the database cluster is not scaled sufficiently for the workload or there are abnormal spikes in the workload. See Section B.5.3, "Writing a Status Listener to Interpret Other Errors" for more information.
- Once the procedure is queued, any subsequent errors (such as an exception in the stored procedure itself or loss of connection to the database) are returned as error conditions to the callback procedure.

Callback Implementation

The callback procedure (`MyCallback()` in this example) is invoked after the stored procedure completes on the server. The following is an example of a callback procedure implementation:

```
static class MyCallback implements ProcedureCallback {
  @Override
  public void clientCallback(ClientResponse clientResponse) {
    if (clientResponse.getStatus() != ClientResponse.SUCCESS) {
       System.err.println(clientResponse.getStatusString());
   } else {
       myEvaluateResultsProc(clientResponse.getResults());
    }
  }
}
```

The callback procedure is passed the same `ClientResponse` structure that is returned in a synchronous invocation. `ClientResponse` contains information about the results of execution. In particular, the methods `getStatus()` and `getResults()` let your callback procedure determine whether the stored procedure was successful and evaluate the results of the procedure.

The VoltDB Java client is single threaded, so callback procedures are processed one at a time. Consequently, it is a good practice to keep processing in the callback to a minimum, returning control to the main thread as soon as possible. If more complex processing is required by the callback, creating a separate thread pool and spawning worker methods on a separate thread from within the asynchronous callback is recommended.

B.4. Closing the Connection

When the client application is done interacting with the VoltDB database, it is a good practice to close the connection. This ensures that any pending transactions are completed in an orderly way. The following example demonstrates how to close the client connection:

```
try {
    client.drain();
    client.close();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

There are two steps to closing the connection:

1. Call `drain()` to make sure all asynchronous calls have completed. The `drain()` method pauses the current thread until all outstanding asynchronous calls (and their callback procedures) complete. This call is not necessary if the application only makes synchronous procedure calls. However, there is no penalty for calling `drain()` and so it can be included for completeness in all applications.
2. Call `close()` to close all of the connections and release any resources associated with the client.

B.5. Handling Errors

A special situation to consider when calling VoltDB stored procedures is error handling. The VoltDB client interface catches most exceptions, including connection errors, errors thrown by the stored procedures themselves, and even exceptions that occur in asynchronous callbacks. These error conditions are not returned to the client application as exceptions. However, the application can still receive notification and interpret these conditions using the client interface.

The following sections explain how to identify and interpret errors that occur when executing stored procedures and in asynchronous callbacks. These include:

- Interpreting Execution Errors
- Handling Timeouts
- Writing a Status Listener to Interpret Other Errors

B.5.1. Interpreting Execution Errors

If an error occurs in a stored procedure (such as an SQL constraint violation), VoltDB catches the error and returns information about it to the calling application as part of the `ClientResponse` class. The `ClientResponse` class provides several methods to help the calling application determine whether the stored procedure completed successfully and, if not, what caused the failure. The two most important methods are `getStatus()` and `getStatusString()`.

```
static class MyCallback implements ProcedureCallback {
  @Override
  public void clientCallback(ClientResponse clientResponse) {
    final byte AppCodeWarm = 1;
    final byte AppCodeFuzzy = 2;
    if (clientResponse.getStatus() != ClientResponse.SUCCESS) {      ❶

