Using VoltDB

Abstract

This book explains how to use VoltDB to design, build, and run high performance applications.

V6.2

Using VoltDB

V6.2

This document was generated on April 11, 2016.

Table of Contents

List of Figures

List of Tables

List of Examples

About This Book

This book is a complete guide to VoltDB. It describes what VoltDB is, how it works, and — more importantly — how to use it to build high performance, data intensive applications. The book is divided into five parts:

Part 1: Getting Started	Explains what VoltDB is, how it works, how to install it, and how to start using VoltDB. The chapters in this section are: • Chapter 1, *Overview* • Chapter 2, *Installing VoltDB* • Chapter 3, *Starting the Database*
Part 2: Developing VoltDB Database Applications	Describes how to design and develop applications using VoltDB. The chapters in this section are: • Chapter 4, *Designing the Database Schema* • Chapter 5, *Designing Stored Procedures to Access the Database* • Chapter 6, *Designing VoltDB Client Applications* • Chapter 7, *Simplifying Application Development* • Chapter 8, *Using VoltDB with Other Programming Languages*
Part 3: Running VoltDB in a Cluster	Describes additional features useful for running a database in a cluster. The chapters in this section are: • Chapter 9, *Using VoltDB in a Cluster* • Chapter 10, *Availability* • Chapter 11, *Database Replication* • Chapter 12, *Security*
Part 4: Managing the Data	Provides techniques for ensuring data durability and integrity. The chapters in this section are: • Chapter 13, *Saving & Restoring a VoltDB Database* • Chapter 14, *Command Logging and Recovery* • Chapter 15, *Importing and Exporting Live Data*
Part 5: Reference Material	Provides reference information about the languages and interfaces used by VoltDB, including: • Appendix A, *Supported SQL DDL Statements* • Appendix B, *Supported SQL Statements* • Appendix C, *SQL Functions* • Appendix D, *VoltDB CLI Commands*

	• Appendix E, *Deployment File (deployment.xml)* • Appendix G, *System Procedures*

This book provides the most complete description of the VoltDB product. It includes features from both the open source Community Edition and the commercial Enterprise Edition. In general, the features described in Parts 1 and 2 are available in both versions of the product. Several features in Parts 3 and 4 are unique to the Enterprise Edition.

If you are new to VoltDB, the *VoltDB Tutorial* provides an introduction to the product and its features. The tutorial, and other books, are available on the web from http://docs.voltdb.com/.

Chapter 1. Overview

1.1. What is VoltDB?

VoltDB is a revolutionary new database product. Designed from the ground up to be the best solution for high performance business-critical applications, the VoltDB architecture is able to achieve 45 times higher throughput than current database products. The architecture also allows VoltDB databases to scale easily by adding processors to the cluster as the data volume and transaction requirements grow.

Current commercial database products are designed as general-purpose data management solutions. They can be tweaked for specific application requirements. However, the one-size-fits-all architecture of traditional databases limits the extent to which they can be optimized.

Although the basic architecture of databases has not changed significantly in 30 years, computing has. As have the demands and expectations of business applications and the corporations that depend on them.

VoltDB is designed to take full advantage of the modern computing environment:

- VoltDB uses in-memory storage to maximize throughput, avoiding costly disk access.
- Further performance gains are achieved by serializing all data access, avoiding many of the time-consuming functions of traditional databases such as locking, latching, and maintaining transaction logs.
- Scalability, reliability, and high availability are achieved through clustering and replication across multiple servers and server farms.

VoltDB is a fully ACID-compliant transactional database, relieving the application developer from having to develop code to perform transactions and manage rollbacks within their own application. By using ANSI standard SQL for the schema definition and data access, VoltDB also reduces the learning curve for experienced database designers.

1.2. Who Should Use VoltDB

VoltDB is not intended to solve all database problems. It is targeted at a specific segment of business computing.

VoltDB focuses specifically on *fast data*. That is, applications that must process large streams of data quickly. This includes financial applications, social media applications, and the burgeoning field of the Internet of Things. The key requirements for these applications are scalability, reliability, high availability, and outstanding throughput.

VoltDB is used today for traditional high performance applications such as capital markets data feeds, financial trade, telco record streams and sensor-based distribution systems. It's also used in emerging applications like wireless, online gaming, fraud detection, digital ad exchanges and micro transaction systems. Any application requiring high database throughput, linear scaling and uncompromising data accuracy will benefit immediately from VoltDB.

However, VoltDB is *not* optimized for all types of queries. For example, VoltDB is not the optimal choice for collecting and collating extremely large historical data sets which must be queried across multiple tables. This sort of activity is commonly found in business intelligence and data warehousing solutions, for which other database products are better suited.

To aid businesses that require *both* exceptional transaction performance *and* ad hoc reporting, VoltDB includes integration functions so that historical data can be exported to an analytic database for larger scale data mining.

1.3. How VoltDB Works

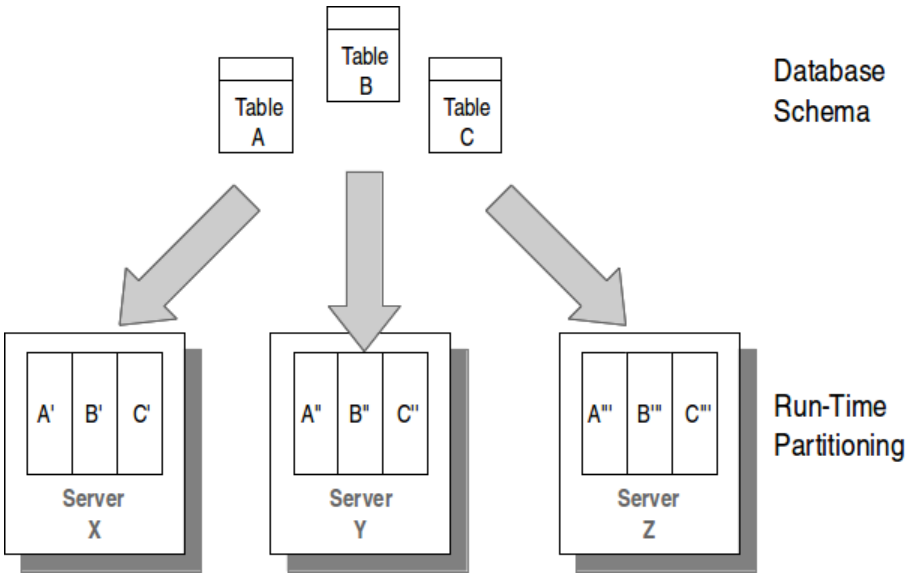
VoltDB is not like traditional database products. Each VoltDB database is optimized for a specific application by partitioning the database tables and the stored procedures that access those tables across multiple "sites" or partitions on one or more host machines to create the distributed database. Because both the data and the work is partitioned, multiple queries can be run in parallel. At the same time, because each site operates independently, each transaction can run to completion without the overhead of locking individual records that consumes much of the processing time of traditional databases. Finally, VoltDB balances the requirements of maximum performance with the flexibility to accommodate less intense but equally important queries that cross partitions. The following sections describe these concepts in more detail.

1.3.1. Partitioning

In VoltDB, each stored procedure is defined as a transaction. The stored procedure (i.e. transaction) succeeds or rolls back as a whole, ensuring database consistency.

By analyzing and precompiling the data access logic in the stored procedures, VoltDB can distribute both the data and the processing associated with it to the individual partitions on the cluster. In this way, each partition contains a unique "slice" of the data and the data processing. Each node in the cluster can support multiple partitions.

Figure 1.1. Partitioning Tables



1.3.2. Serialized (Single-Threaded) Processing

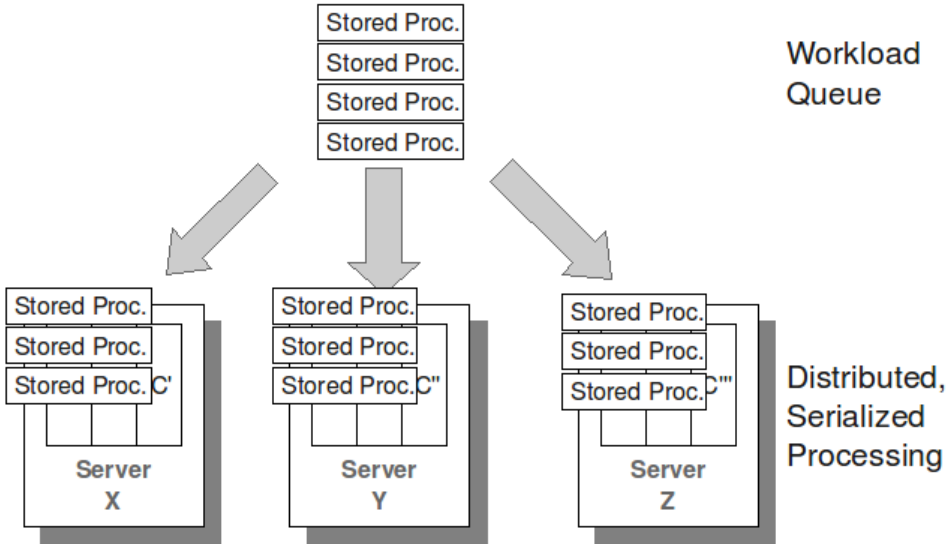
At run-time, calls to the stored procedures are passed to the appropriate partition. When procedures are "single-partitioned" (meaning they operate on data within a single partition) the server process executes the procedure by itself, freeing the rest of the cluster to handle other requests in parallel.

By using serialized processing, VoltDB ensures transactional consistency without the overhead of locking, latching, and transaction logs, while partitioning lets the database handle multiple requests at a time. As a

general rule of thumb, the more processors (and therefore the more partitions) in the cluster, the more transactions VoltDB completes per second, providing an easy, almost linear path for scaling an application's capacity and performance.

When a procedure does require data from multiple partitions, one node acts as a coordinator and hands out the necessary work to the other nodes, collects the results and completes the task. This coordination makes multi-partitioned transactions slightly slower than single-partitioned transactions. However, transactional integrity is maintained and the architecture of multiple parallel partitions ensures throughput is kept at a maximum.

Figure 1.2. Serialized Processing



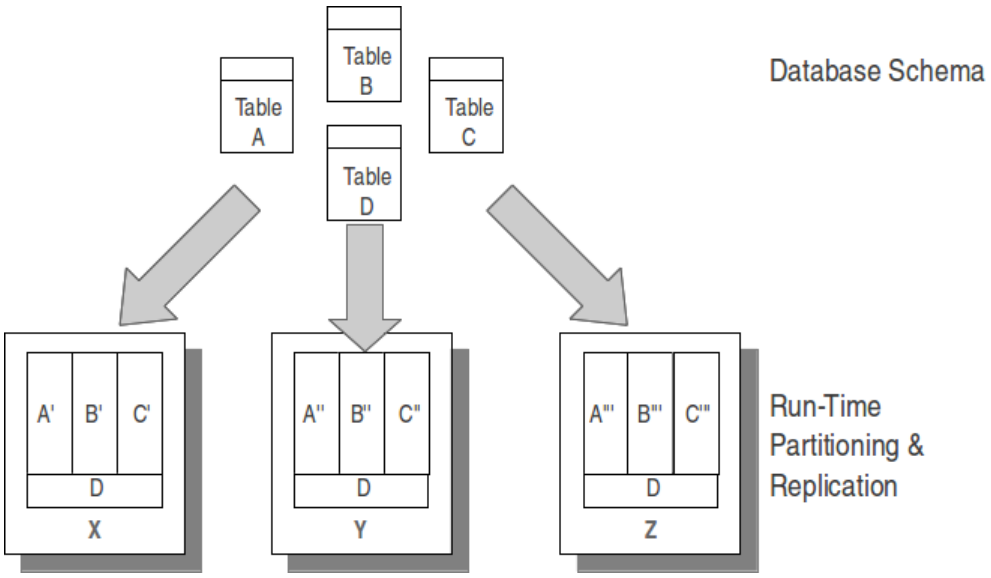
It is important to note that the VoltDB architecture is optimized for throughput over latency. The latency of any one transaction (the time from when the transaction begins until processing ends) is similar in VoltDB to other databases. However, the number of transactions that can be completed in a second (i.e. throughput) is orders of magnitude higher because VoltDB reduces the amount of time that requests sit in the queue waiting to be executed. VoltDB achieves this improved throughput by eliminating the overhead required for locking, latching, and other administrative tasks.

1.3.3. Partitioned vs. Replicated Tables

Tables are partitioned in VoltDB based on a column that you, the developer or designer, specify. When you choose partitioning columns that match the way the data is accessed by the stored procedures, it optimizes execution at runtime.

To further optimize performance, VoltDB allows certain database tables to be replicated to all partitions of the cluster. For small tables that are largely read-only, this allows stored procedures to create joins between this table and another larger table while remaining a single-partitioned transaction. For example, a retail merchandising database that uses product codes as the primary key may have one table that simply correlates the product code with the product's category and full name, Since this table is relatively small and does not change frequently (unlike inventory and orders) it can be replicated to all partitions. This way stored procedures can retrieve and return user-friendly product information when searching by product code without impacting the performance of order and inventory updates and searches.

Figure 1.3. Replicating Tables



1.3.4. Ease of Scaling to Meet Application Needs

The VoltDB architecture is designed to simplify the process of scaling the database to meet the changing needs of your application. Increasing the number of nodes in a VoltDB cluster both increases throughput (by increasing the number of simultaneous queues in operation) and increases the data capacity (by increasing the number of partitions used for each table).

Scaling up a VoltDB database is a simple process that doesn't require any changes to the database schema or application code. You can either:

- Save the database (using a snapshot or command logging), update the deployment file to identify the number of nodes for the resized cluster, then restart the database using either restore or recover to reload the data.
- Add nodes "on the fly" while the database is running.

1.4. Working with VoltDB Effectively

It is possible to use VoltDB like any other SQL database, creating tables and performing ad hoc SQL queries using standard SQL statements. However, to take full advantage of VoltDB's capabilities, it is best to design your schema and your stored procedures to maximize the use of partitioned tables and procedures. There are also additional features of VoltDB to increase the availability and durability of your data. The following sections explain how to work effectively with VoltDB, including:

- Chapters 2 and 3 explain how to install VoltDB and create a new database.
- Chapters 4 through 8 explain how to design your database, stored procedures, and client applications to maximize performance.
- Chapters 9 through 12 explain how to create and use VoltDB clusters to increase scalability and availability.
- Chapters 13 through 15 explain how VoltDB ensures the durability of your data and how you can integrate VoltDB with other data sources using export for complete business solutions

Chapter 2. Installing VoltDB

VoltDB is available in both an open source and an enterprise edition. The open source, or community, edition provides basic database functionality with all the transactional performance benefits of VoltDB. The enterprise edition provides additional features needed to support production environments, such as high availability, durability, and dynamic scaling.

Depending on which version you choose, the VoltDB software comes as either pre-built distributions or as source code. This chapter explains the system requirements for running VoltDB, how to install and upgrade the software, and what resources are provided in the kit.

2.1. Operating System and Software Requirements

The following are the requirements for developing and running VoltDB applications.

Table 2.1. Operating System and Software Requirements

Operating System	VoltDB requires a 64-bit Linux-based operating system. Kits are built and qualified on the following platforms: • CentOS version 6.6 or later, including 7.0 • Red Hat (RHEL) version 6.6 or later, including 7.0 • Ubuntu versions 12.04 and 14.04 Development builds are also available for Macintosh OS X 10.9 and later[1].
CPU	• Dual core[2] x86_64 processor • 64 bit • 1.6 GHz
Memory	4 Gbytes[3]
Java[4]	VoltDB Server: Java 8 Java and JDBC Client: Java 7 or 8
Required Software	NTP[5] Python 2.6 or later release of 2.x
Recommended Software	Eclipse 3.x (or other Java IDE)

Footnotes:

1. CentOS 6.6, CentOS 7.0, RHEL 6.6, RHEL 7.0, and Ubuntu 12.04 and 14.04 are the only officially supported operating systems for VoltDB. However, VoltDB is tested on several other POSIX-compliant and Linux-based 64-bit operating systems, including Macintosh OS X 10.9.

2. Dual core processors are a minimum requirement. Four or eight physical cores are recommended for optimal performance.

3. Memory requirements are very specific to the storage needs of the application and the number of nodes in the cluster. However, 4 Gigabytes should be considered a minimum configuration.

4. VoltDB supports JDKs from OpenJDK or Oracle/Sun.

5. NTP minimizes time differences between nodes in a database cluster, which is critical for VoltDB. All nodes of the cluster should be configured to synchronize against the same NTP server. Using a single local NTP server is recommended, but not required.

2.2. Installing VoltDB

VoltDB is distributed as a compressed tar archive for each of the supported platforms. The file name identifies the platform, the edition (community or enterprise) and the version number. The best way to install VoltDB is to unpack the distribution kit as a folder in the home directory of your personal account, like so:

```
$ tar -zxvf voltdb-ent-6.0.tar.gz -C $HOME/
```

Installing into your personal directory gives you full access to the software and is most useful for development.

If you are installing VoltDB on a production server where the database will be run, you may want to install the software into a standard system location so that the database cluster can be started with the same commands on all nodes. The following shell commands install the VoltDB software in the folder `/opt/voltdb`:

```
$ sudo tar -zxvf voltdb-ent-6.0.tar.gz -C /opt
$ cd /opt
$ sudo mv voltdb-ent-6.0 voltdb
```

Note that installing as root using the sudo command makes the installation folders read-only for non-privileged accounts. Which is why installing in $HOME is recommended for running the sample applications and other development activities.

2.2.1. Upgrading From Older Versions

When upgrading from a previous version of VoltDB — especially with an existing database — there are a few key steps you should take to ensure a smooth migration. The recommended steps for upgrading an existing database are:

1. Place the database in admin mode (**voltadmin pause**).
2. Perform a manual snapshot of the database (**voltadmin save --blocking**).
3. Shutdown the database (**voltadmin shutdown**).
4. Upgrade VoltDB.
5. Create a new database using the **voltdb create --force** command, and starting in admin mode (specified in the deployment file).
6. Restore the snapshot created in Step #2 (**voltadmin restore**).
7. Return the database to normal operations (**voltadmin resume**).

2.2.2. Building a New VoltDB Distribution Kit

If you want to build the open source VoltDB software from source (for example, if you want to test recent development changes), you must first fetch the VoltDB source files. The VoltDB sources are stored in a GitHub repository.

The VoltDB sources are designed to build and run on 64-bit Linux-based or 64-bit Macintosh platforms. However, the build process has not been tested on all possible configurations. Attempts to build the sources on other operating systems may require changes to the build files and possibly to the sources as well.

Once you obtain the sources, use Ant 1.7 or later to build a new distribution kit for the current platform:

```
$ ant dist
```

The resulting distribution kit is created as `obj/release/volt-n.n.nn.tar.gz` where *n.n.nn* identifies the current version and build numbers. Use this file to install VoltDB according to the instructions in Section 2.2, “Installing VoltDB”.

2.3. Setting Up Your Environment

VoltDB comes with shell command scripts that simplify the process of developing and deploying VoltDB applications. These scripts are in the /bin folder under the installation root and define short-cut commands for executing many VoltDB actions. To make the commands available to your session, you must include the /bin directory as part your PATH environment variable.

You can add the `/bin` directory to your PATH variable by redefining PATH. For example, the following shell command adds `/bin` to the end of the environment PATH, assuming you installed the VoltDB Enterprise Edition as `/voltdb-ent-n.n` in your $HOME directory:

```
$ export PATH="$PATH:$HOME/voltdb-ent-n.n/bin"
```

To avoid having to redefine PATH every time you create a new session, you can add the preceding command to your shell login script. For example, if you are using the bash shell, you would add the preceding command to the `$HOME/.bashrc` file.

2.4. What is Included in the VoltDB Distribution

Table 2.2 lists the components that are provided as part of the VoltDB distribution.

Table 2.2. Components Installed by VoltDB

Component	Description
VoltDB Software & Runtime	The VoltDB software comes as Java archives (.JAR files) and a callable library that can be found in the `/voltdb` subfolder. Other software libraries that VoltDB depends on are included in a separate `/lib` subfolder.
Example Applications	VoltDB comes with several example applications that demonstrate VoltDB capabilities and performance. They can be found in the `/examples` subfolder.
VoltDB Management Center	VoltDB Management Center is a browser-based management tool for monitoring, examining, and querying a running VoltDB database. The Management Center is bundled with the VoltDB server software. You can start the Management Center by connecting to the HTTP port of a running VoltDB database server. For example, `http://`

Component	Description
	`voltsvr:8080/`. Note that the httpd server and JSON interface must be enabled on the server to be able to access the Management Center.
Shell Commands	The `/bin` subfolder contains executable scripts to perform common VoltDB tasks, such as starting the VoltDB server process and issuing database queries from the command line using sqlcmd, Add the `/bin` subfolder to your PATH environment variable to use the following shell commands: csvloader jdbcloader kafkaloader sqlcmd voltadmin voltdb
Documentation	Online documentation, including the full manuals and javadoc describing the Java programming interface, is available in the `/doc` subfolder.

2.5. VoltDB in Action: Running the Sample Applications

Once you install VoltDB, you can use the sample applications to see VoltDB in action and get a better understanding of how it works. The easiest way to do this is to set directory to the `/examples` folder where VoltDB is installed. Each sample application has its own subdirectory and a run.sh script to simplify building and running the application. See the README file in the `/examples` subfolder for a complete list of the applications and further instructions.

Once you get a taste for what VoltDB can do, we recommend following the VoltDB tutorial to understand how to create your own applications using VoltDB.

Chapter 3. Starting the Database

This chapter describes the procedures for starting and stopping a VoltDB database and includes details about configuring the database. The chapter contains the following sections:

- Section 3.1, “Initializing a VoltDB Database”
- Section 3.2, “Initializing the Database on a Cluster”
- Section 3.3, “Updating Nodes on the Cluster”
- Section 3.4, “Stopping a VoltDB Database”
- Section 3.5, “Restarting a VoltDB Database”
- Section 3.6, “Defining the Cluster Configuration”

3.1. Initializing a VoltDB Database

Use the **voltdb** command with the **create** action to start an empty, single-node database suitable for developing and testing a database and application:

```
$ voltdb create
```

Other database startup actions include adding or rejoining nodes to the cluster, and recovering the database from snapshots after the database stops. More startup arguments identify such information as a host to manage startup in a cluster, and a deployment file containing cluster configuration options. The rest of this chapter covers these issues and more in detail.

Important

If the database you are working on has stopped, use **voltdb recover** to restart it. *Do not rerun the voltdb create command or your schema and data will be reinitialized to an empty database.* Later in this chapter we explain how to safely stop and restart a VoltDB database.

3.2. Initializing the Database on a Cluster

To start an empty VoltDB database on a cluster, you will need the following information for the **voltdb create** command:

- **Deployment file location:** The deployment file defines the cluster configuration including the number of nodes. The deployment file must be identical on all nodes for the cluster to start, so be sure you copy the deployment file to all nodes of the cluster. We’ll describe details about the deployment file in Section 3.6, “Defining the Cluster Configuration”.
- **Host name:** Provide the hostname or IP address of the cluster’s host node, which coordinates the startup of all the nodes in the cluster.
- **License file location:** If you are using the VoltDB Enterprise Edition, provide a license file on the host node. Only the host node requires the license file when starting a cluster.

For each node of the cluster, log in and start the server process using the same command. For example, the following **voltdb create** command starts the database cluster specifying the location and name of the

deployment file and naming voltsvr1 as the host node. Be sure the number of nodes on which you run the command match the number of nodes defined in the deployment file.

```
$ voltdb create --deployment=deployment.xml -–host=voltsvr1
```

Or you can also use shortened forms for the argument flags:

```
$ voltdb create -d deployment.xml -H voltsvr1
```

VoltDB looks for the license file on the host as a file named `license.xml` in three locations, in the following order:

1. The current working directory
2. The directory where the VoltDB image files are installed (usually in the `/voltdb` subfolder of the installation directory)
3. The current user's home directory

If the license file is not in any of these locations, you must explicitly identify it when you run the **voltdb** command on the host node using the `--license` or `-l` flag. For example, the command on the host node might be:

```
$ voltdb create -d deployment.xml -H voltsvr1 \
                -l /usr/share/voltdb-license.xml
```

When starting a VoltDB database on a cluster, the VoltDB server process performs the following actions:

1. If you are starting the database on the node identified as the host node, it waits for initialization messages from the remaining nodes. The host can be any node in the cluster and plays a special role during startup by managing the cluster initiation process. It is important that all nodes in the cluster can resolve the hostname or IP address of the host node you specify.
2. If you are starting the database on a non-host node, it sends an initialization message to the host indicating that it is ready. The database is not operational until the correct number of nodes (as specified in the deployment file) have connected.
3. Once all the nodes have sent initialization messages, the host sends out a message to the other nodes that the cluster is complete. Once the startup procedure is complete, the host's role is over and it becomes a peer like every other node in the cluster. It performs no further special functions.

Manually logging on to each node of the cluster every time you want to start the database can be tedious. Instead, you can use secure shell (ssh) to execute shell commands remotely. By creating an ssh script (with the appropriate permissions) you can copy files and/or start the datab ase on each node in the cluster from a single script.

3.3. Updating Nodes on the Cluster

A cluster is a dynamic system in which nodes might be stopped either deliberately or by unforeseen circumstances, or nodes might be added to the cluster on-the-fly to scale the database for improved performance. The **voltdb** command provides the following additional startup actions for nodes of a running VoltDB database:

- Use the **voltdb add** command to start up and add a new node to the running database cluster. See Section 9.2.1, “Adding Nodes with Elastic Scaling”.

- Use the **voltdb rejoin** command to restart a node that was previously part of the cluster but had stopped running. See Section 10.3, “Recovering from System Failures”.

3.4. Stopping a VoltDB Database

Once the VoltDB database is up and running, you can shut it down by stopping the VoltDB server processes on each cluster node. However, it is easier to stop the database as a whole on the entire cluster with a single command. You can do this either programmatically with the @Shutdown system procedure (from any node) or interactively with the **voltadmin shutdown** command. You do not have to issue commands on each node. For example, entering the following command without specifying a host server will shut down the database cluster the current system is part of.

```
$ voltadmin shutdown
```

To shutdown a remote database running on servers of a different cluster, use the `--host`, `--user`, and `--password` arguments to access the remote database. For example, the following command shuts down the VoltDB database that includes the server zeus:

```
$ voltadmin shutdown --host=zeus
```

Because VoltDB is an in-memory database, once the database server process stops, the database schema and the data itself are removed from memory. However, VoltDB saves the information to disk. To retain the schema and data across sessions, VoltDB provides database snapshots and command logging. A snapshot is a point-in-time copy of the database contents written to disk. Command logging provides, in addition to periodic snapshots, a log of all stored procedures that are initiated at each partition. Command logging is enabled by default to ensure your database is not lost. To learn more about how to save and restore snapshots of the database, see Chapter 13, *Saving & Restoring a VoltDB Database*. To learn more about using command logging and recovery to save and reload the database automatically, see Chapter 14, *Command Logging and Recovery*.

You can pause the database using the @Pause system procedure or **voltadmin pause** to restrict clients from accessing it while you perform changes in administration mode. You resume the database using the @Resume system procedure or the **voltadmin resume** command. See the *VoltDB Administrator's Guide* for more about modes of operation.

3.5. Restarting a VoltDB Database

To restart a VoltDB database use the **voltdb** command with the **recover** action. For example, the following command restarts a single-node database:

```
$ voltdb recover
```

To restart a database on a cluster, execute **voltdb recover** on each node and specify the deployment file and the cluster’s host node. For example, assuming voltsvr1 is the cluster’s host node, a command such as the following would be executed on all nodes of the cluster. Be sure the number of nodes on which you run the command match the number of nodes defined in the deployment file:

```
$ voltdb recover -d deployment.xml -H voltsvr1
```

Remember that when executing the command on the host node, VoltDB needs to access the license file. If the license file is not in any of the standard VoltDB locations you must explicitly identify it. For example, the command on the host node might be:

```
$ voltdb recover -d deployment.xml  -H voltsvr1 \
```

```
-l /usr/share/voltdb-license.xml
```

3.6. Defining the Cluster Configuration

An important aspect of a VoltDB database is the physical layout of the cluster that runs the database. You define the cluster configuration in the deployment file. The deployment file is an XML file, which you specify when you start the database to establish the correct cluster topology. The basic syntax of the deployment file is as follows:

```
<?xml version="1.0"?>
<deployment>
   <cluster hostcount="n"
            kfactor="n"
   />
</deployment>
```

The attributes of the `<cluster>` tag define the physical layout of the hardware that will run the database. The key attributes are:

- **hostcount** — specifies the number of nodes in the cluster.
- **kfactor** — specifies the K-safety value to use for durability when creating the database. The K-safety value controls the duplication of database partitions. This attribute is optional, so if you do not specify a value the default is zero, which means there is no partition duplication. See Chapter 10, *Availability* for more information about K-safety.

In the simplest case — when running on a single node with no special options enabled — you can skip the deployment file altogether on the **voltdb** command line. If you do not specify a deployment file or host, VoltDB defaults to one node, eight execution sites per host, and a K-safety value of zero.

The deployment file is used to enable and configure many other runtime options related to the database, which are described later in this book. For example, the deployment file can specify:

- The number of execution sites per host on which partitions can be distributed. This setting defaults to eight sites per host, which is appropriate for most situations. If you choose to tune the number of sites per host, see Section 3.6.1, "Determining How Many Sites per Host" for how to determine the optimal value.
- Whether security is enabled and what users and passwords are needed to authenticate clients at runtime. See Chapter 12, *Security* for more information.
- A schedule for saving automatic snapshots of the database. See Section 13.2, "Scheduling Automated Snapshots".
- Control of network fault protection to avoid partition errors. See Section 10.4.2, "Using Network Fault Protection".
- Properties for exporting data to other databases. See Chapter 15, *Importing and Exporting Live Data*.

For the complete deployment file features, see Appendix E, *Deployment File (deployment.xml)*.

3.6.1. Determining How Many Sites per Host

There is very little penalty for allocating more sites than needed for the partitions the database will use (except for incremental memory usage). Consequently, VoltDB defaults to eight sites per node to provide reasonable performance on most modern system configurations. This default does not normally need to be

changed. However, for systems with a large number of available processes (16 or more) or older machines with fewer than 8 processors and limited memory, you may wish to tune the `sitesperhost` attribute.

The number of sites needed per node is related to the number of processor cores each system has, the optimal number being approximately 3/4 of the number of CPUs reported by the operating system. For example, if you are using a cluster of dual quad-core processors (in other words, 8 cores per node), the optimal number of partitions is likely to be 6 or 7 sites per node.

```
<?xml version="1.0"?>
<deployment>
   <cluster . . .
            sitesperhost="6"
   />
</deployment>
```

For systems that support hyperthreading (where the number of physical cores support twice as many threads), the operating system reports twice the number of physical cores. In other words, a dual quad-core system would report 16 virtual CPUs. However, each partition is not quite as efficient as on non-hyperthreading systems. So the optimal number of sites is more likely to be between 10 and 12 per node in this situation.

Because there are no hard and set rules, the optimal number of sites per node is best calculated by actually benchmarking the application to see what combination of cores and sites produces the best results. However, it is important to remember that all nodes in the cluster will use the same number of sites. So the best performance is achieved by using a cluster with all nodes having the same physical architecture (i.e. cores).

3.6.2. Configuring Paths for Runtime Features

An important aspect of some runtime features is that they make use of disk resources for persistent storage across sessions. For example, automatic snapshots need a directory for storing snapshots of the database contents. Similarly, export uses disk storage for writing overflow data if the export connector cannot keep up with the export queue.

You can specify individual paths for each feature, or you can specify a root directory where VoltDB will create subfolders for each feature as needed. If you do not specify a root path or a specific feature path, the root path defaults to ./voltdbroot in the current default directory and VoltDB creates the directory (and subfolders) as needed.

To specify a common root, use the `<voltdbroot>` tag (as a child of `<paths>`) to specify where VoltDB will store disk files. If you specify a root directory path, the directory must exist and the process running VoltDB must have write access to it. VoltDB does not attempt to create an explicitly named root directory path if it does not exist. For example, the following `<paths>` tag specifies `/opt/voltdb` as the root directory:

```
<paths>
   <voltdbroot path="/opt/voltdb" />
</paths>
```

You can also identify specific path locations for individual features including:

- `<commandlog>`
- `<commandlogsnapshot>`
- `<exportoverflow>`

- `<snapshots>`

If you name a specific feature path and it does not exist, VoltDB will attempt to create it for you. For example, the `<exportoverflow>` path contains temporary data which can be deleted periodically. The following excerpt from a deployment file specifies `/opt/voltdb` as the default root but `/opt/overflow` as the directory for export overflow.

```
<paths>
   <voltdbroot path="/opt/voltdb" />
   <exportoverflow path="/opt/overflow" />
</paths>
```

3.6.3. Verifying your Hardware Configuration

The deployment file defines the expected configuration of your database cluster. However, there are several important aspects of the physical hardware and operating system configuration that you should be aware of before running VoltDB:

- VoltDB can operate on heterogeneous clusters. However, best performance is achieved by running the cluster on similar hardware with the same type of processors, number of processors, and amount of memory on each node.
- All nodes must be able to resolve the IP addresses and host names of the other nodes in the cluster. That means they must all have valid DNS entries or have the appropriate entries in their local hosts file.
- You must run the Network Time Protocol (NTP) on all of the cluster nodes, preferably synchronizing against the same local time server. If the time skew between nodes in the cluster is greater than 200 milliseconds, VoltDB cannot start the database.
- It is strongly recommended that you run NTP with the `-x` argument. Using `ntpd -x` stops the server from adjusting time backwards for all but very large increments. If the server time moves backward, VoltDB must pause and wait for time to catch up.

Chapter 4. Designing the Database Schema

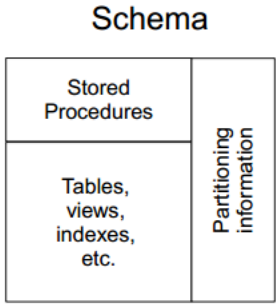
VoltDB is a relational database product. Relational databases consist of tables and columns, with constraints, indexes, and views. VoltDB also uses standard SQL database definition language (DDL) statements to specify the database schema. So designing the schema for a VoltDB database uses the same skills and knowledge as designing a database for Oracle, MySQL, or any other relational database product.

This guide describes the stages of application design by dividing the work into three chapters:

- **Design the schema** in DDL to define the database structure. Schema design is covered in this chapter.
- **Design stored procedures** to access data in the database. Stored procedures provide client applications an application programming interface (API) to the database. Stored procedures are covered in Chapter 5, *Designing Stored Procedures to Access the Database*.
- **Design clients** to provide business logic and also connect to the database to access data. Client application design is covered in Chapter 6, *Designing VoltDB Client Applications*.

The database schema is a specification that describes the structure of the VoltDB database such as tables and indexes, identifies the stored procedures that access data in the database, and defines the way tables and stored procedures are partitioned for fast data access. When designing client applications to use the database, the schema specifies the details needed about data types, tables, columns, and so on.

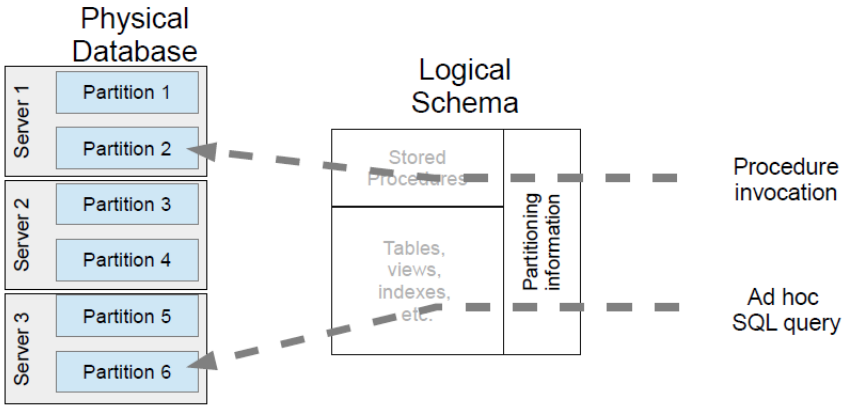
Figure 4.1. Components of a Database Schema



Along with designing your database tables, an important aspect of VoltDB database design is partitioning, which provides much more efficient access to data and processing. Partitioning distributes the rows of a table and the processing to access the table across several, independent partitions instead of one. Your design requires coordinating the partitioning of both database tables and the stored procedures that access the tables. At design time you choose a column on which to partition a table's rows. You also partition stored procedures on the same column if they use the column to identify which rows to operate on in the table.

At runtime, VoltDB decides which cluster nodes and partitions to use for the table partitions and consistently allocates rows to the appropriate partition. Figure 4.2, “Partitions Distribute Table Data and Stored Procedure Processing” shows how when data is inserted into a partitioned table, VoltDB automatically allocates the data to the correct partition. Also, when a partitioned stored procedure is invoked, VoltDB automatically executes the stored procedure in the single partition that has the data requested.

Figure 4.2. Partitions Distribute Table Data and Stored Procedure Processing



The following sections of this chapter provide guidelines for designing VoltDB database schemas. Although gathering business requirements is a typical first step in database application design, it is outside the scope of this guide.

4.1. How to Enter DDL Statements

You use standard SQL DDL statements to design your schema. For a full list of valid VoltDB DDL, see Appendix A, *Supported SQL DDL Statements*. The easiest way to enter your DDL statements is using VoltDB's command line utility, sqlcmd. Using sqlcmd you can input DDL statements in several ways:

- Redirect standard input from a file when you start sqlcmd:

```
$ sqlcmd < myschema.sql
```

- Import from a file using the sqlcmd file directive:

```
$ sqlcmd
1> file myschema.sql;
```

- Enter DDL directly at the sqlcmd prompt:

```
$ sqlcmd
1>
2> CREATE TABLE Customer (
3>   CustomerID INTEGER UNIQUE NOT NULL,
4>   FirstName VARCHAR(15),
5>   LastName VARCHAR (15),
6>   PRIMARY KEY(CustomerID)
7> );
```

- Copy DDL from another application and paste it into the sqlcmd prompt:

```
$ sqlcmd
1> CREATE TABLE Flight (
2>   FlightID INTEGER UNIQUE NOT NULL,
3>   DepartTime TIMESTAMP NOT NULL,
4>   Origin VARCHAR(3) NOT NULL,
5>   Destination VARCHAR(3) NOT NULL,
6>   NumberOfSeats INTEGER NOT NULL,
```

```
7>   PRIMARY KEY(FlightID)
8> );
```

The following sections show how to design and create schema objects. DDL statements and techniques for changing a schema are described later in Section 4.6, “Modifying the Schema”.

4.2. Creating Tables and Primary Keys

The schema in this section is referred to throughout the design chapters of this guide. Let's assume you are designing a flight reservation system. At its simplest, the application requires database tables for the flights, the customers, and the reservations. Example 4.1, “DDL Example of a Reservation Schema” shows how the schema looks as defined in standard SQL DDL. For the VoltDB-specific details for creating tables, see CREATE TABLE. When defining the data types for table columns, refer to Table A.1, “Supported SQL Datatypes”.

Example 4.1. DDL Example of a Reservation Schema

```
CREATE TABLE Flight (
   FlightID INTEGER UNIQUE NOT NULL,
   DepartTime TIMESTAMP NOT NULL,
   Origin VARCHAR(3) NOT NULL,
   Destination VARCHAR(3) NOT NULL,
   NumberOfSeats INTEGER NOT NULL,
   PRIMARY KEY(FlightID)
);

CREATE TABLE Reservation (
   ReserveID INTEGER NOT NULL,
   FlightID INTEGER NOT NULL,
   CustomerID INTEGER NOT NULL,
   Seat VARCHAR(5) DEFAULT NULL,
   Confirmed TINYINT DEFAULT '0'
);

CREATE TABLE Customer (
   CustomerID INTEGER UNIQUE NOT NULL,
   FirstName VARCHAR(15),
   LastName VARCHAR (15),
   PRIMARY KEY(CustomerID)
);
```

To satisfy entity integrity you can specify a table's primary key by providing the usual PRIMARY KEY constraint on one or more of the table’s columns. To create a simple key, apply the PRIMARY KEY constraint to one of the table's existing columns whose values are unique and not null, as shown in Example 4.1, “DDL Example of a Reservation Schema”.

To create a composite primary key from a combination of columns in a table, apply the PRIMARY KEY constraint to multiple columns with typical DDL such as the following:

```
$ sqlcmd
1> CREATE TABLE Customer (
2>   FirstName VARCHAR(15),
3>   LastName VARCHAR (15),
4>   CONSTRAINT pkey PRIMARY KEY (FirstName, LastName)
5> );
```

4.3. Analyzing Data Volume and Workload

A schema is not all you need to define the database effectively. You also need to know the expected volume and workload on the database. For our example, let's assume that we expect the following volume of data at any given time:

- Flights: 2,000
- Reservations: 200,000
- Customers: 1,000,000

This additional information about the volume and workload affects the design of both the database and the client application, because it impacts what SQL queries need to be written for accessing the data and what attributes (columns) to share between tables. Table 4.1, “Example Application Workload” defines a set of procedures the application must perform. The table also shows the estimated workload as expected frequency of each procedure. Procedures in bold modify the database.

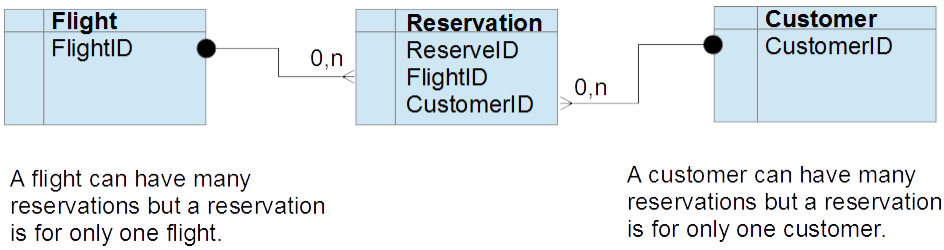
Table 4.1. Example Application Workload

Use Case	Frequency
Look up a flight (by origin and destination)	10,000/sec
See if a flight is available	5,000/sec
Make a reservation	1,000/sec
Cancel a reservation	200/sec
Look up a reservation (by reservation ID)	200/sec
Look up a reservation (by customer ID)	100/sec
Update flight info	1/sec
Take off (close reservations and archive associated records)	1/sec

You can make your procedures that access the database transactional by defining them as VoltDB stored procedures. This means each stored procedure call completes or rolls back if necessary, thus maintaining data integrity. Stored procedures are described in detail in Chapter 5, *Designing Stored Procedures to Access the Database*.

In our analysis we also need to consider referential integrity, where relationships are maintained between tables with shared columns that link tables together. For example, Figure 4.3, “Diagram Representing the Flight Reservation System” shows that the Flight table links to the Reservation table where FlightID is the shared column. Similarly, the Customer table links to the Reservation table where CustomerID is the common column.

Figure 4.3. Diagram Representing the Flight Reservation System



Since VoltDB stored procedures are transactional, you can use stored procedures to maintain referential integrity between tables as data is added or removed. For example, if a customer record is removed from the Customer table, all reservations for that customer need to be removed from the Reservations table as well.

With VoltDB, you use all this additional information about volume and workload to configure the database and optimize performance. Specifically, you want to partition the individual tables to ensure efficiency. Partitioning is described next.

4.4. Partitioning Database Tables

This section discusses how to partition a database to maximize throughput, using the flight reservation case study as an example. To partition a table, you choose a column of the table that VoltDB can use to uniquely identify and distribute the rows into partitions. The goal of partitioning a database table is to ensure that the most frequent transactions on the table execute in the same partition as the data accessed. We call this a *single-partitioned transaction*. Thus the stored procedure must uniquely identify a row by the partitioning column value. This is particularly important for queries that modify the data, such as INSERT, UPDATE, and DELETE statements.

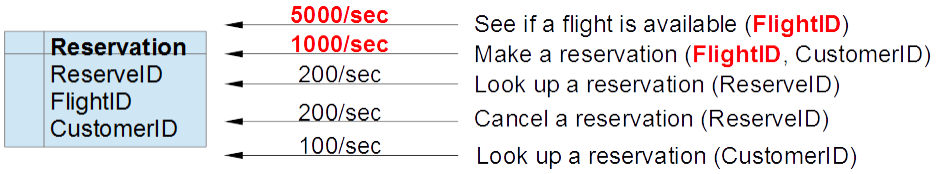
Looking at the workload for the reservation system in the previous section, the important transactions to focus on are:

- Look up a flight
- See if a flight is available
- Look up a reservation
- **Make a reservation**

Of these transactions, only the last modifies the database.

4.4.1. Choosing a Column on which to Partition Table Rows

We will discuss the Flight table later, but first let's look at the Reservation table. Looking at the schema alone (Example 4.1), ReserveID might look like a good attribute to use to partition the table rows. However, looking at the workload, there are only two transactions that are keyed to the ReserveID (“Cancel a reservation” and “Look up a reservation (by reservation ID)”), each of which occur only 200 times a second. Whereas, “See if a flight is available” , which requires looking up reservations by the FlightID, occurs 5,000 times a second, or 25 times as frequently. Therefore, the Reservation table is best partitioned on the FlightID column.



Moving to the Customer table, CustomerID is used for most data access. Although customers might need to look up their record by name, the first and last names are not guaranteed to be unique. Therefore, CustomerID is the best column to use for partitioning the Customer table.

```
CREATE TABLE Customer (
   CustomerID INTEGER UNIQUE NOT NULL,
```

```
   FirstName VARCHAR(15),
   LastName VARCHAR (15),
   PRIMARY KEY(CustomerID)
);
```

4.4.2. Specifying Partitioned Tables

Once you choose the column to use for partitioning a database table, you define your partitioning choices in the database schema. Specifying the partitioning along with the schema DDL helps keep all of the database structural information in one place.

You define the partitioning scheme using VoltDB's PARTITION TABLE statement, specifying the partitioning column for each table. For example, to specify FlightID and CustomerID as the partitioning columns for the Reservation and Customer tables, respectively, your database schema must include the following statements:

```
$ sqlcmd
1> PARTITION TABLE Reservation ON COLUMN FlightID;
2> PARTITION TABLE Customer ON COLUMN CustomerID;
```

4.4.3. Design Rules for Partitioning Tables

The following are the rules to keep in mind when choosing a column by which to partition table rows:

- **There can be only one partition column per table.** If you need to partition a table on two columns (for example first and last name), add an additional column (fullname) that combines the values of the two columns and use this new column to partition the table.
- **If the table has a primary key, the partitioning column must be included in the primary key.**
- **Any integer or string column can identify the partition.** VoltDB can partition rows on any column that is an integer (TINYINT, SMALLINT, INTEGER, or BIGINT) or string (VARCHAR) datatype. (See also Table A.1, “Supported SQL Datatypes”.)
- **Partition column values cannot be null.** The partition columns do not need to have unique values, but you must specify NOT NULL in the schema for the partition column. Numeric fields can be zero and string or character fields can be empty, but the column cannot contain a null value.

The following are some additional recommendations:

- Choose a column with a reasonable distribution of values so that rows of data will be evenly partitioned.
- Choose a column that maximizes use of single-partitioned stored procedures. If one procedure uses column A to lookup data and two procedures use column B to lookup data, partition on column B. The goal of partitioning is to make the most frequent transactions single-partitioned.
- If you partition more than one table on the same column attribute, VoltDB will partition them together.

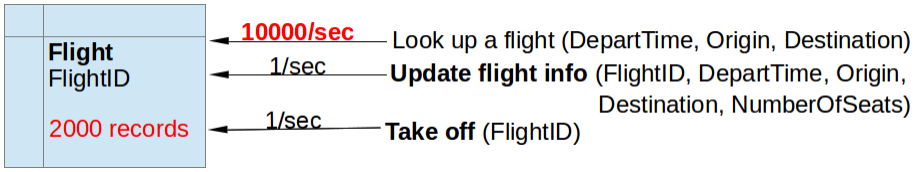
4.5. Replicating Database Tables

With VoltDB, tables are either partitioned or replicated across all nodes and sites of a VoltDB database. Smaller, mostly read-only tables are good candidates for replication. Note also that if a table needs to be accessed frequently by columns other than the partitioning column, the table should be replicated instead because there is no guarantee that a particular partition includes the data that the query seeks.

The previous section describes how to partition the Reservation and Customer tables as examples, but what about the Flight table? It is possible to partition the Flight table (for example, on the FlightID column). However, not all tables benefit from partitioning.

4.5.1. Choosing Replicated Tables

Looking at the workload of the flight reservation example, the Flight table has the most frequent accesses (at 10,000 a second). However, these transactions are read-only and may involve any combination of three columns: the departure time, the point of origin, and the destination. This makes it hard to partition the table in a way that would make the transaction single-partitioned because the lookup is not restricted to one table column.



Fortunately, the number of flights available for booking at any given time is limited (estimated at 2,000) and so the size of the table is relatively small (approximately 36 megabytes). In addition, the vast majority of the transactions involving the Flight table are read-only except when new flights are added and at take-off (when the records are deleted). Therefore, Flight is a good candidate for replication.