        System.err.println(clientResponse.getStatusString());        ❷
    } else {

        if (clientResponse.getAppStatus() == AppCodeFuzzy) {         ❸
           System.err.println(clientResponse.getAppStatusString());
        };
        myEvaluateResultsProc(clientResponse.getResults());
    }
  }
}
```

❶ The `getStatus()` method tells you whether the stored procedure completed successfully and, if not, what type of error occurred. It is good practice to always check the status of the `ClientResponse` before evaluating the results of a procedure call, because if the status is anything but SUCCESS, there will not be any results returned. The possible values of `getStatus()` are:

- **CONNECTION_LOST** — The network connection was lost before the stored procedure returned status information to the calling application. The stored procedure may or may not have completed successfully.
- **CONNECTION_TIMEOUT** — The stored procedure took too long to return to the calling application. The stored procedure may or may not have completed successfully. See Section B.5.2, “Handling Timeouts” for more information about handling this condition.

- **GRACEFUL_FAILURE** — An error occurred and the stored procedure was gracefully rolled back.
- **RESPONSE_UNKNOWN** — This is a rare error that occurs if the coordinating node for the transaction fails before returning a response. The node to which your application is connected cannot determine if the transaction failed or succeeded before the coordinator was lost. The best course of action, if you receive this error, is to use a new query to determine if the transaction failed or succeeded and then take action based on that knowledge.
- **SUCCESS** — The stored procedure completed successfully.
- **UNEXPECTED_FAILURE** — An unexpected error occurred on the server and the procedure failed.
- **USER_ABORT** — The code of the stored procedure intentionally threw a UserAbort exception and the stored procedure was rolled back.

❷ If a `getStatus()` call identifies an error status other than SUCCESS, you can use the `getStatusString()` method to return a text message providing more information about the specific error that occurred.

❸ If you want the stored procedure to provide additional information to the calling application, there are two more methods to the `ClientResponse` that you can use. The methods `getAppStatus()` and `getAppStatusString()` act like `getStatus()` and `getStatusString()`, but rather than returning information set by VoltDB, `getAppStatus()` and `getAppStatusString()` return information set in the stored procedure code itself.

In the stored procedure, you can use the methods `setAppStatusCode()` and `setAppStatusString()` to set the values returned to the calling application by the stored procedure. For example:

```
/* stored procedure code */
final byte AppCodeWarm = 1;
final byte AppCodeFuzzy = 2;
               . . .
setAppStatusCode(AppCodeFuzzy);
setAppStatusString("I'm not sure about that...");
               . . .
```

B.5.2. Handling Timeouts

One particular error that needs special handling is if a connection or a stored procedure call times out. By default, the client interface only waits a specified amount of time (two minutes) for a stored procedure to complete. If no response is received from the server before the timeout period expires, the client interface returns control to your application, notifying it of the error. For synchronous procedure calls, the client interface returns the error CONNECTION_TIMEOUT to the procedure call. For asynchronous calls, the client interface invokes the callback including the error information in the `clientResponse` object.

It is important to note that CONNECTION_TIMEOUT does not necessarily mean the synchronous procedure failed. In fact, it is very possible that the procedure may complete and return information after the timeout error is reported. The timeout is provided to avoid locking up the client application when procedures are delayed or the connection to the cluster hangs for any reason.

Similarly, if no response of any kind is returned on a connection (even if no transactions are pending) within the specified timeout period, the client connection will timeout. When this happens, the connection is closed, any open stored procedures on that connection are closed with a return status of CONNEC-

TION_LOST, and then the client status listener callback method `connectionLost()` is invoked. Unlike a procedure timeout, when the connection times out, the connection no longer exists, so your client application will receive no further notifications concerning pending procedures, whether they succeed or fail.

CONNECTION_LOST does not necessarily mean a pending asynchronous procedure failed. It is possible that the procedure completed but was unable to return its status due to a connection failure. The goal of the connection timeout is to notify the client application of a lost connection in a timely manner, even if there are no outstanding procedures using the connection.

There are several things you can do to address potential timeouts in your application:

- Change the timeout period by calling either or both the methods `setProcedureCallTimeout()` and `setConnectionResponseTimeout()` on the `ClientConfig` object. The default timeout period is 2 minutes for both procedures and connections. You specify the timeout period in milliseconds, where a value of zero disables the timeout altogether. For example, the following client code resets the procedure timeout to 90 seconds and the connection timeout period to 3 minutes, or 180 seconds:

  ```
  config = new ClientConfig("advent","xyzzy");
  config.setProcedureCallTimeout(90 * 1000);
  config.setConnectionResponseTimeout(180 * 1000);
  client = ClientFactory.createClient(config);
  ```

- Catch and respond to the timeout error as part of the response to a procedure call. For example, the following code excerpt from a client callback procedure reports the error to the console and ends the callback:

  ```
  static class MyCallback implements ProcedureCallback {
    @Override
    public void clientCallback(ClientResponse response) {

      if (response.getStatus() == ClientResponse.CONNECTION_TIMEOUT) {
           System.out.println("A procedure invocation has timed out.");
           return;
      };
      if (response.getStatus() == ClientResponse.CONNECTION_LOST) {
           System.out.println("Connection lost before procedure response.");
           return;
      };
  ```

- Set a status listener to receive the results of any procedure invocations that complete after the client interface times out. See the following Section B.5.3, “Writing a Status Listener to Interpret Other Errors” for an example of creating a status listener for delayed procedure responses.

B.5.3. Writing a Status Listener to Interpret Other Errors

Certain types of errors can occur that the `ClientResponse` class cannot notify you about immediately. In these cases, an error happens and is caught by the client interface outside of the normal stored procedure execution cycle. If you want your application to address these situations, you need to create a listener, which is a special type of asynchronous callback that the client interface will notify whenever such errors occur. The types of errors that a listener addresses include:

Lost Connection

If a connection to the database cluster is lost or times out and there are outstanding asynchronous requests on that connection, the `ClientResponse` for those procedure calls will indicate that the

connection failed before a return status was received. This means that the procedures may or may not have completed successfully. If no requests were outstanding, your application might not be notified of the failure under normal conditions, since there are no callbacks to identify the failure. Since the loss of a connection can impact the throughput or durability of your application, it is important to have a mechanism for general notification of lost connections outside of the procedure callbacks.

Backpressure

If backpressure causes the client interface to wait, the stored procedure is never queued and so your application does not receive control until after the backpressure is removed. This can happen if the client applications are queuing stored procedures faster than the database cluster can process them. The result is that the execution queue on the server gets filled up and the client interface will not let your application queue any more procedure calls. Two ways to handle this situation programmatically are to:

- Let the client pause momentarily to let the queue subside. The asynchronous client interface does this automatically for you.
- Create multiple connections to the cluster to better distribute asynchronous calls across the database nodes.

Exceptions in a Procedure Callback

An error can occur in an asynchronous callback after the stored procedure completes. These exceptions are also trapped by the VoltDB client, but occur after the `ClientResponse` is returned to the application.

Late Procedure Responses

Procedure invocations that time out in the client may later complete on the server and return results. Since the client application can no longer react to this response inline (for example, with asynchronous procedure calls, the associated callback has already received a connection timeout error) the client may want a way to process the returned results.

For the sake of example, the following status listener does little more than display a message on standard output. However, in real world applications the listener would take appropriate actions based on the circumstances.

```
/*
*  Declare the status listener
*/
ClientStatusListenerExt mylistener = new ClientStatusListenerExt()       ❶
  {
    @Override
    public void connectionLost(String hostname, int port,                ❷
                               int connectionsLeft,
                               DisconnectCause cause)
      {
        System.out.printf("A connection to the database has been lost."
         + "There are %d connections remaining.\n", connectionsLeft);
      }
    @Override
    public void backpressure(boolean status)
      {
        System.out.println("Backpressure from the database "
         + "is causing a delay in processing requests.");
```

```
      }
    @Override
    public void uncaughtException(ProcedureCallback callback,
                                  ClientResponse r, Throwable e)
     {
        System.out.println("An error has occurred in a callback "
         + "procedure. Check the following stack trace for details.");
        e.printStackTrace();
     }
    @Override
    public void lateProcedureResponse(ClientResponse response,
                                    String hostname, int port)
     {
        System.out.printf("A procedure that timed out on host %s:%d"
           + " has now responded.\n", hostname, port);
     }
  };
/*
*  Declare the client configuration, specifying
*  a username, a password, and the status listener
*/
ClientConfig myconfig = new ClientConfig("username",                    ❸
                                         "password",
                                         mylistener);
/*
*  Create the client using the specified configuration.
*/
Client myclient = ClientFactory.createClient(myconfig);                ❹
```

By performing the operations in the order as described here, you ensure that all connections to the VoltDB database cluster use the same credentials for authentication and will notify the status listener of any error conditions outside of normal procedure execution.

❶ Declare a `ClientStatusListenerExt` listener callback. Define the listener before you define the VoltDB client or open a connection.

❷ The `ClientStatusListenerExt` interface has four methods that you can implement, one for each type of error situation:

- `connectionLost()`
- `backpressure()`
- `uncaughtException()`
- `lateProcedureResponse()`

❸ Define the client configuration `ClientConfig` object. After you declare your `ClientStatus-ListenerExt`, you define a `ClientConfig` object to use for all connections, which includes the username, password, and status listener. This configuration is then used to define the client next.

❹ Create a client with the specified configuration.