Note that the Customer table is also largely read-only. However, because of the volume of data in the Customer table (a million records), it is not a good candidate for replication, which is why it is partitioned.

4.5.2. Specifying Replicated Tables

In VoltDB, you do not explicitly state that a table is replicated. If you do not specify a partitioning column in the database schema, the table will by default be replicated.

So, in our flight reservation example, there is no explicit action required to replicate the Flight table. However, it is very important to specify partitioning information for tables that you want to partition. If not, they will be replicated by default, significantly changing the performance characteristics of your application.

4.6. Modifying the Schema

You can use DDL to add, modify, or remove schema objects as the database is running. For a list of all valid DDL you can use, see Appendix A, *Supported SQL DDL Statements*. You can do the following types of schema changes:

- Modifying Tables — You can add, modify (alter), and remove (drop) table columns. You can also add and drop table constraints. Finally, you can drop entire tables.
- Adding and Dropping Indexes — You can add and remove (drop) named indexes.
- Modifying Partitioning for Tables and Stored Procedures — You can un-partition stored procedures and re-partition stored procedures on a different column, For tables you can change a table between partitioned and replicated, and repartition a table on a different column,
- Modify roles and users — To learn about modifying roles and users, see Chapter 12, *Security*.

VoltDB safely handles sqlcmd DDL entered by different users on different nodes of the cluster because it manages sqlcmd commands as transactions, just like stored procedures. To demonstrate the DDL statements to modify the schema, the following sections use a new table, Airport, added to the fight reservation as shown below:

```
CREATE TABLE Airport (
   AirportID integer NOT NULL,
   Name varchar(15) NOT NULL,
   City varchar(25),
   Country varchar(15),
   PRIMARY KEY (AirportID)
);
```

4.6.1. Effects of Schema Changes on Data and Clients

You can make many schema changes on empty tables with few restrictions. However, be aware that if a table has data, some schema changes are not allowed and other schema changes may modify or even remove data. When working with test data in your database, you can use TRUNCATE TABLE to empty the data from a table you are working on. Note that all DDL examples in this chapter assume the tables are empty.

We can think of the effects of schema changes on data in three severity levels:

- Schema change completes without damage to data
- Schema change fails to complete to avoid damage to data
- Schema change destroys data

VoltDB error messages and the documentation can help you avoid schema change attempts that fail to complete. For example, you cannot drop a table that has referencing procedures or views.

Obviously you need to be most aware of which schema changes cause data to be destroyed. In particular, removing objects from the schema will also remove the data they contain. Note that schema objects cannot be renamed with DDL, but objects can be replaced by performing a DROP and then ADD. However, it is important to realize that as a result of a DROP operation, such as DROP TABLE, the data associated with that table will be deleted before the new definition is added.

Plan and coordinate changes with client development. Stored procedures and ad hoc queries provide an API that clients use to access the database correctly. Changes to the schema can break the stored procedure calls client applications have developed, so use well-planned schedules to communicate database schema changes to others. Client applications depend on many schema definition features including (but not limited to):

- Table names
- Column names
- Column data types
- Primary key definitions
- Table partitions
- Stored procedure names
- Stored procedure partitioning

Plan and test carefully before making schema changes to a production database. Be aware that clients may experience connection issues during schema changes, especially for changes that take longer to complete, such as view or index changes.

Schema changes not only affect data, but the existence of data in the database affects the time it takes to process schema changes. For example, when there are large amounts of data, some DDL statements can block processing, resulting in a noticeable delay for other pending transactions. Examples include adding indexes, creating new table columns, and modifying views.

4.6.2. Viewing the Schema

The VoltDB Management Center provides a web browser view of database information, including the DDL schema source. Use a web browser to view the VoltDB Management Center on port 8080 of one of the cluster hosts (http://host-name:8080).

You can also use the sqlcmd **show** directive to see a list of the current database tables and all procedures. For additional details about the schema, execute the @SystemCatalog system procedure. Use any of the following arguments to @SystemCatalog to obtain details about a component of the database schema:

- TABLES
- COLUMNS
- INDEXINFO
- PRIMARYKEYS
- PROCEDURES
- PROCEDURECOLUMNS

For example:

```
$ sqlcmd
1> SHOW TABLES;
2> SHOW PROCEDURES;
3> EXEC @SystemCatalog COLUMNS;
```

4.6.3. Modifying Tables

After creating a table in a database with CREATE TABLE, you can use ALTER TABLE to make the following types of table changes:

- Altering a Table Column's Data Definition
- Adding and Dropping Table Columns
- Adding and Dropping Table Constraints

To drop an entire table, use the DROP TABLE DDL statement.

4.6.3.1. Altering a Table Column's Data Definition

You can make the following types of alterations to a table column's data definition:

```
$ sqlcmd
```

```
1> ALTER TABLE Airport ALTER COLUMN Name VARCHAR(25);            ❶
2> ALTER TABLE Airport ALTER COLUMN Country SET DEFAULT 'USA';   ❷
3> ALTER TABLE Airport ALTER COLUMN Name SET NOT NULL;           ❸
```

The examples are described as follows:

❶ Change a column's data type. In our example we decided we needed more than 15 characters for the Airport Name so we changed it to 25 characters.

If the table has no existing data, you can make any data type changes. However, if the table already contains data, the new type must be larger than the old one. This restriction prevents corrupting existing data values that might be larger than the size of the new data type (See also Table A.1, “Supported SQL Datatypes”.)

❷ Set or drop the column's DEFAULT value. In our example we assume the application is to be used mostly for US domestic travel so we can set a default value for the Airport Country of 'USA'.

To remove a default, redefine the column data definition, for example:

```
ALTER TABLE Airport ALTER COLUMN Country VARCHAR(15);
```

❸ Change whether the column is NULL or NOT NULL. In our example we set the AirportID to be not null because this is a required field.

If the table has existing data, you cannot change a column to not null.

4.6.3.2. Adding and Dropping Table Columns

```
$ sqlcmd
1> ALTER TABLE Airport ADD COLUMN AirportCode VARCHAR(3)          ❶
2> BEFORE AirportID;
3> ALTER TABLE Airport DROP COLUMN AirportID;                     ❷
```

The examples are described as follows:

❶ Add table columns. In our example, we have decided not to use the integer AirportID for airport identification but to instead add an AirportCode, which uses a unique three-letter code for any airport as defined by the International Air Transport Association's airport codes.

You cannot rename or overwrite a column but you can drop and add columns. When adding a column, you must include the new column name and the data type. Options you may include are:

- DEFAULT value — If a table contains data, the values for the new column will be automatically filled in with the default value.
- NOT NULL — If the table contains data, you must include a default value if you specify a NOT NULL column.
- One of the following index type constraints including PRIMARY KEY, UNIQUE, or ASSUME-UNIQUE.

  Note, we recommend that you not define the UNIQUE or ASSUMEUNIQUE constraint directly on a column definition when adding a column or creating a table. If you do, the constraint has no name so you cannot drop the constraint without dropping the entire column. Instead, we recommend you apply UNIQUE or ASSUMEUNIQUE by adding the constraint (see Section 4.6.3.3, “Adding and Dropping Table Constraints”) or by adding an index with the constraint (see Section 4.6.4, “Adding and Dropping Indexes”). Defining these constraints this way names the constraint, which makes it easier to drop later if necessary.

- BEFORE column-name — Table columns cannot be reordered but the BEFORE clause allows you to place a new column in a specific position with respect to the existing columns of the table.

❷ Drop table columns. In our example we drop the AirportID column because we are replacing it with the AirportCode column.

You cannot remove a column that has a reference to it. You have to remove all references to the column first. References to a column may include:

- A stored procedure
- An index
- A view

4.6.3.3. Adding and Dropping Table Constraints

You cannot alter a table constraint but you can add and drop table constraints. If the table contains existing data, you cannot add UNIQUE, ASSUMEUNIQUE, or PRIMARY KEY constraints.

```
$ sqlcmd
1> ALTER TABLE Airport ADD CONSTRAINT                          ❶
2>    uniquecode UNIQUE (Airportcode);
3> ALTER TABLE Airport ADD PRIMARY KEY (AirportCode);          ❷
```

The examples are described as follows:

❶ Add named constraints UNIQUE or ASSUMEUNIQUE. In our example, we add the UNIQUE constraint to the AirportCode column. To drop a named constraint, include the name using the format in the following example:

```
ALTER TABLE Airport DROP CONSTRAINT uniquecode;
```

❷ Add unnamed constraints PRIMARY KEY or LIMIT PARTITION ROWS, each of which can apply to a table only once. In our example, we add the PRIMARY KEY constraint to the new AirportCode column.

When adding a table constraint, it must not conflict with the other columns of the table. For example, only one primary key is allowed for a table so you cannot add the PRIMARY KEY constraint to an additional column.

To drop the PRIMARY KEY or LIMIT PARTITION ROWS constraint, include the type of constraint using the format in the following example:

```
ALTER TABLE Airport DROP PRIMARY KEY;
```

4.6.4. Adding and Dropping Indexes

Use CREATE INDEX to create an index on one or more columns of a table. Use DROP INDEX to remove an index from the schema. The following example modifies the flight reservation schema by adding an index to the Flight table to improve performance when looking up flights.

```
$ sqlcmd
1> CREATE INDEX flightTimeIdx ON Flight (departtime);
```

The CREATE INDEX statement explicitly creates an index. VoltDB creates an index implicitly when you specify the table constraints UNIQUE, PRIMARY KEY, or ASSUMEUNIQUE. Use the ALTER

TABLE statement to add or drop these table constraints along with their associated indexes, as shown in Section 4.6.3, “Modifying Tables”.

4.6.5. Modifying Partitioning for Tables and Stored Procedures

Any changes to the schema must be carefully coordinated with the design and development of stored procedures. This not only applies to column names, data types, and so on, but also to the partition plan.

How to partition tables and stored procedures using the PARTITION TABLE and PARTITION PROCEDURE statements is explained in Section 4.4, “Partitioning Database Tables” and Section 5.3.3, “Partitioning Stored Procedures in the Schema”.

You can change the partitioning of stored procedures, and you can change a table to a replicated table or repartition it on a different column. However, because of the intricate dependencies of partitioned tables and stored procedures, this can only be done by dropping and re-adding the tables and procedures. Also, you must pay close attention to the order in which objects are dropped and added.

The following DDL examples demonstrate some partitioning modifications to a table and stored procedures.

- Un-partitioning a Stored Procedure
- Changing a Partitioned Table to a Replicated Table
- Re-partitioning a Table to a Different Column
- Updating a Stored Procedure
- Removing a Stored Procedure from the Database

The following DDL is added to the Flight reservation schema to help demonstrate the DDL partition changes described in this section.

```
$ sqlcmd
1> PARTITION TABLE Airport ON COLUMN Name;
2> CREATE PROCEDURE FindAirportCodeByName AS
3>    SELECT TOP 1 AirportCode FROM Airport WHERE Name=?;
4> PARTITION PROCEDURE FindAirportCodeByName
5>    ON TABLE Airport COLUMN Name;
6> CREATE PROCEDURE FindAirportCodeByCity AS
7>    SELECT TOP 1 AirportCode FROM Airport WHERE City=?;
```

The stored procedures are tested with the following sqlcmd directives:

```
$ sqlcmd
1> exec FindAirportCodeByName 'Logan Airport';
2> exec FindAirportCodeByCity 'Boston';
```

4.6.5.1. Un-partitioning a Stored Procedure

In the simplest case, you can un-partition a single-partitioned stored procedure by dropping and re-creating that procedure without including the PARTITION PROCEDURE statement. In this example we drop the single-partitioned FindAirportCodeByName procedure and re-create it as multi-partitioned because it needs to search all partitions to find an airport code by name.

```
$ sqlcmd
```

```
1> DROP PROCEDURE FindAirportCodeByName;
2> CREATE PROCEDURE FindAirportCodeByName AS
3>    SELECT TOP 1 AirportCode FROM Airport WHERE Name=?;
```

4.6.5.2. Changing a Partitioned Table to a Replicated Table

Important

You cannot change the partitioning of a table that has data in it. To change a partitioned table to a replicated one, you drop and re-create the table, which deletes any data that might be in the table.

Before executing the following steps, save the existing schema so you can easily re-create the table. The VoltDB Management Center provides a view of the existing database schema DDL source, which you can download and save.

```
$ sqlcmd
1> DROP PROCEDURE FindAirportCodeByName;                    ❶
2> DROP PROCEDURE FindAirportCodeByCity;
3> DROP TABLE Airport IF EXISTS CASCADE;                    ❷
4> CREATE TABLE AIRPORT (                                   ❸
5>    AIRPORTCODE varchar(3) NOT NULL,
6>    NAME varchar(25),
7>    CITY varchar(25),
8>    COUNTRY varchar(15) DEFAULT 'USA',
9>    CONSTRAINT UNIQUECODE UNIQUE (AIRPORTCODE),
10>    PRIMARY KEY (AIRPORTCODE)
11> );
12> CREATE PROCEDURE FindAirportCodeByName AS               ❹
13>    SELECT TOP 1 AirportCode FROM Airport WHERE Name=?;
14> CREATE PROCEDURE FindAirportCodeByCity AS
15>    SELECT TOP 1 AirportCode FROM Airport WHERE City=?;
```

The example is described as follows:

❶ Drop all stored procedures that reference the table. You cannot drop a table if stored procedures reference it.

❷ Drop the table. Options you may include are:

- IF EXISTS — Use the IF EXISTS option to avoid command errors if the named table is already removed.
- CASCADE — A table cannot be removed if it has index or view references. You can remove the references explicitly first or use the CASCADE option to have VoltDB remove the references along with the table.

❸ Re-create the table. By default, a newly created table is a replicated table.

❹ Re-create the stored procedures that access the table. If the stored procedure is implemented with Java and changes are required, modify and reload the code before re-creating the stored procedures. For more, see Section 5.3, “Installing Stored Procedures into the Database”.

4.6.5.3. Re-partitioning a Table to a Different Column

Important

You cannot change the partitioning of a table that has data in it. In order to re-partition a table you have to drop and re-create the table, which deletes any data that might be in the table.

Follow these steps to re-partition a table:

1. Un-partition the table by following the instructions in Section 4.6.5.2, “Changing a Partitioned Table to a Replicated Table”. The sub-steps are summarized as follows:

   a. Drop all stored procedures that reference the table.

   b. Drop the table.

   c. Re-create the table.

   d. Re-create the stored procedures that access the table.

2. Partition the table on the new column. In our example, it makes sense to partition the Airport table on the AirportCode column, where each row must be unique and non null.

   ```
   $ sqlcmd
   1> PARTITION TABLE Airport ON COLUMN AirportCode;
   ```

3. Re-partition stored procedures that should be single-partitioned. See Section 4.6.5.4, “Updating a Stored Procedure”.

4.6.5.4. Updating a Stored Procedure

This section describes how to update a stored procedure that has already been declared in the database with the CREATE PROCEDURE statement. The steps to update a stored procedure are summarized as follows:

1. If the procedure is implemented in Java, update the procedure's code, recompile, and repackage the jar file. For details, see Section 5.3, “Installing Stored Procedures into the Database”.

2. Ensure all tables and columns the procedure accesses are in the database schema.

3. Update the procedure in the database.

   - If the procedure is implemented in Java, use the sqlcmd **load classes** directive to update the class in the database. For example:

     ```
     $ sqlcmd
     1> load classes GetAirport.jar;
     ```

   - If the procedure is implemented with SQL, use the CREATE PROCEDURE AS command to update the SQL.

4. If required, partition the stored procedure. If the procedure is currently multi-partitioned, use the PARTITION PROCEDURE command to partition on the same column as the table being accessed. Note that if you previously re-partitioned a table, it required that you drop and then re-create the stored procedures as multi-partitioned.

   If the procedure is already single-partitioned but needs to be re-partitioned on a different column, do the following steps:

   a. Use DROP PROCEDURE to remove the stored procedure.

   b. Use CREATE PROCEDURE to re-declare the stored procedure.

   c. c. Use PARTITION PROCEDURE to partition on the new column.

In our example so far, we have three stored procedures that are adequate to access the Airport table, so no additional procedures need to be partitioned:

- VoltDB automatically defined a default select stored procedure, which is partitioned on the Airport-Code column. It takes an AirportCode as input and returns a table structure containing the Airport-Code, Name, City, and Country.
- The FindAirportCodeByName stored procedure should remain multi-partitioned because it needs to search in all partitions.
- The FindAirportCodeByCity stored procedure should also remain multi-partitioned because it needs to search in all partitions.

4.6.5.5. Removing a Stored Procedure from the Database

If you've decided a stored procedure is no longer needed, use the following steps to remove it from the database:

1. Drop the stored procedure from the database.

```
$ sqlcmd
1> DROP PROCEDURE GetAirport;
```

2. Remove the code from the database. If the procedure is implemented with Java, use the sqlcmd **remove classes** directive to remove the procedure's class from the database.

```
2> remove classes myapp.procedures.GetAirport;
```

Chapter 5. Designing Stored Procedures to Access the Database

As you can see from Chapter 4, *Designing the Database Schema*, defining the database schema and the partitioning plan go hand in hand with understanding how the data is accessed. The two must be coordinated to ensure optimum performance. Your stored procedures must use the same attribute for partitioning as the table being accessed. Proper partitioning ensures that the table rows the stored procedure requests are in the same partition in which the procedure executes, thereby ensuring maximum efficiency.

It doesn't matter whether you design the partitioning first or the data access first, as long as in the end they work together. However, for the sake of example, we will use the schema and partitioning outlined in Chapter 4, *Designing the Database Schema* when discussing how to design the data access.

5.1. How Stored Procedures Work

The key to designing the data access for VoltDB applications is that complex or performance sensitive access to the database should be done through stored procedures. It is possible to perform ad hoc queries on a VoltDB database. However, ad hoc queries do not benefit as fully from the performance optimizations VoltDB specializes in and therefore should not be used for frequent, repetitive, or complex transactions.

Within the stored procedure, you access the database using standard SQL syntax, with statements such as SELECT, UPDATE, INSERT, and DELETE. You can also include your own code within the stored procedure to perform calculations on the returned values, to evaluate and execute conditional statements, or to perform many other functions your applications may need.

5.1.1. VoltDB Stored Procedures are Transactional

In VoltDB, a stored procedure and a transaction are one and the same. Thus when you define a stored procedure, VoltDB automatically provides ACID transaction guarantees for the stored procedure. This means that stored procedures fully succeed or automatically roll back as a whole if an error occurs (atomic). When stored procedures change the data, the database is guaranteed to remain consistent. Stored procedures execute and access the database completely isolated from each other, including when they execute concurrently. Finally, stored procedure changes to the database are guaranteed to be saved and available for subsequent database access (durable).

Because the transaction is defined in advance as a stored procedure, there is no need for your application to manage transactions using specific transaction commands such as BEGIN, ROLLBACK, COMMIT or END.[1]

5.1.2. VoltDB Stored Procedures are Deterministic

To ensure data consistency and durability, VoltDB procedures must be deterministic. That is, given specific input values, the outcome of the procedure is consistent and predictable. Determinism is critical because it allows the same stored procedure to run in multiple locations and give the same results. It is determinism that makes it possible to run redundant copies of the database partitions without impacting performance. (See Chapter 10, *Availability* for more information on redundancy and availability.)

[1]One side effect of transactions being precompiled as stored procedures is that external transaction management frameworks, such as Spring or JEE, are not supported by VoltDB.

5.1.2.1. Use Sorted SQL Queries

One key to deterministic behavior is avoiding ambiguous SQL queries in stored procedures. Specifically, performing unsorted queries can result in a nondeterministic outcome. VoltDB does not guarantee a consistent order of results unless you use a tree index to scan the records in a specific order or you specify an ORDER BY clause in the query itself. In the worst case, a limiting query, such as `SELECT TOP 10 Emp_ID FROM Employees` without an index or ORDER BY clause, can result in a different set of rows being returned. However, even a simple query such as `SELECT * from Employees` can return the same rows in a different order.

The problem is that even if a non-deterministic query is read-only, its results might be used as input to an INSERT, UPDATE, or DELETE statement elsewhere in the stored procedure. For clusters with a K-safety value greater than zero, this means unsorted query results returned by two copies of the same partition, which may not match, could be used for separate update queries. If this happens, VoltDB detects the mismatch, reports it as potential data corruption, and shuts down the cluster to protect the database contents.

This is why VoltDB issues a warning for any non-deterministic queries in read-write stored procedures. This is also why use of an ORDER BY clause or a tree index in the WHERE constraint is strongly recommended for all SELECT statements that return multiple rows.

5.1.2.2. Avoid Introducing Non-deterministic Values from External Functions

Another key to deterministic behavior is avoiding calls within your stored procedures to external functions or procedures that can introduce arbitrary data. External functions include file and network I/O (which should be avoided any way because they can impact latency), as well as many common system-specific procedures such as Date and Time.

However, this limitation does not mean you cannot use arbitrary data in VoltDB stored procedures. It just means you must either generate the arbitrary data before the stored procedure call and pass it in as input parameters or generate it in a deterministic way. For example, if you need to load a set of records from a file, you can open the file in your application and pass each row of data to a stored procedure that loads the data into the VoltDB database. This is the best method when retrieving arbitrary data from sources (such as files or network resources) that would impact latency.

The other alternative is to use data that can be generated deterministically. For two of the most common cases, timestamps and random values, VoltDB provides methods for this:

- `VoltProcedure.getTransactionTime()` returns a timestamp that can be used in place of the Java Date or Time classes.
- `VoltProcedure.getSeededRandomNumberGenerator()` returns a pseudo random number that can be used in place of the Java Util.Random class.

These procedures use the current transaction ID to generate a deterministic value for the timestamp and the random number. See the VoltDB Java Stored Procedure API for more.

5.1.2.3. Stored Procedures have no Persistence

Finally, even seemingly harmless programming techniques, such as static variables can introduce nondeterministic behavior. VoltDB provides no guarantees concerning the state of the stored procedure class instance across invocations. Any information that you want to persist across invocations must either be stored in the database itself or passed into the stored procedure as a parameter.

5.2. The Anatomy of a VoltDB Stored Procedure

You can write VoltDB stored procedures as Java classes. The following code sample illustrates the basic structure of a VoltDB java stored procedure.

```
import org.voltdb.*;

public class Procedure-name extends VoltProcedure {

                 // Declare SQL statements ...

    public datatype run ( arguments ) throws VoltAbortException {

                // Body of the Stored Procedure ...

    }
}
```

The key points to remember are to:

1. Import the VoltDB classes from `org.voltdb.*`
2. Include the class definition, which extends the abstract class `VoltProcedure`
3. Define the method `run()`, which performs the SQL queries and processing that make up the transaction

It is important to understand the details of how to design and develop stored procedures for your application as described in the following sections. However, for simple data access, the following techniques may suffice for some of your stored procedures:

- VoltDB defines default stored procedures to perform the most common table access such as inserting, selecting, updating, and deleting records based on a specific key value. See Section 7.1, “Using Default Procedures” for more.
- You can create stored procedures without writing any Java code by using the DDL statement CREATE PROCEDURE AS, where you define a single SQL query as a stored procedure. See Section 7.2, “Shortcut for Defining Simple Stored Procedures”.

The following sections describe the components of a stored procedure in more detail.

5.2.1. The Structure of the Stored Procedure

The stored procedures themselves are written as Java classes, each procedure being a separate class. Example 5.1, “Components of a VoltDB Java Stored Procedure” shows the stored procedure that looks up a flight to see if there are any available seats. The callouts identify the key components of a VoltDB stored procedure.

Example 5.1. Components of a VoltDB Java Stored Procedure

```
package fadvisor.procedures;

import org.voltdb.*;                                                    ❶

public class HowManySeats extends VoltProcedure {                      ❷

   public final SQLStmt GetSeatCount = new SQLStmt(                     ❸
        "SELECT NumberOfSeats, COUNT(ReserveID) " +
        "FROM Flight AS F, Reservation AS R " +
        "WHERE F.FlightID=R.FlightID AND R.FlightID=? " +
        "GROUP BY NumberOfSeats;");

    public long run( int flightid)                                      ❹
        throws VoltAbortException {

        long numofseats;
        long seatsinuse;
        VoltTable[] queryresults;

        voltQueueSQL( GetSeatCount, flightid);                          ❺

        queryresults = voltExecuteSQL();                                ❻

        VoltTable result = queryresults[0];                             ❼
        if (result.getRowCount() < 1) { return -1; }
        numofseats = result.fetchRow(0).getLong(0);                     ❽
        seatsinuse = result.fetchRow(0).getLong(1);
        numofseats = numofseats - seatsinuse;
        return numofseats; // Return available seats                    ❾
    }
}
```

❶ Stored procedures are written as Java classes. To access the VoltDB classes and methods, be sure to `import org.voltdb.*`.

Although VoltDB stored procedures must be written in Java and the primary client interface is Java (as described in Chapter 6, *Designing VoltDB Client Applications*), it is possible to write client applications using other programming languages. See Chapter 8, *Using VoltDB with Other Programming Languages* for more information on alternate client interfaces.

❷ Each stored procedure extends the generic class `VoltProcedure`.

❸ Within the stored procedure you access the database using ANSI-standard SQL statements. To do this, you declare the statement as a special Java type called `SQLStmt`.

In the SQL statement, you insert a question mark (?) everywhere you want to replace a value by a variable at runtime. In this example, the query GetSeatCount has one input variable, FlightID. (See Appendix B, *Supported SQL Statements* for details on the supported SQL statements.)

To ensure the stored procedure code is single partitioned, queries must filter on the partitioning column for a single value (using equal, =). Filtering for a range of values will not be single-partitioned because the code will have to look up in all the partitions to ensure the entire range is found.

❹ The bulk of the stored procedure is the `run()` method, whose input specifies the input arguments for the stored procedure. See Section 5.2.2, “Passing Arguments to a Stored Procedure” next for details.

Note that the `run()` method throws the exception `VoltAbortException` if any exceptions are not caught. `VoltAbortException` causes the stored procedure transaction to rollback. (See Section 5.2.6, “Rolling Back a Transaction” for more information about rollback.)

❺ To perform database queries, you queue SQL statements, specifying both the SQL statement and the variables it requires, using the `voltQueueSQL()` method. More details are described in Section 5.2.3, “Creating and Executing SQL Queries in Stored Procedures”.

❻ After you queue all of the SQL statements you want to perform, use `voltExecuteSQL()` to execute the statements in the queue.

❼ Each statement returns its results in a `VoltTable` structure. Because the queue can contain multiple queries, `voltExecuteSQL()` returns an array of `VoltTable` structures, one array element for each query. More details are described in Section 5.2.4, “Interpreting the Results of SQL Queries”.

❽ In addition to queueing and executing queries, stored procedures can contain custom code. However, you should limit the amount of custom code in stored procedures to only that processing that is necessary to complete the transaction, so as not to delay subsequent transactions.

❾ Stored procedures can return a long integer, a `VoltTable` structure, or an array of `VoltTable` structures. For more details, see Section 5.2.5, “Returning Results from a Stored Procedure”.

5.2.2. Passing Arguments to a Stored Procedure

You specify the number and type of the arguments that the stored procedure accepts in the `run()` method. For example, the following is the declaration of the `run()` method for an `Initialize()` stored procedure from the voter sample application. This procedure accepts two arguments: an integer and a string.

```
public long run(int maxContestants, String contestants) { . . .
```

VoltDB stored procedures can accept parameters of any of the following Java and VoltDB datatypes:

Integer types	byte, short, int, long, Byte, Short, Integer, and Long
Floating point types	float, double, Float, Double
Fixed decimal types	BigDecimal
String and binary types	String and byte[]
Timestamp types	org.voltdb.types.TimestampType java.util.Date, java.sql.Date, java.sql.Timestamp
VoltDB type	VoltTable

The arguments can be scalar objects or arrays of any of the preceding types. For example, the following `run()` method defines three arguments: a scalar long and two arrays, one array of timestamps and one array of Strings:

```
import org.voltdb.*;
public class LogMessagesByEvent extends VoltProcedure {

     public long run (
           long eventType,
          org.voltdb.types.TimestampType[] eventTimeStamps,
          String[] eventMessages
    ) throws VoltAbortException {
```

The calling client application can use any of the preceding datatypes when invoking the `callProcedure()` method and, where necessary, VoltDB makes the appropriate type conversions (for example,

from int to String or from String to Double). See Section 6.2, “Invoking Stored Procedures” for more on using the `callProcedure()` method.

5.2.3. Creating and Executing SQL Queries in Stored Procedures

The main function of the stored procedure is to perform database queries. In VoltDB this is done in two steps:

1. Queue the queries using the `voltQueueSQL()` function

2. Execute the queue and return the results using the `voltExecuteSQL()` function

Queuing SQL Statements The first argument to `voltQueueSQL()` is the SQL statement to be executed. The SQL statement is declared using a special class, `SQLStmt`, with question marks as placeholders for values that will be inserted at runtime. The remaining arguments to `voltQueueSQL()` are the actual values that VoltDB inserts into the placeholders. For example, if you want to perform a SELECT of a table using two columns in the WHERE clause, your SQL statement might look something like this:

```
SELECT CustomerID FROM Customer WHERE FirstName=? AND LastName=?;
```

At runtime, you want the questions marks replaced by values passed in as arguments from the calling application. So the actual `voltQueueSQL()` invocation might look like this:

```
public final SQLStmt getcustid = new SQLStmt(
                                  "SELECT CustomerID FROM Customer " +
                                  "WHERE FirstName=? AND LastName=?;");

     ...

voltQueueSQL(getcustid, firstnm, lastnm);
```

Your stored procedure can call `voltQueueSQL()` more than once to queue up multiple SQL statements before they are executed. Queuing multiple SQL statements improves performance when the SQL queries execute because it minimizes the amount of network traffic within the cluster. Once you have queued all of the SQL statements you want to execute together, you then process the queue using the `voltExecuteSQL()` function.

```
VoltTable[] queryresults = voltExecuteSQL();
```

Cycles of Queue and Execute

Your procedure can queue and execute SQL statements in as many cycles as necessary to complete the transaction. For example, if you want to make a flight reservation, you may need to access the database and verify that the flight exists before creating the reservation in the database. One way to do this is to look up the flight, verify that a valid row was returned, then insert the reservation, like so:

Example 5.2. Cycles of Queue and Execute in a Stored Procedure

```
final String getflight = "SELECT FlightID FROM Flight WHERE FlightID=?;";     ❶
final String makeres = "INSERT INTO Reservation (?,?,?,?,?);";

public final SQLStmt getflightsql = new SQLStmt(getflight);
public final SQLStmt makeressql = new SQLStmt(makeres);

public VoltTable[] run( int reservenum, int flightnum, int customernum )      ❷
        throws VoltAbortException {

        // Verify flight ID
  voltQueueSQL(getflightsql, flightnum);                                      ❸
  VoltTable[] queryresults = voltExecuteSQL();

        // If there is no matching record, rollback
  if (queryresults[0].getRowCount() == 0 ) throw new VoltAbortException();    ❹

        // Make reservation
  voltQueueSQL(makeressql, reservenum, flightnum, customernum,0,0);           ❺
  return voltExecuteSQL();
}
```

This stored procedure code to make a reservation is described as follows:

❶ Define the SQL statements to use. The *getflight* string contains an SQL statement that verifies the flight ID, and the *makeres* string contains the SQL statement that makes the reservation.

❷ Define the `run()` method for the stored procedure. This stored procedure takes as input arguments the reservation number, the flight number, and the customer number.

❸ Queue and execute an SQL statement. In this example the `voltExecuteSQL()` method processes the single `getflightsql()` function, which executes the SQL statement specified in the *getflight* string.

❹ Process results. If the flight is not available, the exception `VoltAbortException` aborts the stored procedure and rolls back the transaction.

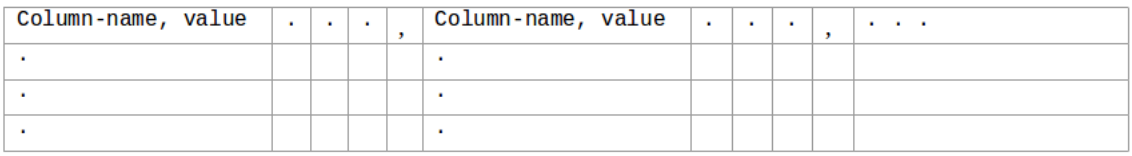
❺ The second SQL statement to make the reservation is then queued and executed. The `voltExecuteSQL()` method processes the single `makeressql()` function, which executes the SQL statement specified in the *makeres* string.

5.2.4. Interpreting the Results of SQL Queries

With the `voltExecuteSQL()` call, the results of all the queued SQL statements are returned in an array of `VoltTable` structures. The array contains one `VoltTable` for each SQL statement in the queue. The `VoltTable` structures are returned in the same order as the respective SQL statements in the queue.

The `VoltTable` itself consists of rows, where each row contains columns, and each column has the column name and a value of a fixed datatype. The number of rows and columns per row depends on the specific query.

Figure 5.1. Array of VoltTable Structures



For example, if you queue two SQL SELECT statements, one looking for the destination of a specific flight and the second looking up the ReserveID and Customer name (first and last) of reservations for that flight, the code for the stored procedure might look like the following:

```
public final SQLStmt getdestsql = new SQLStmt(
              "SELECT Destination FROM Flight WHERE FlightID=?;");
public final SQLStmt getressql = new SQLStmt(
             "SELECT r.ReserveID, c.FirstName, c.LastName " +
             "FROM Reservation AS r, Customer AS c " +
             "WHERE r.FlightID=? AND r.CustomerID=c.CustomerID;");

         ...

   voltQueueSQL(getdestsql,flightnum);
   voltQueueSQL(getressql,flightnum);
   VoltTable[] results = voltExecuteSQL();
```

The array returned by `voltExecuteSQL()` will have two elements:

- The first array element is a `VoltTable` with one row (FlightID is defined as unique) containing one column, because the SELECT statement returns only one value.
- The second array element is a `VoltTable` with as many rows as there are reservations for the specific flight, each row containing three columns: ReserveID, FirstName, and LastName.

Assuming the stored procedure call input was a FlightID value of 134, the data returned for the second array element might be represented as follows:

Figure 5.2. One VoltTable Structure is returned for each Queued SQL Statement

FlightID, 134	,	ReserveID, 4747	FirstName, Will	LastName, Poger
		ReserveID, 9879	FirstName, Janice	LastName, Josly
		ReserveID, 3456	FirstName, Holly	LastName, Eagan
		ReserveID, 1098	FirstName, Ralph	LastName, Finess

VoltDB provides a set of convenience methods for accessing the contents of the `VoltTable` array. Table 5.1, “Methods of the VoltTable Classes” lists some of the most common methods. (See also Java Stored Procedure API.)

Table 5.1. Methods of the VoltTable Classes

Method	Description
int fetchRow(int index)	Returns an instance of the VoltTableRow class for the row specified by index.
int getRowCount()	Returns the number of rows in the table.
int getColumnCount()	Returns the number of columns for each row in the table.
Type getColumnType(int index)	Returns the datatype of the column at the specified index. Type is an enumerated type with the following possible values: BIGINT DECIMAL FLOAT GEOGRAPHY GEOGRAPHY_POINT INTEGER INVALID NULL NUMERIC SMALLINT STRING TIMESTAMP TINYINT VARBINARY VOLTTABLE
String getColumnName(int index)	Returns the name of the column at the specified index.
double getDouble(int index) long getLong(int index) String getString(int index) BigDecimal getDecimalAsBigDecimal(int index) double getDecimalAsDouble(int index) Date getTimestampAsTimestamp(int index) long getTimestampAsLong(int index) byte[] getVarbinary(int index)	Methods of VoltTable.Row Return the value of the column at the specified index in the appropriate datatype. Because the datatype of the columns vary depending on the SQL query, there is no generic method for returning the value. You must specify what datatype to use when fetching the value.

It is also possible to retrieve the column values by name. You can invoke any of the getDatatype() methods and pass a string argument specifying the name of the column, rather than the numeric index. Accessing the columns by name can make code easier to read and less susceptible to errors due to changes in the SQL schema (such as changing the order of the columns). On the other hand, accessing column values by numeric index is potentially more efficient under heavy load conditions.

Example 5.3, “Displaying the Contents of VoltTable Arrays” shows a generic routine for “walking” through the return results of a stored procedure. In this example, the contents of the `VoltTable` array are written to standard output.

Example 5.3. Displaying the Contents of VoltTable Arrays

```
public void displayResults(VoltTable[] results) {
  int table = 1;
     for (VoltTable result : results) {
      System.out.printf("*** Table %d ***\n",table++);
      displayTable(result);
   }
}

public void displayTable(VoltTable t) {

   final int colCount = t.getColumnCount();
   int rowCount = 1;
   t.resetRowPosition();
   while (t.advanceRow()) {
      System.out.printf("--- Row %d ---\n",rowCount++);

      for (int col=0; col<colCount; col++) {
         System.out.printf("%s: ",t.getColumnName(col));
         switch(t.getColumnType(col)) {
            case TINYINT: case SMALLINT: case BIGINT: case INTEGER:
               System.out.printf("%d\n", t.getLong(col));
               break;
            case STRING:
               System.out.printf("%s\n", t.getString(col));
               break;
            case DECIMAL:
               System.out.printf("%f\n", t.getDecimalAsBigDecimal(col));
               break;
            case FLOAT:
               System.out.printf("%f\n", t.getDouble(col));
               break;
         }
      }
   }
}
```

For further details on interpreting the VoltTable structure, see the Java documentation that is provided online in the `doc/` subfolder for your VoltDB installation.

5.2.5. Returning Results from a Stored Procedure

Stored procedures can return the following types:

- Long integer
- Single VoltTable
- Array of VoltTable structures

You can return all of the query results by returning the `VoltTable` array, or you can return a scalar value that is the logical result of the transaction. (For example, the stored procedure in Example 5.1, “Components of a VoltDB Java Stored Procedure” returns a long integer representing the number of remaining seats available in the flight.)

Whatever value the stored procedure returns, make sure the `run()` method includes the appropriate datatype in its definition. For example, the following two definitions specify different return datatypes; the first returns a long integer and the second returns the results of a SQL query as a `VoltTable` array.

```
public long run( int flightid)

public VoltTable[] run ( String lastname, String firstname)
```

Note that you can interpret the results of SQL queries either in the stored procedure or in the client application. However, for performance reasons, it is best to limit the amount of additional processing done by the stored procedure to ensure it executes quickly and frees the queue for the next stored procedure. So unless the processing is necessary for subsequent SQL queries, it is usually best to return the query results (in other words, the `VoltTable` array) directly to the calling application and interpret them there.

5.2.6. Rolling Back a Transaction

Finally, if a problem arises while a stored procedure is executing, whether the problem is anticipated or unexpected, it is important that the transaction rolls back. *Rollback* means that any changes made during the transaction are undone and the database is left in the same state it was in before the transaction started.

VoltDB is a fully transactional database, which means that if a transaction (stored procedure) fails, the transaction is automatically rolled back and the appropriate exception is returned to the calling application. Exceptions that can cause a rollback include the following:

- Runtime errors in the stored procedure code, such as division by zero or datatype overflow.
- Violating database constraints in SQL queries, such as inserting a duplicate value into a column defined as unique.

The atomicity of the stored procedure depends on VoltDB being able to roll back incomplete database changes. VoltDB relies on Java exception handling outside the stored procedure to perform the roll back. Therefore, you should not attempt to alter any exceptions thrown by the voltExecuteSql method. If your procedure code *does* catch exceptions thrown as a result of executing SQL statements, make sure that the exception handler re-throws the exception to allow VoltDB to perform the necessary roll back activities before the stored procedure returns to the calling program.

On the other hand, there may be situations where an exception occurs in the program logic. The issue might not be one that is caught by Java or VoltDB, but still there is no practical way for the transaction logic to complete. In these situations, you can force a rollback by explicitly throwing the `VoltAbortException` exception. For example, if a flight ID does not exist, you do not want to create a reservation so the stored procedure can force a rollback like so:

```
if (!flightid) { throw new VoltAbortException(); }
```

See Section 7.3, “Verifying Expected Query Results” for another way to roll back procedures when queries do not meet necessary conditions.

5.3. Installing Stored Procedures into the Database

When your stored procedure code is ready, you need to get the procedures into the database and ready to use. You first compile the procedure code, create a jar file, and load the resulting jar file into the database. Then you need to declare in the schema which procedures are stored procedures. Finally, depending on which table each stored procedure accesses, you need to partition each procedure to match the table partitioning. These processes are covered in the following sections:

- Compiling, Packaging, and Loading Stored Procedures
- Declaring Stored Procedures in the Schema
- Partitioning Stored Procedures in the Schema

These sections show how to use DDL to declare and partition stored procedures in the database schema. If you find you need to modify the schema, see Section 4.6, “Modifying the Schema”.

5.3.1. Compiling, Packaging, and Loading Stored Procedures

The VoltDB stored procedures are written as Java classes, so you compile them using the Java compiler. Anytime you update your stored procedure code, remember to recompile, package, and reload it into the database using the following steps:

```
$ javac -classpath "./:/opt/voltdb/voltdb/*" \          ❶
        -d ./obj  \
         *.java
$ jar  cvf  myproc.jar -C obj .                         ❷
$ sqlcmd                                                ❸
1> load classes myproc.jar;
2> show classes;
```

The steps are described as follows:

❶ Use the **javac** command to compile the procedure Java code.

You include libraries by using the `-classpath` argument on the command line or by defining the environment variable CLASSPATH. You must include the VoltDB libraries in the classpath so Java can resolve references to the VoltDB classes and methods. This example assumes that the VoltDB software has been installed in the folder `/opt/voltdb`. If you installed VoltDB in a different directory, you need to include your installation path. Also, if your client application depends on other libraries, they need to be included in the classpath as well.

Use the `-d` flag to specify an output directory in which to create the resulting class files.

❷ Use the **jar** command to package your Java classes into a Java archive, or JAR file.

The JAR file must have the same Java package structure as the classes in the JAR file. For example, if a class has a structure such as `myapp.procedures.ProcedureFoo`, then the JAR file has to have `myapp/procedures/ProcedureFoo.class` as the class structure for this file.

The JAR file must include any inner classes or other dependent classes used by the stored procedures.

❸ Use the sqlcmd **load classes** directive to load the stored procedure classes into the database.

You can use the **show classes** command to display information about the classes installed in the cluster.

Before a stored procedure can be called by a client application, you need to declare it in the schema, which is described next.

5.3.2. Declaring Stored Procedures in the Schema

To make your stored procedures accessible in the database, you must declare them in the schema using the CREATE PROCEDURE statement. Be sure to identify all of your stored procedures or they will not be available to the client applications at runtime. Also, before you declare a procedure, ensure the tables and columns the procedure accesses are in the schema.

The following DDL statements declare five stored procedures, identifying them by their class name:

```
$ sqlcmd
1> CREATE PROCEDURE FROM CLASS fadvisor.procedures.LookupFlight;
2> CREATE PROCEDURE FROM CLASS fadvisor.procedures.HowManySeats;
3> CREATE PROCEDURE FROM CLASS fadvisor.procedures.MakeReservation;
4> CREATE PROCEDURE FROM CLASS fadvisor.procedures.CancelReservation;
5> CREATE PROCEDURE FROM CLASS fadvisor.procedures.RemoveFlight;
```

For some situations, you can create stored procedures directly in the schema using SQL instead of loading Java code. See how to use the CREATE PROCEDURE AS statement in Section 7.2, “Shortcut for Defining Simple Stored Procedures”.

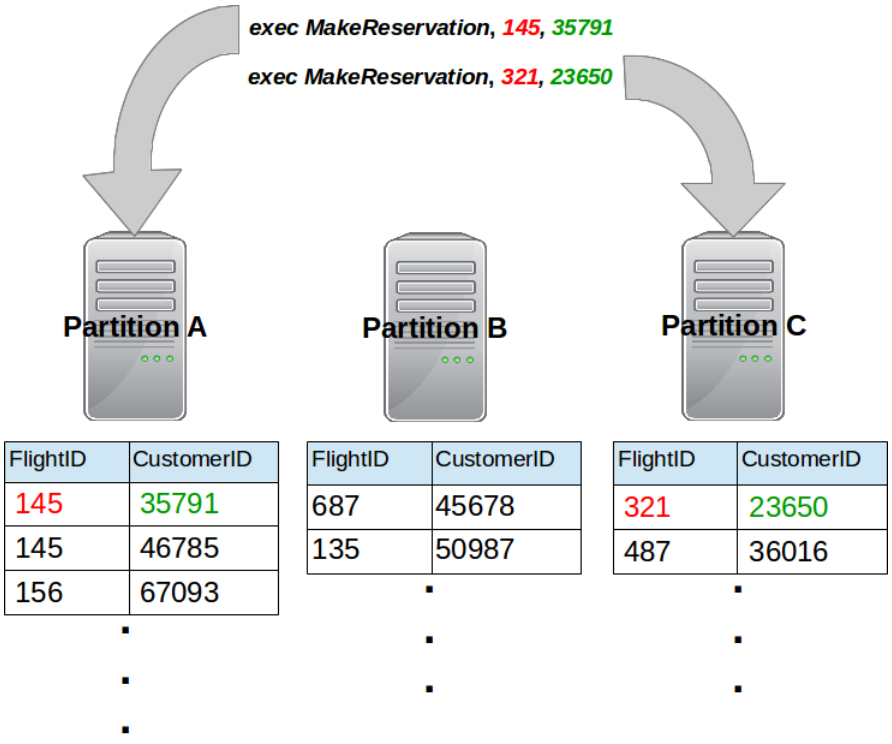
For more about modifying a schema with DDL, see Section 4.6, “Modifying the Schema”.

5.3.3. Partitioning Stored Procedures in the Schema

We want the most frequently used stored procedures to be single-partitioned. This means that the procedure executes in the one partition that also has the data it needs. Single-partitioned stored procedures do not have the overhead of processing across multiple partitions and servers, wasting time searching through the data of the entire table. To ensure single-partitioned efficiency, the parameter the stored procedure uses to identify its required data must be the same as the column on which the table rows are partitioned.

Remember that in our sample application the RESERVATION table is partitioned on FLIGHTID. Let's say you create a stored procedure, *MakeReservation()*, with two arguments, *flight_id* and *customer_id*. The following figure shows how the stored procedure will automatically execute in the partition that has the requested row.

Figure 5.3. Stored Procedures Execute in the Appropriate Partition Based on the Partitioned Parameter Value



If you do not declare a procedure as single-partitioned, it is assumed to be multi-partitioned by default. The advantage of multi-partitioned stored procedures is that they have full access to all of the data in the database, across all partitions. However, the real focus of VoltDB, and the way to achieve maximum throughput for your application, is through the use of single-partitioned stored procedures.

5.3.3.1. How to Declare Single-Partition Procedures

Before declaring a single-partitioned procedure, ensure the following prerequisites:

1. The table that the stored procedure accesses has been partitioned in the schema. See Section 4.4, “Partitioning Database Tables”.

2. If the procedure is implemented with Java code, it is loaded into the database. See Section 5.3.1, “Compiling, Packaging, and Loading Stored Procedures”.

3. The stored procedure has been declared in the schema with either of the CREATE PROCEDURE statements. See Section 5.3.2, “Declaring Stored Procedures in the Schema”.

When you declare a stored procedure as single-partitioned, you must specify both the associated table and the column on which it is partitioned using the PARTITION PROCEDURE statement in the schema. The following example uses the RESERVATION table and the FLIGHTID column as the partitioning column. For example:

```
PARTITION PROCEDURE MakeReservation ON TABLE Reservation COLUMN FlightID;
```

The PARTITION PROCEDURE statement assumes that the partitioning column value is also the first parameter to the stored procedure. Suppose you wish to partition a stored procedure on the third parameter such as the procedure `GetCustomerDetails()`, where the third parameter is a customer_id. You must specify the partitioning parameter using the PARAMETER clause and an index for the parameter position. The index is zero-based so the third parameter would be "2" and the PARTITION PROCEDURE statement would be as follows:

```
PARTITION PROCEDURE GetCustomerDetails
    ON TABLE Customer COLUMN CustomerID
    PARAMETER 2;
```

5.3.3.2. Queries in Single-Partitioned Stored Procedures

Single-partitioned stored procedures are special because they operate independently of other partitions, which is why they are so fast. At the same time, single-partitioned stored procedures operate on only a subset of the entire data, that is, only the data within the specified partition.

Caution

It is the application developer's responsibility to ensure that the queries in a single-partitioned stored procedure are truly single-partitioned. VoltDB *does not* warn you about SELECT or DELETE statements that might return incomplete results. For example, if your single-partitioned procedure attempts to operate on a range of values for the partitioning column, the range is incomplete and includes only a subset of the table data that is in the current partition.

VoltDB does generate a runtime error if you attempt to INSERT a row that does not belong in the current partition.

After you partition a procedure, your stored procedure can operate on only those records in the partitioned table that are identified by the partitioning column, in this example the RESERVATION table identified

by a FLIGHTID. Your stored procedure can operate on records in replicated tables because all partitions have the same copy of a replicated table. However, for other partitioned tables, the stored procedure can only operate on those records *if both tables are partitioned on the same attribute*. In this example that would be FLIGHTID.

In other words, the following rules apply:

- Any SELECT, UPDATE, or DELETE queries must use the constraint, `WHERE identifier=?`

  The question mark is replaced at runtime by the input value that identifies the row of data in the table. In our example, queries on the RESERVATION table must use the constraint, `WHERE FLIGHTID=?`

- SELECT statements can join the partitioned table to replicated tables, as long as the preceding WHERE constraint is also applied.

- SELECT statements can join the partitioned table to other partitioned tables as long as the following are true:

  - The two tables are partitioned on the same attribute or column (in our example, FLIGHTID).

  - The tables are joined on the shared partitioning column.

  - The following WHERE constraint is also used: `WHERE partitioned-table. identifier=?` In this example, `WHERE RESERVATION.FLIGHTID=?`

For example, the RESERVATION table can be joined with the FLIGHT table (which is replicated). However, the RESERVATION table *cannot* be joined with the CUSTOMER table in a single-partitioned stored procedure because the two tables use different partitioning columns. (CUSTOMER is partitioned on the CUSTOMERID column.)

The following are examples of invalid SQL queries for a single-partitioned stored procedure partitioned on FLIGHTID:

- **INVALID:** `SELECT * FROM reservation WHERE reservationid=?`

  The RESERVATION table is being constrained by a column (RESERVATIONID) which is not the partitioning column.

- **INVALID:** `SELECT c.lastname FROM reservation AS r, customer AS c WHERE r.flightid=? AND c.customerid = r.customerid`

  The correct partitioning column is being used in the WHERE clause, but the tables are being joined on a different column. As a result, not all CUSTOMER rows are available to the stored procedure since the CUSTOMER table is partitioned on a different column than RESERVATION.

Chapter 6. Designing VoltDB Client Applications

After you design and partition your database schema (Chapter 4, *Designing the Database Schema*), and after you design the necessary stored procedures (Chapter 5, *Designing Stored Procedures to Access the Database*), you are ready to write the client application logic. The client code contains all the business-specific logic required for the application, including business rule validation and keeping track of constraints such as proper data ranges for arguments entered in stored procedure calls.

The three steps to using VoltDB from a client application are:

1. Creating a connection to the database

2. Calling stored procedures

3. Closing the client connection

The following sections explain how to perform these functions using the standard VoltDB Java client interface. (See VoltDB Java Client API.) The VoltDB Java Client is a thread-safe class library that provides runtime access to VoltDB databases and functions.

It is possible to call VoltDB stored procedures from programming languages other than Java. However, reading this chapter is still recommended to understand the process for invoking and interpreting the results of a VoltDB stored procedure. See Chapter 8, *Using VoltDB with Other Programming Languages* for more information about using VoltDB from client applications written in other languages.

6.1. 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 6.5, “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 deployment file, the call to `createConnection()` will throw an exception. See Chapter 12, *Security* for more information about the use of security with VoltDB databases.

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 6.4, “Closing the Connection”.

6.1.1. 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. See Chapter 10, *Availability* for more.

6.1.2. Using an Auto-Reconnecting Client

If the client application loses contact with a server (either because the server goes down or a temporary network glitch), the connection to that server is closed. Assuming the application has connections to multiple servers in the cluster, it can continue to submit stored procedures through the remaining connections. However, the lost connection is not, by default, restored.

The application can use error handling to detect and recover from broken connections, as described in Section 6.5.2, “Handling Timeouts”. Or you can enable auto-reconnecting when you initialize the client object. You set auto-reconnecting in the client configuration before creating the client object, as in the following example:

```
org.voltdb.client.Client client = null;
ClientConfig config = new ClientConfig("","");
config.setReconnectOnConnectionLoss(true);
 try {
       client = ClientFactory.createClient(config);
       client.createConnection("server1.xyz.net");
       client.createConnection("server2.xyz.net");
       client.createConnection("server3.xyz.net");
            . . .
```

When `setReconnectOnConnectionLoss()` is set to true, the client library will attempt to reestablish lost connections, attempts starting every second and backing off to every eight seconds. As soon as the connection is reestablished, the reconnected server will begin to receive its share of the procedure calls.

6.2. 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. See Section 5.2.4, “Interpreting the Results of SQL Queries” for more information about interpreting the results of VoltDB stored procedures.

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.

6.3. 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 6.5.3, "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.

6.4. 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.

6.5. 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

6.5.1. 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 6.5.2, "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...");
                . . .
```

6.5.2. 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 CONNECTION_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 6.5.3, "Writing a Status Listener to Interpret Other Errors" for an example of creating a status listener for delayed procedure responses.

6.5.3. 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 `ClientStatusListenerExt`, 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.

6.6. Compiling and Running Client Applications

VoltDB client applications written in Java compile and run like other Java applications. (See Chapter 8, *Using VoltDB with Other Programming Languages* for more on writing client applications using other languages.) To compile, you must include the VoltDB libraries in the classpath so Java can resolve references to the VoltDB classes and methods. It is possible to do this manually by defining the environment variable `CLASSPATH` or by using the `-classpath` argument on the command line. If your client application depends on other libraries, they need to be included in the classpath as well. You can also specify where to create the resulting class files using the `-d` flag to specify an output directory, as in the following example:

```
$ javac -classpath "./:/opt/voltdb/voltdb/*" \
        -d ./obj  \
         *.java
```

The preceding example assumes that the VoltDB software has been installed in the folder `/opt/voltdb`. If you installed VoltDB in a different directory, you need to include your installation path in the `-classpath` argument.

If you are using Apache Maven to manage your application development, the VoltDB Java client library is available from the central Maven repository. So rather than installing VoltDB locally, you can simply include it as a dependency in your Maven project object model, or pom.xml, like so:

```
<dependency>
    <groupId>org.voltdb</groupId>
    <artifactId>voltdbclient</artifactId>
    <version>5.1</version>
</dependency>
```

6.6.1. Starting the Client Application

Before you start your client application, the VoltDB database must be running. When you start your client application, you must ensure that the VoltDB library JAR file is in the classpath. For example:

```
$ java -classpath "./:/opt/voltdb/voltdb/*"  MyClientApp
```

If you develop your application using one of the sample applications as a template, the `run.sh` file manages this dependency for you.

6.6.2. Running Clients from Outside the Cluster

If you are running the database on a cluster and the client applications on separate machines, you do not need to include all of the VoltDB software with your client application. The VoltDB distribution comes with two separate libraries: `voltdb-`*`n.n.nn`*`.jar` and `voltdbclient-`*`n.n.nn`*`.jar` (where *n.n.nn* is the VoltDB version number). The first file is a complete library that is required for building and running a VoltDB database server.

The second file, `voltdbclient-`*`n.n.nn`*`.jar`, is a smaller library containing only those components needed to run a client application. If you are distributing your client applications, you only need to distribute the client classes and the VoltDB client library. You do not need to install all of the VoltDB software distribution on the client nodes.

Chapter 7. Simplifying Application Development

The previous chapter (Chapter 6, *Designing VoltDB Client Applications*) explains how to develop your VoltDB database application using the full power and flexibility of the Java client interface. However, some database tasks — such as inserting records into a table or retrieving a specific column value — do not need all of the capabilities that the Java API provides.

Now that you know how the VoltDB programming interface works, VoltDB has features to simplify common tasks and make your application development easier. Those features include:

- Using Default Procedures
- Shortcut for Defining Simple Stored Procedures
- Verifying Expected Query Results
- Writing Stored Procedures Inline Using Groovy

The following sections describe each of these features separately.

7.1. Using Default Procedures

Although it is possible to define quite complex SQL queries, often the simplest are also the most common. Inserting, selecting, updating, and deleting records based on a specific key value are the most basic operations for a database. Another common practice is upsert, where if a row matching the primary key already exists, the record is updated — if not, a new record is inserted. To simplify these operations, VoltDB defines these default stored procedures for tables.

The default stored procedures use a standard naming scheme, where the name of the procedure is composed of the name of the table (in all uppercase), a period, and the name of the query in lowercase. For example, the Hello World tutorial (`doc/tutorials/helloworld`) contains a single table, HELLOWORLD, with three columns and the partitioning column, DIALECT, as the primary key. As a result, five default stored procedures are included in addition to any user-defined procedures declared in the schema. The parameters to the procedures differ based on the procedure.

VoltDB defines a default insert stored procedure when any table is defined:

HELLOWORLD.insert	The parameters are the table columns, in the same order as defined in the schema.

VoltDB defines default update, upsert, and delete stored procedures if the table has a primary key:

HELLOWORLD.update	The parameters are the new column values, in the order defined by the schema, followed by the primary key column values. This means the primary key column values are specified twice: once as their corresponding new column values and once as the primary key value.
HELLOWORLD.upsert	The parameters are the table columns, in the same order as defined in the schema.
HELLOWORLD.delete	The parameters are the primary key column values, listed in the order they appear in the primary key definition.

VoltDB defines a default select stored procedure if the table has a primary key and the table is partitioned:

HELLOWORLD.select	The parameters are the primary key column values, listed in the order they appear in the primary key definition.

Use the sqlcmd command **show procedures** to list all the stored procedures available including the number and type of parameters required. Use `@SystemCatalog with the PROCEDURECOLUMNS` selector to show more details about the order and meaning of each procedure's parameters.

The following code example uses the default procedures for the HELLOWORLD table to insert, retrieve (select), update, and delete a new record with the key value "American":

```
VoltTable[] results;
client.callProcedure("HELLOWORLD.insert",
                     "American","Howdy","Earth");
results = client.callProcedure("HELLOWORLD.select",
                     "American").getResults();
client.callProcedure("HELLOWORLD.update",
                     "American","Yo","Biosphere",
                     "American");
client.callProcedure("HELLOWORLD.delete",
                     "American");
```

7.2. Shortcut for Defining Simple Stored Procedures

Sometimes all you want is to execute a single SQL query and return the results to the calling application. In these simple cases, writing the necessary Java code can be tedious, so VoltDB provides a shortcut. For very simple stored procedures that execute a single SQL query and return the results, you can define the entire stored procedure as part of the database schema.

Recall from Section 5.3.2, “Declaring Stored Procedures in the Schema”, that normally you use the CREATE PROCEDURE statement to specify the class name of the Java procedure you coded, for example:

```
CREATE PROCEDURE FROM CLASS MakeReservation;
CREATE PROCEDURE FROM CLASS CancelReservation;
```

However, to create procedures without writing any Java, you can simply insert the SQL query in the AS clause:

```
CREATE PROCEDURE CountReservations AS
    SELECT COUNT(*) FROM RESERVATION;
```

VoltDB creates the procedure when you include the SQL query in the CREATE PROCEDURE AS statement. Note that you must specify a unique class name for the procedure, which is unique among all stored procedures, including both those declared in the schema and those created as Java classes. (You can use the sqlcmd command **show procedures** to display a list of all stored procedures.)

It is also possible to pass arguments to the SQL query in simple stored procedures. If you use the question mark placeholder in the SQL, any additional arguments you pass in client applications through the `callProcedure()` method are used to replace the placeholders, in their respective order. For example, the following simple stored procedure expects to receive three additional parameters:

```
CREATE PROCEDURE MyReservationsByTrip AS
        SELECT R.RESERVEID, F.FLIGHTID, F.DEPARTTIME
```

```
        FROM RESERVATION AS R, FLIGHT AS F
        WHERE R.CUSTOMERID = ?
        AND R.FLIGHTID = F.FLIGHTID
        AND F.ORIGIN=? AND F.DESTINATION=?;
```

Finally, you can also specify whether the simple procedure is single-partitioned or not. By default, simple stored procedures are assumed to be multi-partitioned. But if your procedure should be single-partitioned, specify its partitioning in a PARTITION PROCEDURE statement. In the following example, the stored procedure is partitioned on the FLIGHTID column of the RESERVATION table using the first parameter as the partitioning key.

```
CREATE PROCEDURE FetchReservations AS
   SELECT * FROM RESERVATION WHERE FLIGHTID=?;
PARTITION PROCEDURE FetchReservations
   ON TABLE Reservation COLUMN flightid;
```

7.3. Verifying Expected Query Results

The automated default and simple stored procedures reduce the coding needed to perform simple queries. However, another substantial chunk of stored procedure and client application code is often required to verify the correctness of the results returned by the queries. Did you get the right number of records? Does the query return the correct value?

Rather than you having to write the code to validate the query results manually, VoltDB provides a way to perform several common validations as part of the query itself. The Java client interface includes an `Expectation` object that you can use to define the expected results of a query. Then, if the query does not meet those expectations, the executing stored procedure automatically throws a `VoltAbortException` and rolls back.

You specify the expectation as the second parameter (after the SQL statement but before any arguments) when queuing the query. For example, when making a reservation in the Flight application, the procedure must make sure there are seats available. To do this, the procedure must determine how many seats the flight has. This query can also be used to verify that the flight itself exists, because there should be one and only one record for every flight ID.

The following code fragment uses the EXPECT_ONE_ROW expectation to both fetch the number of seats and verify that the flight itself exists and is unique.

```
import org.voltdb.Expectation;
           .
           .
           .
public final SQLStmt GetSeats = new SQLStmt(
        "SELECT numberofseats FROM Flight WHERE flightid=?;");

voltQueueSQL(GetSeats, EXPECT_ONE_ROW, flightid);
VoltTable[] recordset = voltExecuteSQL();
Long numofseats = recordset[0].asScalarLong();
```

By using the expectation, the stored procedure code does not need to do additional error checking to verify that there is one and only one row in the result set. The following table describes all of the expectations that are available to use in stored procedures.

Expectation	Description
EXPECT_EMPTY	The query must return no rows.

Expectation	Description
EXPECT_ONE_ROW	The query must return one and only one row.
EXPECT_ZERO_OR_ONE_ROW	The query must return no more than one row.
EXPECT_NON_EMPTY	The query must return at least one row.
EXPECT_SCALAR	The query must return a single value (that is, one row with one column).
EXPECT_SCALAR_LONG	The query must return a single value with a datatype of Long.
EXPECT_SCALAR_MATCH(long)	The query must return a single value equal to the specified Long value.

7.4. Writing Stored Procedures Inline Using Groovy

Note

Use of embedded Groovy stored procedures is supported for compiled catalogs only. See the appendix on Using Application Catalogs in the *VoltDB Administrator's Guide* for more information.

Writing stored procedures as separate Java classes is good practice; Java is a structured language that encourages good programming habits and helps modularize your code base. However, sometimes — especially when prototyping — you just want to do something quickly and keep everything in one place.

You can write simple stored procedures directly in the schema by embedding the procedure code using the Groovy programming language (http:// groovy.codehaus.org/). Groovy is an object-oriented language that dynamically compiles to Java Virtual Machine (JVM) byte code. Groovy is not as strict as Java and promotes simpler coding through implicit typing and other shortcuts. It is important to note that Groovy is an interpreted language. It is very useful for quick coding and prototyping. However, Groovy procedures do not perform as well as the equivalent compiled Java classes. For optimal performance, Java stored procedures are recommended.

You embed a Groovy stored procedure in the schema by including the code in the CREATE PROCEDURE AS statement, enclosed by a special marker — three pound signs (###) — before and after the code. For example, the following DDL uses the CREATE PROCEDURE AS statement to implement the Insert stored procedure from the Hello World tutorial (`doc/tutorials/helloworld`) using Groovy:

```
CREATE PROCEDURE Insert AS ###                              ❶
   sql = new SQLStmt(                                       ❷
      "INSERT INTO HELLOWORLD VALUES (?, ?, ?);" )
   transactOn = { String language,                          ❸
                  String hello,
                  String world ->
      voltQueueSQL(sql, hello, world, language)
      voltExecuteSQL()
   }
### LANGUAGE GROOVY;                                        ❹
```

Some important things to note when using embedded Groovy stored procedures:

❶ Begin with three pound signs (###) before the code. The definitions for `VoltTypes`, `VoltProcedure`, and `VoltAbortException` are automatically included and can be used without explicit import statements.

❷ As with Java stored procedures, you must declare all SQL queries as `SQLStmt` objects at the beginning of the Groovy procedure.

❸ You must also define a closure called `transactOn`, which is invoked the same way the run() method is invoked in a Java stored procedure. This closure performs the actual work of the procedure and can accept any arguments that the Java run method can accept. It can also return a `VoltTable`, an array of `VoltTable`, or a long value.

❹ End the DDL statement with three pound signs (###) after the Groovy code.

In addition, VoltDB provides special wrappers, `tuplerator()` and `buildTable()`, that help you access `VoltTable` results and construct `VoltTable` structures from scratch. For example, the following code fragment shows the `ContestantWinningStates()` stored procedure from the Voter sample application (`examples/voter`) written in Groovy:

```
transactOn = { int contestantNumber, int max ->
    voltQueueSQL(resultStmt)

    results = []
    state = ""

    tuplerator(voltExecuteSQL()[0]).eachRow {
        isWinning = state != it[1]
        state = it[1]

        if (isWinning && it[0] == contestantNumber) {
            results << [state: state, votes: it[2]]
        }
    }
    if (max > results.size) max = results.size
    buildTable(state:STRING, num_votes:BIGINT) {
        results.sort { a,b -> b.votes - a.votes }[0..<max].each {
            row it.state, it.votes
        }
    }
}
```

Chapter 8. Using VoltDB with Other Programming Languages

VoltDB stored procedures are written in Java and the primary client interface also uses Java. However, that is not the only programming language you can use with VoltDB.

It is possible to have client interfaces written in almost any language. These client interfaces allow programs written in different programming languages to interact with a VoltDB database using native functions of the language. The client interface then takes responsibility for translating those requests into a standard communication protocol with the database server as described in the VoltDB wire protocol.

Some client interfaces are developed and packaged as part of the standard VoltDB distribution kit while others are compiled and distributed as separate client kits. As of this writing, the following client interfaces are available for VoltDB:

- C#
- C++
- Erlang
- Go
- Java (packaged with VoltDB)
- JDBC (packaged with VoltDB)
- JSON (packaged with VoltDB)
- Node.js
- PHP
- Python
- Ruby

The JSON client interface may be of particular interest if your favorite programming language is not listed above. JSON is a data format, rather than a programming interface, and the JSON interface provides a way for applications written in any programming language to interact with VoltDB via JSON messages sent across a standard HTTP protocol.

The following sections explain how to use the C++, JSON, and JDBC client interfaces.

8.1. C++ Client Interface

VoltDB provides a client interface for programs written in C++. The C++ client interface is available precompiled as a separate kit from the VoltDB web site, or in source format from the VoltDB github repository (http://github.com/VoltDB/voltdb-client-cpp). The following sections describe how to write VoltDB client applications in C++.

8.1.1. Writing VoltDB Client Applications in C++

When using the VoltDB client library, as with any C++ library, it is important to include all of the necessary definitions at the beginning of your source code. For VoltDB client applications, this includes defin-

itions for the VoltDB methods, structures, and datatypes as well as the libraries that VoltDB depends on (specifically, boost shared pointers). For example:

```
#define __STDC_CONSTANT_MACROS
#define __STDC_LIMIT_MACROS

#include <vector>
#include <boost/shared_ptr.hpp>
#include "Client.h"
#include "Table.h"
#include "TableIterator.h"
#include "Row.hpp"
#include "WireType.h"
#include "Parameter.hpp"
#include "ParameterSet.hpp"
#include "ProcedureCallback.hpp"
```

Once you have included all of the necessary declarations, there are three steps to using the interface to interact with VoltDB:

1. Create and open a client connection
2. Invoke stored procedures
3. Interpret the results

The following sections explain how to perform each of these functions.

8.1.2. Creating a Connection to the Database Cluster

Before you can call VoltDB stored procedures, you must create a client instance and connect to the database cluster. For example:

```
voltdb::ClientConfig config("myusername", "mypassword");
voltdb::Client client = voltdb::Client::create(config);
client.createConnection("myserver");
```

As with the Java client interface, you can create connections to multiple nodes in the cluster by making multiple calls to the **createConnection** method specifying a different IP address for each connection.

8.1.3. Invoking Stored Procedures

The C++ client library provides both a synchronous and asynchronous interface. To make a synchronous stored procedure call, you must declare objects for the parameter types, the procedure call itself, the parameters, and the response. Note that the datatypes, the procedure, and the parameters need to be declared in a specific order. For example:

```
/* Declare the number and type of parameters */
std::vector<voltdb::Parameter> parameterTypes(3);
parameterTypes[0] = voltdb::Parameter(voltdb::WIRE_TYPE_BIGINT);
parameterTypes[1] = voltdb::Parameter(voltdb::WIRE_TYPE_STRING);
parameterTypes[2] = voltdb::Parameter(voltdb::WIRE_TYPE_STRING);

/* Declare the procedure and parameter structures */
voltdb::Procedure procedure("AddCustomer", parameterTypes);
```

```
voltdb::ParameterSet* params = procedure.params();

/* Declare a client response to receive the status and return values */
voltdb::InvocationResponse response;
```

Once you instantiate these objects, you can reuse them for multiple calls to the stored procedure, inserting different values into *params* each time. For example:

```
params->addInt64(13505).addString("William").addString("Smith");
response = client.invoke(procedure);
params->addInt64(13506).addString("Mary").addString("Williams");
response = client.invoke(procedure);
params->addInt64(13507).addString("Bill").addString("Smythe");
response = client.invoke(procedure);
```

8.1.4. Invoking Stored Procedures Asynchronously

To make asynchronous procedure calls, you must also declare a callback structure and method that will be used when the procedure call completes.

```
class AsyncCallback : public voltdb::ProcedureCallback
{
public:
    bool callback
         (voltdb::InvocationResponse response)
         throw (voltdb::Exception)
    {
           /*
           * The work of your callback goes here...
           */
    }
};
```

Then, when you go to make the actual stored procedure invocation, you declare an callback instance and invoke the procedure, using both the procedure structure and the callback instance:

```
boost::shared_ptr<AsyncCallback> callback(new AsyncCallback());
client.invoke(procedure, callback);
```

Note that the C++ interface is single-threaded. The interface is not thread-safe and you should not use instances of the client, client response, or other client interface structures from within multiple concurrent threads. Also, the application must release control occasionally to give the client interface an opportunity to issue network requests and retrieve responses. You can do this by calling either the run() or runOnce() methods.

The run() method waits for and processes network requests, responses, and callbacks until told not to. (That is, until a callback returns a value of false.)

The runOnce() method processes any outstanding work and then returns control to the client application.

In most applications, you will want to create a loop that makes asynchronous requests and then calls runOnce(). This allows the application to queue stored procedure requests as quickly as possible while also processing any incoming responses in a timely manner.

Another important difference when making stored procedure calls asynchronously is that you must make sure all of the procedure calls complete before the client connection is closed. The client objects destructor

automatically closes the connection when your application leaves the context or scope within which the client is defined. Therefore, to make sure all asynchronous calls have completed, be sure to call the *drain* method until it returns true before leaving your client context:

```
while (!client.drain()) {}
```

8.1.5. Interpreting the Results

Both the synchronous and asynchronous invocations return a client response object that contains both the status of the call and the return values. You can use the status information to report problems encountered while running the stored procedure. For example:

```
if (response.failure())
{
    std::cout << "Stored procedure failed. " << response.toString();
    exit(-1);
}
```

If the stored procedure is successful, you can use the client response to retrieve the results. The results are returned as an array of VoltTable structures. Within each VoltTable object you can use an iterator to walk through the rows. There are also methods for retrieving each datatype from the row. For example, the following example displays the results of a single VoltTable containing two strings in each row:

```
/* Retrieve the results and an iterator for the first volttable */
std::vector<voltdb::Table> results = response.results();
voltdb::TableIterator iterator = results[0].iterator();

/* Iterate through the rows */
while (iterator.hasNext())
{
    voltdb::Row row = iterator.next();
    std::cout << row.getString(0) << ", " << row.getString(1) << std::endl;
}
```

8.2. JSON HTTP Interface

JSON (JavaScript Object Notation) is not a programming language; it is a data format. The JSON "interface" to VoltDB is actually a web interface that the VoltDB database server makes available for processing requests and returning data in JSON format.

The JSON interface lets you invoke VoltDB stored procedures and receive their results through HTTP requests. To invoke a stored procedure, you pass VoltDB the procedure name and parameters as a querystring to the HTTP request, using either the GET or POST method.

Although many programming languages provide methods to simplify the encoding and decoding of JSON strings, you still need to understand the data structures that are created. So if you are not familiar with JSON encoding, you may want to read more about it at `http://www.json.org`.

8.2.1. How the JSON Interface Works

To use the VoltDB JSON interface, you must first enable JSON in the deployment file. You do this by adding the following tags to the deployment file:

```
<httpd>
```

```
   <jsonapi enabled="true"/>
</httpd>
```

With JSON enabled, when a VoltDB database starts it opens port 8080[1] on the local machine as a simple web server. Any HTTP requests sent to the location /api/1.0/ on that port are interpreted as requests to run a stored procedure. The structure of the request is:

URL	http://<server>:8080/api/1.0/
Arguments	Procedure=<procedure-name> Parameters=<procedure-parameters> User=<username for authentication> Password=<password for authentication> Hashedpassword=<Hashed password for authentication> admin=<true|false> jsonp=<function-name>

The arguments can be passed either using the GET or the POST method. For example, the following URL uses the GET method (where the arguments are appended to the URL) to execute the system procedure @SystemInformation on the VoltDB database running on node voltsvr.mycompany.com:

```
http://voltsvr.mycompany.com:8080/api/1.0/?Procedure=@SystemInformation
```

Note that only the `Procedure` argument is required. You can authenticate using the `User` and `Password` (or `Hashedpassword`) arguments if security is enabled for the database. Use `Password` to send the password as plain text or `Hashedpassword` to send the password as an encoded string. (The hashed password must be either a 40-byte hex-encoding of the 20-byte SHA-1 hash or a 64-byte hex-encoding of the 32-byte SHA-256 hash.)[2]

You can also include the parameters on the request. However, it is important to note that the parameters — and the response returned by the stored procedure — are JSON encoded. The parameters are an array (even if there is only one element to that array) and therefore must be enclosed in square brackets. Also, although there is an upper limit of 2 megabytes for the entire length of the parameter string, large parameter sets must be sent using POST to avoid stricter limitations on allowable URL lengths.

The `admin` argument specifies whether the request is submitted on the standard client port (the default) or the admin port (when you specify `admin=true`). When the database is in admin mode, the client port is read-only; so you must submit write requests with `admin=true` or else the request is rejected by the server.

The `jsonp` argument is provided as a convenience for browser-based applications (such as Javascript) where cross-domain browsing is disabled. When you include the `jsonp` argument, the entire response is wrapped as a function call using the function name you specify. Using this technique, the response is a complete and valid Javascript statement and can be executed to create the appropriate language-specific object. For example, calling the @Statistics system procedure in Javascript using the jQuery library looks like this:

```
$.getJSON('http://myserver:8080/api/1.0/?Procedure=@Statistics' +
          '&Parameters=["MANAGEMENT",0]&jsonp=?',
          {},MyCallBack);
```

[1] You can specify an alternate port for the JSON interface when you start the VoltDB server by including the port number as an attribute of the <httpd> tag in the deployment file. For example: `<httpd port="{port-number}">`.

[2] Hashing the password stops the text of your password from being detectable from network traffic. However, it does not make the database access any more secure. To secure the transmission of credentials and data between client applications and VoltDB, use an SSL proxy server in front of the database servers.

Perhaps the best way to understand the JSON interface is to see it in action. If you build and start the Hello World example application that is provided in the VoltDB distribution kit (including the client that loads data into the database), you can then open a web browser and connect to the local system through port 8080, to retrieve the French translation of "Hello World". For example:

```
http://localhost:8080/api/1.0/?Procedure=Select&Parameters=["French"]
```

The resulting display is the following:

```
{"status":1,"appstatus":-128,"statusstring":null,"appstatusstring":null,
"exception":null,"results":[{"status":-128,"schema":[{"name":"HELLO",
"type":9},{"name":"WORLD","type":9}],"data":[["Bonjour","Monde"]]}]}
```

As you can see, the results (which are a JSON-encoded string) are not particularly easy to read. But then, the JSON interface is not really intended for human consumption. It's real purpose is to provide a generic interface accessible from almost any programming language, many of which already provide methods for encoding and decoding JSON strings and interpreting their results.

8.2.2. Using the JSON Interface from Client Applications

The general process for using the JSON interface from within a program is:

1. Encode the parameters for the stored procedure as a JSON-encoded string
2. Instantiate and execute an HTTP request, passing the name of the procedure and the parameters as arguments using either GET or POST.
3. Decode the resulting JSON string into a language-specific data structure and interpret the results.

The following are examples of invoking the Hello World Insert stored procedure from several different languages. In each case, the three arguments (the name of the language and the words for "Hello" and "World") are encoded as a JSON string.

PHP

```
// Construct the procedure name, parameter list, and URL.

   $voltdbserver = "http://myserver:8080/api/1.0/";
   $proc = "Insert";
   $a = array("Croatian","Pozdrav","Svijet");
   $params = json_encode($a);
   $params = urlencode($params);
   $querystring = "Procedure=$proc&Parameters=$params";

// create a new cURL resource and set options
   $ch = curl_init();
   curl_setopt($ch, CURLOPT_URL, $voltdbserver);
   curl_setopt($ch, CURLOPT_HEADER, 0);
   curl_setopt($ch, CURLOPT_FAILONERROR, 1);
   curl_setopt($ch, CURLOPT_POST, 1);
   curl_setopt($ch, CURLOPT_POSTFIELDS, $querystring);
   curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

// Execute the request
```

```
$resultstring = curl_exec($ch);
```

Python

```
import urllib
import urllib2
import json

# Construct the procedure name, parameter list, and URL.
url = 'http://myserver:8080/api/1.0/'
voltparams = json.dumps(["Croatian","Pozdrav","Svijet"])
httpparams = urllib.urlencode({
    'Procedure': 'Insert',
    'Parameters' : voltparams
})
print httpparams
# Execute the request
data = urllib2.urlopen(url, httpparams).read()

# Decode the results
result = json.loads(data)
```

Perl

```
use LWP::Simple;

my $server = 'http://myserver:8080/api/1.0/';

# Insert "Hello World" in Croatian
my $proc = 'Insert';
my $params = '["Croatian","Pozdrav","Svijet"]';
my $url = $server . "?Procedure=$proc&Parameters=$params";
my $content = get $url;
die "Couldn't get $url" unless defined $content;
```

C#

```
using System;
using System.Text;
using System.Net;
using System.IO;

namespace hellovolt

{
  class Program
  {
    static void Main(string[] args)
    {
      string VoltDBServer = "http://myserver:8080/api/1.0/";
      string VoltDBProc = "Insert";
      string VoltDBParams = "[\"Croatian\",\"Pozdrav\",\"Svijet\"]";
      string Url = VoltDBServer + "?Procedure=" + VoltDBProc
                  + "&Parameters=" + VoltDBParams;
```

```
        string result = null;
        WebResponse response = null;
        StreamReader reader = null;

        try
        {
          HttpWebRequest request = (HttpWebRequest)WebRequest.Create(Url);
          request.Method = "GET";
          response = request.GetResponse();
          reader = new StreamReader(response.GetResponseStream(),Encoding.UTF8 );
          result = reader.ReadToEnd();

        }
        catch (Exception ex)

        {     // handle error
          Console.WriteLine( ex.Message );
        }
        finally
        {
          if (reader != null)reader.Close();
          if (response != null) response.Close();

        }
      }
    }
  }
}
```

8.2.3. How Parameters Are Interpreted

When you pass arguments to the stored procedure through the JSON interface, VoltDB does its best to map the data to the datatype required by the stored procedure. This is important to make sure partitioning values are interpreted correctly.

For integer values, the JSON interface maps the parameter to the smallest possible integer type capable of holding the value. (For example, BYTE for values less than 128). Any values containing a decimal point are interpreted as DOUBLE.

String values (those that are quoted) are handled in several different ways. If the stored procedure is expecting a BIGDECIMAL, the JSON interface will try to interpret the quoted string as a decimal value. If the stored procedure is expecting a TIMESTAMP, the JSON interface will try to interpret the quoted string as a JDBC-encoded timestamp value. (You can alternately pass the argument as an integer value representing the number of microseconds from the epoch.) Otherwise, quoted strings are interpreted as a string datatype.

Table 8.1, “Datatypes in the JSON Interface” summarizes how to pass different datatypes in the JSON interface.

Table 8.1. Datatypes in the JSON Interface

Datatype	How to Pass	Example
Integers (Byte, Short, Integer, Long)	An integer value	12345

Datatype	How to Pass	Example
DOUBLE	A value with a decimal point	123.45
BIGDECIMAL	A quoted string containing a value with a decimal point	"123.45"
TIMESTAMP	Either an integer value or a quoted string containing a JDBC-encoded date and time	12345 "2010-07-01 12:30:21"
String	A quoted string	"I am a string"

8.2.4. Interpreting the JSON Results

Making the request and decoding the result string are only the first steps. Once the request is completed, your application needs to interpret the results.

When you decode a JSON string, it is converted into a language-specific structure within your application, composed of objects and arrays. If your request is successful, VoltDB returns a JSON-encoded string that represents the same ClientResponse object returned by calls to the callProcedure method in the Java client interface. Figure 8.1, “The Structure of the VoltDB JSON Response” shows the structure of the object returned by the JSON interface.

Figure 8.1. The Structure of the VoltDB JSON Response

```
{  appstatus           (integer, boolean)
   appstatusstring     (string)
   exception           (integer)
   results             (array)
     [                 (object, VoltTable)
         {  data       (array)
              [        (any type)
              ]
            schema     (array)
              [  name  (string)
                 type  (integer, enumerated)
              ]
            status     (integer, boolean)
         }
     ]
   status              (integer)
   statusstring        (string)
}
```

The key components of the JSON response are the following:

appstatus — Returns additional information, provided by the application developer, about the success or failure of the stored procedure. The values of *appstatus* and *appstatusstring* can be set programmatically in the stored procedure. (See Section 6.5.1, “Interpreting Execution Errors” for details.)

results — An array of objects representing the data returned by the stored procedure. This is an array of VoltTable objects. If the stored procedure does not return a value (i.e. is void or null), then *results* will be null.

data — Within each VoltTable object, *data* is the array of values.

schema — Within each VoltTable, object *schema* is an array of objects with two elements: the name of the field and the datatype of that field (encoded as an enumerated integer value).

status — Indicates the success or failure of the stored procedure. If *status* is false, *statusstring* contains the text of the status message..

It is possible to create a generic procedure for testing and evaluating the result values from any VoltDB stored procedure. However, in most cases it is far more expedient to evaluate the values that you know the individual procedures return.

For example, again using the Hello World example that is provided with the VoltDB software, it is possible to use the JSON interface to call the Select stored procedure and return the values for "Hello" and "World" in a specific language. Rather than evaluate the entire results array (including the name and type fields), we know we are only receiving one VoltTable object with two string elements. So we can simplify the code, as in the following python example:

```
import urllib
import urllib2
import json
import pprint

# Construct the procedure name, parameter list, and URL.
url = 'http://localhost:8080/api/1.0/'
voltparams = json.dumps(["French"])
httpparams = urllib.urlencode({
    'Procedure': 'Select',
    'Parameters' : voltparams
})

# Execute the request
data = urllib2.urlopen(url, httpparams).read()

# Decode the results
result = json.loads(data)

# Get the data as a simple array and display them
foreignwords = result[u'results'][0][u'data'][0]

print foreignwords[0], foreignwords[1]
```

8.2.5. Error Handling using the JSON Interface

There are a number of different reasons why a stored procedure request using the JSON interface may fail: the VoltDB server may be unreachable, the database may not be started yet, the stored procedure name may be misspelled, the stored procedure itself may fail... When using the standard Java client interface, these different situations are handled at different times. (For example, server and database access issues are addressed when instantiating the client, whereas stored procedure errors can be handled when the procedures themselves are called.) The JSON interface simplifies the programming by rolling all of these activities into a single call. But you must be more organized in how you handle errors as a consequence.

When using the JSON interface, you should check for errors in the following order:

1. First check to see that the HTTP request was submitted without errors. How this is done depends on what language-specific methods you use for submitting the request. In most cases, you can use the appropriate programming language error handlers (such as try-catch) to catch and interpret HTTP request errors.

2. Next check to see if VoltDB successfully invoked the stored procedure. You can do this by verifying that the HTTP request returned a valid JSON-encoded string and that its *status* is set to true.

3. If the VoltDB server successfully invoked the stored procedure, then check to see if the stored procedure itself succeeded, by checking to see if *appstatus* is true.

4. Finally, check to see that the results are what you expect. (For example, that the *data* array is non-empty and contains the values you need.)

8.3. JDBC Interface

JDBC (Java Database Connectivity) is a programming interface for Java programmers that abstracts database specifics from the methods used to access the data. JDBC provides standard methods and classes for accessing a relational database and vendors then provide JDBC drivers to implement the abstracted methods on their specific software.

VoltDB provides a JDBC driver for those who would prefer to use JDBC as the data access interface. The VoltDB JDBC driver supports ad hoc queries, prepared statements, calling stored procedures, and methods for examining the metadata that describes the database schema.

8.3.1. Using JDBC to Connect to a VoltDB Database

The VoltDB driver is a standard class within the VoltDB software jar. To load the driver you use the Class.forName method to load the class org.voltdb.jdbc.Driver.

Once the driver is loaded, you create a connection to a running VoltDB database server by constructing a JDBC url using the "jdbc:" protocol, followed by "voltdb://", the server name, a colon, and the port number. In other words, the complete JDBC connection url is "jdbc:voltdb://{server}:{port}". To connect to multiple nodes in the cluster, use a comma separated list of server names and port numbers after the "jdbc:voltdb://" prefix.

For example, the following code loads the VoltDB JDBC driver and connects to the servers svr1 and svr2 using the default client port:

```
Class.forName("org.voltdb.jdbc.Driver");
Connection c = DriverManager.getConnection(
     "jdbc:voltdb://svr1:21212,svr2:21212");
```

If security is enabled for the database, you must also provide a username and password. Set these as properties using the setProperty method before creating the connection and then pass the properties as a second argument to getConnection. For example, the following code uses the username/password pair of "Hemingway" and "KeyWest" to authenticate to the VoltDB database:

```
Class.forName("org.voltdb.jdbc.Driver");
Properties props = new Properties();
props.setProperty("user", “Hemingway");
props.setProperty("password", “KeyWest");
Connection c = DriverManager.getConnection(
     "jdbc:voltdb://svr1:21212,svr2:21212", props);
```

8.3.2. Using JDBC to Query a VoltDB Database

Once the connection is made, you use the standard JDBC classes and methods to access the database. (See the JDBC documentation at `http://download.oracle.com/javase/6/docs/technotes/`

guides/jdbc for details.) Note, however, when running the JDBC application, you must make sure both the VoltDB software jar and the Guava library are in the Java classpath. Guava is a third party library that is shipped as part of the VoltDB kit in the /lib directory. Unless you include both components in the classpath, your application will not be able to find and load the necessary driver class.

The following is a complete example that uses JDBC to access the Hello World tutorial that comes with the VoltDB software in the subdirectory /doc/tutorials/helloworld. The JDBC demo program executes both an ad hoc query and a call to the VoltDB stored procedure, Select.

```
import java.sql.*;
import java.io.*;

public class JdbcDemo {

    public static void main(String[] args) {

        String driver = "org.voltdb.jdbc.Driver";
        String url = "jdbc:voltdb://localhost:21212";
        String sql = "SELECT dialect FROM helloworld";

        try {
                // Load driver. Create connection.
            Class.forName(driver);
            Connection conn = DriverManager.getConnection(url);

                // create a statement
            Statement query = conn.createStatement();
            ResultSet results = query.executeQuery(sql);
            while (results.next()) {
                System.out.println("Language is " + results.getString(1));
            }

                // call a stored procedure
            CallableStatement proc = conn.prepareCall("{call Select(?)}");
            proc.setString(1, "French");
            results = proc.executeQuery();
            while (results.next()) {
                System.out.printf("%s, %s!\n", results.getString(1),
                                                results.getString(2));
            }

                 //Close statements, connections, etc.
            query.close();
            proc.close();
            results.close();
            conn.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Chapter 9. Using VoltDB in a Cluster

It is possible to run VoltDB on a single server and still get all the advantages of parallelism because VoltDB creates multiple partitions on each server. However, there are practical limits to how much memory or processing power any one server can sustain.

One of the key advantages of VoltDB is its ease of expansion. You can increase both capacity and processing (i.e. the total number of partitions) simply by adding servers to the cluster to achieve almost linear scalability. Using VoltDB in a cluster also gives you the ability to increase the availability of the database — protecting it against possible server failures or network glitches.

This chapter explains how to create a cluster of VoltDB servers running a single database. It also explains how to expand the cluster when additional capacity or processing power is needed. The following chapters explain how to increase the availability of your database through the use of K-safety and database replication, as well as how to enable security to limit access to the data.

9.1. Starting a Database Cluster

As described in Chapter 3, *Starting the Database*, starting a VoltDB cluster is similar to starting VoltDB on a single server — you use the same commands. Starting a single server database, you simply use the **voltdb create** command by itself. Or, to customize database features, you can specify a deployment file as well.

To start a cluster you must:

- Specify the number of nodes in the cluster in the deployment file using the hostcount attribute:

  ```
  <cluster  hostcount="5" />
  ```

- Choose one of the nodes as the lead or "host" node and specify that node using the --host argument on the start command

- Issue the start command on all nodes of the cluster

For example, if you are creating a new five node cluster and choose node server3 as the host, you would issue a command like the following on all five nodes:

```
$ voltdb create --host=server3 --deployment=deployment.xml
```

To restart a cluster using commands logs or automatic snapshots, you repeat this process replacing the **create** action with **recover**:

```
$ voltdb recover --host=server3 --deployment=deployment.xml
```

In both cases you choose one node, any node, to act as the leader for initiating the cluster. Once the database cluster is running the leader's special role is complete and all nodes are peers.

9.2. Updating the Cluster Configuration

If you choose to change the configuration of your cluster — adding or removing nodes or changing the K-safety value or number of partitions per server — you can save the database as a snapshot, shutdown, edit the deployment file, restart with the new number of servers, and restore the database. (See Chapter 13, *Saving & Restoring a VoltDB Database* for information on using **save** and **restore**).When doing benchmarking, where you need to change the number of partitions or other runtime options, this is the correct approach.

However, if you are simply adding nodes to the cluster to add capacity or increase performance, you can add the nodes while the database is running. Adding nodes "on the fly" is also known as *elastic* scaling.

9.2.1. Adding Nodes with Elastic Scaling

When you are ready to extend the cluster by adding one or more nodes, you simply start the VoltDB database process on the new nodes using the **voltdb add** command specifying the name of one of the existing cluster nodes as the host. For example, if you are adding node ServerX to a cluster where ServerA is already a member, you can execute the following command on ServerX:

```
me@ServerX:~$ voltdb add -l ~/license.xml --host=ServerA
```

Once the add action is initiated, the cluster performs the following tasks:

1. The cluster acknowledges the presence of a new server.
2. The active application catalog and deployment settings are sent to the new node.
3. Once sufficient nodes are added, copies of all replicated tables and their share of the partitioned tables are sent to the new nodes.
4. As the data is redistributed (or *rebalanced*), the added nodes begin participating as full members of the cluster.

There are some important notes to consider when expanding the cluster using elastic scaling:

- You must add a sufficient number of nodes to create an integral K-safe unit. That is, K+1 nodes. For example, if the K-safety value for the cluster is two, you must add three nodes at a time to expand the cluster. If the cluster is not K-safe (in other words it has a K-safety value of zero), you can add one node at a time.
- When you add nodes to a K-safe cluster, the nodes added first will complete steps #1 and #2 above, but will not complete steps #3 and #4 until the correct number of nodes are added, at which point all nodes rebalance together.
- While the cluster is rebalancing (Step #3), the database continues to handle incoming requests. However, depending on the workload and amount of data in the database, rebalancing may take a significant amount of time.
- When using database replication (DR), the master and replica databases must have the same configuration. If you use elasticity to add nodes to the master cluster, replication stops. Once rebalancing is complete on the master database, you can restart the replica with additional nodes matching the new master cluster configuration and restart replication.

9.2.2. Configuring How VoltDB Rebalances New Nodes

Once you add the necessary number of nodes (based on the K-safety value), VoltDB rebalances the cluster, moving data from existing partitions to the new nodes. During the rebalance operation, the database remains available and actively processing client requests. How long the rebalance operation takes is dependent on two factors: how often rebalance tasks are processed and how much data each transaction moves.

Rebalance tasks are fully transactional, meaning they operate within the database's ACID-compliant transactional model. Because they involve moving data between two or more partitions, they are also multi-partition transactions. This means that each rebalance work unit can incrementally add to the latency of pending client transactions.

You can control how quickly the rebalance operation completes versus how much rebalance work impacts ongoing client transactions using two attributes of the `<elastic>` element in the deployment file:

- The **duration** attribute sets a target value for the length of time each rebalance transaction will take, specified in milliseconds. The default is 50 milliseconds.
- The **throughput** attribute sets a target value for the number of megabytes per second that will be processed by the rebalance transactions. The default is 2 megabytes.

When you change the target duration, VoltDB adjusts the amount of data that is moved in each transaction to reach the target execution time. If you increase the duration, the volume of data moved per transaction increases. Similarly, if you reduce the duration, the volume per transaction decreases.

When you change the target throughput, VoltDB adjusts the frequency of rebalance transactions to achieve the desired volume of data moved per second. If you increase the target throughout, the number of rebalance transactions per second increases. Similarly, if you decrease the target throughout, the number of transactions decreases.

The <elastic> element is a child of the <systemsettings> element. For example, the following deployment file sets the target duration to 15 milliseconds and the target throughput to 1 megabyte per second before starting the database:

```
<deployment>
   . . .
   <systemsettings>
       <elastic duration="15" throughput="1"/>
   </systemsettings>
</deployment>
```

Chapter 10. Availability

Durability is one of the four key ACID attributes required to ensure the accurate and reliable operation of a transactional database. Durability refers to the ability to maintain database consistency and availability in the face of external problems, such as hardware or operating system failure. Durability is provided by four features of VoltDB: snapshots, command logging, K-safety, and disaster recovery through database replication.

- *Snapshots* are a "snapshot" of the data within the database at a given point in time written to disk. You can use these snapshot files to restore the database to a previous, known state after a failure which brings down the database. The snapshots are guaranteed to be transactionally consistent at the point at which the snapshot was taken. Chapter 13, *Saving & Restoring a VoltDB Database* describes how to create and restore database snapshots.

- *Command Logging* is a feature where, in addition to periodic snapshots, the system keeps a log of every stored procedure (or "command") as it is invoked. If, for any reason, the servers fail, they can "replay" the log on startup to reinstate the database contents completely rather than just to an arbitrary point-in-time. Chapter 14, *Command Logging and Recovery* describes how to enable, configure, and replay command logs.

- *K-safety* refers to the practice of duplicating database partitions so that the database can withstand the loss of cluster nodes without interrupting the service. For example, a K value of zero means that there is no duplication and losing any servers will result in a loss of data and database operations. If there are two copies of every partition (a K value of one), then the cluster can withstand the loss of at least one node (and possibly more) without any interruption in service.

- *Database Replication* is similar to K-safety, since it involves replicating data. However, rather than creating redundant partitions within a single database, database replication involves creating and maintaining a complete copy of the entire database. Database replication has a number of uses, but specifically in terms of durability, replication lets you maintain two copies of the database in separate geographic locations. In case of catastrophic events, such as fires, earthquakes, or large scale power outages, the replica can be used as a replacement for a disabled cluster.

Subsequent chapters describe snapshots and command logging. The next chapter describes how you can use database replication for disaster recovery. This chapter explains how K-safety works, how to configure your VoltDB database for different values of K, and how to recover in the case of a system failure.

10.1. How K-Safety Works

K-safety involves duplicating database partitions so that if a partition is lost (either due to hardware or software problems) the database can continue to function with the remaining duplicates. In the case of VoltDB, the duplicate partitions are fully functioning members of the cluster, including all read and write operations that apply to those partitions. (In other words, the duplicates function as peers rather than in a master-slave relationship.)

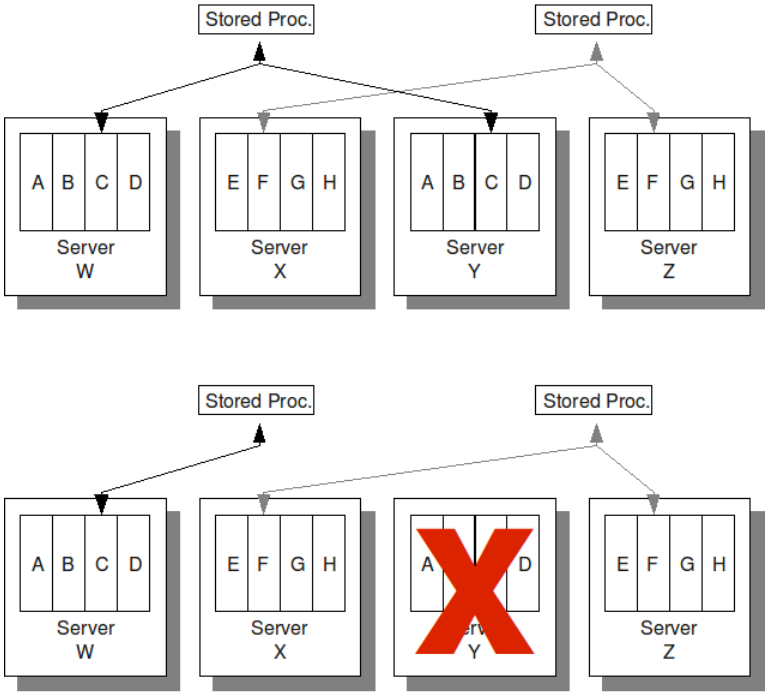
It is also important to note that K-safety is different than WAN replication. In replication the entire database cluster is replicated (usually at a remote location to provide for disaster recovery in case the entire cluster or site goes down due to catastrophic failure of some type).

In replication, the replicated cluster operates independently and cannot assist when only part of the active cluster fails. The replicate is intended to take over only when the primary database cluster fails entirely. So, in cases where the database is mission critical, it is not uncommon to use both K-safety and replication to achieve the highest levels of service.

To achieve K=1, it is necessary to duplicate all partitions. (If you don't, failure of a node that contains a non-duplicated partition would cause the database to fail.) Similarly, K=2 requires two duplicates of every partition, and so on.

What happens during normal operations is that any work assigned to a duplicated partition is sent to all copies (as shown in Figure 10.1, “K-Safety in Action”). If a node fails, the database continues to function sending the work to the unaffected copies of the partition.

Figure 10.1. K-Safety in Action



10.2. Enabling K-Safety

You specify the desired K-safety value as part of the cluster configuration in the VoltDB deployment file for your application. By default, VoltDB uses a K-safety value of zero (no duplicate partitions). You can specify a larger K-safety value using the kfactor attribute of the <cluster> tag. For example, in the following deployment file, the K-safety value for a 6-node cluster with 4 partitions per node is set to 2:

```
<?xml version="1.0"?>
<deployment>
   <cluster hostcount="6"
            sitesperhost="4"
            kfactor="2"
   />
</deployment>
```

When you start the database specifying a K-safety value greater than zero, the appropriate number of partitions out of the cluster will be assigned as duplicates. For example, in the preceding case where there are 6 nodes and 4 partitions per node, there are a total of 24 partitions. With K=1, half of those partitions (12) will be assigned as duplicates of the other half. If K is increased to 2, the cluster would be divided into 3 copies consisting of 8 partitions each.

The important point to note when setting the K value is that, if you do not change the hardware configuration, you are dividing the available partitions among the duplicate copies. Therefore performance (and capacity) will be proportionally decreased as K-safety is increased. So running K=1 on a 6-node cluster will be approximately equivalent to running a 3-node cluster with K=0.

If you wish to increase reliability without impacting performance, you must increase the cluster size to provide the appropriate capacity to accommodate for K-safety.

10.2.1. What Happens When You Enable K-Safety

Of course, to ensure a system failure does not impact the database, not only do the partitions need to be duplicated, but VoltDB must ensure that the duplicates are kept on separate nodes of the cluster. To achieve this, VoltDB calculates the maximum number of unique partitions that can be created, given the number of nodes, partitions per node, and the desired K-safety value.

When the number of nodes is an integral multiple of the duplicates needed, this is easy to calculate. For example, if you have a six node cluster and choose K=1, VoltDB will create two instances of three nodes each. If you choose K=2, VoltDB will create three instances of two nodes each. And so on.

If the number of nodes is not a multiple of the number of duplicates, VoltDB does its best to distribute the partitions evenly. For example, if you have a three node cluster with two partitions per node, when you ask for K=1 (in other words, two of every partition), VoltDB will duplicate three partitions, distributing the six total partitions across the three nodes.

10.2.2. Calculating the Appropriate Number of Nodes for K-Safety

By now it should be clear that there is a correlation between the K value and the number of nodes and partitions in the cluster. Ideally, the number of nodes is a multiple of the number of copies needed (in other words, the K value plus one). This is both the easiest configuration to understand and manage.

However, if the number of nodes is not an exact multiple, VoltDB distributes the duplicated partitions across the cluster using the largest number of unique partitions possible. This is the highest whole integer where the number of unique partitions is equal to the total number of partitions divided by the needed number of copies:

```
Unique partitions = (nodes * partitions/node) / (K + 1)
```

Therefore, when you specify a cluster size that is not a multiple of K+1, but where the total number of partitions is, VoltDB will use all of the partitions to achieve the required K-safety value.

Note that the total number of partitions must be a whole multiple of the number of copies (that is, K+1). If neither the number of nodes nor the total number of partitions is divisible by K+1, then VoltDB will not let the cluster start and will display an appropriate error message. For example, if the deployment file specifies a three node cluster with 3 sites per host and a K-safety value of 1, the cluster cannot start because the total number of partitions (3X3=9) is not a multiple of the number of copies (K+1=2). To start the cluster, you must either increase the K-safety value to 2 (so the number of copies is 3) or change the sites per host to 2 or 4 so the total number of partitions is divisible by 2.

Finally, if you specify a K value higher than the available number of nodes, it is not possible to achieve the requested K-safety. Even if there are enough partitions to create the requested duplicates, VoltDB cannot distribute the duplicates to distinct nodes. For example, if you have a 3 node cluster with 4 partitions per node (12 total partitions), there are enough partitions to achieve a K value of 3, but not without some duplicates residing on the same node. In this situation, VoltDB issues an error message. You must either reduce the K-safety or increase the number of nodes.

10.3. Recovering from System Failures

When running without K-safety (in other words, a K-safety value of zero) any node failure is fatal and will bring down the database (since there are no longer enough partitions to maintain operation). When running with K-safety on, if a node goes down, the remaining nodes of the database cluster log an error indicating that a node has failed.

By default, these error messages are logged to the console terminal. Since the loss of one or more nodes reduces the reliability of the cluster, you may want to increase the urgency of these messages. For example, you can configure a separate Log4J appender (such as the SMTP appender) to report node failure messages. To do this, you should configure the appender to handle messages of class HOST and severity level ERROR or greater. See the chapter on Logging in the *VoltDB Administrator's Guide* for more information about configuring logging.

When a node fails with K-safety enabled, the database continues to operate. But at the earliest possible convenience, you should repair (or replace) the failed node.

To replace a failed node to a running VoltDB cluster, you restart the VoltDB server process specifying the deployment file, **rejoin** as the start action, and the address of one of the remaining nodes of the cluster as the *host*. For example, to rejoin a node to the VoltDB cluster where myclusternode5 is one of the current member nodes, you use the following command:

```
$ voltdb  rejoin --host=myclusternode5 \
          --deployment=mydeployment.xml
```

Note that the node you specify may be any active cluster node; it *does not* have to be the node identified as the host when the cluster was originally started. Also, the deployment file you specify must be the currently active deployment settings for the running database cluster.

10.3.1. What Happens When a Node Rejoins the Cluster

When you issue the rejoin command, the node first rejoins the cluster, then retrieves a copy of the database schema and the appropriate data for its partitions from other nodes in the cluster. Rejoining the cluster only takes seconds and once this is done and the schema is received, the node can accept and distribute stored procedure requests like any other member.

However, the new node will not actively participate in the work until a full working copy of its partition data is received. The rejoin process can happen in two different ways: blocking and "live".

During a *blocking* rejoin, the update process for each partition operates as a single transaction and will block further transactions on the partition which is providing the data. While the node is rejoining and being updated, the cluster continues to accept work. If the work queue gets filled (because the update is blocking further work), the client applications will experience back pressure. Under normal conditions, this means the calls to submit stored procedures with the callProcedure method (either synchronously or asynchronously) will wait until the back pressure clears before returning control to the calling application. The time this update process takes varies in length depending on the volume of data involved and network bandwidth. However, the process should not take more than a few minutes.

During a *live* rejoin, the update separates the rejoin process from the standard transactional workflow, allowing the database to continue operating with a minimal impact to throughput or latency. The advantage of a live rejoin is that the database remains available and responsive to client applications throughout the rejoin procedure. The deficit of a live rejoin is that, for large datasets, the rejoin process can take longer to complete than with a blocking rejoin.

By default, VoltDB performs live rejoins, allowing the work of the database to continue. If, for any reason, you choose to perform a blocking rejoin, you can do this by using the `--blocking` flag on the command line. For example, the following command performs a blocking rejoin to the database cluster including the node myclusternode5:

```
$ voltdb rejoin --blocking --host=myclusternode5 \
         --deployment mydeployment.xml
```

In rare cases, if the database is near capacity in terms of throughput, a live rejoin cannot keep up with the ongoing changes made to the data. If this happens, VoltDB reports that the live rejoin cannot complete and you must wait until database activity subsides or you can safely perform a blocking rejoin to reconnect the server.

It is important to remember that the cluster is not fully K-safe until the restoration is complete. For example, if the cluster was established with a K-safety value of two and one node failed, until that node rejoins and is updated, the cluster is operating with a K-safety value of one. Once the node is up to date, the cluster becomes fully operational and the original K-safety is restored.

10.3.2. Where and When Recovery May Fail

It is possible to rejoin any appropriately configured node to the cluster. It does not have to be the same physical machine that failed. This way, if a node fails for hardware reasons, it is possible to replace it in the cluster immediately with a new node, giving you time to diagnose and repair the faulty hardware without endangering the database itself.

It is also possible, when doing blocking rejoins, to rejoin multiple nodes simultaneously, if multiple nodes fail. That is, assuming the cluster is still viable after the failures. As long as there is at least one active copy of every partition, the cluster will continue to operate and be available for nodes to rejoin. Note that with live rejoin, only one node can rejoin at a time.

There are a few conditions in which the rejoin operation may fail. Those situations include the following:

- Insufficient K-safety

  If the database is running without K-safety, or more nodes fail simultaneously than the cluster is capable of sustaining, the entire cluster will fail and must be restarted from scratch. (At a minimum, a VoltDB database running with K-safety can withstand at least as many simultaneous failures as the K-safety value. It may be able to withstand more node failures, depending upon the specific situation. But the K-safety value tells you the minimum number of node failures that the cluster can withstand.)

- Mismatched deployment file

  If the deployment file that you specify when issuing the rejoin command does not match the current deployment configuration of the database, the cluster will refuse to let the node rejoin.

- More nodes attempt to rejoin than have failed

  If one or more nodes fail, the cluster will accept rejoin requests from as many nodes as failed. For example, if one node fails, the first node requesting to rejoin will be accepted. Once the cluster is back to the correct number of nodes, any further requests to rejoin will be rejected. (This is the same behavior as if you tried to add more nodes than specified in the deployment file when initially starting the database.)

- The rejoining node does not specify a valid username and/or password

  When rejoining a cluster with security enabled, you must specify a valid username and password when issuing the rejoin command. The username and password you specify must have sufficient privileges to

execute system procedures. If not, the rejoin request will be rejected and an appropriate error message displayed.

10.4. Avoiding Network Partitions

VoltDB achieves scalability by creating a tightly bound network of servers that distribute both data and processing. When you configure and manage your own server hardware, you can ensure that the cluster resides on a single network switch, guaranteeing the best network connection between nodes and reducing the possibility of network faults interfering with communication.

However, there are situations where this is not the case. For example, if you run VoltDB "in the cloud", you may not control or even know what is the physical configuration of your cluster.

The danger is that a network fault — between switches, for example — can interrupt communication between nodes in the cluster. The server nodes continue to run, and may even be able to communicate with others nodes on their side of the fault, but cannot "see" the rest of the cluster. In fact, both halves of the cluster think that the other half has failed. This condition is known as a *network partition*.

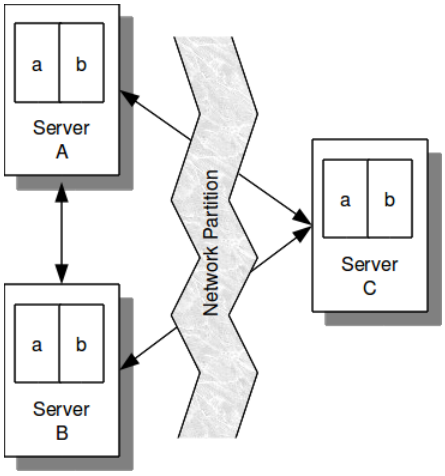
10.4.1. K-Safety and Network Partitions

When you run a VoltDB cluster without availability (in other words, no K-safety) the danger of a network partition is simple: loss of the database. Any node failure makes the cluster incomplete and the database will stop, You will need to reestablish network communications, restart VoltDB, and restore the database from the last snapshot.

However, if you are running a cluster with K-safety, it is possible that when a network partition occurs, the two separate segments of the cluster might have enough partitions each to continue running, each thinking the other group of nodes has failed.

For example, if you have a 3 node cluster with 2 sites per node, and a K-safety value of 2, each node is a separate, self-sustaining copy of the database, as shown in Figure 10.2, “Network Partition”. If a network partition separates nodes A and B from node C, each segment has sufficient partitions remaining to sustain the database. Nodes A and B think node C has failed; node C thinks that nodes A and B have failed.

Figure 10.2. Network Partition



The problem is that you never want two separate copies of the database continuing to operate and accepting requests thinking they are the only viable copy. If the cluster is physically on a single network switch,

the threat of a network partition is reduced. But if the cluster is on multiple switches, the risk increases significantly and must be accounted for.

10.4.2. Using Network Fault Protection

VoltDB provides a mechanism for guaranteeing that a network partition does not accidentally create two separate copies of the database. The feature is called network fault protection.

Because the consequences of a partition are so severe, use of network partition detection is strongly recommended and VoltDB enables partition detection by default. In addition it is recommended that, wherever possible, K-safe clusters be configured with an odd number of nodes.

However, it is possible to disable network fault protection in the deployment file, if you choose. You enable and disable partition detection using the <partition-detection> tag. The <partition-detection> tag is a child of <deployment> and peer of <cluster>. For example:

```
<deployment>
   <cluster hostcount="4"
            sitesperhost="2"
            kfactor="1" />
   <partition-detection enabled="true">
        <snapshot prefix="netfault"/>
   </partition-detection>
</deployment>
```

If a partition is detected, the affected nodes automatically do a snapshot of the current database before shutting down. You can use the <snapshot> tag to specify the file prefix for the snapshot files. If you do not explcitly enable partition detection, the default prefix is "partition_detection".

Network partition snapshots are saved to the same directory as automated snapshots. By default, this is a subfolder of the VoltDB root directory as described in Section 3.6.2, “Configuring Paths for Runtime Features”. However, you can select a specific path using the <paths> tag set. For example, the following example sets the path for snapshots to `/opt/voltdb/snapshots/`.

```
   <partition-detection enabled="true">
       <snapshot prefix="netfaultsave"/>
   </partition-detection>
   <paths>
       <snapshots path="/opt/voltdb/snapshots/" />
   </paths>
```

When network fault protection is enabled, and a fault is detected (either due to a network fault or one or more servers failing), any viable segment of the cluster will perform the following steps:

1. Determine what nodes are missing

2. Determine if the missing nodes are also a viable self-sustained cluster. If so...

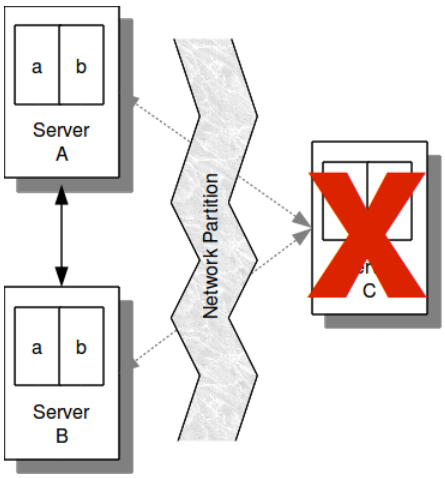
3. Determine which segment is the larger segment (that is, contains more nodes).

   - If the current segment is larger, continue to operate assuming the nodes in the smaller segment have failed.

   - If the other segment is larger, perform a snapshot of the current database content and shutdown to avoid creating two separate copies of the database.

For example, in the case shown in Figure 10.2, “Network Partition”, if a network partition separates nodes A and B from C, the larger segment (nodes A and B) will continue to run and node C will write a snapshot and shutdown (as shown in Figure 10.3, “Network Fault Protection in Action”).

Figure 10.3. Network Fault Protection in Action



If a network partition creates two viable segments of the same size (for example, if a four node cluster is split into two two-node segments), a special case is invoked where one segment is uniquely chosen to continue, based on the internal numbering of the host nodes. Thereby ensuring that only one viable segment of the partitioned database continues.

Network fault protection is a very valuable tool when running VoltDB clusters in a distributed or uncontrolled environment where network partitions may occur. The one downside is that there is no way to differentiate between network partitions and actual node failures. In the case where network fault protection is turned on and no network partition occurs but a large number of nodes actually fail, the remaining nodes may believe they are the smaller segment. In this case, the remaining nodes will shut themselves down to avoid partitioning.

For example, in the previous case shown in Figure 10.3, “Network Fault Protection in Action”, if rather than a network partition, nodes A and B fail, node C is the only node still running. Although node C is viable and could continue because the cluster was started with K-safety set to 2, if fault protection is enabled node C will shut itself down to avoid a partition.

In the worst case, if half the nodes of a cluster fail, the remaining nodes may actually shut themselves down under the special provisions for a network partition that splits a cluster into two equal parts. For example, consider the situation where a two node cluster with a k-safety value of one has network partition detection enabled. If one of the nodes fails (half the cluster), there is only a 50/50 chance the remaining node is the "blessed" node chosen to continue under these conditions. If the remaining node is *not* the chosen node, it will shut itself down to avoid a conflict, taking the database out of service in the process.

Because this situation — a 50/50 split — could result in either a network partition or a viable cluster shutting down, VoltDB recommends always using network partition detection and using clusters with an odd number of nodes. By using network partitioning, you avoid the dangers of a partition. By using an odd number of servers, you avoid even the possibility of a 50/50 split, whether caused by partitioning or node failures.

Chapter 11. Database Replication

There are times when it is useful to create multiple copies of a database. Not just a snapshot of a moment in time, but live, constantly updated copies.

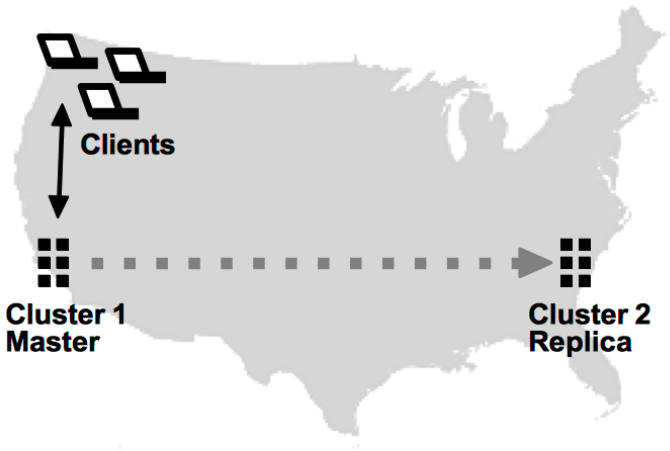
K-safety maintains redundant copies of partitions within a single VoltDB database, which helps protect the database cluster against individual node failure. Database replication also creates a copy. However, database replication creates and maintains copies in separate, often remote, databases.

VoltDB supports two forms of database replication:

- One-way (Passive)
- Two-way (Cross Datacenter)

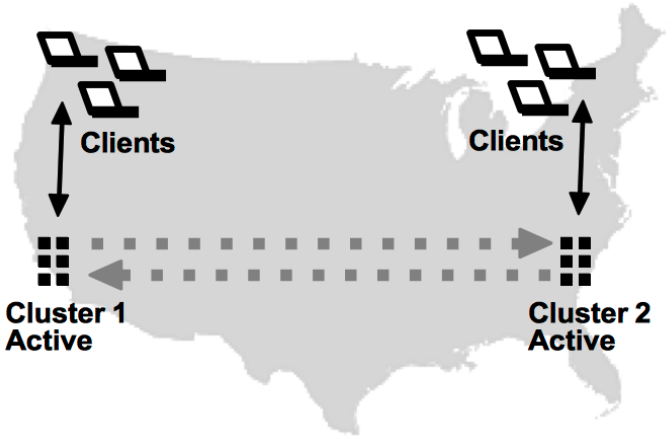
Passive replication copies the contents from one database, known as the *master* database, to the other, known as the *replica*. In passive replication, replication occurs in one direction: from the master to the replica. Clients can connect to the master database and perform all normal database operations, including INSERT, UPDATE, and DELETE statements. As shown in Figure 11.1, "Passive Database Replication" changes are copied from the master to the replica. To ensure consistency between the two databases, the replica is started as a read-only database, where only transactions replicated from the master can modify the database contents.

Figure 11.1. Passive Database Replication



Cross Datacenter Replication (XDCR), or active replication, copies changes in *both* directions. It is possible for client applications to perform read/write operations on either cluster and changes in one database are then copied and applied to the other database. Figure 11.2, "Cross Datacenter Replication" shows how XDCR can support client applications attached to each database instance.

Figure 11.2. Cross Datacenter Replication



Database replication (DR) provides two key business advantages. The first is protecting your business data against catastrophic events, such as power outages or natural disasters, which could take down an entire cluster. This is often referred to as *disaster recovery*. Because the two clusters can be in different geographic locations, both passive DR and XDCR allow one of the clusters to continue unaffected when the other becomes inoperable. Because the replica is available for read-only transactions, passive DR also allows you to offload read-only workloads, such as reporting, from the main database instance.

The second business issue that DR addresses is the need to maintain separate, active copies of the database in two separate locations. For example, XDCR allows you to maintain copies of a product inventory database at two separate warehouses, close to the applications that need the data. This feature makes it possible to support massive numbers of clients that could not be supported by a single database instance or might result in unacceptable latency when the database and the users are geographically separated. The databases can even reside on separate continents.

It is important to note, however, that database replication is not instantaneous. The transactions are committed locally, then copied to the other database. So when using XDCR to maintain two active clusters you must be careful to design your applications to avoid possible conflicts when transactions change the same record in the two databases at approximately the same time. See Section 11.3.5, “Understanding Conflict Resolution” for more information about conflict resolution.

The remainder of this chapter discusses the following topics:

- Section 11.1, “How Database Replication Works”
- Section 11.2, “Using Passive Database Replication”
- Section 11.3, “Using Cross Datacenter Replication”
- Section 11.4, “Monitoring Database Replication”

11.1. How Database Replication Works

Database replication (DR) involves duplicating the contents of selected tables between two database clusters. In passive DR, the contents are copied in one direction: from master to replica. In active or cross datacenter DR, changes are copied in both directions.

You identify which tables to replicate in the schema, by specifying the table name in a DR TABLE statement. For example, to replicate all tables in the voter sample application, you would execute three DR TABLE statements when defining the database schema:

```
DR TABLE contestants;
DR TABLE votes;
DR TABLE area_code_state;
```

11.1.1. Starting Database Replication

You enable DR by including the `<dr>` tag in the deployment files of the two databases. The `<dr>` element identifies the unique cluster ID for each database (a number between 0 and 127) and the connection source of replication as the host name or IP address of one or more nodes from the other producer database. For example:

```
<dr id="2">
  <connection source="serverA1,serverA2" />
</dr>
```

Each cluster must have a unique ID. For passive DR, only the replica needs a `<connection>` element, since replication occurs in only one direction. For active or cross datacenter replication (XDCR), both clusters must include the `<connection>` element pointing at each other.

Finally, for XDCR, you must include the DDL statement `SET DR=ACTIVE;` as part of the schema on both clusters before DR begins. For passive DR, you must start the replica database with the `--replica` flag on the command line to ensure the replica is in read-only mode. Once the clusters are configured properly and the schema of the DR tables match in both databases, replication starts.

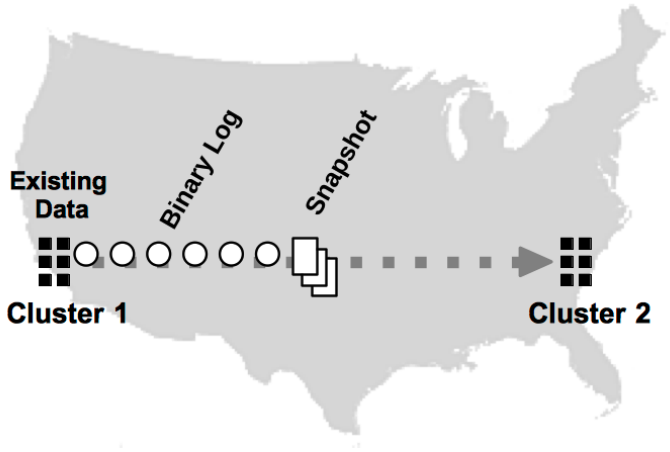
The actual replication process is performed in multiple parallel streams; each unique partition on one cluster sends a binary log of completed transactions to the other cluster. Replicating by partition has two key advantages:

- The process is faster — Because the replication process uses a binary log of the results of the transaction (rather than the transaction itself), the receiving cluster (or *consumer*) does not need to reprocess the transaction; it simply applies the results. Also, since each partition replicates autonomously, multiple streams of data are processed in parallel, significantly increasing throughout.

- The process is more durable — In a K-safe environment, if a server fails on either cluster, individual partition streams can be redirected to other nodes or a stream can wait for the server to rejoin — without interfering with the replication of the other partitions.

If data already exists in one of the clusters before database replication starts for the first time, that database sends a snapshot of the existing data to the other, as shown in Figure 11.3, “Replicating an Existing Database”. Once the snapshot is received and applied (and the two clusters are in sync), the partitions start sending binary logs of transaction results to keep the clusters synchronized.

Figure 11.3. Replicating an Existing Database



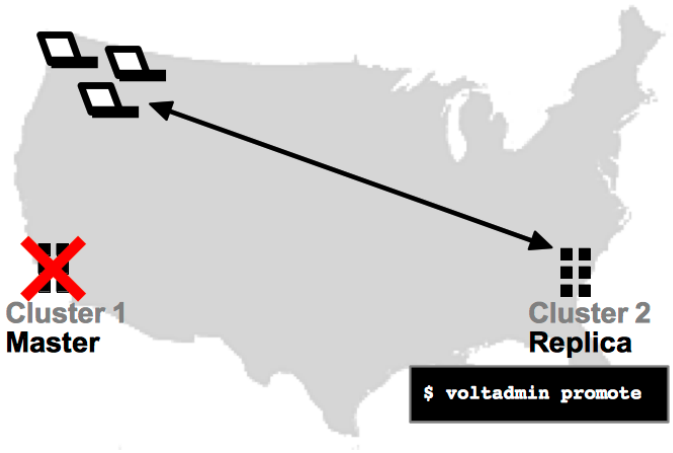
For passive DR, only the master database can have existing data before starting replication for the first time. The replica's DR tables must be empty. For XDCR, only one of the two databases can have data in the DR tables. If both clusters contain data, replication cannot start. Once DR has started, the databases can stop and recover using command logging without having to restart DR from the beginning.

11.1.2. Database Replication, Availability, and Disaster Recovery

Once replication begins, the DR process is designed to withstand normal failures and operational downtime. When using K-safety, if a node fails on either cluster, you can rejoin the node (or a replacement) using the **voltdb rejoin** command without breaking replication. Similarly, if either cluster shuts down, you can use **voltdb recover** to restart the database and restart replication where it left off. The ability to restart DR using **recover** assumes you are using command logging. Specifically, *synchronous* command logging is recommended to ensure complete durability.

If unforeseen events occur that make either database unreachable, database replication lets you replace the missing database with its copy. This process is known as *disaster recovery*. For cross datacenter replication (XDCR), you simply need to redirect your client applications to the remaining cluster. For passive DR, there is an extra step. To replace the master database with the replica, you must issue the **voltadmin promote** command on the replica to switch it from read-only mode to a fully operational database.

Figure 11.4. Promoting the Replica



See Section 11.2.5.3, “Promoting the Replica When the Master Becomes Unavailable” for more information on promoting the replica database.

11.1.3. Database Replication and Completeness

It is important to note that, unlike K-safety where multiple copies of each partition are updated simultaneously, database replication involves shipping the results of completed transactions from one database to another. Because replication happens after the fact, there is no guarantee that the contents of the two clusters are identical at any given point in time. Instead, the receiving database (or consumer) "catches up" with the sending database (or producer) after the binary logs are received and applied by each partition.

Also, because DR occurs on a per partition basis, changes to partitions may not occur in the same order on the consumer, since one partition may replicate faster than another. Normally this is not a problem because the results of all single-partitioned transactions are atomic in the binary log. Also, any changes to replicated tables are handled as atomic in the binary logs, to ensure all copies of the table on the consumer remain consistent. However, changes to partitioned tables from within a *multi-partitioned* transaction will result in separate logs that can arrive at the consumer's partitions at different times.

If the producer cluster crashes, there is no guarantee that the consumer has managed to retrieve all the logs that were queued. Therefore, it is possible that some transactions that completed on the producer are not reflected on the consumer. More importantly, if any multi-partitioned transactions update partitioned tables, you should be aware of the possibility that all of the results of that transaction did not arrive simultaneously. You may need to check the contents of the consumer to see if any such transactions were interrupted in flight.

Fortunately, using command logging and the **voltdb recover** command to restart the failed cluster, any unacknowledged transactions will be replayed from the failed cluster's disk-based DR cache, allowing the two clusters to recover and resume DR where they left off. However, if the failed cluster does not recover, you will need to decide how to proceed. You can choose to restart DR from scratch or, if you are using passive DR, you can promote the replica to replace the master.

To ensure effective recovery, the use of synchronous command logging is recommended for DR. Synchronous command logging guarantees that all transactions are recorded in the command log and no transactions are lost. If you use asynchronous command logging, there is a possibility that a binary log is applied but not captured by the command log before the cluster crashes. Then when the database recovers, the two clusters will not agree on the last acknowledged DR transaction, and DR will not be able to resume.

The decision whether to promote the replica or wait for the master to return (and hopefully recover all transactions from the command log) is not an easy one. Promoting the replica and using it to replace the original master may involve losing one or more transactions per partition. However, if the master cannot be recovered or cannot not be recovered quickly, waiting for the master to return can result in significant business loss or interruption.

Your own business requirements and the specific situation that caused the outage will determine which choice to make — whether to wait for the failed cluster to recover or to continue operations on the remaining cluster only. The important point is that database replication makes the choice possible and significantly eases the dangers of unforeseen events.

11.2. Using Passive Database Replication

The following sections provide step-by-step instructions for setting up and running passive replication between two VoltDB clusters. The steps include:

1. Specifying what tables to replicate in the schema
2. Configuring the master and replica clusters for DR
3. Starting the databases
4. Loading the schema

The remaining sections discuss other aspects of managing passive DR, including:

- Stopping database replication
- Promoting the replica database
- Using the replica for read-only transactions

11.2.1. Specifying the DR Tables in the Schema

First, you must identify which tables you wish to copy from the master to the replica. Only the selected tables are copied. You identify the tables in both the master and the replica database schema with the DR TABLE statement, For example, the following statements identify two tables to be replicated, the *Customers* and *Orders* tables:

```
CREATE TABLE customers (
  customerID INTEGER NOT NULL,
  firstname VARCHAR(128),
  lastname VARCHAR(128)
);
CREATE TABLE orders (
  orderID INTEGER NOT NULL,
  customerID INTEGER NOT NULL,
  placed TIMESTAMP
);
DR TABLE customers;
DR TABLE orders;
```

You can identify any regular table, whether partitioned or not, as a DR table, as long as the table is empty. That is, the table must have no data in it when you issue the DR TABLE statement.

The important point to remember is that the schema for both databases must contain matching table definitions for any tables identified as DR tables, including the associated DR TABLE declarations. Although

it is easiest to have the master and replica databases use the exact same schema, that is not necessary. The replica can have a subset or superset of the tables in the master, as long as it contains matching definitions for all of the DR tables. The replica schema can even contain additional objects not in the master schema, such as additional views. Which can be useful when using the replica for read-only or reporting workloads, just as long as the DR tables match.

11.2.2. Configuring the Clusters

The next step is to properly configure the master and replica clusters. The two database clusters can have different physical configurations (that is, different numbers of nodes, different sites per host, or a different K factor). Identical cluster configurations guarantee the most efficient replication, because the replica does not need to repartition the incoming binary logs. Differing configurations, on the other hand, may incrementally increase the time needed to apply the binary logs.

Before you start the databases, you must configure DR in the deployment file for both clusters. You enable DR in the deployment file using the `<dr>` element, including a unique cluster ID for each database cluster. The ID is a number between 0 and 127 which VoltDB uses to uniquely identify each cluster as part of the DR process. For example, you could assign ID=1 for the master cluster and ID=2 for the replica. On the replica, you must also include a `<connection>` sub-element that points to the master database. For example:

Master Cluster
```
<dr id="1" />
```
Replica Cluster
```
<dr id="2">
   <connection source="MasterSvrA,MasterSvrB" />
</dr>
```

11.2.3. Starting the Clusters

The next step is to start the databases. You start the master database as normal, specifying the DR-enabled deployment file. If you create a new database, you can then load the schema, including the necessary DR TABLE statements. Or you can recover a previous database instance if desired. Once the master database starts, it is ready and can interact with client applications.

For the replica database, when you first start DR, you must create a new database using the **voltdb create** command. You must also use the `--replica` flag and specify your customized deployment file. For example:

```
$ voltdb create --replica --deployment=dr-deploy.xml
```

When you specify the `--replica` argument, the database is marked as read-only. You can execute DDL statements to load the database schema, but you cannot perform any data manipulation queries such as INSERT, UPDATE, or DELETE.

The `source` attribute of the `<connection>` tag in the deployment file identifies the hostname or IP address (and optionally port number) of one or more servers in the master cluster. You can specify multiple servers so that DR can start even if one of the listed servers on the master cluster is currently down.

It is usually convenient to specify the connection information when starting the database. But this property can be changed after the database starts, in case you do not know the address of the master cluster nodes before starting. (Note, however, that the cluster ID *cannot* be changed once the database starts.)

11.2.4. Loading the Schema and Starting Replication

As soon as the replica database starts with DR enabled, it will attempt to contact the master database to start replication. The replica will issue warnings that the schema does not match, since the replica does

not have any schema defined yet. This is normal. The replica will periodically contact the master until the schema for DR objects on the two databases match. This gives you time to load a matching schema.

As soon as the replica database has started, you can load the appropriate schema. Loading the same schema as the master database is the easiest and recommended approach. The key point is that once a matching schema is loaded, replication will begin automatically.

When replication starts, the following actions occur:

1. The replica and master databases verify that the DR tables match on the two clusters.
2. If data already exists in the DR tables on the master, the master sends a snapshot of the current contents to the replica where it is restored into the appropriate tables.
3. Once the snapshot, if any, is restored, the master starts sending binary logs of changes to the DR tables to the replica.

If any errors occur during the snapshot transmission, replication stops and must be restarted from the beginning. However, once the third step is reached, replication proceeds independently for each unique partition and, in a K safe environment, the DR process becomes durable across node failures and rejoins and other non-fatal events.

If either the master or the replica database crashes and needs to restart, it is possible to restart DR where it left off, assuming the databases are using command logging for recovery. If the master fails, you can perform a **voltdb recover** action to restart the master database. The replica will wait for the master to recover. The master will then replay any DR logs on disk and resume DR where it left off.

If the replica fails, the master will queue the DR logs to disk waiting for the replica to return. If you perform a **voltdb recover** action, including the **--replica** flag, on the replica cluster, the replica will perform the following actions:

1. Restart the replica database, restoring both the schema and the data, and placing the database in read-only mode.
2. Contact the master cluster and attempt to re-establish DR.
3. If both clusters agree on where (that is, what transaction), DR was interrupted, DR will resume from that point, starting with the DR logs that the master database has queued in the interim.

Note that you must use the **--replica** flag when recovering the replica database if you want to resume DR where it left off. For example:

```
$ voltdb recover --replica --deployment=dr-deploy.xml
```

If you do not include the **--replica** flag, the database will resume as a normal, read/write database and not attempt to contact the master database. Also, if the clusters do not agree on where DR stopped during step #3, the replica database will generate an error and stop replication. For example, if you recover from an asynchronous command log where the last few DR logs were ACKed to the master but not written to the command log, the master and the replica will be in different states when the replica recovers.

If this occurs, you must restart DR from the beginning, by creating a new, empty replica database and reloading a compatible schema. Similarly, if you are not using command logging, you cannot recover the replica database and must start DR from scratch.

11.2.5. Stopping Replication

If, for any reason, you wish to stop replication of a database, there are two ways to do this: you can stop sending data from the master or you can "promote" the replica to stop it from receiving data. Since the

individual partitions are replicating data independently, if possible you want to make sure all pending transfers are completed before turning off replication.

So, under the best circumstances, you should perform the following steps to stop replication:

1. Stop write transactions on the master database by putting it in admin mode using the **voltadmin pause** command.
2. Wait for all pending DR log transfers to be completed.
3. Reset DR on the master cluster using the **voltadmin dr reset** command.
4. Depending on your goals, either shut down the replica or promote it to a fully-functional database as described in Section 11.2.5.3, “Promoting the Replica When the Master Becomes Unavailable”.

11.2.5.1. Stopping Replication on the Master if the Replica Becomes Unavailable

If the replica becomes unavailable *and is not going to be recovered or restarted*, you should consider stopping DR on the master database, to avoid consuming unnecessary disk space.

The DR process is resilient against network glitches and node or cluster failures. This durability is achieved by the master database continually queueing DR logs in memory and — if too much memory is required — to disk while it waits for the replica to ACK the last message. This way, when the network interruption or other delay is cleared, the DR process can pick up where it left off. However, the master database has no way to distinguish a temporary network failure from an actual stoppage of DR on the replica.

Therefore, if the replica stops unexpectedly, it is a good idea to restart the replica and re-initiate DR as soon as convenient. Or, if you are not going to restart DR, you should reset DR on the master to cancel the queuing of DR logs and to delete any pending logs. To reset the DR process on the master database, use the **voltadmin dr reset** command. For example:

```
$ voltadmin dr reset --host=serverA
```

Of course, if you do intend to recover and restart DR on the replica, you do *not* want to reset DR on the master. Resetting DR on the master will delete any queued DR logs and make restarting replication where it left off impossible and force you to start DR over from the beginning.

11.2.5.2. Database Replication and Disaster Recovery

If unforeseen events occur that make the master database unreachable, database replication lets you replace the master with the replica and restore normal business operations with as little downtime as possible. You switch the replica from read-only to a fully functional database by *promoting* it. To do this, perform the following steps:

1. Make sure the master is actually unreachable, because you do not want two live copies of the same database. If it is reachable but not functioning properly, be sure to pause or shut down the master database.
2. Promote the replica to a read/write mode using the **voltadmin promote** command.
3. Redirect the client applications to the newly promoted database.

Figure 11.4, “Promoting the Replica” illustrates how database replication reduces the risk of major disasters by allowing the replica to replace the master if the master becomes unavailable.

Once the master is offline and the replica is promoted, the data is no longer being replicated. As soon as normal business operations have been re-established, it is a good idea to also re-establish replication. This can be done using any of the following options:

- If the original master database hardware can be restarted, take a snapshot of the current database (that is, the original replica), restore the snapshot on the original master and redirect client traffic back to the original. Replication can then be restarted using the original configuration.
- An alternative, if the original database hardware can be restarted but you do not want to (or need to) redirect the clients away from the current database, is to use the original master hardware to create a replica of the newly promoted cluster — essentially switching the roles of the master and replica databases — as described in Section 11.2.5.4, “Reversing the Master/Replica Roles”.
- If the original master hardware cannot be recovered effectively, create a new database cluster in a third location to use as a replica of the current database.

11.2.5.3. Promoting the Replica When the Master Becomes Unavailable

If the master database becomes unreachable for whatever reason (such as catastrophic system or network failure) it may not be possible to turn off DR in an orderly fashion. In this case, you may choose to “turn on” the replica as a fully active (writable) database to replace the master. To do this, you use the **voltadmin promote** command. When you promote the replica database, it exits read-only mode and becomes a fully operational VoltDB database. For example, the following Linux shell command uses **voltadmin** to promote the replica node serverB:

```
$ voltadmin promote --host=serverB
```

11.2.5.4. Reversing the Master/Replica Roles

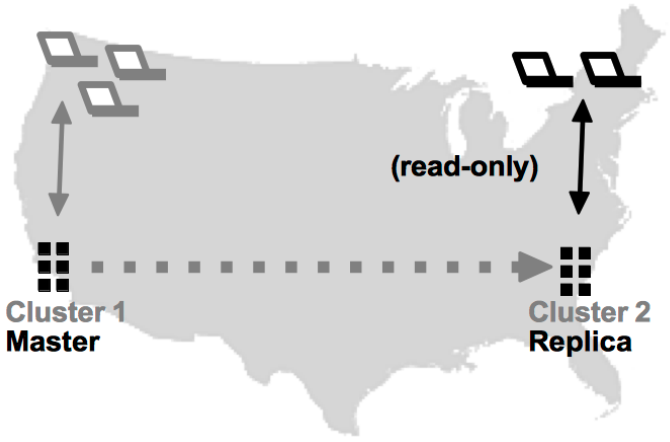
If you do promote the replica and start using it as the primary database, you will likely want to establish a new replica as soon as possible to return to the original production configuration and level of durability. You can do this by creating a new replica cluster and connecting to the promoted database as described in Section 11.2.3, “Starting the Clusters”. Or, if the master database can be restarted, you can reuse that cluster as the new replica, by modifying the deployment file to include the necessary `<connection>` element and starting the database cluster with **voltdb create --replica**.

11.2.6. Database Replication and Read-only Clients

While database replication is occurring, the only changes to the replica database come from the binary logs. Client applications can connect to the replica and use it for read-only transactions, including read-only ad hoc queries and system procedures. However, any attempt to perform a write transaction from a client application returns an error.

There will always be some delay between a transaction completing on the master and its results being applied on the replica. However, for read operations that do not require real-time accuracy (such as reporting), the replica can provide a useful source for offloading certain less-frequent, read-only transactions from the master.

Figure 11.5. Read-Only Access to the Replica



11.3. Using Cross Datacenter Replication

The following sections provide step-by-step instructions for setting up and running cross datacenter replication (XDCR) between two VoltDB clusters. The sections describe how to:

1. Design your schema, including:
   - Enabling XDCR, or active DR
   - Identifying the DR tables
2. Configure the database clusters, including:
   - Choosing unique cluster IDs
   - Identifying the DR connections
3. Start the databases
4. Load the schema and start replication

Later sections discuss other aspects of managing XDCR, including:

- Stopping database replication
- Resolving conflicts

Important

XDCR is a separately licensed feature. If your current VoltDB license does not include a key for XDCR you will not be able to complete the tasks described in this section. See your VoltDB sales representative for more information on licensing XDCR.

11.3.1. Designing Your Schema for Active Replication

If you plan to use XDCR or active DR, you need to design your database schema appropriately. Specifically you must:

- Enable the use of XDCR

- Identify the tables that will be replicated

11.3.1.1. Enabling Active Replication in the Schema

To manage XDCR, VoltDB stores a small amount (8 bytes) of extra metadata with every row of data that is shared. To allocate this additional space, you must tell VoltDB you will be using active DR. You do this with the SET DR=ACTIVE statement in your schema:

```
SET DR=ACTIVE;
```

You must execute the SET DR=ACTIVE before there is any data in the tables that will be replicated. Consequently, it is easiest to include it at the beginning of your schema. However, it can be executed at any time — even after one or more tables are declared as DR tables with the DR TABLE statement — as long as the DR tables are empty.

11.3.1.2. Identifying the DR Tables in the Schema

Next, you must identify which tables you wish to share between the two databases. Only the selected tables are copied. You identify the tables in the schema for both databases with the DR TABLE statement. For example, the following statements identify two tables to be replicated, the *Customers* and *Orders* tables:

```
SET DR=ACTIVE;
CREATE TABLE customers (
  customerID INTEGER NOT NULL,
  firstname VARCHAR(128),
  LASTNAME varchar(128)
);
CREATE TABLE orders (
  orderID INTEGER NOT NULL,
  customerID INTEGER NOT NULL,
  placed TIMESTAMP
);
DR TABLE customers;
DR TABLE orders;
```

You can identify any regular table, whether partitioned or not, as a DR table, as long as the table is empty. That is, the table must have no data in it when you issue the DR TABLE statement. The important point to remember is that the schema definitions for the tables participating in DR, including the DR TABLE statements, must be identical on the two clusters.

11.3.2. Starting the Database Clusters

The next step is to start the databases. The two database clusters must have the same physical configuration — that is, the same number of nodes, sites per host, and K factor. You must also enable and configure DR in the deployment file, including:

- Choosing a unique ID for each cluster
- Identifying the DR connections

11.3.2.1. Choosing Unique IDs

You enable DR in the deployment file using the `<dr>` element and including a unique cluster ID for each database cluster.

To manage the DR process VoltDB needs to uniquely identify the clusters. You provide this unique identifier as a number between 0 and 127 when you configure the clusters. For example, if we assign ID=1 to a cluster in New York and ID=2 to another in Chicago, their respective deployment files must contain the following `<dr>` elements:

New York Cluster

```
<dr id="1" />
```

Chicago Cluster

```
<dr id="2" />
```

11.3.2.2. Identifying the DR Connections

For each database cluster, you must also the specify the source of replication in the `<connection>` subelement. You do this by pointing each cluster at the other, specifying one or more servers on the other cluster in the source attribute.

For example, say the New York cluster has nodes NYserverA, NYserverB, and NYserverC. While the Chicago cluster has CHIserverX, CHIserverY, and CHIserverZ. The deployment files for the two clusters might look like this:

New York Cluster

```
<dr id="1">
   <connection source="CHIserverX,CHIserverY" />
</dr>
```

Chicago Cluster

```
<dr id="2">
   <connection source="NYserverA,NYserverB,NYserverC" />
</dr>
```

Note that *both* clusters must have a connection defined for active replication to start. Once the deployment files have the necessary declarations, you can start the database clusters. However, it is important to note that as soon as both databases start, they will attempt to contact each other, verify that the DR table schema match, and then start the DR process.

So often the easiest method for starting the databases is to:

1. Start one cluster

2. Load the schema (including the DR table declarations) on that cluster

3. Once the first cluster is fully configured, start the second cluster and load the schema

Using this approach, DR will not start until step #3 is complete and both clusters are fully configured. An alternative approach is to start both databases leaving the source attribute of the <connection> element empty. For example:

```
<dr id="2">
   <connection source="" />
</dr>
```

You can then load the schema on both databases and perform any other preparatory work you require. Then edit the deployment files filling in the source attribute for each cluster to point at the other. Then use the **voltadmin update** command to update the deployment files on the running databases. As soon as the source attribute is defined and the schema match, the DR process will begin.

Note

Although the source attribute can be modified on a running database, the unique cluster ID *cannot* be changed after the database starts. So it is important to include the <dr> element with the unique ID in the initial deployment file when starting the databases.

11.3.3. Loading a Matching Schema and Starting Replication

As soon as the databases start with DR enabled, they will attempt to contact the cooperating database to start replication. Each cluster will issue warnings until the schema for both databases match. This is normal and gives you time to load a matching schema. The key point is that once matching schema are loaded on both databases, replication will begin automatically.

When replication starts, the following actions occur:

1. The clusters verify that they have the same physical configuration and that the DR tables match on both clusters.

2. If data already exists in the DR tables on one of the databases, that cluster sends a snapshot of the current contents to the other cluster where it is restored into the appropriate tables.

3. Once the snapshot, if any, is restored, both databases start sending binary logs of changes to any DR tables to the other cluster.

If any errors occur during the snapshot transmission, replication stops and must be restarted from the beginning. However, once the third step is reached, replication proceeds independently for each unique partition and, in a K safe environment, the DR process becomes durable across node failures and rejoins as well as cluster shutdowns and recoveries.

11.3.4. Stopping Replication

If, for any reason, you need to break replication between the two databases, all you need to do is issue the **voltadmin dr reset** command to either cluster. For example, if one cluster goes down and will not be brought back online for an extended period, you will want to issue a **voltadmin dr reset** command on the remaining cluster to tell it to stop queuing binary logs. If not, the logs will be saved on disk, waiting for the other cluster to recover, until you run out of disk space.

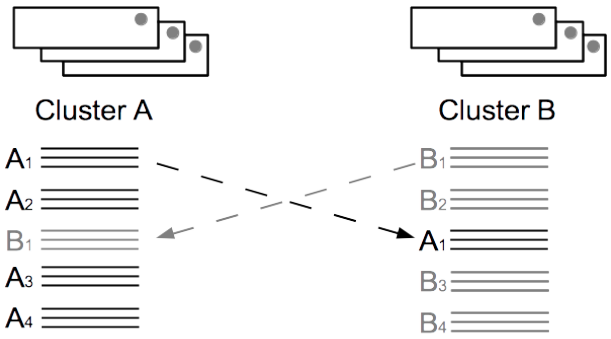
For similar reasons, if you break replication while both databases are running, you should issue the **voltadmin dr reset** command to *both* clusters. Although using **dr reset** on one cluster is sufficient to break the DR process, the cluster that does not receive the reset command will continue to queue binary logs until it, too, is reset.

11.3.5. Understanding Conflict Resolution

One aspect of database replication that is unique to cross datacenter replication (XDCR) is the need to prepare for and manage conflicts between the two databases. Conflict resolution is not an issue for passive replication since changes travel in only one direction. However, with XDCR it is possible for changes to be made to the same data at approximately the same time on the two databases. Those changes are then sent to the other database, resulting in possible inconsistencies or invalid transactions.

For example, say clusters A and B are processing transactions as shown in Figure 11.6, "Transaction Order and Conflict Resolution". Cluster A executes a transaction that modifies a specific record and this transaction is included in the binary log A_1. By the time cluster B receives the binary log and processes A_1, cluster B has already processed its own transactions B_1 and B_2. Those transactions may have modified the same record as the transaction in A_1, or another record that would conflict with the change in A_1, such as a matching unique index entry.

Figure 11.6. Transaction Order and Conflict Resolution



Under these conditions, cluster B cannot simply apply the changes in A_1 because doing so could violate the uniqueness constraints of the schema and, more importantly, is likely to result in the content of the two database clusters diverging. Instead, cluster B must decide which change takes priority. That is, what resolution to the conflict is most likely to produce meaningful results or match the *intent* of the business application. This decision making process is called *conflict resolution*.

No matter what the resolution, it is important that the database administrators are notified of the conflict, why it occurred, and what action was taken. The following sections explain:

- How to avoid conflicts
- How VoltDB resolves conflicts when they do occur
- What types of conflicts can occur
- How those conflicts are reported

11.3.5.1. Designing Your Application to Avoid Conflicts

VoltDB uses well-defined rules for resolving conflicts. However, the best protection against conflicts and the problems they can cause is to design your application to avoid conflicts in the first place. There are at least two things you can do in your client applications to avoid conflicts:

- **Use Primary Keys**

  It is best, wherever possible, to define a primary key for all DR tables. The primary key index greatly improves performance for finding the matching row to apply the change on the consumer cluster. It is also required if you want conflicts to be resolved using the standard rules described in the following section. Any conflicting action *without* a primary key is rejected.

- **Apply related transactions to the same cluster**

  Another tactic for avoiding conflicts is to make sure any autonomous set of transactions affecting a set of rows are all applied on the same cluster. For example, ensuring that all transactions for a single user session, or associated with a particular purchase order, are directed to the same cluster.

11.3.5.2. How Conflicts are Resolved

Even with the best application design possible, errors in program logic or operation may occur that result in conflicting records being written to the two databases. When a conflict does occur, VoltDB follows specific rules for resolving the issue. The conflict resolution rules are:

- Conflicts are resolved on a per action basis. That is, resolution rules apply to the individual INSERT, UPDATE, or DELETE operation on a specific tuple. Resolutions are not applied to the transaction as a whole.
- The resolution is that the incoming action is accepted (that is, applied to the receiving database) or rejected.
- Only actions involving a table with a primary key can be accepted, all other conflicting actions are rejected.
- Accepted actions are applied as a whole — the entire record is changed to match the result on the producer cluster. That means for UPDATE actions, all columns are written not just the columns specified in the SQL statement.
- For tables with primary keys, the rules for which transaction wins are, in order:

  1. DELETE transactions always win
  2. If neither action is a DELETE, the last transaction (based on the timestamp) wins

Let's look at a simple example to see how these rules work. Assume that the database stores user records, using a numeric user ID as the primary key and containing columns for the user's name and password. A user logs on simultaneously in two locations and performs two separate updates: one on cluster A changing their name and one on cluster B changing the password. These updates are almost simultaneous. However, cluster A timestamps its transaction as occurring at 10:15.00.003 and cluster B timestamps its transaction at 10:15.00.001.

The binary logs from the two transactions include the type of action, the contents of the record before and after the change, and the timestamps — both of the last previous transaction and the timestamp of the new transaction. (Note that the timestamp includes both the time and the cluster ID where the transaction occurred.) So the two binary logs might look like the following.

Binary Log A_1:

```
Action: UPDATE
Current Timestamp:  1, 10:15.00.003
Previous Timestamp: 1, 06:30.00.000
```

Before	After
UserID: 12345 Name: Joe Smith Password: abalone	UserID: 12345 Name: **Joseph Smith** Password: abalone

Binary Log B_1:

```
Action: UPDATE
Current Timestamp:  2, 10:15.00.001
Previous Timestamp: 1, 06:30.00.000
```

Before	After
UserID: 12345 Name: Joe Smith Password: abalone	UserID: 12345 Name: Joe Smith Password: **flounder**

When the binary log A_1 arrives at cluster B, the DR process performs the following steps:

1. Uses the primary key (12345) to look up the current record in the database.

2. Compares the current timestamp in the database with the previous timestamp in the binary log.

3. Because the transaction in B_1 has already been applied on cluster B, the time stamps do not match. A conflict is recognized.

4. A primary key exists, so cluster B attempts to resolve the conflict by comparing the new timestamp, 10:15.00.003, to the current timestamp, 10:15.00.001.

5. Because the new timestamp is the later of the two, the new transaction "wins" and the change is applied to the database.

6. Finally, the conflict and resolution is logged. (See Section 11.3.5.4, “Reporting Conflicts” for more information about how conflicts are reported.)

Note that when the UPDATE from A_1 is applied, the change to the password in B_1 is overwritten and the password is reset to "abalone". Which at first looks like a problem. However, when the binary log B_1 arrives at cluster A, the same steps are followed. But when cluster A reaches steps #4 and 5, it finds that the new timestamp from B_1 is older than the current timestamp, and so the action is rejected and the record is left unchanged. As a result both databases end up with the same value for the record. Essentially, the password change is dropped.

If the transaction on cluster B had been to delete the user record rather than change the password, then the outcome would be different, but still consistent. In that case, when binary log A_1 reaches cluster B, it would not be able to find the matching record in step #1. This is recognized as a DELETE action having occurred. Since DELETE always wins, the incoming UPDATE is rejected. Similarly, when binary log B_1 reaches cluster A, the previous timestamps do not match but, even though the incoming action in B_1 has an older timestamp than the UPDATE action in A_1, B_1 "wins" because it is a delete action and the record is deleted from cluster A. Again, the result is consistent across the two databases.

The real problem with conflicts is when there is no primary key on the database table. Primary keys uniquely identify a record. Without a primary key, there is no way for VoltDB to tell, even if there are one or more unique indexes on the table, whether two records are the same record modified or two different records with the same unique key values.

As a result, if there is a conflict between two transactions without a primary key, VoltDB has no way to resolve the conflict and simply rejects the incoming action. Going back to our example, if the user table had a unique index on the user ID rather than a primary key, and both cluster A and cluster B update the user record at approximately the same time, when binary log A_1 arrives at cluster B, it would look for the record based on all columns in the record and fail to find a match.

However, when it attempts to insert the record, it will encounter a constraint violation on the unique index. Again, since there is no primary key, VoltDB cannot resolve the conflict and rejects the incoming action, leaving the record with the changed password. On cluster A, the same process occurs and the password change in B_1 gets rejected, leaving cluster A with a changed name column and database B with a changed password column — the databases diverge.

11.3.5.3. What Types of Conflict Can Occur

The preceding section uses a simple case of conflicting UPDATE transactions to illustrate the steps involved in conflict resolution. However, there are several different types of conflict that can occur. First, there are three possible actions that the binary log can contain: INSERT, UPDATE, or DELETE. There are also three types of conflicts that can be generated:

- **Missing row** — The affected row is missing from the consumer database.
- **Timestamp mismatch** — The affected row exists in the consumer database, but has a different timestamp than expected (in other words, it has been modified).
- **Constraint violation** — Applying the incoming action would result in one or more constraint violations on unique indexes.

A missing row means that the binary log contains an UPDATE or DELETE action, but the affected row cannot be found in the consumer database. (A missing row conflict cannot occur for INSERT actions, since INSERT assumes no such row exists.) In the case of a missing row conflict, VoltDB assumes a DELETE action has removed the affected row. Since the rule is that DELETE wins, this means the incoming action is rejected.

Note that if the table does not have a primary key, the assumption that a DELETE action removed the row is not guaranteed to be true, since it is possible an UPDATE changed the row. Without a primary key, there is no way for the DR process to find the matching row when some columns may have changed, so it assumes it was deleted. As a result, an UPDATE could occur on one cluster and a DELETE on the other. This is why assigning primary keys is recommended for DR tables when using XDCR.

If the matching primary key *is* found, it is still possible that the contents of the row have been changed. In which case, the timestamps will not match and a timestamp mismatch conflict occurs. Again, this can happen for UPDATE and DELETE actions where an existing row is being modified. If the incoming action is a DELETE, it takes precedence and the row is deleted. If not, if the incoming action has the later of the two timestamps, it is accepted. If the existing record has the later timestamp, the incoming action is rejected.

Finally, whether the timestamps match or not, with an INSERT or UPDATE action, it is possible that applying the action would violate one of more unique index constraints. This can happen because another row has been updated with matching values for the unique index or another record has been inserted with similar values. Whatever the cause, VoltDB cannot apply the incoming action so it is rejected. Note that for a single action there can be more than one unique index that applies to the table, so there can be multiple constraint violations as well as a possible incorrect timestamp. When a conflict occurs, all conflicts associated with the action are included in the conflict log.

To summarize, the following chart shows the conflicts that can occur with each type of action and the result for tables with a primary key.

Action	Possible Conflict	Result for Tables with Primary Key
INSERT	Constraint violation	Rejected
UPDATE	Missing row Timestamp mismatch Constraint violation	Rejected Last transaction wins Rejected
DELETE	Missing row Timestamp mismatch	Accepted (no op) Accepted

11.3.5.4. Reporting Conflicts

VoltDB makes a record of every conflict that occurs when processing the DR binary logs. These conflict logs include:

- The intended action
- The type of conflict
- The timestamp and contents of the row before and after the action from the binary log
- The timestamp and contents of the row(s) in the consumer database that caused the conflict

By default, these logs are written as comma-separated value (CSV) files on the cluster where the conflicts occur. These files are usually written to a subfolder of the voldbroot directory (`voltdbroot/xdcr_conflicts`) using the file prefix `LOG`. However, you can configure the logs to be written to different destinations or locations using the VoltDB export deployment settings.

The DR process writes the conflicts as export data to the export stream VOLTDB_XDCR_CONFLICTS. You do not need to explicitly configure export — the DR process automatically declares the necessary export tables, establishes a default export configuration for the file connector, and enables the export stream. However, if you want the data to be sent to a different location or using a different export connector, you can do this by configuring the export stream yourself.

For example, if you want to export the XDCR conflicts to a Kafka stream where they can be used for automatic notifications, you can change the export configuration in the deployment file. The following deployment file code writes the conflict logs to the Kafka topic sysops on the broker kafkabroker.mycompany.com:

```
<export>
   <configuration enabled="true" type="kafka"
                  stream="VOLTDB_XDCR_CONFLICTS">
      <property name="broker">kafkabroker.mycompany.com</property>
      <property name="topic">sysops</property>
   </configuration>
</export>
```

Each action in the binary log can generate one or more conflicts. When this occurs, VoltDB logs the conflict(s) as multiple rows in the conflict report. Each row is identified by the type of action (INSERT, UPDATE, DELETE) as well as the type of information the row contains:

- EXISTING (EXT) — The timestamp and contents of an existing row in the consumer database that caused a conflict. There can be multiple existing row logs, if there are multiple conflicts.
- EXPECTED (EXP) — The timestamp and contents of the row that is expected before the action is applied (from the binary log).
- NEW (NEW) — The new timestamp and contents for the row once the action is applied (from the binary log).

For an INSERT action, there is no EXPECTED row and for a DELETE action there is no NEW row. The order of the rows in the report is as follows:

1. The EXISTING row, if there is a timestamp mismatch
2. The EXPECTED row, if there is a timestamp mismatch

3. One or more EXISTING rows, if there are any constraint violations

4. The NEW row, for all actions but DELETE

Table 11.1, "Structure of the XDCR Conflict Logs" describes the structure and content of the conflict log records in the export stream.

Table 11.1. Structure of the XDCR Conflict Logs

Column Name	Datatype	Description
ROW_TYPE	3 Byte string	The type of row, specified as: `EXT` — existing `EXP` — expected `NEW` — new
ACTION_TYPE	1 Byte string	The type of action, specified as: `I` — insert `U` — update `D` — delete
CONFLICT_TYPE	4 Byte string	The type of conflict, specified as: `MISS` — missing row `MSMT` — timestamp mismatch `CNST` — constraint violation `NONE` — no violation[a]
CONFLICTS_ON _PRIMARY_KEY	TINYINT	Whether a constraint violation is associated with the primary key. 1 for true and 0 for false.
DECISION	1 Byte string	How the conflict was resolved, specified as: A — the incoming action is accepted R — the incoming action is rejected
CLUSTER_ID	TINYINT	The DR cluster ID of the cluster that last modified the row
TIMESTAMP	BIGINT	The timestamp of the row.
DIVERGENCE	1 Byte string	Whether the resulting action could cause the two cluster to diverge, specified as: `C` — the clusters are consistent `D` — the cluster may have diverged
TABLE_NAME	String	The name of the table.
TUPLE	JSON-encoded string	The contents of the row, as a JSON-encoded string.

[a]Update operations are executed as two separate statements: a delete and an insert, where only one of the two statements might result in a violation. For example, the delete may trigger a missing row violation but the insert not generate a violation. In which case the EXT row of the conflict log reports the MISS conflict and the NEW row reports NONE.

11.4. Monitoring Database Replication

Database replication runs silently in the background. To ensure replication is proceeding effectively, VoltDB provides statistics on both the producer and consumer clusters that help you understand the current state of the DR process. Specifically, the statistics can tell you:

- The amount of DR data waiting to be sent from the producer
- The timestamp and unique ID of the last transaction received by the consumer
- Whether any partitions are "falling behind" in processing DR data

This information is available from the @Statistics system procedure using the "DR" selector on the producer database and "DRCONSUMER" on the consumer. For one-way (passive) DR, the master database acts as the producer and the replica acts as the consumer. For two-way (cross datacenter) replication, both clusters act as both producer and consumer and can provide statistics on both roles:

- On the producer database, the @Statistics DR procedure includes columns for the transaction ID and timestamp of the last queued transaction and for the last transaction ACKed by the consumer. The difference between these two events can tell you the approximate latency between the two databases.
- On the consumer database, the @Statistics DRCONSUMER procedure includes statistics, on a per partition basis, showing whether it has an identified "host" server from the producer cluster "covering" it, or in other words, providing it DR logs. The system procedure results also include columns listing the ID and timestamp of the last received transaction. If a consumer partition is not covered, it means it has lost contact with the server on the producer database that was providing it logs (possibly due to a node failure). It is possible for the partition to recover, once the covering server rejoins. However, the difference between the last received timestamp of that partition and the other partitions may give you an indication of how long the interruption has persisted and how far behind that partition may be.

Chapter 12. Security

Security is an important feature of any application. By default, VoltDB does not perform any security checks when a client application opens a connection to the database or invokes a stored procedure. This is convenient when developing and distributing an application on a private network.

However, on public or semi-private networks, it is important to make sure only known client applications are interacting with the database. VoltDB lets you control access to the database through settings in the schema and deployment files. The following sections explain how to enable and configure security for your VoltDB application.

12.1. How Security Works in VoltDB

When an application creates a connection to a VoltDB database (using ClientFactory.clientCreate), it passes a username and password as part of the client configuration. These parameters identify the client to the database and are used for authenticating access.

At runtime, if security is enabled, the username and password passed in by the client application are validated by the server against the users defined in the deployment file. If the client application passes in a valid username and password pair, the connection is established. When the application calls a stored procedure, permissions are checked again. If the schema identifies the user as being assigned a role having access to that stored procedure, the procedure is executed. If not, an error is returned to the calling application.

Note

VoltDB uses hashing rather than encryption when passing the username and password between the client and the server. The Java and C++ clients use SHA-2 hashing while the older clients currently use SHA-1. The passwords are also hashed within the database. For an encrypted solution, you can consider implementing Kerberos security, described in Section 12.7, “Integrating Kerberos Security with VoltDB”.

There are three steps to enabling security for a VoltDB application:

1. Add the `<security enabled="true"/>` tag to the deployment file to turn on authentication and authorization.

2. Define the users and roles you need to authenticate.

3. Define which roles have access to each stored procedure.

The following sections describe each step of this process, plus how to enable access to system procedures and ad hoc queries.

12.2. Enabling Authentication and Authorization

By default VoltDB does not perform authentication and client applications have full access to the database. To enable authentication, add the <security> tag to the deployment file:

```
<deployment>
   <security enabled="true"/>
       . . .
</deployment>
```

12.3. Defining Users and Roles

The key to security for VoltDB applications is the users and roles defined in the schema and deployment files. You define users in the deployment file and roles in the schema.

This split is deliberate because it allows you to define the overall security structure globally in the schema, assigning permissions to generic roles (such as operator, dbuser, apps, and so on). You then define specific users and assign them to the generic roles as part of the deployment. This way you can create one configuration (including cluster information and users) for development and testing, then move the database to a different configuration and a different set of users for production by changing only one file: the deployment file.

You define users within the <users> ... </users> tag set in the deployment file. The syntax for defining users is as follows.

```
<deployment>
   <users>
      <user name="user-name"
            password="password-string"
            roles="role-name[,...]"  />
        [ ... ]
   </users>
   ...
</deployment>
```

Include a <user> tag for every username/password pair you want to define.

Then within the schema you define the roles the users can belong to. You define roles with the CREATE ROLE statement.

```
CREATE ROLE role-name;
```

You specify which roles a user belongs to as part of the user definition in the deployment file using the roles attribute to the <user> tag. For example, the following code defines three users, assigning operator and developer the ops role and developer and clientapp the dbuser role. When a user is assigned more than one role, you specify the role names as a comma-delimited list.

```
<deployment>
   <users>
      <user name="operator" password="mech" roles="ops" />
      <user name="developer" password="tech" roles="ops,dbuser" />
      <user name="clientapp" password="xyzzy" roles="dbuser" />
   </users>

</deployment>
```

Two important notes concerning the assignment of users and roles:

- Users must be assigned at least one role, or else they have no permissions. (Permissions are assigned by role.)
- There must be a corresponding role defined in the schema for any roles listed in the deployment file.

12.4. Assigning Access to Stored Procedures

Once you define the users and roles you need, you assign them access to individual stored procedures using the ALLOW clause of the CREATE PROCEDURE statement in the schema. In the following example, users assigned the roles dbuser and ops are permitted access to both the MyProc1 and MyProc2 procedures. Only users assigned the ops role have access to the MyProc3 procedure.

```
CREATE PROCEDURE ALLOW dbuser,ops FROM CLASS MyProc1;
CREATE PROCEDURE ALLOW dbuser,ops FROM CLASS MyProc2;
CREATE PROCEDURE ALLOW ops FROM CLASS MyProc3;
```

Usually, when security is enabled, you must specify access rights for each stored procedure. If a procedure declaration does not include an ALLOW clause, no access is allowed. In other words, calling applications will not be able to invoke that procedure.

12.5. Assigning Access by Function (System Procedures, SQL Queries, and Default Procedures)

It is not always convenient to assign permissions one at a time. You might want a special role for access to all user-defined stored procedures. Also, there are special capabilities available within VoltDB that are not called out individually in the schema so cannot be assigned using the CREATE PROCEDURE statement.

For these special cases VoltDB provides named permissions that you can use to assign functions as a group. For example, the ALLPROC permission grants a role access to all user-defined stored procedures so the role does not need to be granted access to each procedure individually.

Several of the special function permissions have two versions: a full access permission and a read-only permission. So, for example, DEFAULTPROC assigns access to all default procedures while DEFAULTPROCREAD allows access to only the read-only default procedures; that is, the *TABLE*.select procedures. Similarly, the SQL permission allows the user to execute both read and write SQL queries interactively while SQLREAD only allows read-only (SELECT) queries to be executed.

One additional functional permission is access to the read-only system procedures, such as @Statistics and @SystemInformation. This permission is special in that it does not have a name and does not need to be assigned; all authenticated users are automatically assigned read-only access to these system procedures.

Table 12.1, “Named Security Permissions” describes the named functional permissions.

Table 12.1. Named Security Permissions

Permission	Description	Inherits
DEFAULTPROCREAD	Access to read-only default procedures (*TABLE*.select)	
DEFAULTPROC	Access to all default procedures (*TABLE*.select, *TABLE*.insert, *TABLE*.delete, *TABLE*.update, and *TABLE*.upsert)	DEFAULTPROCREAD
SQLREAD	Access to read-only ad hoc SQL queries (SELECT)	DEFAULTPROCREAD
SQL	Access to all ad hoc SQL queries and default procedures	SQLREAD, DEFAULTPROC
ALLPROC	Access to all user-defined stored procedures	

Permission	Description	Inherits
ADMIN	Full access to all system procedures, all user-defined procedures, as well as default procedures, ad hoc SQL, and DDL statements.	ALLPROC, DEFAULTPROC, SQL
Note: For backwards compatibility, the special permissions ADHOC and SYSPROC are still recognized. They are interpreted as synonyms for SQL and ADMIN, respectively.		

In the CREATE ROLE statement you enable access to these functions by including the permission name in the WITH clause. (The default, if security is enabled and the keyword is not specified, is that the role is not allowed access to the corresponding function.)

Note that the permissions are additive. So if a user is assigned one role that allows access to SQLREAD but not DEFAULTPROC, but that user is also assigned another role that allows DEFAULTPROC, the user has both permissions.

The following example assigns full access to members of the ops role, access to interactive SQL queries (and default procedures by inheritance) and all user-defined procedures to members of the developer role, and no special access beyond read-only system procedures to members of the apps role.

```
CREATE ROLE ops WITH admin;
CREATE ROLE developer WITH sql, allproc;
CREATE ROLE apps;
```

12.6. Using Default Roles

To simplify the development process, VoltDB predefines two roles for you when you enable security: administrator and user. *Administrator* has ADMIN permissions: access to all functions including interactive SQL queries, DDL, system procedures, and user-defined procedures. *User* has SQL and ALLPROC pemissions: access to ad hoc SQL and all default and user-defined stored procedures.

These predefined roles are important, because when you start the database there is no schema and therefore no user-defined roles available to assign to users. So you should always include at least one user who is assigned the Administrator role when starting a database with security enabled. You can use this account to then load the schema — including additional security roles and permissions — and then update the deployment file to add more users as necessary.

12.7. Integrating Kerberos Security with VoltDB

For environments where more secure communication is required than hashed usernames and passwords, it is possible for a VoltDB database to use Kerberos to authenticate clients and servers. Kerberos is a popular network security protocol that you can use to authenticate the Java client processes when they connect to VoltDB database servers. Use of Kerberos is supported for the Java client library only.

To use Kerberos authentication for VoltDB security, you must perform the following steps:

1. Set up and configure Kerberos on your network, servers, and clients.

2. Install and configure the Java security extensions on your VoltDB servers and clients.

3. Configure the VoltDB cluster and client applications to use Kerberos.

The following sections describe these steps in detail.

12.7.1. Installing and Configuring Kerberos

Kerberos is a complete software solution for establishing a secure network environment. It includes network protocols and software for handling authentication and authorization in a secure, encrypted fashion. Kerberos requires one or more servers known as key distribution centers (KDC) to authenticate and authorize services and the users who access them.

To use Kerberos for VoltDB authentication you must first set up Kerberos within your network environment. If you do not already have a Kerberos KDC, you will need to create one. You will also need to install the Kerberos client libraries on all of the VoltDB servers and clients and set up the appropriate principals and services. Because Kerberos is a complete network environment rather than a single platform application, it is beyond the scope of this document to explain how to install and configure Kerberos itself. This section only provides notes specific to configuring Kerberos for use by VoltDB. For complete information about setting up and using Kerberos, please see the Kerberos documentation.

Part of the Kerberos setup is the creation of a configuration file on both the VoltDB server and client machines. By default, the configuration file is located in /etc/krb5.conf (or /private/etc/krb5.conf on Macintosh). Be sure this file exists and points to the correct realm and KDC.

Once a KDC exists and the nodes are configured correctly, you must create the necessary Kerberos accounts — known as "user principals" for the accounts that run the VoltDB client applications and a "service principal" for the VoltDB cluster. For example, to create the service keytab file for the VoltDB database, you can issue the following commands on the Kerberos KDC:

```
$ sudo kadmin.local
kadmin.local: addprinc -randkey service/voltdb
kadmin.local: ktadd -k voltdb.keytab service/voltdb
```

Then copy the keytab file to the database servers, making sure it is only accessible by the user account that starts the database process:

```
$ scp voltdb.keytab voltadmin@voltsvr:voltdb.keytab
$ ssh voltadmin@voltsvr chmod 0600 voltdb.keytab
```

12.7.2. Installing and Configuring the Java Security Extensions

The next step is to install and configure the Java security extension known as Java Cryptography Extension (JCE). JCE enables the more robust encryption required by Kerberos within the Java Authentication and Authorization Service (JAAS). This is necessary because VoltDB uses JAAS to interact with Kerberos.

The JCE that needs to be installed is specific to the version of Java you are running. See the the Java web site for details. Again, you must install JCE on both the VoltDB servers and client nodes

Once JCE is installed, you create a JAAS login configuration file so Java knows how to authenticate the current process. By default, the JAAS login configuration file is `$HOME/.java.login.config`. On the database servers, the configuration file must define the *VoltDBService* module and associate it with the keytab created in the previous section.

Server JAAS Login Configuration File

```
VoltDBService {
    com.sun.security.auth.module.Krb5LoginModule required
        useKeyTab=true keyTab="/home/voltadmin/voltdb.keytab"
```

```
        doNotPrompt=true
        principal="service/voltdb@MYCOMPANY.LAN" storeKey=true;
};
```

On the client nodes, the JAAS login configuration defines the *VoltDBClient* module.

Client JAAS Login Configuration File

```
VoltDBClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useTicketCache=true renewTGT=true doNotPrompt=true;
};
```

12.7.3. Configuring the VoltDB Servers and Clients

Finally, once Kerberos and the Java security extensions are installed and configured, you must configure the VoltDB database cluster and client applications to use Kerberos.

On the database servers, you enable Kerberos security using the <security> element in the deployment file, specifying "kerberos" as the provider. For example:

```
<?xml version="1.0"?>
<deployment>
     <security enabled="true" provider="kerberos"/>
       . . .
</deployment>
```

You then assign roles to individual users as described in Section 12.3, “Defining Users and Roles”, except in place of generic usernames, you specify the Kerberos user — or "principal" — names, including their realm. Since Kerberos uses encrypted certificates, the password attribute is ignored and can be filled in with arbitrary text. For example:

```
<?xml version="1.0"?>
<deployment>
   <security enabled="true" provider="kerberos"/>
       . . .
   <users>
       <user name="mtwain@MYCOMPANY.LAN" password="n/a" role="admin"/>
       <user name="cdickens@MYCOMPANY.LAN" password="n/a" role="dev"/>
       <user name="hbalzac@MYCOMPANY.LAN" password="n/a" role="adhoc"/>
   </users>
</deployment>
```

Having configured Kerberos in the deployment file, you are ready to start the VoltDB cluster. When starting the VoltDB process, Java must know how to access the Kerberos and JAAS login configuration files created in the preceding sections. If the files are not in their default locations, you can override the default location using the VOLTDB_OPTS environment variable and setting the flags `java.security.krb5.conf` and `java.security.auth.login.config`, respectively.[1]

In the client application, you specify Kerberos as the security protocol when you create the client connection, using the enableKerberosAuthentication method as part of the configuration. For example:

```
import org.voltdb.client.ClientConfig;
```

[1] On Macintosh systems, you must always specify the `java.security.krb5.conf` property.

```
import org.voltdb.client.ClientFactory;

ClientConfig config = new ClientConfig();
      // specify the JAAS login module
config.enableKerberosAuthentication("VoltDBClient");

VoltClient client = ClientFactory.createClient(config);
client.createConnection("voltsvr");
```

Note that the VoltDB client automatically picks up the Kerberos cached credentials of the current process, the user's Kerberos "principal". So you do not need to — and should *not* — specify a username or password as part of the VoltDB client configuration.

It is also important to note that once the cluster starts using Kerberos authentication, only Java clients can connect to the cluster and they must also use Kerberos authentication, including the CLI command **sqlcmd**. To authenticate to a VoltDB server with Kerberos security enabled using **sqlcmd**, you must include the `--kerberos` flag identifying the name of the Kerberos client service module. For example:

```
$ sqlcmd --kerberos=VoltDBClient
```

Again, if the configuration files are not in the default location, you must specify their location on the command line:

```
$ sqlcmd --kerberos=VoltDBClient -J-Djava.security.krb5.conf=/etc/krb5.conf
```

You cannot use clients in other programming languages or CLI commands other than **sqlcmd** to access a cluster with Kerberos security enabled.

Chapter 13. Saving & Restoring a VoltDB Database

There are times when it is necessary to save the contents of a VoltDB database to disk and then restore it. For example, if the cluster needs to be shut down for maintenance, you may want to save the current state of the database before shutting down the cluster and then restore the database once the cluster comes back online. Performing periodic backups of the data can also provide a fallback in case of unexpected failures — either physical failures, such as power outages, or logic errors where a client application mistakenly corrupts the database contents.

VoltDB provides shell commands, system procedures, and an automated snapshot feature that help you perform these operations. The following sections explain how to save and restore a running VoltDB cluster, either manually or automatically.

13.1. Performing a Manual Save and Restore of a VoltDB Cluster

Manually saving and restoring a VoltDB database is useful when you need to do maintenance on the database itself or the cluster it runs on. The normal use of save and restore, when performing such a maintenance operation, is as follows:

1. Stop database activities (using pause).
2. Use save to write a snapshot of the current data to disk.
3. Shutdown the cluster.
4. Make changes to the VoltDB schema, cluster configuration, and/or deployment file as desired.
5. Restart the cluster in admin mode.
6. Optionally, reload the schema and stored procedures
7. Restore the previous snapshot.
8. Restart client activity (using resume).

The key is to make sure that all database activity is stopped before the save and shutdown are performed. This ensures that no further changes to the database are made (and therefore lost) after the save and before the shutdown. Similarly, it is important that no client activity starts until the database has started and the restore operation completes.

Also note that Step #6, reloading the schema, is optional. If you are going to reuse the same schema and there are no tables currently defined in the database, the restore operation will automatically load the schema from the snapshot itself. If you want to modify the schema in any way, such as changing indexes or tables and columns, you should load the modified schema before restoring the data from the snapshot. If tables are defined, only the data is loaded from the snapshot. See Section 13.1.3.2, “Modifying the Database Schema and Stored Procedures” for more information on modifying the schema when restoring snapshots.

Save and restore operations are performed either by calling VoltDB system procedures or using the corresponding **voltadmin** shell commands. In most cases, the shell commands are simpler since they do not

require program code to use. Therefore, this chapter uses **voltadmin** commands in the examples. If you are interested in programming the save and restore procedures, see Appendix G, *System Procedures* for more information about the corresponding system procedures.

When you issue a save command, you specify a path where the data will be saved and a unique identifier for tagging the files. VoltDB then saves the current data on each node of the cluster to a set of files at the specified location (using the unique identifier as a prefix to the file names). This set of files is referred to as a snapshot, since it contains a complete record of the database for a given point in time (when the save operation was performed).

The `--blocking` option lets you specify whether the save operation should block other transactions until it completes. In the case of manual saves, it is a good idea to use this option since you do not want additional changes made to the database during the save operation.

Note that every node in the cluster uses the same absolute path, so the path specified must be valid, must exist on every node, and must not already contain data from any previous saves using the same unique identifier, or the save will fail.

When you issue a restore command, you specify the same absolute path and unique identifier used when creating the snapshot. VoltDB checks to make sure the appropriate save set exists on each node, then restores the data into memory.

13.1.1. How to Save the Contents of a VoltDB Database

To save the contents of a VoltDB database, use the **voltadmin save** command. The following example creates a snapshot at the path */tmp/voltdb/backup* using the unique identifier *TestSnapshot*.

```
$ voltadmin save --blocking /tmp/voltdb/backup "TestSnapshot"
```

In this example, the command tells the save operation to block all other transactions until it completes. It is possible to save the contents without blocking other transactions (which is what automated snapshots do). However, when performing a manual save prior to shutting down, it is normal to block other transactions to ensure you save a known state of the database.

Note that it is possible for the save operation to succeed on some nodes of the cluster and not others. When you issue the **voltadmin save** command, VoltDB displays messages from each partition indicating the status of the save operation. If there are any issues that would stop the process from starting, such as a bad file path, they are displayed on the console. It is a good practice to examine these messages to make sure all partitions are saved as expected.

13.1.2. How to Restore the Contents of a VoltDB Database Manually

The easiest way to restore a snapshot is to let VoltDB do it for you as part of the recover operation. If you are not changing the cluster configuration you can use an automated snapshot or other snapshot saved into the `voltdbroot/snapshots` directory by simply restarting the cluster nodes using the **voltdb recover** command. With the recover action VoltDB automatically starts and restores the most recent snapshot. This approach has the added benefit that VoltDB automatically loads the previous schema as well as part of the snapshot.

However, you cannot use **voltdb recover** to restore a snapshot or command log if the cluster configuration has changed, if you updated the VoltDB software itself, or if you want to restore an earlier snapshot or a snapshot stored in an alternate location. In these cases you must do a manual restore.

To manually restore a VoltDB database from a snapshot previously created by a save operation, you use the **voltadmin restore** command. You must specify the same pathname and unique identifier used during the save.

The following example restores the snapshot created by the example in Section 13.1.1.

```
$ voltadmin restore /tmp/voltdb/backup "TestSnapshot"
```

As with save operations, it is always a good idea to check the status information displayed by the command to ensure the operation completed as expected.

13.1.3. Changing the Cluster Configuration Using Save and Restore

Between a save and a restore, it is possible to make changes to the the database and cluster configuration. You can:

- Modify the schema and/or stored procedures
- Add or remove nodes from the cluster
- Change the number of sites per host
- Change the K-safety value

To make these changes, you must make appropriate modifications to the schema, restart the cluster as an empty database, reload the schema and stored procedures, and then perform the restore. The following sections discuss these steps in more detail.

13.1.3.1. Adding and Removing Nodes from the Database

To add nodes to the cluster, use the following procedure:

1. Save the database.
2. Edit the deployment file, specifying the new number of nodes in the hostcount attribute of the <cluster> tag.
3. Restart the cluster (including the new nodes).
4. Issue a restore command.

When the snapshot is restored, the database (and partitions) are redistributed over the new cluster configuration.

It is also possible to remove nodes from the cluster using this procedure. However, to make sure that no data is lost in the process, you must copy the snapshot files from the nodes that are being removed to one of the nodes that is remaining in the cluster. This way, the restore operation can find and restore the data from partitions on the missing nodes.

13.1.3.2. Modifying the Database Schema and Stored Procedures

To modify the database schema or stored procedures between a save and restore, make the appropriate changes to the source files (that is, the database DDL and the stored procedure Java source files). If you modify the stored procedures, be sure to repackage any Java stored procedures into a JAR file. Then you can:

1. Restart the cluster as an empty database.
2. Reload the schema.
3. Reload the stored procedures using the sqlcmd **load classes** directive.
4. Issue the restore command.

Two points to note when modifying the database structure before restoring a snapshot are:

- When existing rows are restored to tables where new columns have been added, the new columns are filled with either the default value (if defined by the schema) or nulls.
- When changing the datatypes of columns, it is possible to decrease the datatype size (for example, going from an INT to an TINYINT). However, if any existing values exceed the capacity of the new datatype (such as an integer value of 5,000 where the datatype has been changed to TINYINT), the entire restore will fail.

If you remove or modify stored procedures (particularly if you change the number and/or datatype of the parameters), you must make sure the corresponding changes are made to all client applications as well.

13.2. Scheduling Automated Snapshots

Save and restore are useful when planning for scheduled down times. However, these functions are also important for reducing the risk from unexpected outages. VoltDB assists in contingency planning and recovery from such worst case scenarios as power failures, fatal system errors, or data corruption due to application logic errors.

In these cases, the database stops unexpectedly or becomes unreliable. By automatically generating snapshots at set intervals, VoltDB gives you the ability to restore the database to a previous valid state.

You schedule automated snapshots of the database as part of the deployment file. The <snapshot> tag lets you specify:

- The frequency of the snapshots. You can specify any whole number of seconds, minutes, or hours (using the suffix "s", "m", or "h", respectively, to denote the unit of measure). For example "3600s", "60m", and "1h" are all equivalent. The default frequency is 24 hours.
- The unique identifier to use as a prefix for the snapshot files. The default prefix is "AUTOSNAP".
- The number of snapshots to retain. Snapshots are marked with a timestamp (as part of the file names), so multiple snapshots can be saved. The `retain` attribute lets you specify how many snapshots to keep. Older snapshots are purged once this limit is reached. The default number of snapshots retained is two.

The following example enables automated snapshots every thirty minutes using the prefix "flightsave" and keeping only the three most recent snapshots.

```
<snapshot prefix="flightsave"
          frequency="30m"
          retain="3"
/>
```

By default, automated snapshots are stored in a subfolder of the VoltDB default path (as described in Section 3.6.2, “Configuring Paths for Runtime Features”). You can save the snapshots to a specific path by adding the <snapshots> tag within to the <paths>...</paths> tag set. For example, the following example defines the path for automated snapshots as `/etc/voltdb/autobackup/`.

```
<paths>
   <snapshots path="/etc/voltdb/autobackup/" />
</paths>
```

13.3. Managing Snapshots

VoltDB does not delete snapshots after they are restored; the snapshot files remain on each node of the cluster. For automated snapshots, the oldest snapshot files are purged according to the settings in the deployment file. But if you create snapshots manually or if you change the directory path or the prefix for automated snapshots, the old snapshots will also be left on the cluster.

To simplify maintenance, it is a good idea to observe certain guidelines when using save and restore:

- Create dedicated directories for use as the paths for VoltDB snapshots.
- Use separate directories for manual and automated snapshots (to avoid conflicts in file names).
- Do not store any other files in the directories used for VoltDB snapshots.
- Periodically cleanup the directories by deleting obsolete, unused snapshots.

You can delete snapshots manually. To delete a snapshot, use the unique identifier, which is applied as a filename prefix, to find all of the files in the snapshot. For example, the following commands remove the snapshot with the ID TestSave from the directory /etc/voltdb/backup/. Note that VoltDB separates the prefix from the remainder of the file name with a dash for manual snapshots:

```
$ rm /etc/voltdb/backup/TestSave-*
```

However, it is easier if you use the system procedures VoltDB provides for managing snapshots. If you delete snapshots manually, you must make sure you execute the commands on all nodes of the cluster. When you use the system procedures, VoltDB distributes the operations across the cluster automatically.

VoltDB provides several system procedures to assist with the management of snapshots:

- @SnapshotStatus provides information about the most recently performed snapshots for the current database. The response from SnapshotStatus includes information about up to ten recent snapshots, including their location, when they were created, how long the save took, whether they completed successfully, and the size of the individual files that make up the snapshot. See the reference section on @SnapshotStatus for details.
- @SnapshotScan lists all of the snapshots available in a specified directory path. You can use this system procedure to determine what snapshots exist and, as a consequence, which ought to be deleted. See the reference section on @SnapshotScan for details.
- @SnapshotDelete deletes one or more snapshots based on the paths and prefixes you provide. The parameters to the system procedure are two string arrays. The first array specifies one or more directory paths. The second array specifies one or more prefixes. The array elements are taken in pairs to determine which snapshots to delete. For example, if the first array contains paths A, B, and C and the second array contains the unique identifiers X, Y, and Z, the following three snapshots will be deleted: A/X, B/Y, and C/Z. See the reference section on @SnapshotDelete for details.

13.4. Special Notes Concerning Save and Restore

The following are special considerations concerning save and restore that are important to keep in mind:

- Save and restore do not check the cluster health (whether all nodes exist and are running) before executing. The user can find out what nodes were saved by looking at the messages displayed by the save operation.

- Both the save and restore calls do a pre-check to see if the action is likely to succeed before the actual save/restore is attempted. For save, VoltDB checks to see if the path exists, if there is any data that might be overwritten, and if it has write access to the directory. For restore, VoltDB verifies that the saved data can be restored completely.

- You should use separate directories for manual and automated snapshots to avoid naming conflicts.

- It is possible to provide additional protection against failure by copying the automated snapshots to remote locations. Automated snapshots are saved locally on the cluster. However, you can set up a network process to periodically copy the snapshot files to a remote system. (Be sure to copy the files from all of the cluster nodes.) Another approach would be to save the snapshots to a SAN disk that is already set up to replicate to another location. (For example, using iSCSI.)

Chapter 14. Command Logging and Recovery

By executing transactions in memory, VoltDB, frees itself from much of the management overhead and I/O costs of traditional database products. However, accidents do happen and it is important that the contents of the database be safeguarded against loss or corruption.

Snapshots provide one mechanism for safeguarding your data, by creating a point-in-time copy of the database contents. But what happens to the transactions that occur between snapshots?

Command logging provides a more complete solution to the durability and availability of your VoltDB database. Command logging keeps a record of every transaction (that is, stored procedure) as it is executed. Then, if the servers fail for any reason, the database can restore the last snapshot and "replay" the subsequent logs to re-establish the database contents in their entirety.

The key to command logging is that it logs the invocations, not the consequences, of the transactions. A single stored procedure can include many individual SQL statements and each SQL statement can modify hundreds or thousands of table rows. By recording only the invocation, the command logs are kept to a bare minimum, limiting the impact the disk I/O will have on performance.

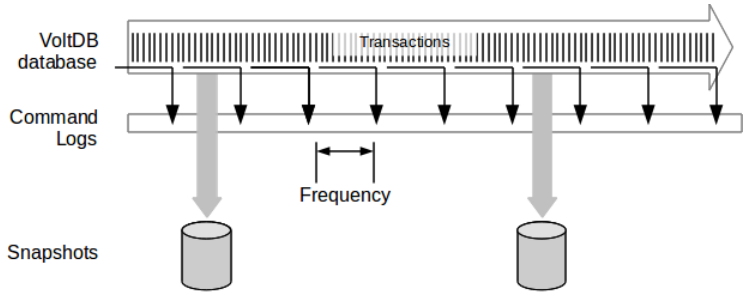
However, any additional processing can impact overall performance, especially when it involves disk I/O. So it is important to understand the tradeoffs concerning different aspects of command logging and how it interacts with the hardware and any other options you are utilizing. The following sections explain how command logging works and how to configure it to meet your specific needs.

14.1. How Command Logging Works

When command logging is enabled, VoltDB keeps a log of every transaction (that is, stored procedure) invocation. At first, the log of the invocations are held in memory. Then, at a set interval the logs are physically written to disk. Of course, at a high transaction rate, even limiting the logs to just invocations, the logs begin to fill up. So at a broader interval, the server initiates a snapshot. Once the snapshot is complete, the command logging process is able to free up — or "truncate" — the log keeping only a record of procedure invocations since the last snapshot.

This process can continue indefinitely, using snapshots as a baseline and loading and truncating the command logs for all transactions since the last snapshot.

Figure 14.1. Command Logging in Action

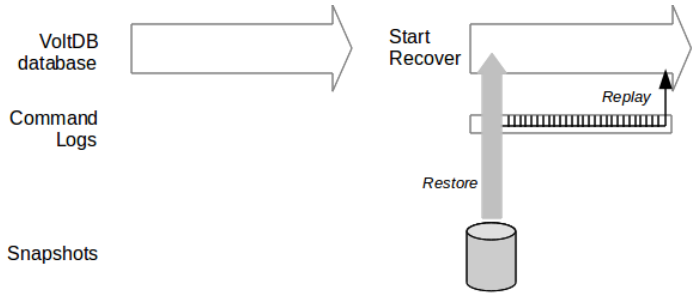


The frequency with which the transactions are written to the command log is configurable (as described in Section 14.3, “Configuring Command Logging for Optimal Performance”). By adjusting the frequency and

type of logging (synchronous or asynchronous) you can balance the performance needs of your application against the level of durability desired.

In reverse, when it is time to "replay" the logs, if you start the database with the **recover** action (as described in Section 3.5, “Restarting a VoltDB Database”) once the server nodes establish a quorum, they start by restoring the most recent snapshot. Once the snapshot is restored, they then replay all of the transactions in the log since that snapshot.

Figure 14.2. Recovery in Action



14.2. Controlling Command Logging

Command logging is enabled by default in the VoltDB Enterprise Edition. Using command logging is recommended to ensure durability of your data. However, you can choose whether to have command logging enabled or not using the `<commandlog>` element in the deployment file. For example:

```
<deployment>
   <cluster hostcount="4" sitesperhost="2" kfactor="1" />
   <commandlog enabled="true"/>
</deployment>
```

In its simplest form, the `<commandlog/>` tag enables or disables command logging by setting the `enabled` attribute to "true" or "false". You can also use other attributes and child elements to control specific characteristics of command logging. The following section describes those options in detail.

14.3. Configuring Command Logging for Optimal Performance

Command logging can provide complete durability, preserving a record of every transaction that is completed before the database stops. However, the amount of durability must be balanced against the performance impact and hardware requirements to achieve effective I/O.

VoltDB provides three settings you can use to optimize command logging:

- The amount of disk space allocated to the command logs
- The frequency between writes to the command logs
- Whether logging is synchronous or asynchronous

The following sections describe these options. A fourth section discusses the impact of storage hardware on the different logging options.

14.3.1. Log Size

The command log size specifies how much disk space is preallocated for storing the logs on disk. The logs are divided into three "segments" Once a segment is full, it is written to a snapshot (as shown in Figure 14.1, “Command Logging in Action”).

For most workloads, the default log size of one gigabyte is sufficient. However, if your workload writes large volumes of data or uses large strings for queries (so the procedure invocations include large parameter values), the log segments fill up very quickly. When this happens, VoltDB can end up snapshotting continuously, because by the time one snapshot finishes, the next log segment is full.

To avoid this situation, you can increase the total log size, to reduce the frequency of snapshots. You define the log size in the deployment file using the `logsize` attribute of the `<commandlog>` tag. Specify the desired log size as an integer number of megabytes. For example:

```
<commandlog enabled="true" logsize="3072" />
```

When increasing the log size, be aware that the larger the log, the longer it may take to recover the database since any transactions in the log since the last snapshot must be replayed before the recovery is complete. So, while reducing the frequency of snapshots, you also may be increasing the time needed to restart.

The minimum log size is three megabytes. Note that the log size specifies the *initial* size. If the existing segments are filled before a snapshot can truncate the logs, the server will allocate additional segments.

14.3.2. Log Frequency

The log frequency specifies how often transactions are written to the command log. In other words, the interval between writes, as shown in Figure 14.1, “Command Logging in Action”. You can specify the frequency in either or both time and number of transactions.

For example, you might specify that the command log is written every 200 milliseconds or every 500 transactions, whichever comes first. You do this by adding the `<frequency>` element as a child of `<commandlog>` and specifying the individual frequencies as attributes. For example:

```
<commandlog enabled="true">
   <frequency time="200" transactions="500"/>
</commandlog>
```

Time frequency is specified in milliseconds and transaction frequency is specified as the number of transactions. You can specify either or both types of frequency. If you specify both, whichever limit is reached first initiates a write.

14.3.3. Synchronous vs. Asynchronous Logging

If the command logs are being written *asynchronously* (which is the default), results are returned to the client applications as soon as the transactions are completed. This allows the transactions to execute uninterrupted.

However, with asynchronous logging there is always the possibility that a catastrophic event (such as a power failure) could cause the cluster to fail. In that case, any transactions completed since the last write and before the failure would be lost. The smaller the frequency, the less data that could be lost. This is how you "dial up" the amount of durability you want using the configuration options for command logging.

In some cases, no loss of data is acceptable. For those situations, it is best to use *synchronous logging*. When you select synchronous logging, no results are returned to the client applications until those transactions

are written to the log. In other words, the results for all of the transactions since the last write are held on the server until the next write occurs.

The advantage of synchronous logging is that no transaction is "complete" and reported back to the calling application until it is guaranteed to be logged — no transactions are lost. The obvious disadvantage of synchronous logging is that the interval between writes (i.e. the frequency) while the results are held, adds to the latency of the transactions. To reduce the penalty of synchronous logging, you need to reduce the frequency.

When using synchronous logging, it is recommended that the frequency be limited to between 1 and 4 milliseconds to avoid adding undue latency to the transaction rate. A frequency of 1 or 2 milliseconds should have little or no measurable affect on overall latency. However, low frequencies can only be achieved effectively when using appropriate hardware (as discussed in the next section, Section 14.3.4, “Hardware Considerations”).

To select synchronous logging, use the `synchronous` attribute of the `<commandlog>` tag. For example:

```
<commandlog enabled="true" synchronous="true" >
   <frequency time="2"/>
</commandlog>
```

14.3.4. Hardware Considerations

Clearly, synchronous logging is preferable since it provides complete durability. However, to avoid negatively impacting database performance you must not only use very low frequencies, but you must have storage hardware that is capable of handling frequent, small writes. Attempting to use aggressively low log frequencies with storage devices that cannot keep up will also hurt transaction throughput and latency.

Standard, uncached storage devices can quickly become overwhelmed with frequent writes. So you should not use low frequencies (and therefore synchronous logging) with slower storage devices. Similarly, if the command logs are competing for the device with other disk I/O, performance will suffer. So do not write the command logs to the same device that is being used for other I/O, such as snapshots or export overflow.

On the other hand, fast, cached devices such as disks with a battery-backed cache, are capable of handling frequent writes. So it is strongly recommended that you use such devices when using synchronous logging.

To specify where the command logs and their associated snapshots are written, you use tags within the `<paths>...</paths>` tag set. For example, the following example specifies that the logs are written to `/fastdisk/voltdblog` and the snapshots are written to `/opt/voltdb/cmdsnaps`:

```
<paths>
   <commandlog path="/faskdisk/voltdblog/" />
   <commandlogsnapshot path="/opt/voltdb/cmdsnaps/" />
</paths>
```

Note that the default paths for the command logs and the command log snapshots are both subfolders of the voltdbroot directory. To avoid overloading a single device on production servers, it is recommended that you specify an explicit path for the command logs, at a minimum, and preferably for both logs and snapshots.

To summarize, the rules for balancing command logging with performance and throughput on production databases are:

- Use asynchronous logging with slower storage devices.

- Write command logs to a dedicated device. Do not write logs and snapshots to the same device.
- Use low (1-2 milisecond) frequencies when performing synchronous logging.
- Use moderate (100 millisecond or greater) frequencies when performing asynchronous logging.

Chapter 15. Importing and Exporting Live Data

VoltDB is an in-memory, transaction processing database. It excels at managing large volumes of transactions in real-time.

However, transaction processing is often only one aspect of the larger business context and data needs to transition from system to system as part of the overall solution. The process of moving from one database to another as data moves through the system is often referred to as Extract, Transform, and Load (ETL). VoltDB supports ETL through the ability to selectively export data as it is committed to the database, as well as the ability to import data through multiple standard protocols.

Exporting data differs from save and restore (as described in Chapter 13, *Saving & Restoring a VoltDB Database*) in several ways:

- You only export selected data (as required by the business process)
- Export is an ongoing process rather than a one-time event
- The outcome of exporting data is that information is used by other business processes, not as a backup copy for restoring the database

The target for exporting data from VoltDB may be another database, a repository (such as a sequential log file), or a process (such as a system monitor or accounting system). No matter what the target, VoltDB helps automate the process for you. This chapter explains how to plan for and implement the exporting of live data using VoltDB.

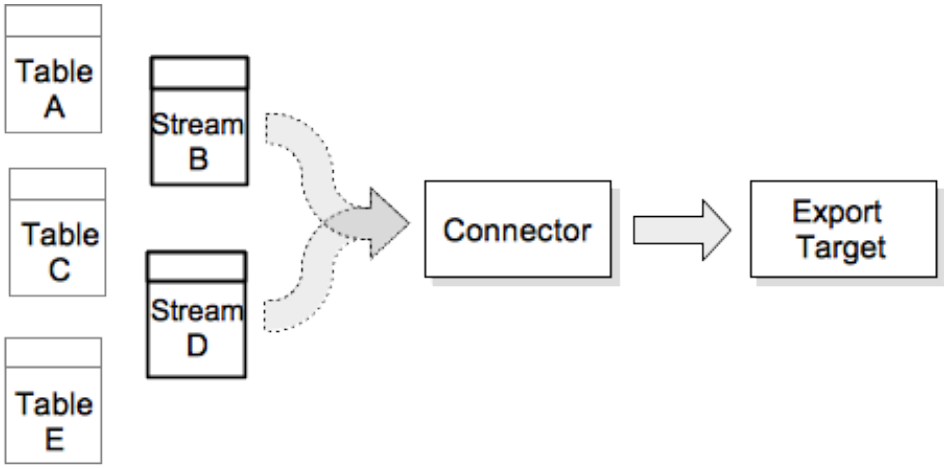
For import, VoltDB supports both one-time import through data loading utilities and ongoing import as part of the database process. The following sections describe how to use VoltDB export and import in detail.

15.1. Understanding Export

VoltDB lets you automate the export process by specifying streams in the schema as sources for export. Streams act just like tables, except no data is actually stored in the database. At runtime, any data written to the specified streams is sent to the selected export *connector*, which queues the data for export. Then, asynchronously, the connector sends the queued export data to the selected output target. Which export connector runs depends on the target you choose when configuring export in the deployment file. Currently, VoltDB provides connectors for exporting to files, for exporting to other business processes via a distributed message queue or HTTP, and for exporting to other databases via JDBC. The connector processes are managed by the database servers themselves, helping to distribute the work and ensure maximum throughput.

Figure 15.1, “Overview of the Export Process” illustrates the basic export procedure, where streams B and D are specified as export streams. (Streams can be used for other things besides export. This chapter focuses on their use for export but see the description of the CREATE STREAM statement for other uses.)

Figure 15.1. Overview of the Export Process



Note that you do not need to modify the schema or the client application to turn exporting of live data on and off. The application's stored procedures insert data into the streams; but it is the deployment file that determines whether export actually occurs at runtime.

When a stored procedure uses an SQL INSERT statement to write data into a export stream, rather than storing that data in the database, it is handed off to the connector when the stored procedure successfully commits the transaction.[1] Export streams have several important characteristics:

- Streams let you limit the export to only the data that is required. For example, in the preceding example, Stream B may contain a subset of columns from Table A. Whenever a new record is written to Table A, the corresponding columns can be written to Stream B for export to the remote database.
- Streams let you combine fields from several existing tables into a single exported row. This technique is particularly useful if your VoltDB database and the target of the export have different schema. The stream can act as a transformation of VoltDB data to a representation of the target schema.
- Streams let you control *when* data is exported. Again, in the previous example, Stream D might be an exact replica of Table C. However, the records in Table C are updated frequently. The client application can choose to copy records from Table C to Stream D only when all of the updates are completed and the data is finalized, significantly reducing the amount of data that must pass through the connector.

Of course, there are restrictions to export streams. Since they have no storage associated with them, they are for INSERT only. Any attempt to SELECT, UPDATE, or DELETE data from streams will result in an error.

15.2. Planning your Export Strategy

The important point when planning to export data, is deciding:

- What data to export
- When to export the data
- Where to export data to

[1]There is no guarantee on the latency of export between the connector and the export target. The export function is transactionally correct; no export occurs if the stored procedure rolls back and the export data is in the appropriate transaction order. But the flow of export data from the connector to the target is not synchronous with the completion of the transaction. There may be several seconds delay before the export data reaches the target.

It is possible to export all of the data in a VoltDB database. You would do this by creating export stream replicas of all tables in the schema and writing to the corresponding stream whenever you insert into the normal table. However, this means the same number of transactions and volume of data that is being processed by VoltDB will be exported through the connector. There is a strong likelihood, given a high transaction volume, that the target database will not be able to keep up with the load VoltDB is handling. As a consequence you will usually want to be more selective about what data is exported when.

If you have an existing target database, the question of what data to export is likely decided for you (that is, you need to export the data matching the target's schema). If you are defining both your VoltDB database and your target at the same time, you will need to think about what information is needed "downstream" and create the appropriate export streams within VoltDB.

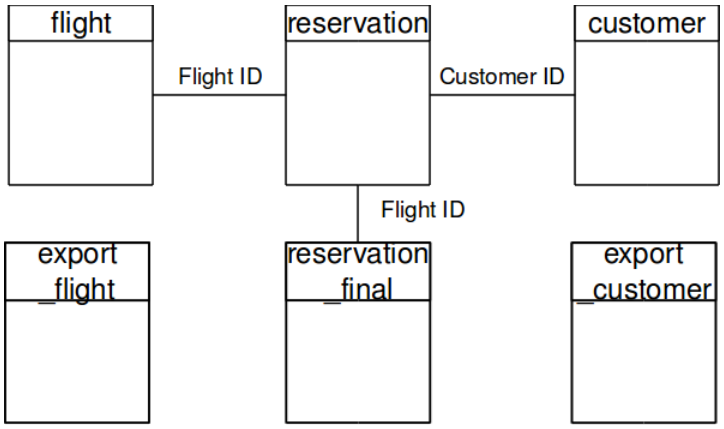
The second consideration is *when* to export the data. For tables that are not updated frequently, inserting the data to a complementary export stream whenever data is inserted into the real table is the easiest and most practical approach. For tables that are updated frequently (hundreds or thousands of times a second) you should consider writing a copy of the data to an export stream at an appropriate milestone.

Using the flight reservation system as an example, one aspect of the workflow not addressed by the application described in Chapter 6, *Designing VoltDB Client Applications* is the need to archive information about the flights after takeoff. Changes to reservations (additions and cancellations) are important in real time. However, once the flight takes off, all that needs to be recorded (for billing purposes, say) is what reservations were active at the time.

In other words, the archiving database needs information about the customers, the flights, and the final reservations. According to the workload in Table 4.1, “Example Application Workload”, the customer and flight tables change infrequently. So data can be inserted into the export streams at the same time as the "live" flight and reservation tables. (It is a good idea to give the export stream a meaningful name so its purpose is clear. In this example we identify the streams with the export_ prefix or, in the case of the reservation stream which is not an exact copy, the _final suffix.)

The reservation table, on the other hand, is updated frequently. So rather than export all changes to a reservation to the reservation stream in real-time, a separate stored procedure is invoked when a flight takes off. This procedure copies the final reservation data to the export stream and deletes the associated flight and reservation records from the VoltDB database. Figure 15.2, “Flight Schema with Export Streams” shows the modified database schema with the added export streams, EXPORT_FLIGHT, EXPORT_CUSTOMER, and RESERVATION_FINAL.

Figure 15.2. Flight Schema with Export Streams



This design adds a transaction to the VoltDB application, which is executed approximately once a second (when a flight takes off). However, it reduces the number of reservation transactions being exported from

1200 a second to less than 200 a second. These are the sorts of trade offs you need to consider when adding export functionality to your application.

The third decision is where to export the data to. As described in Section 15.4, “Configuring Export in the Deployment File”, you can export the data through multiple different protocols: files, HTTP. JDBC, etc. Your choice of protocol will depend on the ultimate target destination for your exported data.

You can also export to multiple destinations at once. When you declare a stream, you can assign it to a specific export *target*. If you want different streams to be exported to different destinations, you declare the streams to belong to different targets. Then in the deployment file you can configure each target to be exported to a different destination.

15.3. Identifying Export Streams in the Schema

Once you decide what data to export, you are ready to declare your export streams in the schema. Streams are defined in the database schema much like tables, except you use the CREATE STREAM statement instead of CREATE TABLE. So in the case of the flight application, we need to add the export streams to our schema. The following example illustrates (in bold) the addition of a stream for reservations with a subset of columns from the normal reservation table.

```
    . . .

CREATE TABLE Reservation (
   ReserveID INTEGER UNIQUE NOT NULL,
   FlightID INTEGER NOT NULL,
   CustomerID INTEGER NOT NULL,
   Seat VARCHAR(5) DEFAULT NULL,
   Confirmed TINYINT DEFAULT '0',
   PRIMARY KEY(ReserveID)
);
CREATE STREAM Reservation_final
  EXPORT TO TARGET archive (
   ReserveID INTEGER UNIQUE NOT NULL,
   FlightID INTEGER NOT NULL,
   CustomerID INTEGER NOT NULL,
   Seat VARCHAR(5) DEFAULT NULL
);
     . . .
```

When you declare your streams in the schema, you need to assign them to the appropriate target, using the EXPORT TO TARGET clause. In our example, the three streams are being exported to the same target, so would name the same target in the declaration, as shown in the following abbreviated example:

```
CREATE STREAM export_customer
   EXPORT TO TARGET archive (
         . . .
);
CREATE STREAM export_flight
   EXPORT TO TARGET archive (
         . . .
);
CREATE STREAM reservation_final
   EXPORT TO TARGET archive (
         . . .
);
```

If a stream does not specify an export target, it is not exported. In the preceding example, *export_customer*, *export_flight*, and *reservation_final* streams are identified as the streams that will be sent to the export target called *archive*. Note that, even if an export target is specified in the CREATE STREAM statement, inserting data into these streams will have no effect until export is enabled in the deployment file for the *archive* target.

If you want to export to different locations, you can assign the streams to different targets, then export each stream separately. For example, if you want to export the reservations to a log file but the customer and flight records to an archival database, you can assign the streams to two different targets:

```
CREATE STREAM export_customer
   EXPORT TO TARGET archive (
         . . .
);
CREATE STREAM export_flight
   EXPORT TO TARGET archive (
         . . .
);
CREATE STREAM reservation_final
   EXPORT TO TARGET log (
         . . .
);
```

Note that no changes are required to the client application. The configuration of streams and export targets is all done through the schema and deployment file.

You can also specify whether the streams are partitioned or not using the PARTITION ON COLUMN clause in the CREATE STREAM statement. For example, if an export stream is a copy of a normal data table, it can be partitioned on the same column. However, partitioning is not necessary for export streams. Whether they are partitioned or "replicated", since no storage is associated with the stream, you can INSERT into the stream in either a single-partitioned or multi-partitioned stored procedure. In either case, the export connector ensures that at least one copy of the tuple is written to the export target.

15.4. Configuring Export in the Deployment File

To enable export at runtime, you include the `<export>` and `<configuration>` tags in the deployment file, specifying the target you are configuring and which export connector to use (with the `type` attribute). For example:

```
<export>
   <configuration enabled="true" target="log" type="file">
     . . .
   </configuration>
</export>
```

To export to multiple destinations, you include multiple `<configuration>` tags, each specifying the target it is configuring. Any streams identified as exporting to that target (in a CREATE STREAM... EXPORT TO TARGET statement), then use that configuration for export. For example:

```
<export>
   <configuration enabled="true" type="file" target="log">
     . . .
   </configuration>
   <configuration enabled="true" type="jdbc" target="archive">
     . . .
   </configuration>
</export>
```

You must also configure each export connector by specifying properties as one or more `<property>` tags within the `<configuration>` tag. For example, the following XML code enables export to comma-separated (CSV) text files using the file prefix "MyExport".

```
<export>
   <configuration enabled="true" stream="log" type="file">
     <property name="type">csv</property>
     <property name="nonce">MyExport</property>
  </configuration>
</export>
```

The properties that are allowed and/or required depend on the export connector you select. VoltDB comes with six export connectors:

- Export to file (type="file")
- Export to HTTP, including Hadoop (type="http")
- Export to JDBC (type="jdbc")
- Export to Kafka (type="kafka")
- Export to RabbitMQ (type="rabbitmq")
- Export to Elasticsearch (type="elasticsearch")

As the name implies, the file connector writes the exported data to local files, either as comma-separated or tab-delimited files. Similarly, the JDBC connector writes data to a variety of possible destination databases through the JDBC protocol. The Kafka connector writes export data to an Apache Kafka distributed message queue, where one or more other processes can read the data. In all three cases you configure the specific features of the connector using the `<property>` tag as described in the following sections.

15.5. How Export Works

Two important aspects of export to keep in mind are:

- Export is automatic. When you enable an export configuration in the deployment file, the database servers take care of starting and stopping the connector on each server when the database starts and stops, including if nodes fail and rejoin the cluster. You can also start and stop export on a running database by updating the deployment file using the **voltadmin update** command.
- Export is asynchronous. The actual delivery of the data to the export target is asynchronous to the transactions that initiate data transfer.

The advantage of an asynchronous approach is that any delays in delivering the exported data to the target system do not interfere with the VoltDB database performance. The disadvantage is that VoltDB must

handle queueing export data pending its actual transmission to the target, including ensuring durability in case of system failures. Again, this task is handled automatically by the VoltDB server process. But it is useful to understand how the export queuing works and its consequences.

One consequence of this durability guarantee is that VoltDB will send at least one copy of every export record to the target. However, it is possible when recovering command logs or rejoining nodes, that certain export records are resent. It is up to the downstream target to handle these duplicate records. For example, using unique indexes or including a unique record ID in the export stream.

15.5.1. Export Overflow

For the export process to work, it is important that the connector keep up with the queue of exported information. If too much data gets queued to the connector by the export function without being delivered by the target system, the VoltDB server process consumes increasingly large amounts of memory.

If the export target does not keep up with the connector and the data queue fills up, VoltDB starts writing overflow data in the export buffer to disk. This protects your database in several ways:

- If the destination is intermittently unreachable or cannot keep up with the data flow, writing to disk helps VoltDB avoid consuming too much memory while waiting for the destination to catch up.
- If the database is stopped, the export data is retained across sessions. When the database restarts, the connector will retrieve the overflow data and reinsert it in the export queue.

You can specify where VoltDB writes the overflow export data using the <exportoverflow> element in the deployment file. For example:

```
<paths>
   <voltdbroot path="/opt/voltdb/" />
   <exportoverflow path="/tmp/export/"/>
</paths>
```

If you do not specify a path for export overflow, VoltDB creates a subfolder in the root directory (in the preceding example, `/opt/voltdb`). See Section 3.6.2, “Configuring Paths for Runtime Features” for more information about configuring paths in the deployment file.

15.5.2. Persistence Across Database Sessions

It is important to note that VoltDB only uses the disk storage for overflow data. However, you can force VoltDB to write all queued export data to disk by either calling the @Quiesce system procedure or by requesting a blocking snapshot. (That is, calling @SnapshotSave with the blocking flag set.) This means it is possible to perform an orderly shutdown of a VoltDB database and ensure all data (including export data) is saved with the following procedure:

1. Put the database into admin mode with the **voltadmin pause** command.
2. Perform a blocking snapshot with **voltadmin save**, saving both the database and any existing queued export data.
3. Shutdown the database with **voltadmin shutdown**.

You can then restore the database — and any pending export queue data — by starting the database in admin mode, restoring the snapshot, and then exiting admin mode.

15.6. The File Connector

The file connector receives the serialized data from the export streams and writes it out as text files (either comma or tab separated) to disk. The file connector writes the data out one file per stream, "rolling" over to new files periodically. The filenames of the exported data are constructed from:

- A unique prefix (specified with the `nonce` property)
- A unique value identifying the current version of the database schema
- The stream name
- A timestamp identifying when the file was started

While the file is being written, the file name also contains the prefix "active-". Once the file is complete and a new file started, the "active-" prefix is removed. Therefore, any export files without the prefix are complete and can be copied, moved, deleted, or post-processed as desired.

There are two properties that must be set when using the file connector:

- The `type` property lets you choose between comma-separated files (csv) or tab-delimited files (tsv).
- The `nonce` property specifies a unique prefix to identify all files that the connector writes out for this database instance.

Table 15.1, “File Export Properties” describes the supported properties for the file connector.

Table 15.1. File Export Properties

Property	Allowable Values	Description
type*	csv, tsv	Specifies whether to create comma-separated (CSV) or tab-delimited (TSV) files,
nonce*	string	A unique prefix for the output files.
outdir	directory path	The directory where the files are created. If you do not specify an output path, VoltDB writes the output files to the current default directory.
period	Integer	The frequency, in minutes, for "rolling" the output file. The default frequency is 60 minutes.
binaryencoding	hex, base64	Specifies whether VARBINARY data is encoded in hexadecimal or BASE64 format. The default is hexadecimal.
dateformat	format string	The format of the date used when constructing the output file names. You specify the date format as a Java SimpleDateFormat string. The default format is "yyyyMMddHHmmss".
timezone	string	The time zone to use when formatting the timestamp. Specify the time zone as a Java timezone identifier. The default is GMT.
delimiters	string	Specifies the delimiter characters for CSV output. The text string specifies four characters: the field delimiter, the enclosing character, the escape character, and the record delimiter. To use special or non-printing characters (including the space character) encode the character as an HTML entity. For example "<" for the "less than" symbol.

Property	Allowable Values	Description
batched	true, false	Specifies whether to store the output files in subfolders that are "rolled" according to the frequency specified by the period property. The subfolders are named according to the nonce and the timestamp, with "active-" prefixed to the subfolder currently being written.
skipinternals	true, false	Specifies whether to include six columns of VoltDB metadata (such as transaction ID and timestamp) in the output. If you specify skip-internals as "true", the output files contain only the exported stream data.
with-schema	true, false	Specifies whether to write a JSON representation of each stream's schema as part of the export. The JSON schema files can be used to ensure the appropriate datatype and precision is maintained if and when the output files are imported into another system.

*Required

Whatever properties you choose, the order and representation of the content within the output files is the same. The export connector writes a separate line of data for every INSERT it receives, including the following information:

- Six columns of metadata generated by the export connector. This information includes a transaction ID, a timestamp, a sequence number, the site and partition IDs, as well as an integer indicating the query type.
- The remaining columns are the columns of the database stream, in the same order as they are listed in the database definition (DDL) file.

15.7. The HTTP Connector

The HTTP connector receives the serialized data from the export streams and writes it out via HTTP requests. The connector is designed to be flexible enough to accommodate most potential targets. For example, the connector can be configured to send out individual records using a GET request or batch multiple records using POST and PUT requests. The connector also contains optimizations to support export to Hadoop via WebHDFS.

15.7.1. Understanding HTTP Properties

The HTTP connector is a general purpose export utility that can export to any number of destinations from simple messaging services to more complex REST APIs. The properties work together to create a consistent export process. However, it is important to understand how the properties interact to configure your export correctly. The four key properties you need to consider are:

- **batch.mode** — whether data is exported in batches or one record at a time
- **method** — the HTTP request method used to transmit the data
- **type** — the format of the output
- **endpoint** — the target HTTP URL to which export is written

The properties are described in detail in Table 15.2, “HTTP Export Properties”. This section explains the relationship between the properties.

There are essentially two types of HTTP export: batch mode and one record at a time. Batch mode is appropriate for exporting large volumes of data to targets such as Hadoop. Exporting one record at a time is less efficient for large volumes but can be very useful for writing intermittent messages to other services.

In batch mode, the data is exported using a POST or PUT method, where multiple records are combined in either command-separated value (CSV) or Avro format in the body of the request. When writing one record at a time, you can choose whether to submit the HTTP request as a POST, PUT or GET (that is, as a querystring attached to the URL). When exporting in batch mode, the method must be either `POST` or `PUT` and the type must be either `csv` or `avro`. When exporting one record at a time, you can use the `GET`, `POST`, or `PUT` method, but the output type must be `form`.

Finally, the endpoint property specifies the target URL where data is being sent, using either the http: or https: protocol. Again, the endpoint must be compatible with the possible settings for the other properties. In particular, if the endpoint is a WebHDFS URL, batch mode must enabled.

The URL can also contain placeholders that are filled in at runtime with metadata associated with the export data. Each placeholder consists of a percent sign (%) and a single ASCII character. The following are the valid placeholders for the HTTP endpoint property:

Placeholder	Description
%t	The name of the VoltDB export stream. The stream name is inserted into the endpoint in all uppercase.
%p	The VoltDB partition ID for the partition where the INSERT query to the export stream is executing. The partition ID is an integer value assigned by VoltDB internally and can be used to randomly partition data. For example, when exporting to webHDFS, the partition ID can be used to direct data to different HDFS files or directories.
%g	The export generation. The generation is an identifier assigned by VoltDB. The generation increments each time the database starts or the database schema is modified in any way.
%d	The date and hour of the current export period. Applicable to WebHDFS export only. This placeholder identifies the start of each period and the replacement value remains the same until the period ends, at which point the date and hour is reset for the new period. You can use this placeholder to "roll over" WebHDFS export destination files on a regular basis, as defined by the `period` property. The `period` property defaults to one hour.

When exporting in batch mode, the endpoint must contain at least one instance each of the %t, %p, and %g placeholders. However, beyond that requirement, it can contain as many placeholders as desired and in any order. When not in batch mode, use of the placeholders are optional.

Table 15.2, “HTTP Export Properties” describes the supported properties for the HTTP connector.

Table 15.2. HTTP Export Properties

Property	Allowable Values	Description
endpoint*	string	Specifies the target URL. The endpoint can contain placeholders for inserting the stream name (%t), the partition ID (%p), the date and hour (%d), and the export generation (%g).
avro.compress	true, false	Specifies whether Avro output is compressed or not. The default is false and this property is ignored if the type is not Avro.

Property	Allowable Values	Description
avro.schema.location	string	Specifies the location where the Avro schema will be written. The schema location can be either an absolute path name on the local database server or a webHDFS URL and must include at least one instance of the placeholder for the stream name (%t). Optionally it can contain other instances of both %t and %g. The default location for the Avro schema is the file path `export/avro/%t_avro_schema.json` on the database server under the voltdbroot directory. This property is ignored if the type is not Avro.
batch.mode	true, false	Specifies whether to send multiple rows as a single request or send each export row separately. The default is true. Batch mode must be enabled for WebHDFS export.
httpfs.enable	true, false	Specifies that the target of WebHDFS export is an Apache HttpFS (Hadoop HDFS over HTTP) server. This property must be set to true when exporting webHDFS to HttpFS targets.
kerberos.enable	true, false	Specifies whether Kerberos authentication is used when connecting to a WebHDFS endpoint. This property is only valid when connecting to WebHDFS servers and is false by default.
method	get, post, put	Specifies the HTTP method for transmitting the export data. The default method is POST. For WebHDFS export, this property is ignored.
period	Integer	Specifies the frequency, in hours, for "rolling" the WebHDFS output date and time. The default frequency is every hour (1). For WebHDFS export only.
timezone	string	The time zone to use when formatting the timestamp. Specify the time zone as a Java timezone identifier. The default is the local time zone.
type	csv, avro, form	Specifies the output format. If batch.mode is true, the default type is CSV. If batch.mode is false, the default and only allowable value for type is form. Avro format is supported for WebHDFS export only (see Section 15.7.2, “Exporting to Hadoop via WebHDFS” for details.)

*Required

15.7.2. Exporting to Hadoop via WebHDFS

As mentioned earlier, the HTTP connector contains special optimizations to support exporting data to Hadoop via the WebHDFS protocol. If the endpoint property contains a WebHDFS URL (identified by the URL path component starting with the string "/webhdfs/v1/"), special rules apply.

First, for WebHDFS URLs, the batch.mode property must be enabled. Also, the endpoint must have at least one instance each of the stream name (%t), the partition ID (%p), and the export generation (%g) placeholders and those placeholders must be part of the URL path, not the domain or querystring.

Next, the method property is ignored. For WebHDFS, the HTTP connector uses a combination of POST, PUT, and GET requests to perform the necessary operations using the WebHDFS REST API.

For example, The following deployment file configuration exports stream data to WebHDFS using the HTTP connector and writing each stream to a separate directory, with separate files based on the partition ID, generation, and period timestamp, rolling over every 2 hours:

```
<export>
   <configuration target="hadoop" enabled="true" type="http">
     <property name="endpoint">
        http://myhadoopsvr/webhdfs/v1/%t/data%p-%g.%d.csv
     </property>
     <property name="batch.mode">true</property>
     <property name="period">2</property>
  </configuration>
</export>
```

Note that the HTTP connector will create any directories or files in the WebHDFS endpoint path that do not currently exist and then append the data to those files, using the POST or PUT method as appropriate for the WebHDFS REST API.

You also have a choice between two formats for the export data when using WebHDFS: comma-separated values (CSV) and Apache Avro™ format. By default, data is written as CSV data with each record on a separate line and batches of records attached as the contents of the HTTP request. However, you can choose to set the output format to Avro by setting the `type` property, as in the following example:

```
<export>
   <configuration target="hadoop" enabled="true" type="http">
     <property name="endpoint">
       http://myhadoopsvr/webhdfs/v1/%t/data%p-%g.%d.avro
     </property>
     <property name="type">avro</property>
     <property name="avro.compress">true</property>
     <property name="avro.schema.location">
       http://myhadoopsvr/webhdfs/v1/%t/schema.json
     </property>
  </configuration>
</export>
```

Avro is a data serialization system that includes a binary format that is used natively by Hadoop utilities such as Pig and Hive. Because it is a binary format, Avro data takes up less network bandwidth than text-based formats such as CSV. In addition, you can choose to compress the data even further by setting the `avro.compress` property to true, as in the previous example.

When you select Avro as the output format, VoltDB writes out an accompanying schema definition as a JSON document. For compatibility purposes, the stream name and columns names are converted, removing underscores and changing the resulting words to lowercase with initial capital letters (sometimes called "camelcase"). The stream name is given an initial capital letter, while columns names start with a lowercase letter. For example, the stream EMPLOYEE_DATA and its column named EMPLOYEE_iD would be converted to EmployeeData and employeeId in the Avro schema.

By default, the Avro schema is written to a local file on the VoltDB database server. However, you can specify an alternate location, including a webHDFS URL. So, for example, you can store the schema in the same HDFS repository as the data by setting the `avro.schema.location` property, as shown in the preceding example.

See the Apache Avro web site for more details on the Avro format.

15.7.3. Exporting to Hadoop Using Kerberos Security

If the WebHDFS service to which you are exporting data is configured to use Kerberos security, the VoltDB servers must be able to authenticate using Kerberos as well. To do this, you must perform the following two extra steps:

- Configure Kerberos security for the VoltDB cluster itself
- Enable Kerberos authentication in the export configuration

The first step is to configure the VoltDB servers to use Kerberos as described in Section 12.7, “Integrating Kerberos Security with VoltDB”. Because use of Kerberos authentication for VoltDB security changes how the clients connect to the database cluster, It is best to set up, enable, and test Kerberos authentication first to ensure your client applications work properly in this environment before trying to enable Kerberos export as well.

Once you have Kerberos authentication working for the VoltDB cluster, you can enable Kerberos authentication in the configuration of the WebHDFS export target as well. Enabling Kerberos authentication in the HTTP connector only requires one additional property, `kerberos.enable`, to be set. To use Kerberos authentication, set the property to "true". For example:

```
<export>
   <configuration target="hadoop" enabled="true" type="http">
     <property name="endpoint">
       http://myhadoopsvr/webhdfs/v1/%t/data%p-%g.%d.csv
     </property>
     <property name="type">csv</property>
     <property name="kerberos.enable">true</property>
  </configuration>
</export>
```

Note that Kerberos authentication is only supported for WebHDFS endpoints.

15.8. The JDBC Connector

The JDBC connector receives the serialized data from the export streams and writes it, in batches, to another database through the standard JDBC (Java Database Connectivity) protocol.

When the JDBC connector opens the connection to the remote database, it first attempts to create tables in the remote database to match the VoltDB export stream by executing CREATE TABLE statements through JDBC. This is important to note because, it ensures there are suitable tables to receive the exported data. The tables are created using either the stream names from the VoltDB schema or (if you do not enable the ignoregenerations property) the stream name prefixed by the database generation ID.

If the target database has existing tables that match the VoltDB export streams in both name and structure (that is, the number, order, and datatype of the columns), be sure to enable the ignoregenerations property in the export configuration to ensure that VoltDB uses those tables as the export target.

It is also important to note that the JDBC connector exports data through JDBC in batches. That is, multiple INSERT instructions are passed to the target database at a time, in approximately two megabyte batches. There are two consequences of the batching of export data:

- For many databases, such as Netezza, where there is a cost for individual invocations, batching reduces the performance impact on the receiving database and avoids unnecessary latency in the export processing.
- On the other hand, no matter what the target database, if a query fails for any reason the entire batch fails.

To avoid errors causing batch inserts to fail, it is strongly recommended that the target database not use unique indexes on the receiving tables that might cause constraint violations.

If any errors *do* occur when the JDBC connector attempts to submit data to the remote database, the VoltDB disconnects and then retries the connection. This process is repeated until the connection succeeds. If the connection does not succeed, VoltDB eventually reduces the retry rate to approximately every eight seconds.

Table 15.3, "JDBC Export Properties" describes the supported properties for the JDBC connector.

Table 15.3. JDBC Export Properties

Property	Allowable Values	Description
jdbcurl*	connection string	The JDBC connection string, also known as the URL.
jdbcuser*	string	The username for accessing the target database.
jdbcpassword	string	The password for accessing the target database.
jdbcdriver	string	The class name of the JDBC driver. The JDBC driver class must be accessible to the VoltDB process for the JDBC export process to work. Place the driver JAR files in the `lib/extension/` directory where VoltDB is installed to ensure they are accessible at runtime. You do not need to specify the driver as a property value for several popular databases, including MySQL, Netezza, Oracle, PostgreSQL, and Vertica. However, you still must provide the driver JAR file.
schema	string	The schema name for the target database. The use of the schema name is database specific. In some cases you must specify the database name as the schema. In other cases, the schema name is not needed and the connection string contains all the information necessary. See the documentation for the JDBC driver you are using for more information.
minpoolsize	integer	The minimum number of connections in the pool of connections to the target database. The default value is 10.
maxpoolsize	integer	The maximum number of connections in the pool. The default value is 100.
maxidletime	integer	The number of milliseconds a connection can be idle before it is removed from the pool. The default value is 60000 (one minute).
maxstatementcached	integer	The maximum number of statements cached by the connection pool. The default value is 50.
ignoregenerations	true, false	Specifies whether a unique ID for the generation of the database is included as part of the output table name(s). The generation ID changes each time a database restarts or the database schema is updated. The default is false.
skipinternals	true, false	Specifies whether to include six columns of VoltDB metadata (such as transaction ID and timestamp) in the output. If you specify skipinternals as true, the output contains only the exported stream data. The default is false.

*Required

15.9. The Kafka Connector

The Kafka connector receives serialized data from the export streams and writes it to a message queue using the Apache Kafka version 0.8.2 protocols. Apache Kafka is a distributed messaging service that lets you

set up message queues which are written to and read from by "producers" and "consumers", respectively. In the Apache Kafka model, VoltDB export acts as a "producer".

Before using the Kafka connector, we strongly recommend reading the Kafka documentation and becoming familiar with the software, since you will need to set up a Kafka 0.8.2 service and appropriate "consumer" clients to make use of VoltDB's Kafka export functionality. The instructions in this section assume a working knowledge of Kafka and the Kafka operational model.

When the Kafka connector receives data from the VoltDB export streams, it establishes a connection to the Kafka messaging service as a Kafka producer. It then writes records to Kafka topics based on the VoltDB stream name and certain export connector properties.

The majority of the Kafka export properties are identical in both in name and content to the Kafka producer properties listed in the Kafka documentation. All but one of these properties are optional for the Kafka connector and will use the standard Kafka default value. For example, if you do not specify the `queue.buffering.max.ms` property it defaults to 5000 milliseconds.

The only required property is `bootstrap.servers`, which lists the Kafka servers that the VoltDB export connector should connect to. You must include this property so VoltDB knows where to send the export data. Specify each server by its IP address (or hostname) and port; for example, myserver:7777. If there are multiple servers in the list, separate them with commas.

In addition to the standard Kafka producer properties, there are several custom properties specific to VoltDB. The properties `binaryencoding`, `skipinternals`, and `timezone` affect the format of the data. The `topic.prefix` and `topic.key` properties affect how the data is written to Kafka.

The `topic.prefix` property specifies the text that precedes the stream name when constructing the Kafka topic. If you do not specify a prefix, it defaults to "voltdbexport". Alternately, you can map individual streams to topics using the `topic.key` property. In the `topic.key` property you associate a VoltDB export stream name with the corresponding Kafka topic as a named pair separated by a period (.). Multiple named pairs are separated by commas (,). For example:

```
Employee.EmpTopic,Company.CoTopic,Enterprise.EntTopic
```

Any stream-specific mappings in the `topic.key` property override the automated topic name specified by `topic.prefix`.

Note that unless you configure the Kafka brokers with the `auto.create.topics.enable` property set to true, you must create the topics for every export stream manually before starting the export process. Enabling auto-creation of topics when setting up the Kafka brokers is recommended.

When configuring the Kafka export connector, it is important to understand the relationship between synchronous versus asynchronous processing and its effect on database latency. If the export data is sent asynchronously, the impact of export on the database is reduced, since the export connector does not wait for the Kafka infrastructure to respond. However, with asynchronous processing, VoltDB is not able to resend the data if the message fails after it is sent.

If export to Kafka is done synchronously, the export connector waits for acknowledgement of each message sent to Kafka before processing the next packet. This allows the connector to resend any packets that fail. The drawback to synchronous processing is that on a heavily loaded database, the latency it introduces means export may not be able to keep up with the influx of export data and and have to write to overflow.

You specify the level of synchronicity and durability of the connection using the Kafka `acks` property. Set `acks` to "0" for asynchronous processing, "1" for synchronous delivery to the Kafka broker, or "all" to

ensure durability on the Kafka broker. Use of "all" is *not* recommended for VoltDB export. See the Kafka documentation for more information.

VoltDB guarantees that at least one copy of all export data is sent by the export connector. But when operating in asynchronous mode, the Kafka connector cannot guarantee that the packet is actually received and accepted by the Kafka broker. By operating in synchronous mode, VoltDB can catch errors returned by the Kafka broker and resend any failed packets. However, you pay the penalty of additional latency and possible export overflow.

Finally, the actual export data is sent to Kafka as a comma-separated values (CSV) formatted string. The message includes six columns of metadata (such as the transaction ID and timestamp) followed by the column values of the export stream.

Table 15.4, “Kafka Export Properties” lists the supported properties for the Kafka connector, including the standard Kafka producer properties and the VoltDB unique properties.

Table 15.4. Kafka Export Properties

Property	Allowable Values	Description
bootstrap.servers*	string	A comma-separated list of Kafka brokers (specified as IP-address:port-number). You can use `metadata.broker.list` as a synonym for `bootstrap.servers`.
acks	0, 1, all	Specifies whether export is synchronous (*1* or *all*) or asynchronous (*0*) and to what extent it ensures delivery. See the Kafka documentation of the producer properties for details.
acks.retry.timeout	integer	Specifies how long, in milliseconds, the connector will wait for acknowledgment from Kafka for each packet. The retry timeout only applies if acknowledgements are enabled. That is, if the `acks` property is set greater than zero. The default timeout is 5,000 milliseconds. When the timeout is reached, the connector will resend the packet of messages.
partition.key	{*stream*}.{*column*}[,...]	Specifies which stream column value to use as the Kafka partitioning key for each stream. Kafka uses the partition key to distribute messages across multiple servers. By default, the value of the stream's partitioning column is used as the Kafka partition key. Using this property you can specify a list of stream column names, where the stream name and column name are separated by a period and the list of stream references is separated by commas. If the stream is not partitioned and you do not specify a key, the server partition ID is used as a default.
binaryencoding	hex, base64	Specifies whether VARBINARY data is encoded in hexadecimal or BASE64 format. The default is hexadecimal.
skipinternals	true, false	Specifies whether to include six columns of VoltDB metadata (such as transaction ID and timestamp) in the output. If you specify skipinternals as true, the output

Property	Allowable Values	Description
		contains only the exported stream data. The default is false.
timezone	string	The time zone to use when formatting the timestamp. Specify the time zone as a Java timezone identifier. The default is GMT.
topic.key	string	A set of named pairs each identifying a VoltDB stream name and the corresponding Kafka topic name to which the stream is written. Separate the names with a period (.) and the name pairs with a comma (,). The specific stream/topic mappings declared by topic.key override the automated topic names specified by topic.prefix.
topic.prefix	string	The prefix to use when constructing the topic name. Each row is sent to a topic identified by {prefix}{stream-name}. The default prefix is "voltdbexport".
Kafka producer properties	various	You can specify standard Kafka producer properties as export connector properties and their values will be passed to the Kafka interface. However, you cannot modify the property `block.on.buffer.full`.

*Required

15.10. The RabbitMQ Connector

The RabbitMQ connector fetches serialized data from the export streams and writes it to a RabbitMQ message exchange. RabbitMQ is a popular message queueing service that supports multiple platforms, multiple languages, and multiple protocols, including AMQP.

Before using the RabbitMQ connector, we strongly recommend reading the RabbitMQ documentation and becoming familiar with the software, since you will need to set up a RabbitMQ exchange, queues, and routing key filters to make use of VoltDB's RabbitMQ export functionality. The instructions in this section assume a working knowledge of RabbitMQ and the RabbitMQ operational model.

You must also install the RabbitMQ Java client library before you can use the VoltDB connector. To install the RabbitMQ Java client library:

1. Download the client library version 3.3.4 or later from the RabbitMQ website (http://www.rabbitmq.com/java-client.html).
2. Copy the client JAR file into the `lib/extension/` folder where VoltDB is installed for each node in the cluster.

When the RabbitMQ connector receives data from the VoltDB export streams, it establishes a connection to the RabbitMQ exchange as a *producer*. It then writes records to the service using the optional exchange name and routing key suffix. RabbitMQ uses the routing key to identify which queue the data is sent to. The exchange examines the routing key and based on the key value (and any filters defined for the exchange) sends each message to the appropriate queue.

Every message sent by VoltDB to RabbitMQ contains a routing key that includes the name of the export stream. You can further refine the routing by appending a suffix to the stream name, based on the contents of individual stream columns. By default, the value of the export stream's partitioning column is used as

a suffix for the routing key. Alternately, you can specify a different column for each stream by declaring the routing.key.suffix property as a list of stream and column name pairs, separating the stream from the column name with a period and separating the pairs with commas. For example:

```
<export>
   <configuration target="queue" enabled="true" type="rabbitmq">
     <property name="broker.host">rabbitmq.mycompany.com</property>
     <property name="routing.key.suffix">
         voter_export.state,contestants_export.contestant_number
     </property>
  </configuration>
</export>
```

The important point to remember is that it is your responsibility to configure a RabbitMQ exchange that matches the name associated with the exchange.name property (or take the default exchange) and create queues and/or filters to match the routing keys generated by VoltDB. At a minimum, the exchange must be able to handle routing keys starting with the export stream names. This can be achieved by using a filter for each export stream. For example, using the flight example in Section 15.2, “Planning your Export Strategy”, you can create filters for EXPORT_FLIGHT.*, EXPORT_CUSTOMER.*, and RESERVATION_FINAL.*.

Table 15.5, “RabbitMQ Export Properties” lists the supported properties for the RabbitMQ connector.

Table 15.5. RabbitMQ Export Properties

Property	Allowable Values	Description
broker.host*	string	The host name of a RabbitMQ exchange server.
broker.port	integer	The port number of the RabbitMQ server. The default port number is 5672.
amqp.uri	string	An alternate method for specifying the location of the RabbitMQ exchange server. Use of amqp.uri allows you to specify additional RabbitMQ options as part of the connection URI. Either `broker.host` or `amqp.uri` must be specified.
virtual.host	string	Specifies the namespace for the RabbitMQ exchange and queues.
username	string	The username for authenticating to the RabbitMQ host.
password	string	The password for authenticating to the RabbitMQ host.
exchange.name	string	The name of the RabbitMQ exchange to use. If you do not specify a value, the default exchange for the RabbitMQ server is used.
routing.key.suffix	{*stream*}.{*column*} [,...]	Specifies which stream columns to use as a suffix for the RabbitMQ routing key. The routing key always starts with the stream name, in uppercase. A suffix is then appended to the stream name, separated by a period. By default, the value of the stream's partitioning column is used as the suffix. Using this property you can specify a list of stream column names, where the stream name and column name are separated by a period and the list of stream references is separated by commas. This syntax allows you to specify a different routing key suffix for each stream.
queue.durable	true, false	Whether the RabbitMQ queue is durable. That is, data in the queue will be retained and restarted if the RabbitMQ server restarts. If you

Property	Allowable Values	Description
		specify the queue as durable, the messages themselves will also be marked as durable to enable their persistence across server failure. The default is true.
binaryencoding	hex, base64	Specifies whether VARBINARY data is encoded in hexadecimal or BASE64 format. The default is hexadecimal.
skipinternals	true, false	Specifies whether to include six columns of VoltDB metadata (such as transaction ID and timestamp) in the output. If you specify skip-internals as true, the output contains only the exported stream data. The default is false.
timezone	string	The time zone to use when formatting the timestamp. Specify the time zone as a Java timezone identifier. The default is GMT.

*Required

15.11. The Elasticsearch Connector

The Elasticsearch connector receives serialized data from the export streams and inserts it into an Elasticsearch server or cluster. Elasticsearch is an open-source full-text search engine built on top of Apache Lucene™. By exporting selected streams from your VoltDB database to Elasticsearch you can perform extensive full-text searches on the data not possible with VoltDB alone.

Before using the Elasticsearch connector, we recommend reading the Elasticsearch documentation and becoming familiar with the software. The instructions in this section assume a working knowledge of Elasticsearch, its configuration and its capabilities.

The only required property when configuring Elasticsearch is the endpoint, which identifies the location of the Elasticsearch server and what index to use when inserting records into the target system. The structure of the Elasticsearch endpoint is the following:

```
<protocol>://<server>:<port>//<index-name>//<type-name>
```

For example, if the target Elasticsearch service is on the server `esearch.lan` using the default port (9200) and the exported records are being inserted into the `employees` index as documents of type `person`, the endpoint declaration would look like this:

```
<property name="endpoint">
   http://esearch.lan:9200/employees/person
</property>
```

You can use placeholders in the endpoint that are replaced at runtime with information from the export data, such as the stream name (%t), the partition ID (%p), the export generation (%g), and the date and hour (%d). For example, to use the stream name as the index name, the endpoint might look like the following:

```
<property name="endpoint">
   http://esearch.lan:9200/%t/person
</property>
```

When you export to Elasticsearch, the export connector creates the necessary index names and types in Elasticsearch (if they do not already exist) and inserts each record as a separate document with the appropriate metadata. Table 15.6, “Elasticsearch Export Properties” lists the supported properties for the Elasticsearch connector.

Table 15.6. Elasticsearch Export Properties

Property	Allowable Values	Description
endpoint*	string	Specifies the root URL of the RESTful interface for the Elasticsearch cluster to which you want to export the data. The endpoint should include the protocol, host name or IP address, port, and path. The path is assumed to include an index name and a type identifier. The export connector will use the Elasticsearch RESTful API to communicate with the server and insert records into the specified locations. You can use placeholders to replace portions of the endpoint with data from the exported records at runtime, including the stream name (%t), the partition ID (%p), the date and hour (%d), and the export generation (%g).
batch.mode	true, false	Specifies whether to send multiple rows as a single request or send each export row separately. The default is true.
timezone	string	The time zone to use when formatting timestamps. Specify the time zone as a Java timezone identifier. The default is the local time zone.

*Required

15.12. Understanding Import

Just as VoltDB can export data from selected streams to various targets, it supports importing data to selected tables from external sources. Import works in two ways:

- One-time import of data using one of several data loading utilities VoltDB provides. These data loaders support multiple standard input protocols and can be run from any server, even remotely from the database itself.
- Streaming import as part of the database server process. For data that is imported on an ongoing basis, use of the built-in import functionality ensures that import starts and stops with the database.

The following sections discuss these two approaches to data import.

15.12.1. One-Time Import Using Data Loading Utilities

Often, when migrating data from one database to another or when pre-loading a set of data into VoltDB as a starting point, you just want to perform the import once and then use the data natively within VoltDB. For these one-time uses, VoltDB provides separate data loader utilities that you can run once and then stop.

Each data loader supports a different source format. You can load data from text files — such as comma-separated value (CSV) files — using the csvloader utility. You can load data from another JDBC-compliant database using the jdbcloader utility. Or you can load data from a streaming message service with the Kafka loader utility, kafkaloader.

All of the data loaders operate in much the same way. For each utility you specify the source for the import and either a table that the data will be loaded into or a stored procedure that will be used to load

the data. So, for example, to load records from a CSV file named staff.csv into the table EMPLOYEES, the command might be the following:

```
$ csvloader employees --file=staff.csv
```

If instead you are copying the data from a JDBC-compliant database, the command might look like this:

```
$ jdbcloader employees \
     --jdbcurl=jdbc:postgresql://remotesvr/corphr \
     --jdbctable=employees \
     --jdbcdriver=org.postgresql.Driver
```

Each utility has arguments unique to the data source (such as `--jdbcurl`) that allow you to properly configure and connect to the source. See the description of each utility in Appendix D, *VoltDB CLI Commands* for details.

15.12.2. Streaming Import Using Built-in Import Features

If importing data is an ongoing business process, rather than a one-time event, then it is desirable to make it an integral part of the database system. This can be done by building a custom application to push data into VoltDB using one of its standard APIs, such as the JDBC interface. Or you can take advantage of VoltDB's built-in import infrastructure.

The built-in importers work in much the same way as the data loading utilities, where incoming data is written into one or more database tables using an existing stored procedure. The difference is that the built-in importers start automatically whenever the database starts and stop when the database stops, making import an integral part of the database process.

You configure the built-in importers in the deployment file the same way you configure export connections. Within the <import> element, you declare each import stream using separate <configuration> elements. Within the <configuration> tag you use attributes to specify the type and format of data being imported and whether the import configuration is enabled or not. Then enclosed within the <configuration> tags you use <property> elements to provide information required by the specific importer. For example:

```
<import>
  <configuration type="kafka" format="csv" enabled="true">
    <property name="brokers">kafkasvr:9092</property>
    <property name="topics">employees</property>
    <property name="procedure">EMPLOYEE.insert</property>
  </configuration>
</import>
```

Note

For the initial release of built-in importers, Kafka is the only supported import type.

VoltDB currently provides support for only one type of import: kafka. VoltDB also provides support for two import formats: comma-separated values (csv) and tab-separated values (tsv). Command-separated values are the default format. So if you are using CSV-formatted input, you can leave out the format attribute, as in the following examples.

When the database starts, the import infrastructure starts any enabled configurations. If you are importing multiple streams to separate tables through separate procedures, you must include multiple configurations, even if they come from the same source. For example, the following configuration imports data from two Kafka topics from the same Kafka servers into separate VoltDB tables.

```
<import>
  <configuration type="kafka" enabled="true">
    <property name="brokers">kafkasvr:9092</property>
    <property name="topics">employees</property>
    <property name="procedure">EMPLOYEE.insert</property>
  </configuration>
  <configuration type="kafka" enabled="true">
    <property name="brokers">kafkasvr:9092</property>
    <property name="topics">managers</property>
    <property name="procedure">MANAGER.insert</property>
  </configuration>
</import>
```

The following section describes the Kafka importer in more detail.

15.12.2.1. The Kafka Importer

The Kafka importer connects to the specified Kafka messaging service and imports one or more Kafka topics and writes the records into the VoltDB database. The data is decoded according to the specified format — comma-separated values (CSV) by default — and is inserted into the VoltDB database using the specified stored procedure.

You must specify the following properties for each configuration:

- **brokers** — Identifies one or more Kafka brokers. That is, servers hosting the Kafka service and desired topics. Specify a single server or a comma-separated list of brokers.
- **topics** — Identifies the Kafka topics that will be imported. The property value can be a single topic name or a comma-separated list of topics.
- **procedure** — Identifies the stored procedure that is invoked to insert the records into the VoltDB database.

When import starts, the importer first checks to make sure the specified stored procedure exists in the database schema. If not (for example, when you first create a database and before a schema is loaded), the importer issues periodic warnings to the console.

Once the specified stored procedure is declared, the importer looks for the specified Kafka brokers and topics. If the specified brokers cannot be found or the specified topics do not exist on the brokers, the importer reports an error and stops. You will need to restart import once this error condition is corrected. You can restart import using any of the following methods:

- Stop and restart or recover the database
- Pause and resume the database using the **voltadmin pause** and **voltadmin resume** commands
- Update the deployment file using the **voltadmin update** command

If the brokers are found and the topics exist, the importer starts fetching data from the Kafka topics and submitting it to the stored procedure to insert into the database. In the simplest case, you can use the default insert procedure for a table to insert records into a single table. For more complex data you can write your own import stored procedure to interpret the data and insert it into the appropriate table(s).

Table 15.7, “Kafka Import Properties” lists the allowable properties for the Kafka importer.

Table 15.7. Kafka Import Properties

Property	Allowable Values	Description
brokers*	string	A comma-separated list of Kafka brokers.
procedure*	string	The stored procedure to invoke to insert the incoming data into the database.
topics*	string	A comma-separated list of Kafka topics.
fetch.message.max.bytes	integer	The maximum size, in bytes, of the message that is fetched from Kafka. The Kafka default for this property is 64 Kilobytes.
groupid	string	A user-defined name for the group that the client belongs to. Kafka maintains a single pointer for the current position within the stream for all clients in the same group. The default group ID is "voltdb". In the rare case where you have two or more databases importing data from the same Kafka brokers and topics, be sure to set this property to give each database a unique group ID and avoid the databases interfering with each other.
socket.timeout.ms	integer	The time, in milliseconds, before the socket times out if no response is received. The Kafka default for this property is 30,000 (30 seconds). If the socket times out when the importer first tries to connect to the brokers, import will stop. If it times out after the initial connection is made, the importer will retry the connection until it succeeds.

*Required

Appendix A. Supported SQL DDL Statements

This appendix describes the subset of the SQL Data Definition Language (DDL) that VoltDB supports when defining the schema for a VoltDB database. VoltDB also supports extensions to the standard syntax to allow for the declaration of stored procedures and partitioning information related to tables and procedures.

The following sections are not intended as a complete description of the standard SQL DDL. Instead, they summarize the subset of standard SQL DDL statements that are allowed when defining a VoltDB schema and any exceptions, extensions, or limitations that application developers should be aware of.

The supported standard SQL DDL statements are:

- ALTER TABLE
- CREATE INDEX
- CREATE TABLE
- CREATE VIEW

The supported VoltDB-specific extensions for declaring stored procedures, streams, and partitioning are:

- CREATE PROCEDURE AS
- CREATE PROCEDURE FROM CLASS
- CREATE ROLE
- CREATE STREAM
- DR TABLE
- DROP INDEX
- DROP PROCEDURE
- DROP ROLE
- DROP STREAM
- DROP TABLE
- DROP VIEW
- IMPORT CLASS
- PARTITION PROCEDURE
- PARTITION TABLE
- SET DR

ALTER TABLE

ALTER TABLE — Modifies an existing table definition.

Syntax

ALTER TABLE *table-name* DROP CONSTRAINT *constraint-name*

ALTER TABLE *table-name* DROP [COLUMN] *column-name* [CASCADE]

ALTER TABLE *table-name* DROP {PRIMARY KEY | LIMIT PARTITION ROWS}

ALTER TABLE *table-name* ADD {constraint-definition | column-definition [BEFORE *column-name*] }

ALTER TABLE *table-name* ALTER column-definition [CASCADE]

ALTER TABLE *table-name* ALTER [COLUMN] *column-name* SET {DEFAULT *value* | [NOT] NULL}

column-definition: [COLUMN] *column-name* datatype [DEFAULT *value*] [NOT NULL] [index-type]

constraint-definition: [CONSTRAINT *constraint-name*] { index-definition | limit-definition }

index-definition: {index-type} (*column-name* [,...])

limit-definition: LIMIT PARTITION ROWS *row-count*

index-type: PRIMARY KEY | UNIQUE | ASSUMEUNIQUE

Description

The ALTER TABLE modifies an existing table definition by adding, removing or modifying a column or constraint. There are several different forms of the ALTER TABLE statement, depending on what attribute you are altering (a column or a constraint) and how you are changing it. The key point to remember is that you only alter one item at a time. To change two columns or a column and a constraint, you need to issue two ALTER TABLE statements.

There are three ALTER TABLE operations:

- ALTER TABLE ADD
- ALTER TABLE DROP
- ALTER TABLE ALTER

The syntax of each statement depends on whether you are modifying a column or a constraint. You can ADD or DROP either a column or an index. However, you can ALTER columns only. To alter an existing constraint you must first drop the constraint and then ADD the new definition.

There are two forms of the ALTER TABLE DROP statement. You can drop a column or constraint by name or you can drop a PRIMARY KEY or LIMIT PARTITION ROWS constraint by identifying the type of constraint, since there is only one such constraint for any given table.

The syntax for the ALTER TABLE ADD statement uses the same syntax to define a new column or constraint as that used in the CREATE TABLE command. When adding columns you can also specify the BEFORE clause to specify where the new columns falls in the order of table columns. If you to not specify BEFORE, the column is added at the end of the list of columns.

The ALTER TABLE ALTER COLUMN statement also has two forms. You can alter the column by providing a complete replacement definition, similar to the ALTER TABLE ADD COLUMN statement, or you can alter a specific attribute using the ALTER TABLE ALTER COLUMN... SET syntax. Use SET DEFAULT to add or modify an existing default. Use SET DEFAULT NULL to remove an existing default. You can also use the SET clause to specify whether the column can be null (SET NULL) or must not contain a null value (SET NOT NULL).

Handling Dependencies

You can only alter tables if there are no dependencies on the table, column, or index that would be violated by the change. For example, you cannot drop the partitioning column from a partitioned table if there are stored procedures partitioned on that table and column as well. You must first drop the partitioned store procedures before dropping the column. Note that by dropping the partitioning column, you are also automatically changing the table into a replicated table.

The most common dependency is if the table already has data in it. You can add, delete, and (within reasonable bounds) modify the columns of a table with existing data as long as those columns are not named in an index, view, or PARTITION statement. If a column is referenced in a view or index, you can specify CASCADE when you drop the column to automatically drop the referring indexes and views.

When a table has records in it, data associated with dropped columns is deleted. Added columns are interpreted as null or filled in with the specified default value. (You cannot add a column that is defined as NOT NULL, but without a default, if the table has existing data in it.) You can even change the datatype of the column within reason. In other words, you can increase the size of the datatype (for example, from INTEGER to BIGINT) but you cannot decrease the size (say, from INTEGER to TINYINT) since some of the existing data may already violate the size constraint.

You can also add non-unique indexes to tables with existing data. However, you cannot add unique constraints (such as PRIMARY KEY) if data exists.

If a table has no records in it, you can make almost any changes you like to it assuming, again, there are no dependencies. You can add and remove unique constraints, add, remove, and modify columns, even change column datatypes at will.

However, if there are dependencies, such as stored procedure queries that reference a dropped or modified column, you may not be allowed to make the change. If there are such dependencies, it is often easier to do drop the stored procedures before making the changes then recreate the stored procedures afterwards.

Examples

The following example uses ALTER TABLE to drop a unique constraint, add a new column, and then recreate the constraint adding the new column.

```
ALTER TABLE Employee DROP CONSTRAINT UniqueNames;
ALTER TABLE Employee ADD COLUMN MiddleInitial VARCHAR(1);
ALTER TABLE Employee ADD CONSTRAINT UniqueNames
      UNIQUE (FirstName, MiddleInitial, LastName);
```

CREATE INDEX

CREATE INDEX — Creates an index for faster access to a table.

Syntax

```
CREATE [UNIQUE|ASSUMEUNIQUE] INDEX index-name
        ON table-name ( index-column [,...])
        [WHERE [NOT] boolean-expression [ {AND | OR} [NOT] boolean-expression]...]
```

Description

Creating an index on a table makes read access to the table faster when using the columns of the index as a key. Note that VoltDB creates an index automatically when you specify a constraint, such as a primary key, in the CREATE TABLE statement.

When you specify that the index is UNIQUE, VoltDB constrains the table to at most one row for each set of index column values. If an INSERT or UPDATE statement attempts to create a row where all the index column values match an existing indexed row, the statement fails.

Because the uniqueness constraint is enforced separately within each partition, only indexes on replicated tables or containing the partitioning column of partitioned tables can ensure global uniqueness for partitioned tables and therefore support the UNIQUE keyword.

If you wish to create an index on a partitioned table that acts like a unique index but does not include the partitioning column, use the keyword ASSUMEUNIQUE instead of UNIQUE. Assumed unique indexes are treated like unique indexes (VoltDB verifies they are unique within the current partition). However, it is your responsibility to ensure these indexes are actually globally unique. Otherwise, it is possible an index will generate a constraint violation during an operation that modifies the partitioning of the database (such as adding nodes on the fly or restoring a snapshot to a different cluster configuration).

The indexed items (*index-column*) are either columns of the specified table or expressions, including functions, based on the table. For example, the following statements index a table based on the calculated area and its distance from a set location:

```
CREATE INDEX areaofplot ON plot (width * height);
CREATE INDEX distancefrom49 ON plot ( ABS(latitude - 49) );
```

You can create a *partial index* by including a WHERE clause in the index definition. The WHERE clause limits the number of rows that get indexed. This is useful if certain columns in the index are not evenly distributed. For example, if you are not interested in records where a column is null, you can use a WHERE clause to exclude those records and optimize the size and performance of the index.

The partial index is utilized by the database when a query's WHERE clause contains the same condition as the partial index definition. A special case is if the index condition is `{column} IS NOT NULL`. In this situation, the index may be applied even in the query does not contain that exact condition, as long as the query contains a WHERE condition that *implies* the column is not null, such as `{column} > 0`.

By default, VoltDB creates a tree index. Tree indexes provide the best general performance for a wide range of operations, including exact value matches and queries involving a range of values, such as `SELECT ... WHERE Score > 1 AND Score < 10`.

If an index is used exclusively for exact matches (such as `SELECT ... WHERE MyHashColumn = 123`), it is possible to create a hash index instead. To create a hash index, include the string "hash" as part of the index name.

Examples

The following example creates two indexes on a single table. The first is, by default, a non-unique index based on the departure time The second is a unique index based on the columns for the airline and flight number.

```
CREATE INDEX flightTimeIdx ON FLIGHT ( departtime );
CREATE UNIQUE INDEX FlightKeyIdx ON FLIGHT ( airline, flightID );
```

You can also use functions in the index definition. For example, the following is an index based on the element *movie* within a JSON-encoded VARCHAR column named *favorites* and the member's ID.

```
CREATE INDEX FavoriteMovie ON MEMBER (
       FIELD( favorites, 'movie' ), memberID
);
```

The following example demonstrates the use of a partial index, by including a WHERE clause, to exclude records with a null column.

```
CREATE INDEX completed_tasks
    ON tasks (task_id, startdate, enddate)
    WHERE enddate IS NOT NULL;
```

CREATE PROCEDURE AS

CREATE PROCEDURE AS — Defines a stored procedure composed of a SQL query.

Syntax

```
CREATE PROCEDURE procedure-name
        [PARTITION ON TABLE table-name COLUMN column-name [PARAMETER position]]
        [ALLOW role-name [,...]]
        AS sql-statement

CREATE PROCEDURE procedure-name
        [PARTITION ON TABLE table-name COLUMN column-name [PARAMETER position]]
        [ALLOW role-name [,...]]
        AS ### source-code ### LANGUAGE GROOVY
```

Description

You must declare stored procedures as part of the schema to make them accessible at runtime. Use CREATE PROCEDURE AS when declaring stored procedures directly within the DDL statement. There are two forms of the CREATE PROCEDURE AS statement:

- The *SQL query form* supports a single SQL query statement in the AS clause. The SQL statement can contain question marks (?) as placeholders that are filled in at runtime with the arguments to the procedure call.
- The *embedded program code form* supports the inclusion of program code in the AS clause. The embedded program code is opened and closed by three pound signs (###) and followed by the LANGUAGE clause specifying the programming language in use. VoltDB currently supports Groovy as an embedded language. (Supported in compiled application catalogs only. See the appendix on Using Application Catalogs in the *VoltDB Administrator's Guide* for details.)

In both cases, the procedure name must follow the naming conventions for Java class names. For example, the name is case-sensitive and cannot contain any white space.

When creating single-partitioned procedures, you can either specify the partitioning in a separate PARTITION PROCEDURE statement or you can include the PARTITION ON clause in the CREATE PROCEDURE statement. Creating and partitioning stored procedures in a single statement is recommended because there are certain cases where procedures with complex queries must be partitioned and cannot be compiled without the partitioning information. For example, queries that join two partitioned tables must be run in a single-partitioned procedure and must join the tables on their partitioning columns.

Partitioning a stored procedure means that the procedure executes within a unique partition of the database. The partition in which the procedure executes is chosen at runtime based on the table and column specified by *table-name* and *column-name*. By default, VoltDB uses the first parameter to the stored procedure as the partitioning value. However, you can use the PARAMETER clause to specify a different parameter. The *position* value specifies the parameter position, counting from zero. (In other words, position 0 is the first parameter, position 1 is the second, and so on.)

The specified table must be a partitioned table and cannot be an export-only or replicated table.

If security is enabled at runtime, only those roles named in the ALLOW clause (or with the ALLPROC or ADMIN permissions) have permission to invoke the procedure. If security is not enabled at runtime, the ALLOW clause is ignored and all users have access to the stored procedure.

Examples

The following example defines a stored procedure, *CountUsersByCountry*, as a single SQL query with a placeholder for matching the *country* column:

```
CREATE PROCEDURE CountUsersByCountry AS
    SELECT COUNT(*) FROM Users WHERE country=?;
```

The next example restricts access to the stored procedure to only users with the *operator* role. It also partitions the stored procedure on the *userID* column of the *Accounts* table. Note that the PARAMETER clause is used since the userID is the second parameter to the procedure:

```
CREATE PROCEDURE ChangeUserPassword
    PARTITION ON TABLE Accounts COLUMN userID PARAMETER 1
    ALLOW operator
    AS UPDATE Accounts SET HashedPassword=? WHERE userID=?;
```

CREATE PROCEDURE FROM CLASS

CREATE PROCEDURE FROM CLASS — Defines a stored procedure associated with a Java class.

Syntax

```
CREATE PROCEDURE
        [PARTITION ON TABLE table-name COLUMN column-name [PARAMETER position]]
        [ALLOW role-name [,...]]
        FROM CLASS class-name
```

Description

You must declare stored procedures to make them accessible to client applications and the sqlcmd utility. CREATE PROCEDURE FROM CLASS lets you declare stored procedures that are written as Java classes.The *class-name* is the name of the Java class.

Before you declare the stored procedure, you must create, compile, and load the associated Java class. It is usually easiest to do this by compiling all of your Java stored procedures and packaging the resulting class files into a single JAR file that can be loaded once. For example:

```
$ javac -d ./obj src/procedures/*.java
$ jar  cvf  myprocs.jar –C obj .
$ sqlcmd
1> load classes myprocs.jar;
2> CREATE PROCEDURE FROM CLASS procedures.AddCustomer;
```

When creating single-partitioned procedures, you can either specify the partitioning in a separate PARTITION PROCEDURE statement or you can include the PARTITION ON clause in the CREATE PROCEDURE statement. Creating and partitioning stored procedures in a single statement is recommended because there are certain cases where procedures with complex queries must be partitioned and cannot be compiled without the partitioning information. For example, queries that join two partitioned tables must be run in a single-partitioned procedure and must join the tables on their partitioning columns.

Partitioning a stored procedure means that the procedure executes within a unique partition of the database. The partition in which the procedure executes is chosen at runtime based on the table and column specified by *table-name* and *column-name*. By default, VoltDB uses the first parameter to the stored procedure as the partitioning value. However, you can use the PARAMETER clause to specify a different parameter. The *position* value specifies the parameter position, counting from zero. (In other words, position 0 is the first parameter, position 1 is the second, and so on.)

The specified table must be a partitioned table and cannot be an export-only or replicated table.

If security is enabled at runtime, only those roles named in the ALLOW clause (or with the ALLPROC or ADMIN permissions) have permission to invoke the procedure. If security is not enabled at runtime, the ALLOW clause is ignored and all users have access to the stored procedure.

Example

The following example declares a stored procedure matching the Java class MakeReservation. Note that the class name includes its location within the current class path (in this case, as a child of *flight* and *procedures*). However, the name itself, *MakeReservation*, must be unique within the schema because at runtime stored procedures are invoked by name only.

```
CREATE PROCEDURE FROM CLASS flight.procedures.MakeReservation;
```

CREATE ROLE

CREATE ROLE — Defines a role and the permissions associated with that role.

Syntax

```
CREATE ROLE role-name [WITH permission [,...]]
```

Description

The CREATE ROLE statement defines a named role that can be used to assign access rights to specific procedures and functions. When security is enabled in the deployment file, the permissions assigned in the CREATE ROLE and CREATE PROCEDURE statements specify which users can access which functions.

Use the CREATE PROCEDURE statement to assign permissions to named roles for accessing specific stored procedures. The CREATE ROLE statement lets you assign certain generic permissions. The following table describes the permissions that can be assigned the WITH clause.

Permission	Description	Inherits
DEFAULTPROCREAD	Access to read-only default procedures (*TABLE*.select)	
DEFAULTPROC	Access to all default procedures (*TABLE*.select, *TABLE*.insert, *TABLE*.delete, *TABLE*.update, and *TABLE*.upsert)	DEFAULTPROCREAD
SQLREAD	Access to read-only ad hoc SQL queries (SELECT)	DEFAULTPROCREAD
SQL	Access to all ad hoc SQL queries and default procedures	SQLREAD, DEFAULTPROC
ALLPROC	Access to all user-defined stored procedures	
ADMIN	Full access to all system procedures, all user-defined procedures, as well as default procedures, ad hoc SQL, and DDL statements.	ALLPROC, DEFAULTPROC, SQL
Note: For backwards compatibility, the special permissions ADHOC and SYSPROC are still recognized. They are interpreted as synonyms for SQL and ADMIN, respectively.		

The generic permissions are denied by default. So you must explicitly enable them for those roles that need them. For example, if users assigned to the "interactive" role need to run ad hoc queries, you must explicitly assign that permission in the CREATE ROLE statement:

```
CREATE ROLE interactive WITH sql;
```

Also note that the permissions are additive. So if a user is assigned to one role that allows access to defaultproc but not allproc, but that user also is assigned to another role that allows allproc, the user has both permissions.

Example

The following example defines three roles — *admin*, *developer*, and *batch* — each with a different set of permissions:

```
CREATE ROLE admin WITH admin;
```

```
CREATE ROLE developer WITH sql, allproc;
CREATE ROLE batch WITH defaultproc;
```

CREATE STREAM

CREATE STREAM — Creates an output stream in the database.

Syntax

```
CREATE STREAM stream-name
        [PARTITION ON COLUMN column-name]
        [EXPORT TO TARGET export-target-name] (
        column-definition [,...]

);

column-definition: column-name datatype [DEFAULT value ] [ NOT NULL ]
```

Description

The CREATE STREAM statement defines a stream and its associated columns in the database. A stream can be thought of as a virtual table. It has the same structure as a table, consisting of a list of columns and supporting all the same datatypes (Table A.1, “Supported SQL Datatypes”) as tables. The columns have the same rules in terms of naming and size. You can also use the INSERT statement to insert data into the stream once it is defined.

The three differences between streams and tables are:

- No data is stored in the database for a stream, it is only used as a passthrough.
- Because no data is stored, you cannot SELECT, UPDATE, or DELETE the stream contents.
- No indexes or constraints (such as primary keys) are allowed on a stream.

Data inserted into the stream is not stored in the database. The stream is an ephemeral container used only for analysis and/or passing data through VoltDB to other systems via the export function.

Combining streams with views lets you perform summary analysis on data passing through VoltDB without having to store all of the underlying data. For example, you might want to know how many times users access a website and their most recent visit. But you do not need to store a record for each visit. In this case, you can create a stream, *visits*, to capture the event and a view, *visit_by_user*, to capture the cumulative data:

```
CREATE STREAM visits PARTITION ON COLUMN user_id (
    user_id BIGINT NOT NULL,
    login TIMESTAMP
);
CREATE VIEW visit_by_user
    ( user_id, total_visits, last_visit )
    AS SELECT user_id, COUNT(*), MAX(login)
       FROM visits GROUP BY user_id;
```

When creating a view on a stream, the stream must be partitioned and the partition column must appear in the view. Another special feature of views on streams is that, because there is no underlying data stored for the view, VoltDB lets you modify the views content manually by issuing UPDATE and DELETE statements on the view. (This ability to manipulate the view is only available for views on streams. You cannot UPDATE or DELETE a view on a table; you must modify the data in the underlying table instead.)

For example, if you only care about a daily rollup of visits, you can use DELETE with the stream name to clear the data at midnight every night:

```
DELETE FROM visit_by_user;
```

Or if you need to adjust the cumulative analysis to, say, "undo" an entry from a specific user, you can use UPDATE:

```
UPDATE visit_by_user
    SET total_visits = total_visits -1, last_visit = NULL
    WHERE user_id = ?;
```

Streams can also be used to export data out of VoltDB into other systems, such as Kafka, CSV files, and so on. To export data into another system, you start by declaring one or more streams defining the data that will be sent to the external system. In the CREATE STREAM statement you also specify the named target for the export:

```
CREATE STREAM visits
  EXPORT TO TARGET archive (
    user_id BIGINT NOT NULL,
    login TIMESTAMP
);
```

If no export targets are configured in the deployment file, inserting data into the *visits* stream has no effect. However, if the export target *archive* is enabled in the deployment file, then any data inserted into the stream is sent to the export connector for delivery to the configured destination. See Chapter 15, *Importing and Exporting Live Data* for more information on configuring export targets.

Finally, you can combine analysis with export by creating a stream with an export target and also creating a view on that stream. So in our earlier example, if we want to warehouse data about each visit but use VoltDB to perform the real-time summary analysis, we would add an export definition, along with the partitioning clause, to the CREATE STREAM statement for the *visits* stream:

```
CREATE STREAM visits
  PARTITION ON COLUMN user_id
  EXPORT TO TARGET warehouse (
    user_id BIGINT NOT NULL,
    login TIMESTAMP
);
```

Example

The following example defines a stream and a view on that stream. Note the use of the PARTITION ON clause to ensure the stream is partitioned, since it is being used in a view.

```
CREATE STREAM flightdata
  PARTITION ON COLUMN airport (
    flight_id BIGINT NOT NULL,
    airport VARCHAR(3) NOT NULL,
    passengers INTEGER,
    eta TIMSTAMP
);
CREATE VIEW all_flights
  (airport, flight_count, passenger_count)
  AS SELECT airport, count(*),sum(passengers)
```

```
FROM flightdata GROUP BY airport;
```

CREATE TABLE

CREATE TABLE — Creates a table in the database.

Syntax

```
CREATE TABLE table-name (
        column-definition [,...]
        [, constraint-definition [,...]]

);

column-definition: column-name datatype [DEFAULT value ] [ NOT NULL ] [index-type]

constraint-definition: [CONSTRAINT constraint-name] { index-definition | limit-definition }

index-definition: {index-type} (column-name [,...])

limit-definition: LIMIT PARTITION ROWS row-count [EXECUTE (delete-statement)]

index-type: PRIMARY KEY | UNIQUE | ASSUMEUNIQUE
```

Description

The CREATE TABLE statement creates a table and its associated columns in the database. The supported datatypes are described in Table A.1, “Supported SQL Datatypes”.

Table A.1. Supported SQL Datatypes

SQL Datatype	Equivalent Java Datatype	Description
TINYINT	byte	1-byte signed integer, -127 to 127[a]
SMALLINT	short	2-byte signed integer, -32,767 to 32,767
INTEGER	int	4-byte signed integer, -2,147,483,647 to 2,147,483,647
BIGINT	long	8-byte signed integer, -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807
FLOAT	double	8-byte numeric, $-(2-2^{-52})\cdot 2^{1023}$ to $(2-2^{-52})\cdot 2^{1023}$ (Note that values less than or equal to -1.7E+308 are interpreted as null.)
DECIMAL	BigDecimal	16-byte fixed scale of 12 and precision of 38, -99999999999999999999999999.999999999999 to 99999999999999999999999999.999999999999
GEOGRAPHY or GEOGRAPHY()		A geospatial region. The storage requirement for geospatial data varies depending on the geometry. The default maximum size in memory is 32768. However, you can specify a different value by specifying the maximum size (in bytes) in the dec-

SQL Datatype	Equivalent Java Datatype	Description
		laration. For example: GEOGRAPHY(80000). See the section on entering geospatial data in the *VoltDB Guide to Performance and Customization* for details.
GEOGRAPHY_POINT		A geospatial location identified by its latitude and longitude. Requires 16 bytes of storage.
VARCHAR()	String	Variable length text string, with a maximum length specified in either characters (the default) or bytes. To specify the length in bytes, use the BYTES keyword after the length value. For example: VARCHAR(28 BYTES).
VARBINARY()	byte array	Variable length binary string (sometimes referred to as a "blob") with a maximum length specified in bytes
TIMESTAMP	long, VoltDB TimestampType	Time in microseconds

[a]For integer and floating-point datatypes, VoltDB reserves the largest possible negative value to denote a null value. For example -128 is interpreted as null for TINYINT, -32768 for SMALLINT, and so on.

The following limitations are important to note when using the CREATE TABLE statement in VoltDB:

- CHECK and FOREIGN KEY constraints are not supported.
- VoltDB does not support AUTO_INCREMENT, the automatic incrementing of column values.
- Each column has a maximum size of one megabyte and the total declared size of all of the columns in a table cannot exceed two megabytes. For VARCHAR columns where the length is specified in characters, the declared size is calculated as four bytes per character to allow for the longest potential UTF-8 string.
- If you intend to use a column to partition a table, that column cannot contain null values. You must specify NOT NULL in the definition of the column or VoltDB issues an error when compiling the schema.
- When you specify an index constraint, by default VoltDB creates a tree index. You can explicitly create a hash index by including the string "hash" as part of the index name. For example, the following declaration creates a hash index, `Version_Hash_Idx`, of three numeric columns.

```
CREATE TABLE Version (
    Major SMALLINT NOT NULL,
    Minor SMALLINT NOT NULL,
    baselevel INTEGER NOT NULL,
    ReleaseDate TIMESTAMP,
    CONSTRAINT Version_Hash_Idx PRIMARY KEY
        (Major, Minor, Baselevel)
);
```

  See the description of CREATE INDEX for more information on the difference between hash and tree indexes.

- To specify an index — either for an individual column or as a table constraint — that is globally unique across the database, use the standard SQL keywords UNIQUE and PRIMARY KEY. However, for

partitioned tables, VoltDB can only ensure uniqueness if the index includes the partitioning column. Otherwise, these keywords are not allowed.

It can be a performance advantage to define indexes or constraints on non-partitioning columns that you, as the developer, know are going to contain unique values. Although VoltDB cannot ensure uniqueness across the entire database, it does allow you to define indexes that are assumed to be unique by using the ASSUMEUNIQUE keyword.

When you define an index on a partitioned table as ASSUMEUNIQUE, VoltDB verifies uniqueness within the current partition when creating an index entry. However, it is your responsibility as developer or administrator to ensure that the values are actually globally unique. If the database is repartitioned due to adding new nodes or restoring a snapshot to a different cluster configuration, non-unique ASSUME-UNIQUE index entries may collide. When this occurs it results in a constraint violation error and the database will not be able to complete its current action.

Therefore, ASSUMEUNIQUE should be used with caution. Also, it is not necessary and should not be used with replicated tables or indexes that contain the partitioning column, which can be defined as UNIQUE.

- VoltDB includes a special constraint, LIMIT PARTITION ROWS, that limits the number of rows of data that can be inserted into any one partition for the table. This constraint is useful for managing memory usage and avoiding accidentally running out of memory due to unbalanced partitions or unexpected data growth.

  Note that the limit, specified as an integer, limits the number of rows per partition, not for the table as a whole. In the case of replicated tables, where each partition contains all rows of the table, the limit applies equally to the table as a whole and each partition. Also, the constraint is applied to INSERT operations. The constraint is not enforced when restoring a snapshot, altering the table declaration, or rebalancing the cluster as part of elastically adding nodes. In these cases, ignoring the limit allows the operation to succeed even if, as a result, a partition ends up containing more rows than specified by the LIMIT PARTITION ROWS constraint. But once the limit has been exceeded, any attempt to INSERT more table rows into that partition will result in an error, until sufficient rows are deleted to reduce the row count below the limit.

  As part of the LIMIT PARTITION ROWS constraint, you can optionally include an EXECUTE clause that specifies a DELETE statement to be executed when an INSERT statement will exceed the partition's row limit. For example, assume the *events* table has the following constraint as part of the CREATE TABLE statement:

```
CREATE TABLE events (
   event_time TIMESTAMP NOT NULL,
   event_code INTEGER NOT NULL.
   event_message VARCHAR(128),
   LIMIT PARTITION ROWS 1000 EXECUTE (
     DELETE FROM events WHERE
        SINCE_EPOCH(second,NOW) - SINCE_EPOCH(second,event_time) > 24*3600
   )
);
```

  At runtime, If an INSERT statement would result in the the current partition having more than 1000 rows, the delete statement will automatically be executed in an attempt to reduce the row count before the INSERT statement is run. In the example, any records with an *event_time* older than 24 hours will be deleted. Note that it is your responsibility as the query designer to provide a DELETE statement that is both deterministic and likely to remove sufficient rows to allow the query to succeed. Several important points to note about the EXECUTE clause:

    - If the DELETE statement does not delete sufficient rows, the INSERT statement will fail. For example, in the previous example, if you attempt to insert more than 1000 rows into a single partition in a 24 hour period, the DELETE statement will not delete enough records when you attempt to insert the 1001st record.
    - The LIMIT PARTITION ROWS constraint is applied per partition. That is, the DELETE statement is executed as a single-partitioned query in the partition where the INSERT statement triggers the row limit constraint, even if the INSERT statement is part of a multi-partitioned stored procedure.

- The length of VARCHAR columns can be specified in either characters (the default) or bytes. To specify the length in bytes, include the BYTES keyword after the length value; for example VARCHAR(16 BYTES).

  Specifying the VARCHAR length in characters is recommended because UTF-8 characters can require a variable number of bytes to store. By specifying the length in characters you can be sure the column has sufficient space to store any string of the specified length. Specifying the length in bytes is only recommended when all values contain only single byte (ASCII) characters or when conserving space is required and the strings are less than 64 bytes in length.

- The VARBINARY datatype provides variable storage for arbitrary strings of binary data and operates similarly to VARCHAR(n BYTES) strings. You assign byte arrays to a VARBINARY column when passing in variables, or you can use a hexidecimal string for assigning literal values in the SQL statement.

- The VoltDB TIMESTAMP datatype is a long integer representing the number of microseconds since the epoch. Two important points to note about this timestamp:

    - The VoltDB TIMESTAMP is not the same as the Java Timestamp datatype or traditional Linux time measurements, which are measured in milliseconds rather than microseconds. Appropriate conversion is needed when casting values between a VoltDB TIMESTAMP and other timestamp datatypes.
    - The VoltDB TIMESTAMP is interpreted as a Greenwich Meantime (GMT) value. Depending on how time values are created, their value may or may not account for the local machine's default time zone. Mixing timestamps from different time zones (for example, in WHERE clause comparisons) can result in unexpected behavior.

- For TIMESTAMP columns, you can define a default value using the NOW or CURRENT_TIMESTAMP keywords in place of a specific value. For example:

  ```
  CREATE TABLE Event (
      Event_Id INTEGER UNIQUE NOT NULL,
      Event_Timestamp TIMESTAMP DEFAULT NOW,
      Event_Description VARCHAR(128)
  );
  ```

  The default value is evaluated at runtime as an approximation, in milliseconds, of when the transaction begins execution.

Example

The following example defines a table with five columns. The first column, *Company*, is not allowed to be null, which is important since it is used as the partitioning column in the following PARTITION TABLE statement. That column is also contained in the PRIMARY KEY constraint. Again, it is important to include the partitioning column in any fully unique indexes for partitioned tables.

```
CREATE TABLE Inventory (
```

```
    Company VARCHAR(32) NOT NULL,
    ProductID BIGINT NOT NULL,
    Price DECIMAL,
    Category VARCHAR(32),
    Description VARCHAR(256),
    PRIMARY KEY (Company, ProductID)
);
PARTITION TABLE Inventory ON COLUMN Company;
```

CREATE VIEW

CREATE VIEW — Creates a view into a table, optimizing access to a summary of its contents.

Syntax

```
CREATE VIEW view-name ( view-column-name [,...] )
        AS SELECT { column-name | selection-expression } [AS alias] [,...]
        FROM table-name
        [WHERE [NOT] boolean-expression [ {AND | OR} [NOT] boolean-expression]...]
        [GROUP BY { column-name | selection-expression } [,...]]
```

Description

The CREATE VIEW statement creates a view of a table with selected columns and aggregates. VoltDB implements views as materialized views. In other words, the view is stored as a special table in the database and is updated each time the corresponding database table is modified. This means there is a small, incremental performance impact for any inserts or updates to the table, but selects on the view will execute efficiently.

The following limitations are important to note when using the CREATE VIEW statement with VoltDB:

- Views are allowed on individual tables only. Joins are not supported.
- The SELECT statement must include a field specified as COUNT(*). Other aggregate functions (COUNT, MAX, MIN, and SUM) are allowed following the COUNT(*).
- If the SELECT statement contains a GROUP BY clause, all of the columns and expressions listed in the GROUP BY must be listed in the same order at the start of the SELECT statement.

Examples

The following example defines a view that counts the number of records for a specific product item grouped by its location (that is, the warehouse the item is in).

```
CREATE VIEW inventory_count_by_warehouse (
     productID,
     warehouse,
     total_inventory
) AS SELECT
     productID,
     warehouse,
     COUNT(*)
FROM inventory GROUP BY productID, warehouse;
```

The next example uses a WHERE clause but no GROUP BY to provide a count and minimum and maximum aggregates of all records that meet a certain criteria.

```
CREATE VIEW small_towns ( number, minimum, maximum )
    AS SELECT count(*), min(population), max(population)
    FROM TOWNS WHERE population < 10000;
```

DR TABLE

DR TABLE — Identifies a table as a participant in database replication (DR)

Syntax

```
DR TABLE table-name [DISABLE]
```

Description

The DR TABLE statement identifies a table as a participant in database replication (DR). If DR is not enabled, the DR TABLE statement has no effect on the operation of the table or the database as a whole. However, once DR is enabled and if the current cluster is the master database for the DR operation, any updates to the contents of tables identified in the DR TABLE statement are copied and applied to the replica database as well.

The DR TABLE ... DISABLE statement reverses the effect of a previous DR TABLE statement, removing the specified table from participation in DR. Because the replica database schema must have DR TABLE statements for any tables being replicated by the master, if DR is actively occurring you must add the DR TABLE statements to the replica before adding them to the master. In reverse, you must issue DR TABLE... DISABLE statements on the master before you issue the matching statements on the replica.

See Chapter 11, *Database Replication* for more information about how database replication works.

Examples

The following example identifies the tables *Employee* and *Department* as participants in database replication.

```
DR TABLE Employee;
DR TABLE Department;
```

DROP INDEX

DROP INDEX — Removes an index.

Syntax

```
DROP INDEX index-name [IF EXISTS]
```

Description

The DROP INDEX statement deletes the specified index, and any data associated with it, from the database. The IF EXISTS clause allows the statement to succeed even if the specified index does not exist. If the index does not exist and you *do not* include the IF EXISTS clause, the statement will return an error.

You must use the name of the index as specified in the original DDL when dropping the index. You cannot drop an index if it was not explicitly named in the CREATE INDEX command. This is why you should always name indexes and other constraints wherever possible.

Examples

The following example removes the index named employee_idx_by_lastname:

```
DROP INDEX Employee_idx_by_lastname;
```

DROP PROCEDURE

DROP PROCEDURE — Removes the definition of a stored procedure.

Syntax

```
DROP PROCEDURE procedure-name [IF EXISTS]
```

Description

The DROP PROCEDURE statement deletes the definition of the named stored procedure. Note that, for procedures declared using CREATE PROCEDURE FROM and a class file, the statement does *not* delete the class that implements the procedure, it only deletes the definition and any partitioning information associated with the procedure. To remove the associated stored procedure class, you must first drop the procedure definition then use the sqlcmd **remove classes** directive to remove the class.

The IF EXISTS clause allows the statement to succeed even if the specified procedure name does not exist. If the stored procedure does not exist and you *do not* include the IF EXISTS clause, the statement will return an error.

Examples

The following example removes the definition of the FindCanceledReservations stored procedure, then uses **remove classes** to remove the corresponding class.

```
$ sqlcmd
1> DROP PROCEDURE FindCanceledReservations;
2> remove classes "*.FindCanceledReservations";
```

DROP ROLE

DROP ROLE — Removes a role.

Syntax

```
DROP ROLE role-name [IF EXISTS]
```

Description

The DROP ROLE statement deletes the specified role. The IF EXISTS clause allows the statement to succeed even if the specified role does not exist. If the role does not exist and you *do not* include the IF EXISTS clause, the statement will return an error.

Examples

The following example removes the role named debug:

```
DROP ROLE debug;
```

DROP STREAM

DROP STREAM — Removes a stream and, optionally, any views associated with it.

Syntax

```
DROP STREAM stream-name [IF EXISTS] [CASCADE]
```

Description

The DROP STREAM statement deletes the specified stream from the database. The IF EXISTS clause allows the statement to succeed even if the specified stream does not exist. If the stream does not exist and you *do not* include the IF EXISTS clause, the statement will return an error.

If you use the CASCADE clause, VoltDB automatically drops any referencing views as well as the stream itself.

Example

The following example uses DROP STREAM with the IF EXISTS clause to remove the MeterReadings stream definition.

```
DROP STREAM MetterReadings IF EXISTS;
```

DROP TABLE

DROP TABLE — Removes a table and any data associated with it.

Syntax

```
DROP TABLE table-name [IF EXISTS] [CASCADE]
```

Description

The DROP TABLE statement deletes the specified table, and any data associated with it, from the database. The IF EXISTS clause allows the statement to succeed even if the specified tables does not exist. If the table does not exist and you *do not* include the IF EXISTS clause, the statement will return an error.

Before dropping a table, you must first remove any stored procedures that reference the table. For example, if the table EMPLOYEE is partitioned and the stored procedure AddEmployee is partitioned on the EMPLOYEE table, you must drop the procedure first before dropping the table:

```
PARTITION TABLE Employee ON COLUMN EmpID;
PARTITION PROCEDURE AddEmployee
   ON TABLE Employee COLUMN EmpID;

 [. . . ]

DROP PROCEDURE AddEmployee;
DROP TABLE Employee;
```

Attempting to drop the table before dropping the procedure will result in an error. The same will normally happen if there are any views or indexes that reference the table. However, if you use the CASCADE clause VoltDB will automatically drop any referencing indexes and views as well as the table itself.

Examples

The following example uses DROP TABLE with the IF EXISTS clause to remove any existing MailAddress table definition and data before adding a new definition.

```
DROP TABLE UserSignin IF EXISTS;
CREATE TABLE UserSignin (
   userID BIGINT NOT NULL,
   lastlogin TIMESTAMP DEFAULT NOW
);
```

DROP VIEW

DROP VIEW — Removes a view and any data associated with it.

Syntax

```
DROP VIEW view-name [IF EXISTS]
```

Description

The DROP VIEW statement deletes the specified view, and any data associated with it, from the database. The IF EXISTS clause allows the statement to succeed even if the specified view does not exist. If the view does not exist and you *do not* include the IF EXISTS clause, the statement will return an error.

Dropping a view has the same constraints as dropping a table, in that you cannot drop a view that is referenced by existing stored procedure queries. Before dropping the view, you must drop any stored procedures that reference it.

Examples

The following example removes the view named Votes_by_state:

```
DROP VIEW votes_by_state;
```

IMPORT CLASS

IMPORT CLASS — Specifies additional Java classes to include in the application catalog.

Syntax

```
IMPORT CLASS class-name
```

Description

Warning: Deprecated

The IMPORT CLASS statement is only valid when precompiling a schema into an application catalog. However, use of precompiled catalogs, and the IMPORT CLASS statement, are deprecated. When using interactive DDL to enter your schema, use the sqlcmd **load classes** directive instead.

The IMPORT CLASS statement lets you specify class files to be added to the application catalog when the schema is compiled. You can include individual class files only; the IMPORT CLASS statement does not extract classes from JAR files. However, you can use Ant-style wildcards in the class specification to include multiple classes. For example:

```
IMPORT CLASS org.mycompany.utils.*;
```

Use the IMPORT CLASS statement to include reusable code that is accessed by multiple stored procedures. Any classes and methods called by stored procedures must follow the same rules for deterministic behavior that stored procedures follow, as described in Section 5.1.2, “VoltDB Stored Procedures are Deterministic”.

Code imported using IMPORT CLASS is included in the application catalog and, therefore, can be updated on a running database through the @UpdateApplicationCatalog system procedure. For static libraries that stored procedures use but that do not need to be modified often, the recommended approach is to include the code by placing JAR files in the /lib directory where VoltDB is installed on the database servers.

Example

The following example imports a class containing common financial algorithms so they can be used by any stored procedures in the catalog:

```
IMPORT CLASS org.mycompany.common.finance;
```

PARTITION PROCEDURE

PARTITION PROCEDURE — Specifies that a stored procedure is partitioned.

Syntax

```
PARTITION PROCEDURE procedure-name ON TABLE table-name COLUMN column-name
    [PARAMETER position ]
```

Description

Partitioning a stored procedure means that the procedure executes within a unique partition of the database. The partition in which the procedure executes is chosen at runtime based on the table and column specified by *table-name* and *column-name* and the value of the first parameter to the procedure. For example:

```
PARTITION TABLE Employees ON COLUMN BadgeNumber;
PARTITION PROCEDURE FindEmployee ON TABLE Employees COLUMN BadgeNumber;
```

The procedure FindEmployee is partitioned on the table Employees, and table Employees is in turn partitioned on the column BadgeNumber. This means that when the stored procedure FindEmployee is invoked VoltDB determines which partition to run the stored procedure in based on the value of the first parameter to the procedure and the corresponding partitioning value for the column BadgeNumber. So to find the employee with badge number 145303 you would invoke the stored procedure as follows:

```
clientResponse response = client.callProcedure("FindEmployee", 145303);
```

By default, VoltDB uses the first parameter to the stored procedure as the partitioning value. However, if you want to use the value of a different parameter, you can use the PARAMETER clause. The PARAMETER clause specifies which procedure parameter to use as the partitioning value, with *position* specifying the parameter position, counting from zero. (In other words, position 0 is the first parameter, position 1 is the second, and so on.)

The specified table must be a partitioned table and cannot be an export-only or replicated table.

You specify the procedure by its simplified class name. Do *not* include any other parts of the class path. Note that the simple procedure name you specify in the PARTITION PROCEDURE may be different than the class name you specify in the CREATE PARTITION statement, which can include a relative path. For example, if the class for the stored procedure is mydb.procedures.FindEmployee, the procedure name in the PARTITION PROCEDURE statement should be FindEmployee:

```
CREATE PROCEDURE FROM CLASS mydb.procedures.FindEmployee;
PARTITION PROCEDURE FindEmployee ON TABLE Employees COLUMN BadgeNumber;
```

Examples

The following example declares a stored procedure, using an inline SQL query, and then partitions the procedure on the *Customer* table, Note that the PARTITION PROCEDURE statement includes the PARAMETER clause, since the partitioning column is not the first of the placeholders in the SQL query. Also note that the PARTITION argument is zero-based, so the value "1" identifies the second placeholder.

```
CREATE PROCEDURE GetCustomerByName AS
    SELECT *  from Customer WHERE FirstName=? AND  LastName = ?
    ORDER BY LastName, FirstName, CustomerID;
```

```
PARTITION PROCEDURE GetCustomerByName
    ON TABLE Customer COLUMN LastName
    PARAMETER 1;
```

The next example declares a stored procedure as a Java class. Since the first argument to the procedure's run method is the value for the *LastName* column, The PARTITION PROCEDURE statement does not require a POSITION clause and can use the default.

```
CREATE PROCEDURE FROM CLASS org.mycompany.ChangeCustomerAddress;

PARTITION PROCEDURE ChangeCustomerAddress
    ON TABLE Customer COLUMN LastName;
```

PARTITION TABLE

PARTITION TABLE — Specifies that a table is partitioned and which is the partitioning column.

Syntax

```
PARTITION TABLE table-name ON COLUMN column-name
```

Description

Partitioning a table specifies that different records are stored in different unique partitions, based on the value of the specified column. The table *table-name* and column *column-name* must be valid, declared elements in the current DDL file or VoltDB generates an error when compiling the schema.

For a table to be partitioned, the partitioning column must be declared as NOT NULL. If you do not declare a partitioning column of a table in the DDL, the table is assumed to be a replicated table.

Example

The following example partitions the table *Employee* on the column *EmployeeID*.

```
PARTITION TABLE Employee on COLUMN EmployeeID;
```

SET DR

SET DR — Enables the use of Cross Datacenter Replication (XDCR).

Syntax

```
SET DR= {ACTIVE | PASSIVE}
```

Description

The SET DR statements enables and disables Cross Datacenter Replication (XDCR). You actually turn on database replication in the deployment file using the `<dr>` and `<connection>` elements. But to use two-way, active replication, you must also enable it in the database schema using the SET DR=ACTIVE statement for both databases involved in the XDCR process. See Chapter 11, *Database Replication* for more information about XDCR.

By default, only passive DR is enabled in the schema. By specifying SET DR=ACTIVE you enable the use of XDCR. When enabled, XDCR assigns an additional 8 bytes per row for every DR table in the database. The additional space is used to store metadata about the row's most recent transaction.

For example, say your schema contains 5 tables which you declare as DR tables and those tables will store a million rows each. This means the database will consume approximately 40 megabytes of additional memory when XDCR is enabled, *even if DR is not yet initiated in the deployment file*. Which is why the SET DR=ACTIVE statement should only be used for databases that will be involved in active XDCR.

If use of XDCR is enabled in the schema, you can use the SET DR=PASSIVE statement to disable it. Note, however, for both the SET DR=ACTIVE and SET DR=PASSIVE statements, any tables declared as DR tables must be empty when the SET DR statement is executed.

Examples

The following example enables the use of XDCR and then declares three tables as DR tables. Because any DR tables must be empty when the SET DR statement is executed, it is often easiest to place the statement at the beginning of the schema.

```
SET DR=ACTIVE;
DR TABLE Employees;
DR TABLE Divisions;
DR TABLE Locations;
```

Appendix B. Supported SQL Statements

This appendix describes the SQL syntax that VoltDB supports in stored procedures and ad hoc queries.

This is not intended as a complete description of the SQL language and how it operates. Instead, it summarizes the subset of standard SQL statements that are allowed in VoltDB and any exceptions or limitations that application developers should be aware of.

The supported SQL statements are:

- DELETE
- INSERT
- SELECT
- TRUNCATE TABLE
- UPDATE
- UPSERT

DELETE

DELETE — Deletes one or more records from the database.

Syntax

```
DELETE FROM table-name
        [WHERE [NOT] boolean-expression [ {AND | OR} [NOT] boolean-expression]...]
        [ORDER BY {column-name [ ASC | DESC ]}[,...] [LIMIT integer] [OFFSET integer]]
```

Description

The DELETE statement deletes rows from the specified table that meet the constraints of the WHERE clause. The following limitations are important to note when using the DELETE statement in VoltDB:

- The DELETE statement can operate on only one table at a time (no joins or subqueries).
- The WHERE expression supports the boolean operators: equals (=), not equals (!= or <>), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), IS NULL, AND, OR, and NOT. Note, however, although OR is supported syntactically, VoltDB does not optimize these operations and use of OR may impact the performance of your queries.
- The ORDER BY clause lets you order the selection results and then select a subset of the ordered records to delete. For example, you could delete only the five oldest records, chronologically, sorting by timestamp:

  ```
  DELETE FROM events ORDER BY event_time ASC LIMIT 5;
  ```

  Similarly, you could choose to keep only the five most recent:

  ```
  DELETE FROM events ORDER BY event_time DESC OFFSET 5;
  ```

- When using ORDER BY, the resulting sort order must be deterministic. In other words, the ORDER BY must include enough columns to uniquely identify each row. (For example, listing all columns or a primary key.)
- You cannot use ORDER BY to delete rows from a partitioned table in a multi-partitioned query. In other words, for partitioned tables DELETE... ORDER BY must be executed as part of a single-partitioned stored procedure or as an ad hoc query with a WHERE clause that uniquely identifies the partitioning column value.

Examples

The following example removes rows from the EMPLOYEE table where the EMPLOYEE_ID column is equal to 145303.

```
DELETE FROM employee WHERE employee_id = 145303;
```

The following example removes rows from the BID table where the BIDDERID is 12345 and the BID-PRICE is less than 100.00.

```
DELETE FROM bid WHERE bidderid=12345 AND bidprice<100.0;
```

INSERT

INSERT — Creates new rows in the database, using the specified values for the columns.

Syntax

INSERT INTO *table-name* [(*column-name* [,...])] VALUES (*value-expression* [,...])

INSERT INTO *table-name* [(*column-name* [,...])] SELECT *select-expression*

Description

The INSERT statement creates one or more new rows in the database. There are two forms of the INSERT statement, INSERT INTO... VALUES and INSERT INTO... SELECT. The INSERT INTO... VALUES statement lets you enter specific values for a adding a single row to the database. The INSERT INTO... SELECT statement lets you insert multiple rows into the database, depending upon the number of rows returned by the select expression.

The INSERT INTO... SELECT statement is often used for copying rows from one table to another. For example, say you want to export all of the records associated with a particular column value. The following INSERT statement copies all of the records from the table ORDERS with a warehouseID of 25 into the table EXPORT_ORDERS:

```
INSERT INTO Export_Orders SELECT * FROM Orders WHERE CustomerID=25;
```

However, the select expression can be more complex, including joining multiple tables. The following limitations currently apply to the INSERT INTO... SELECT statement:

- INSERT INTO... SELECT can join partitioned tables only if they are joined on equality of the partitioning columns. Also, the resulting INSERT must apply to a partitioned table and be inserted using the same partition column value, whether the query is executed in a single-partitioned or multi-partitioned stored procedure.
- INSERT INTO... SELECT does not support UNION statements.

In addition to the preceding limitations, there are certain instances where the select expression is too complex to be processed. Cases of invalid select expressions in INSERT INTO... SELECT include:

- A LIMIT or TOP clause applied to a partitioned table in a multi-partitioned query
- A GROUP BY of a partitioned table where the partitioning column is not in the GROUP BY clause

Deterministic behavior is critical to maintaining the integrity of the data in a K-safe cluster. Because an INSERT INTO... SELECT statement performs both a query and an insert based on the results of that query, if the selection expression would produces non-deterministic results, the VoltDB query planner rejects the statement and returns an error. See Section 5.1.2, “VoltDB Stored Procedures are Deterministic” for more information on the importance of determinism in SQL queries.

If you specify the column names following the table name, the values will be assigned to the columns in the order specified. If you do not specify the column names, values will be assigned to columns based on the order specified in the schema definition. However, if you specify a subset of the columns, you must specify values for any columns that are explicitly defined in the schema as NOT NULL and do not have a default value assigned.

Examples

The following example inserts values into the columns (firstname, mi, lastname, and emp_id) of an EMPLOYEE table:

```
INSERT INTO employee VALUES ('Jane', 'Q', 'Public', 145303);
```

The next example performs the same operation with the same results, except this INSERT statement explicitly identifies the column names and changes the order:

```
INSERT INTO employee (emp_id, lastname, firstname, mi)
       VALUES (145303, 'Public', 'Jane', 'Q');
```

The last example assigns values for the employee ID and the first and last names, but not the middle initial. This query will only succeed if the MI column is nullable or has a default value defined in the database schema.

```
INSERT INTO employee (emp_id, lastname, firstname)
       VALUES (145304, 'Doe', 'John');
```

SELECT

SELECT — Fetches the specified rows and columns from the database.

Syntax

```
Select-statement [{set-operator} Select-statement ] ...

Select-statement:
        SELECT [ TOP integer-value ]
        { * | [ ALL | DISTINCT ] { column-name | selection-expression } [AS alias] [,...] }
        FROM { table-reference } [ join-clause ]...
        [WHERE [NOT] boolean-expression [ {AND | OR} [NOT] boolean-expression]...]
        [clause...]

table-reference:
        { table-name [AS alias] | view-name [AS alias] | sub-query AS alias }

sub-query:
        (Select-statement)

join-clause:
        ,table-reference
        [INNER | {LEFT | RIGHT} [OUTER]] JOIN [{table-reference}] [join-condition]

join-condition:
        ON conditional-expression
        USING (column-reference [,...])

clause:
        ORDER BY { column-name | alias } [ ASC | DESC ] [,...]
        GROUP BY { column-name | alias } [,...]
        HAVING boolean-expression
        LIMIT integer-value [OFFSET row-count]

set-operator:
        UNION [ALL]
        INTERSECT [ALL]
        EXCEPT
```

Description

The SELECT statement retrieves the specified rows and columns from the database, filtered and sorted by any clauses that are included in the statement. In its simplest form, the SELECT statement retrieves the values associated with individual columns. However, the selection expression can be a function such as COUNT and SUM.

The following features and limitations are important to note when using the SELECT statement with VoltDB:

- See Appendix C, *SQL Functions* for a full list of the SQL functions the VoltDB supports.
- VoltDB supports the following operators in expressions: addition (+), subtraction (-), multiplication (*), division (*) and string concatenation (||).

- `TOP` *n* is a synonym for `LIMIT` *n*.

- The WHERE expression supports the boolean operators: equals (=), not equals (!= or <>), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), LIKE, IS NULL, IS DISTINCT, IS NOT DISTINCT, AND, OR, and NOT. Note, however, although OR is supported syntactically, VoltDB does not optimize these operations and use of OR may impact the performance of your queries.

- The boolean expression LIKE provides text pattern matching in a VARCHAR column. The syntax of the LIKE expression is `{string-expression} LIKE '{pattern}'` where the pattern can contain text and wildcards, including the underscore (_) for matching a single character and the percent sign (%) for matching zero or more characters. The string comparison is case sensitive.

  Where an index exists on the column being scanned and the pattern starts with a text prefix (rather than starting with a wildcard), VoltDB will attempt to use the index to maximize performance, For example, a query limiting the results to rows from the EMPLOYEE table where the primary index, the JOB_CODE column, begins with the characters "Temp" looks like this:

  ```
  SELECT * from EMPLOYEE where JOB_CODE like 'Temp%';
  ```

- The boolean expression IN determines if a given value is found within a list of alternatives. For example, in the following code fragment the IN expression looks to see if a record is part of Hispaniola by evaluating whether the column COUNTRY is equal to either "Dominican Republic" or "Haiti":

  ```
  WHERE Country IN ('Dominican Republic', 'Haiti')
  ```

  Note that the list of alternatives must be enclosed in parentheses. The result of an IN expression is equivalent to a sequence of equality conditions separated by OR. So the preceding code fragment produces the same boolean result as:

  ```
  WHERE Country='Dominican Republic' OR Country='Haiti'
  ```

  The advantages are that the IN syntax provides more compact and readable code and can provide improved performance by using an index on the initial expression where available.

- The boolean expression BETWEEN determines if a value falls within a given range. The evaluation is inclusive of the end points. In this way BETWEEN is a convenient alias for two boolean expressions determining if a value is greater than or equal to (>=) the starting value and less than or equal to (<=) the end value. For example, the following two WHERE clauses are equivalent:

  ```
  WHERE salary BETWEEN ? AND ?
  WHERE salary >= ? AND salary <= ?
  ```

- The boolean expressions IS DISTINCT FROM and IS NOT DISTINCT FROM are similar to the equals ("=") and not equals ("<>") operators respectively, except when evaluating null operands. If either or both operands are null, the equals and not equals operators return a boolean null value, or *false*. IS DISTINCT FROM and IS NOT DISTINCT FROM consider null a valid operand. So if only one operand is null IS DISTINCT FROM returns *true* and IS NOT DISTINCT FROM returns *false*. If both operands are null IS DISTINCT FROM returns *false* and IS NOT DISTINCT FROM returns *true*.

- When using placeholders in SQL statements involving the IN list expression, you can either do replacement of individual values within the list or replace the list as a whole. For example, consider the following statements:

  ```
  SELECT * from EMPLOYEE where STATUS IN (?, ?,?);
  SELECT * from EMPLOYEE where STATUS IN ?;
  ```

In the first statement, there are three parameters that replace individual values in the IN list, allowing you to specify exactly three selection values. In the second statement the placeholder replaces the entire list, including the parentheses. In this case the parameter to the procedure call must be an array and allows you to change not only the values of the alternatives but the number of criteria considered.

The following Java code fragment demonstrates how these two queries can be used in a stored procedure, resulting in equivalent SQL statements being executed:

```
String arg1 = "Salary";
String arg2 = "Hourly";
String arg3 = "Parttime";
voltQueueSQL( query1, arg1, arg2, arg3);

String listargs[] = new String[3];
listargs[0] = arg1;
listargs[1] = arg2;
listargs[2] = arg3;
voltQueueSQL( query2, (Object) listargs);
```

Note that when passing arrays as parameters in Java, it is a good practice to explicitly cast them as an object to avoid the array being implicitly expanded into individual call parameters.

- VoltDB supports the use of CASE-WHEN-THEN-ELSE-END for conditional operations. For example, the following SELECT expression uses a CASE statement to return different values based on the contents of the price column:

```
SELECT Prod_name,
    CASE WHEN price > 100.00
          THEN 'Expensive'
          ELSE 'Cheap'
    END
FROM products ORDER BY Prod_name;
```

  For more complex conditional operations with multiple alternatives, use of the DECODE() function is recommended.

- VoltDB supports both inner and outer joins.
- The SELECT statement supports subqueries as a table reference in the FROM clause. Subqueries must be enclosed in parentheses and must be assigned a table alias. Note that subqueries are only supported in the SELECT statement; they cannot be used in data manipulation statements such UPDATE or DELETE.
- You can only join two or more partitioned tables if those tables are partitioned on the same value and joined on equality of the partitioning column. Joining two partitioned tables on non-partitioned columns or on a range of values is not supported. However, there are no limitations on joining to replicated tables.
- Extremely large result sets (greater than 50 megabytes in size) are not supported. If you execute a SELECT statement that generates a result set of more than 50 megabytes, VoltDB will return an error.

Subqueries

The SELECT statement can include subqueries. Subqueries are separate SELECT statements, enclosed in parentheses, where the results of the subquery are used as values, expressions, or arguments within the surrounding SELECT statement.

Subqueries, like any SELECT statement, are extremely flexible and can return a wide array of information. A subquery might return:

- A single row with a single column — this is sometimes known as a *scalar subquery* and represents a single value
- A single row with multiple columns — this is also known as a *row value expression*
- Multiple rows with one or more columns

In general, VoltDB supports subqueries in the FROM clause, in the selection expression, and in boolean expressions in the WHERE clause or in CASE-WHEN-THEN-ELSE-END operations. However, different types of subqueries are allowed in different situations, depending on the type of data returned.

- In the FROM clause, the SELECT statement supports all types of subquery as a table reference. The subquery must be enclosed in parentheses and must be assigned a table alias.
- In the selection expression, scalar subqueries can be used in place of a single column reference.
- In the WHERE clause and CASE operations, both scalar and non-scalar subqueries can be used as part of boolean expressions. Scalar subqueries can be used in place of any single-valued expression. Non-scalar subqueries can be used in the following situations:
  - **Row value comparisons** — Boolean expressions that compare one row value expression to another can use subqueries that resolve to one row with multiple columns. For example:

    ```
    select * from t1
       where (a,c) > (select a, c from t2 where b=t1.b);
    ```

  - **IN and EXISTS** — Subqueries that return multiple rows can be used as an argument to the IN or EXISTS predicate to determine if a value (or set of values) exists within the rows returned by the subquery. For example:

    ```
    select * from t1
       where a in (select a from t2);
    select * from t1
       where (a,c) in (select a, c from t2 where b=t1.b);
    select * from t1 where c > 3 and
       exists (select a, b from t2 where a=t1.a);
    ```

  - **ANY and ALL** — Multi-row subqueries can also be used as the target of an ANY or ALL comparison, using either a scalar or row expression comparison. For example:

    ```
    select * from t1
       where a > ALL (select a from t2);
    select * from t1
       where (a,c) = ANY (select a, c from t2 where b=t1.b);
    ```

Note that subqueries are only supported in the SELECT statement; they cannot be used in data manipulation statements such UPDATE or DELETE or in CREATE VIEW statements or index definitions. Also, VoltDB does not support subqueries in the HAVING, ORDER BY, or GROUP BY clauses.

For the initial release of subqueries in selection and boolean expressions, only replicated tables can be used in the subquery. Both replicated and partitioned tables can be used in subqueries in place of table references in the FROM clause.

Set Operations

VoltDB also supports the set operations UNION, INTERSECT, and EXCEPT. These keywords let you perform set operations on two or more SELECT statements. UNION includes the combined results sets from the two SELECT statements, INTERSECT includes only those rows that appear in both SELECT statement result sets, and EXCEPT includes only those rows that appear in one result set but not the other.

Normally, UNION and INTERSECT provide a set including unique rows. That is, if a row appears in both SELECT results, it only appears once in the combined result set. However, if you include the ALL modifier, all matching rows are included. For example, UNION ALL will result in single entries for the rows that appear in only one of the SELECT results, but two copies of any rows that appear in both.

The UNION, INTERSECT, and EXCEPT operations obey the same rules that apply to joins:

- You cannot perform set operations on SELECT statements that reference the same table.
- All tables in the SELECT statements must either be replicated tables or partitioned tables partitioned on the same column value, using equality of the partitioning column in the WHERE clause.

Examples

The following example retrieves all of the columns from the EMPLOYEE table where the last name is "Smith":

```
SELECT * FROM employee WHERE lastname = 'Smith';
```

The following example retrieves selected columns for two tables at once, joined by the employee_id using an implicit inner join and sorted by last name:

```
SELECT lastname, firstname, salary
    FROM employee AS e, compensation AS c
    WHERE e.employee_id = c.employee_id
    ORDER BY lastname DESC;
```

The following example includes both a simple SQL query defined in the schema and a client application to call the procedure repeatedly. This combination uses the LIMIT and OFFSET clauses to "page" through a large table, 500 rows at a time.

When retrieving very large volumes of data, it is a good idea to use LIMIT and OFFSET to constrain the amount of data in each transaction. However, to perform LIMIT OFFSET queries effectively, the database must include a tree index that encompasses all of the columns of the ORDER BY clause (in this example, the lastname and firstname columns).

Schema:

```
CREATE PROCEDURE EmpByLimit AS
       SELECT lastname, firstname FROM employee
       WHERE company = ?
       ORDER BY lastname ASC, firstname ASC
       LIMIT 500 OFFSET ?;

PARTITION PROCEDURE EmpByLimit ON TABLE Employee COLUMN Company;
```

Java Client Application:

```
long offset = 0;
```

```
String company = "ACME Explosives";
boolean alldone = false;
while ( ! alldone ) {
   VoltTable results[] = client.callProcedure("EmpByLimit",
                         company,offset).getResults();
   if (results[0].getRowCount() < 1) {
        // No more records.
        alldone = true;
   } else {
        // do something with the results.
   }
   offset += 500;
}
```

TRUNCATE TABLE

TRUNCATE TABLE — Deletes all records from the specified table.

Syntax

```
TRUNCATE TABLE table-name
```

Description

The TRUNCATE TABLE statement deletes all of the records from the specified table. TRUNCATE TABLE is the same as the statement `DELETE FROM {table-name}` with no selection clause. These statements contain optimizations to increase performance and reduce memory usage over an equivalent DELETE statement containing a WHERE selection clause.

The following behavior is important to remember when using the TRUNCATE TABLE statement in VoltDB:

- Executing a TRUNCATE TABLE query on a partitioned table within a single-partitioned stored procedure will only delete the records within the current partition. Records in other partitions will be unaffected.
- You cannot execute a TRUNCATE TABLE query on a replicated table from within a single-partition stored procedure. To truncate a replicated table you must execute the query within a multi-partition stored procedure or as an ad hoc query.

Examples

The following example removes all data from the CURRENT_STANDINGS table:

```
TRUNCATE TABLE Current_standings;
```

UPDATE

UPDATE — Updates the values within the specified columns and rows of the database.

Syntax

```
UPDATE table-name SET column-name = value-expression [, ...]
        [WHERE [NOT] boolean-expression [ {AND | OR} [NOT] boolean-expression]...]
```

Description

The UPDATE statement changes the values of columns within the specified records. The following limitations are important to note when using the UPDATE statement with VoltDB:

- VoltDB supports the following arithmetic operators in expressions: addition (+), subtraction (-), multiplication (*), and division (*).
- The WHERE expression supports the boolean operators: equals (=), not equals (!= or <>), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), IS NULL, AND, OR, and NOT. Note, however, although OR is supported syntactically, VoltDB does not optimize these operations and use of OR may impact the performance of your queries.

Examples

The following example changes the ADDRESS column of the EMPLOYEE record with an employee ID of 145303:

```
UPDATE employee
    SET address = '49 Lavender Sweep'
    WHERE employee_id = 145303;
```

The following example increases the starting price by 25% for all ITEM records with a category ID of 7:

```
UPDATE item SET startprice = startprice * 1.25 WHERE categoryid = 7;
```

UPSERT

UPSERT — Either inserts new rows or updates existing rows depending on the primary key value.

Syntax

UPSERT INTO *table-name* [(*column-name* [,...])] VALUES (*value-expression* [,...])

UPSERT INTO *table-name* [(*column-name* [,...])] SELECT *select-expression*

Description

The UPSERT statement has the same syntax as the INSERT statement and will perform the same function, assuming a record with a matching primary key does *not* already exist in the database. If such a record does exist, UPSERT updates the existing record with the new column values. Note that the UPSERT statement can only be executed on tables that have a primary key.

UPSERT has the same two forms as the INSERT statement: UPSERT INTO... VALUES and UPSERT INTO... SELECT. The UPSERT statement also has similar constraints and limitations as the INSERT statement with regards to joining partitioned tables and overly complex SELECT clauses. (See the description of the INSERT statement for details.)

However, UPSERT INTO... SELECT has an additional limitation: the SELECT statement must produce deterministically ordered results. That is, the query must not only produce the same rows, they must be in the same order to ensure the subsequent inserts and updates produce identical results.

Examples

The following examples use two tables, Employee and Manager, both of which define the column emp_id as a primary key. In the first example, the UPSERT statement either creates a new row with the specified values or updates an existing row with the primary key 145303.

```
UPSERT INTO employee (emp_id, lastname, firstname, title, department)
       VALUES (145303, 'Public', 'Jane', 'Manager', 'HR');
```

The next example copies records from the Employee table to the Manager table, if the employee's title is "Manager". Again, new records will be created or existing records updated depending on whether the employee already has a record in the Manager table. Notice the use of the primary key in an ORDER BY clause to ensure deterministic results from the SELECT statement.

```
UPSERT  INTO Manager (emp_id, lastname, firstname, title, department)
       SELECT * from Employee WHERE title='Manager' ORDER BY emp_id;
```

Appendix C. SQL Functions

Functions let you aggregate column values and perform other calculations and transformations on data within your SQL queries. This appendix lists the functions alphabetically, describing for each their syntax and purpose. The functions can also be grouped by the type of data they produce or operate on, as listed below.

Bitwise Function

- BIT_SHIFT_LEFT()
- BIT_SHIFT_RIGHT()
- BITAND()
- BITNOT()
- BITOR()
- BITXOR()

Column Aggregation Functions

- APPROX_COUNT_DISTINCT()
- AVG()
- COUNT()
- MAX()
- MIN()
- SUM()

Date and Time Functions

- CURRENT_TIMESTAMP
- DATEADD()
- DAY(), DAYOFMONTH()
- DAYOFWEEK()
- DAYOFYEAR()
- EXTRACT()
- FROM_UNIXTIME()
- HOUR()
- MINUTE()
- MONTH()
- NOW
- QUARTER()
- SECOND()
- SINCE_EPOCH()
- TO_TIMESTAMP()
- TRUNCATE()
- WEEK(), WEEKOFYEAR()
- WEEKDAY()
- YEAR()

Geospatial Functions

- AREA()
- ASTEXT()
- CENTROID()

- CONTAINS()
- DISTANCE()
- DWITHIN()
- ISINVALIDREASON()
- ISVALID()
- LATITUDE()
- LONGITUDE()
- NUMINTERIORRINGS()
- NUMPOINTS()
- POINTFROMTEXT()
- POLYGONFROMTEXT()
- VALIDPOLYGONFROMTEXT()

JSON Functions

- ARRAY_ELEMENT()
- ARRAY_LENGTH()
- FIELD()
- SET_FIELD()

Logic and Conversion Functions

- CAST()
- COALESCE()
- DECODE()

Math Function

- ABS()
- CEILING()
- EXP()
- FLOOR()
- LN(), LOG()
- MOD()
- PI()
- POWER()
- SQRT()

String Functions

- BIN()
- CHAR()
- CHAR_LENGTH()
- CONCAT()
- FORMAT_CURRENCY()
- HEX()
- LEFT()
- LOWER()
- OCTET_LENGTH()
- OVERLAY()
- POSITION()
- REGEXP_POSITION()
- REPEAT()
- REPLACE()

- RIGHT()
- SPACE()
- SUBSTRING()
- TRIM()
- UPPER()

ABS()

ABS() — Returns the absolute value of a numeric expression.

Syntax

ABS(*numeric-expression*)

Description

The ABS() function returns the absolute value of the specified numeric expression.

Example

The following example sorts the results of a SELECT expression by its proximity to a target value (specified by a placeholder), using the ABS() function to normalize values both above and below the intended target.

```
SELECT price, product_name FROM product_list
    ORDER BY ABS(price - ?) ASC;
```

APPROX_COUNT_DISTINCT()

APPROX_COUNT_DISTINCT() — Returns an approximate count of the number of distinct values for the specified column expression.

Syntax

```
APPROX_COUNT_DISTINCT( column-expression )
```

Description

The APPROX_COUNT_DISTINCT() function returns an approximation of the number of distinct values for the specified column expression. APPROX_COUNT_DISTINCT(*column-expression*) is an alternative to the SQL expression "`COUNT(DISTINCT column-expression)`".

The reason for using APPROX_COUNT_DISTINCT() is because it can be significantly faster and use less temporary memory than performing a precise COUNT DISTINCT operation. This is particularly true when calculating a distinct count of a partitioned table across all of the partitions. The approximation usually falls within ±1% of the actual count.

You can use the APPROX_COUNT_DISTINCT() function on column expressions of decimal, timestamp, or any size integer datatype. You cannot use the function on floating point (FLOAT) or variable length (VARCHAR and VARBINARY) columns.

Example

The following example returns an approximation of the number of distinct products available in each store.

```
SELECT store, APPROX_COUNT_DISTINCT(product_id) FROM catalog
    GROUP BY store ORDER BY store;
```

AREA()

AREA() — Returns the area of a polygon in square meters.

Syntax

AREA(*polygon*)

Description

The AREA() function returns the area of a GEOGRAPHY value in square meters. The area is the total area of the outer ring minus the area of any inner rings within the polygon. The area is returned as a FLOAT value.

Example

The following example calculates the sum of the areas of multiple polygons representing fields on a farm.

```
SELECT farmer, SUM(AREA(field)) FROM farm
    WHERE farmer = 'Old MacDonald' GROUP BY farmer;
```

ARRAY_ELEMENT()

ARRAY_ELEMENT() — Returns the element at the specified location in a JSON array.

Syntax

ARRAY_ELEMENT(*JSON-array*, *element-position*)

Description

The ARRAY_ELEMENT() function extracts a single element from a JSON array. The array position is zero-based. In other words, the first element in the array is in position "0". The function returns the element as a string. For example, the following function invocation returns the string "two":

```
ARRAY_ELEMENT('["zero","one","two","three"]',2)
```

Note that the array element is always returned as a string. So in the following example, the function returns "2" as a string rather than an integer:

```
ARRAY_ELEMENT('[0,1,2,3]',2)
```

Finally, the element may itself be a valid JSON-encoded object. For example, the following function returns the string "[0,1,2,3]":

```
ARRAY_ELEMENT('[[0,1,2,3],["zero","one","two","three"]]',0)
```

The ARRAY_ELEMENT() function can be combined with other functions, such as FIELD(), to traverse more complex JSON structures. The function returns a NULL value if any of the following conditions are true:

- The position argument is less than zero
- The position argument is greater than or equal to the length of the array
- The JSON string does not represent an array (that is, the string is a valid JSON scalar value or object)

The function returns an error if the first argument is not a valid JSON string.

Example

The following example uses the ARRAY_ELEMENT() function along with FIELD() to extract specific array elements from one field in a JSON-encoded VARCHAR column:

```
SELECT language,
       ARRAY_ELEMENT(FIELD(words,'colors'),1) AS color,
       ARRAY_ELEMENT(FIELD(words,'numbers'),2) AS number
       FROM world_languages WHERE language = 'French';
```

Assuming the column *words* has the following structure, the query returns the strings "French', "vert", and "trois".

```
{"colors":["rouge","vert","bleu"],
 "numbers":["un","deux","trois"]}
```

ARRAY_LENGTH()

ARRAY_LENGTH() — Returns the number of elements in a JSON array.

Syntax

```
ARRAY_LENGTH( JSON-array )
```

Description

The ARRAY_LENGTH() returns the length of a JSON array; that is, the number of elements the array contains. The length is returned as an integer.

The ARRAY_LENGTH() function can be combined with other functions, such as FIELD(), to traverse more complex JSON structures.

The function returns NULL if the argument is a valid JSON string but does not represent an array. The function returns an error if the argument is not a valid JSON string.

Example

The following example uses the ARRAY_LENGTH(), ARRAY_ELEMENT(), and FIELD() functions to return the last element of an array in a larger JSON string. The functions perform the following actions:

- Innermost, the FIELD() function extracts the JSON field "alerts", which is assumed to be an array, from the column *messages*.
- ARRAY_LENGTH() determines the number of elements in the array.
- ARRAY_ELEMENT() returns the last element based on the value of ARRAY_LENGTH() minus one (because the array positions are zero-based).

```
SELECT ARRAY_ELEMENT(FIELD(messages,'alerts'),
       ARRAY_LENGTH(FIELD(messages,'alerts'))-1) AS last_alert,
       station FROM reportlog
       WHERE station=?;
```

ASTEXT()

ASTEXT() — Returns the Well Known Text (WKT) representation of a GEOGRAPHY or GEOGRAPHY_POINT value.

Syntax

```
ASTEXT( polygon | point )
```

Description

The ASTEXT() function returns a text string containing a Well Known Text (WKT) representation of a GEOGRAPHY or GEOGRAPHY_POINT value. ASTEXT(*value*) produces the same results as calling CAST(*value* AS VARCHAR).

Note that ASTEXT() does not return the identical text string that was originally input using POINTFROMTEXT() or POLYGONFROMTEXT(). When geospatial data is converted from WKT to its internal representation, the string representations of longitude and latitude are converted to double floating point values. Rounding and differing levels of precision may result in small differences in the stored values. The use of spaces and capitalization may also vary between the original input strings and the computed output of the ASTEXT() function.

Examples

The following SELECT statement uses the ASTEXT() function to return the WKT representation of a GEOGRAPHY_POINT value in the column *location*.

```
SELECT, name, ASTEXT(location) FROM city
   WHERE state = 'NY' ORDER BY name;
```

AVG()

AVG() — Returns the average of a range of numeric column values.

Syntax

AVG(*column-expression*)

Description

The AVG() function returns the average of a range of numeric column values. The values being averaged depend on the constraints defined by the WHERE and GROUP BY clauses.

Example

The following example returns the average price for each product category.

```
SELECT AVG(price), category FROM product_list
    GROUP BY category ORDER BY category;
```

BIN()

BIN() — Returns the binary representation of a BIGINT value as a string.

Syntax

```
BIN( value )
```

Description

The BIN() function returns the binary representation of a BIGINT value as a string. The function will return the shortest valid string representation, truncating any preceding zeros (except in the case of the value zero, which is returned as the string "0").

Example

The following example use the BIN and BITAND functions to return the binary representations of two BIGINT values and their binary intersection.

```
$ sqlcmd
1> create table bits (a bigint, b bigint);
2> insert into bits values(55,99);
3> select bin(a) as int1, bin(b) as int2,
4>     bin(bitand(a,b)) as intersection from bits;
INT1     INT2      INTERSECTION
-------- --------- -------------
110111   1100011   100011
```

BIT_SHIFT_LEFT()

BIT_SHIFT_LEFT() — Shifts the bits of a BIGINT value to the left a specified number of places.

Syntax

```
BIT_SHIFT_LEFT( value, offset )
```

Description

The BIT_SHIFT_LEFT() function shifts the bit values of a BIGINT value to the left the number of places specified by *offset*. The offset must be a positive integer value. The unspecified bits to the right are padded with zeros. So, for example, if the offset is 5, the left-most 5 bits are dropped, the remaining bits are shifted 5 places to the left, and the right-most 5 bits are set to zero. The result is returned as a new BIGINT value — the arguments to the function are not modified.

The left-most bit of an integer number is the sign bit, but has no special meaning for bitwise operations. However, The left-most bit set to 1 followed by all zeros is reserved as the NULL value. If you use a NULL value as an argument, you will receive a NULL response. But in all other circumstances (using non-NULL BIGINT arguments), the bitwise functions should never return a NULL result. Consequently any bitwise operation that would result in only the left-most bit being set, will generate an error at runtime.

Examples

The following example shifts the bits in a BIGINT value three places to the left and displays the hexadecimal representation of both the initial value and the resulting value.

```
$ sqlcmd
1> create table bits (a bigint);
2> insert into bits values (112);
3> select hex(a), hex(bit_shift_left(a,3)) from bits;
C1       C2
-------- ---------
70       380
```

BIT_SHIFT_RIGHT()

BIT_SHIFT_RIGHT() — Shifts the bits of a BIGINT value to the right a specified number of places.

Syntax

```
BIT_SHIFT_RIGHT( value, offset )
```

Description

The BIT_SHIFT_RIGHT() function shifts the bit values of a BIGINT value to the right the number of places specified by *offset*. The offset must be a positive integer value. The unspecified bits to the left are padded with zeros. So, for example, if the offset is 5, the right-most 5 bits are dropped, the remaining bits are shifted 5 places to the right, and the left-most 5 bits are set to zero. The result is returned as a new BIGINT value — the arguments to the function are not modified.

The left-most bit of an integer number is the sign bit, but has no special meaning for bitwise operations. However, The left-most bit set to 1 followed by all zeros is reserved as the NULL value. If you use a NULL value as an argument, you will receive a NULL response. But in all other circumstances (using non-NULL BIGINT arguments), the bitwise functions should never return a NULL result. Consequently any bitwise operation that would result in only the left-most bit being set, will generate an error at runtime.

Examples

The following example shifts the bits in a BIGINT value three places to the right and displays the hexadecimal representation of both the initial value and the resulting value.

```
$ sqlcmd
1> create table bits (a bigint);
2> insert into bits values (112);
3> select hex(a), hex(bit_shift_right(a,3)) from bits;
C1       C2
-------- -------
70       E
```

BITAND()

BITAND() — Returns the mask of bits set in both of two BIGINT values

Syntax

```
BITAND( value, value )
```

Description

The BITAND() function returns the mask of bits set in both of two BIGINT integers. In other words, it performs a bitwise AND operation on the two arguments. The result is returned as a new BIGINT value — the arguments to the function are not modified.

The left-most bit of an integer number is the sign bit, but has no special meaning for bitwise operations. However, The left-most bit set to 1 followed by all zeros is reserved as the NULL value. If you use a NULL value as an argument, you will receive a NULL response. But in all other circumstances (using non-NULL BIGINT arguments), the bitwise functions should never return a NULL result. Consequently any bitwise operation that would result in only the left-most bit being set, will generate an error at runtime.

Examples

The following example writes values into two BIGINT columns of the table *bits* and then returns the bitwise AND of the columns:

```
$ sqlcmd
1> create table bits (a bigint, b bigint);
2> insert into bits (a,b) values (7,13);
3> select bitand(a,b) from bits;
C1
---
 5
```

BITNOT()

BITNOT() — Returns the mask reversing every bit of a BIGINT value.

Syntax

```
BITNOT( value )
```

Description

The BITNOT() function returns the mask reversing every bit in a BIGINT value. In other words, it performs a bitwise NOT operation, returning the complement of the argument. The result is returned as a new BIGINT value — the argument to the function is not modified.

The left-most bit of an integer number is the sign bit, but has no special meaning for bitwise operations. However, The left-most bit set to 1 followed by all zeros is reserved as the NULL value. If you use a NULL value as an argument, you will receive a NULL response. But in all other circumstances (using non-NULL BIGINT arguments), the bitwise functions should never return a NULL result. Consequently any bitwise operation that would result in only the left-most bit being set, will generate an error at runtime.

Examples

The following example writes a value into a BIGINT column of the table *bits* and then returns the bitwise NOT of the column:

```
$ sqlcmd
1> create table bits (a bigint);
2> insert into bits (a) values (1234567890);
3> select bitnot(a) from bits;
C1
------------
 -1234567891
```

BITOR()

BITOR() — Returns the mask of bits set in either of two BIGINT values

Syntax

BITOR(*value*, *value*)

Description

The BITOR) function returns the mask of bits set in either of two BIGINT integers. In other words, it performs a bitwise OR operation on the two arguments. The result is returned as a new BIGINT value — the arguments to the function are not modified.

The left-most bit of an integer number is the sign bit, but has no special meaning for bitwise operations. However, The left-most bit set to 1 followed by all zeros is reserved as the NULL value. If you use a NULL value as an argument, you will receive a NULL response. But in all other circumstances (using non-NULL BIGINT arguments), the bitwise functions should never return a NULL result. Consequently any bitwise operation that would result in only the left-most bit being set, will generate an error at runtime.

Examples

The following example writes values into two BIGINT columns of the table *bits* and then returns the bitwise OR of the columns:

```
$ sqlcmd
1> create table bits (a bigint, b bigint);
2> insert into bits (a,b) values (7,13);
3> select bitor(a,b) from bits;
C1
---
 15
```

BITXOR()

BITXOR() — Returns the mask of bits set in one but not both of two BIGINT values

Syntax

```
BITXOR( value, value )
```

Description

The BITXOR() function returns the mask of bits set in one but not both of two BIGINT integers. In other words, it performs a bitwise XOR operation on the two arguments. The result is returned as a new BIGINT value — the arguments to the function are not modified.

The left-most bit of an integer number is the sign bit, but has no special meaning for bitwise operations. However, The left-most bit set to 1 followed by all zeros is reserved as the NULL value. If you use a NULL value as an argument, you will receive a NULL response. But in all other circumstances (using non-NULL BIGINT arguments), the bitwise functions should never return a NULL result. Consequently any bitwise operation that would result in only the left-most bit being set, will generate an error at runtime.

Examples

The following example writes values into two BIGINT columns of the table *bits* and then returns the bitwise XOR of the columns:

```
$ sqlcmd
1> create table bits (a bigint, b bigint);
2> insert into bits (a,b) values (7,13);
3> select bitxor(a,b) from bits;
C1
---
 10
```

CAST()

CAST() — Explicitly converts an expression to the specified datatype.

Syntax

```
CAST( expression AS datatype )
```

Description

The CAST() function converts an expression to a specified datatype. Cases where casting is beneficial include when converting between numeric types (such as integer and float) or when converting a numeric value to a string.

All numeric datatypes can be used as the source and numeric or string datatypes can be the target. When converting from decimal values to integers, values are truncated. You can also cast from a TIMESTAMP to a VARCHAR or from a VARCHAR to a TIMESTAMP, assuming the text string is formatted as *YYYY-MM-DD* or *YYYY-MM-DD HH:MM:SS.nnnnnn*. Where the runtime value cannot be converted (for example, the value exceeds the maximum allowable value of the target datatype) an error is thrown.

You cannot use VARBINARY as either the target or the source datatype. To convert between numeric and TIMESTAMP values, use the TO_TIMESTAMP(), FROM_UNIXTIME(), and EXTRACT() functions.

The result of the CAST() function of a null value is the corresponding null in the target datatype.

Example

The following example uses the CAST() function to ensure the result of an expression is also a floating point number and does not truncate the decimal portion.

```
SELECT contestant, CAST( (votes * 100) as FLOAT) / ? as percentage
    FROM contest ORDER BY votes, contestant
```

CEILING()

CEILING() — Returns the smallest integer value greater than or equal to a numeric expression.

Syntax

```
CEILING( numeric-expression )
```

Description

The CEILING() function returns the next integer greater than or equal to the specified numeric expression. In other words, the CEILING() function "rounds up" numeric values. For example:

```
CEILING(3.1415) = 4
CEILING(2.0) = 2
CEILING(-5.32) = -5
```

Example

The following example uses the CEILING function to calculate the shipping costs for a product based on its weight in the next whole number of pounds.

```
SELECT shipping.cost_per_lb * CEILING(product.weight),
       product.prod_id FROM product, shipping
       ORDER BY product.prod_id;
```

CENTROID()

CENTROID() — Returns the central point of a polygon.

Syntax

```
CENTROID( polygon )
```

Description

The CENTROID() returns the central point of a GEOGRAPHY polygon. The centroid is the point where any line passing through the centroid divides the polygon into two segments of equal area. The return value of the CENTROID() function is a GEOGRAPHY_POINT value.

Note that the centroid may fall outside of the polygon itself. For example, if the polygon is a ring (that is, a circle with an inner circle removed) or a horseshoe shape.

Example

The following example uses the CENTROID() and LATITUDE() functions to return a list of countries where the majority of the land mass falls above the equator.

```
SELECT name, capital FROM country
   WHERE LATITUDE(CENTROID(outline)) > 0
   ORDER BY name, capital;
```

CHAR()

CHAR() — Returns a string with a single UTF-8 character associated with the specified character code.

Syntax

CHAR(*integer*)

Description

The CHAR() function returns a string containing a single UTF-8 character that matches the specified UNICODE character code. One use of the CHAR() function is to insert non-printing and other hard to enter characters into string expressions.

Example

The following example uses CHAR() to add a copyright symbol into a VARCHAR field.

```
UPDATE book SET copyright_notice= CHAR(169) || CAST(? AS VARCHAR)
       WHERE isbn=?;
```

CHAR_LENGTH()

CHAR_LENGTH() — Returns the number of characters in a string.

Syntax

```
CHAR_LENGTH( string-expression )
```

Description

The CHAR_LENGTH() function returns the number of text characters in a string.

Note that the number of characters and the amount of physical space required to store those characters can differ. To measure the length of the string, in bytes, use the OCTET_LENGTH() function.

Example

The following example returns the string in the column LastName as well as the number of characters and length in bytes of that string.

```
SELECT LastName, CHAR_LENGTH(LastName), OCTET_LENGTH(LastName)
    FROM Customers ORDER BY LastName, FirstName;
```

COALESCE()

COALESCE() — Returns the first non-null argument, or null.

Syntax

COALESCE(*expression* [, ...])

Description

The COALESCE() function takes multiple arguments and returns the value of the first argument that is not null, or — if all arguments are null — the function returns null.

Examples

The following example uses COALESCE to perform two functions:

- Replace possibly null column values with placeholder text
- Return one of several column values

In the second usage, the SELECT statement returns the value of the column State, Province, or Territory depending on the first that contains a non-null value. Or the function returns a null value if none of the columns are non-null.

```
SELECT lastname, firstname,
   COALESCE(address,'[address unkown]'),
   COALESCE(state, province, territory),
   country FROM users ORDER BY lastname;
```

CONCAT()

CONCAT() — Concatenates two or more strings and returns the result.

Syntax

CONCAT(*string-expression* { , ... })

Description

The CONCAT() function concatenates two or more strings and returns the resulting string. The string concatenation operator || performs the same function as CONCAT().

Example

The following example concatenates the contents of two columns as part of a SELECT expression.

```
SELECT price, CONCAT(category,part_name) AS full_part_name
   FROM product_list ORDER BY price;
```

The next example does something similar but uses the || operator as a shorthand to concatenate three strings, two columns and a string constant, as part of a SELECT expression.

```
SELECT lastname || ', ' || firstname AS full_name
   FROM customers ORDER BY lastname, firstname;
```

CONTAINS()

CONTAINS() — Returns true or false depending if a point falls within the specified polygon.

Syntax

CONTAINS(*polygon*, *point*)

Description

The CONTAINS() function determines if a given point falls within the specified GEOGRAPHY polygon. If so, the function returns a boolean value of true. If not, it returns false.

Example

The following example uses the CONTAINS function to see if a specific user is with the boundaries of a city or not by evaluating if the user.location GEOGRAPHY_POINT column value falls within the polygon defined by the city.boundary GEOGRAPHY column.

```
SELECT user.name, user.id, city.name FROM user, city
    WHERE user.id = ? AND CONTAINS(city.boundary,user.location);
```

COUNT()

COUNT() — Returns the number of rows selected containing the specified column.

Syntax

COUNT(*column-expression*)

Description

The COUNT() function returns the number of rows selected for the specified column. Since the actual value of the column is not used to calculate the count, you can use the asterisk (*) as a wildcard for any column. For example the query `SELECT COUNT(*) FROM widgets` returns the number of rows in the table `widgets`, without needing to know what columns the table contains.

The one case where the column name is significant is if you use the DISTINCT clause to constrain the selection expression. For example, `SELECT COUNT(DISTINCT last_name) FROM customer` returns the count of unique last names in the customer table.

Examples

The following example returns the number of rows where the product name starts with the captial letter A.

```
SELECT COUNT(*) FROM product_list
    WHERE product_name LIKE 'A%';
```

The next example returns the total number of unique product categories in the product list.

```
SELECT COUNT(DISTINCT category) FROM product_list;
```

CURRENT_TIMESTAMP

CURRENT_TIMESTAMP — Returns the current time as a timestamp value.

Syntax

```
CURRENT_TIMESTAMP
```

Description

The CURRENT_TIMESTAMP function returns the current time as a VoltDB timestamp. The value of the timestamp is determined when the query or stored procedure is invoked. Several important aspects of how the CURRENT_TIMESTAMP function operates are:

- The value returned is guaranteed to be identical for all partitions that execute the query.
- The value returned is measured in milliseconds then padded to create a timestamp value in microseconds.
- During command logging, the returned value is stored as part of the log, so when the command log is replayed, the same value is used during the replay of the query.
- Similarly, for database replication (DR) the value returned is passed and reused by the replica database when replaying the query.
- You can specify CURRENT_TIMESTAMP as a default value in the CREATE TABLE statement when defining the schema of a VoltDB database.
- The CURRENT_TIMESTAMP function *cannot* be used in the CREATE INDEX or CREATE VIEW statements.

The NOW and CURRENT_TIMESTAMP functions are synonyms and perform an identical function.

Example

The following example uses CURRENT_TIMESTAMP in the WHERE clause to delete alert events that occurred in the past:

```
DELETE FROM Alert_event WHERE event_timestamp < CURRENT_TIMESTAMP;
```

DATEADD()

DATEADD() — Returns a new timestamp value by adding a specified time interval to an existing timestamp value.

Syntax

```
DATEADD( time-unit, interval, timestamp )
```

Description

The DATEADD() function creates a new TIMESTAMP value by adding (or subtracting for negative values) the specified time interval from another TIMESTAMP value. The first argument specifies the time unit of the interval. The valid time unit keywords are:

- MICROSECOND (or MICROS)
- MILLISECOND (or MILLIS)
- SECOND
- MINUTE
- HOUR
- DAY
- MONTH
- QUARTER
- YEAR

The second argument is an integer value specifying the interval to add to the TIMESTAMP value. A positive interval moves the time ahead. A negative interval moves the time value backwards. The third argument specifies the TIMESTAMP value to which the interval is applied.

The DATEADD function takes into account leap years and the variable number of days in a month. Therefore, if the year of either the specified timestamp or the resulting timestamp is a leap year, the day is adjusted to its correct value. For example, DATEADD(YEAR, 1, '2008-02-29') returns '2009-02-28'. Similarly, if the original timestamp is the last day of a month, then the resulting timestamp will be adjusted as necessary. For example, DATEADD(MONTH, 1, '2008-03-31') returns '2008-04-30'.

Example

The following example uses the DATEADD() function to find all records where the TIMESTAMP column, incident, occurs within one day before a specified timestamp (entered as a POSIX time value).

```
SELECT incident, description FROM securityLog
   WHERE DATEADD(DAY, 1, incident) > FROM_UNIXTIME(?)
     AND incident < FROM_UNIXTIME(?)
     ORDER BY incident, description;
```

DAY(), DAYOFMONTH()

DAY(), DAYOFMONTH() — Returns the day of the month as an integer value.

Syntax

DAY(*timestamp-value*)

DAYOFMONTH(*timestamp-value*)

Description

The DAY() function returns an integer value between 1 and 31 representing the timestamp's day of the month. The DAY() and DAYOFMONTH() functions are synonyms. These functions produce the same result as using the DAY or DAY_OF_MONTH keywords with the EXTRACT() function.

Examples

The following example uses the DAY(), MONTH(), and YEAR() functions to return a timestamp column as a formatted date string.

```
SELECT  CAST( MONTH(starttime) AS VARCHAR) || '/' ||
        CAST( DAY(starttime)   AS VARCHAR) || '/' ||
        CAST( YEAR(starttime)  AS VARCHAR), title, description
        FROM event ORDER BY starttime;
```

DAYOFWEEK()

DAYOFWEEK() — Returns the day of the week as an integer between 1 and 7.

Syntax

```
DAYOFWEEK( timestamp-value )
```

Description

The DAYOFWEEK() function returns an integer value between 1 and 7 representing the day of the week in a timestamp value. For the DAYOFTHEWEEK() function, the week starts (1) on Sunday and ends (7) on Saturday.

This function produces the same result as using the DAY_OF_WEEK keyword with the EXTRACT() function.

Examples

The following example uses DAYOFWEEK() and the DECODE() function to return a string value representing the day of the week for the specified TIMESTAMP value.

```
SELECT eventtime,
   DECODE(DAYOFWEEK(eventtime),
       1, 'Sunday',
       2, 'Monday',
       3, 'Tuesday',
       4, 'Wednesday',
       5, 'Thursday',
       6, 'Friday',
       7, 'Saturday') AS eventday
   FROM event ORDER BY eventtime;
```

DAYOFYEAR()

DAYOFYEAR() — Returns the day of the year as an integer between 1 and 366.

Syntax

DAYOFYEAR(*timestamp-value*)

Description

The DAYOFYEAR() function returns an integer value between 1 and 366 representing the day of the year of a timestamp value. This function produces the same result as using the DAY_OF_YEAR keyword with the EXTRACT() function.

Examples

The following example uses the DAYOFYEAR() function to determine the number of days until an event occurs.

```
SELECT DECODE(YEAR(NOW), YEAR(starttime),
     CAST(DAYOFYEAR(starttime) - DAYOFYEAR(NOW) AS VARCHAR)
           || ' days remaining',
     CAST(YEAR(starttime) - YEAR(NOW) AS VARCHAR)
           || ' years remaining'),
  eventname FROM event;
```

DECODE()

DECODE() — Evaluates an expression against one or more alternatives and returns the matching response.

Syntax

DECODE(*expression*, { *comparison-value*, *result* } [,...] [,*default-result*])

Description

The DECODE() function compares an expression against one or more possible comparison values. If the *expression* matches the *comparison-value*, the associated *result* is returned. If the expression does not match any of the comparison values, the *default-result* is returned. If the expression does not match any comparison value and no default result is specified, the function returns NULL.

The DECODE() function operates the same way an IF-THEN-ELSE, or CASE statement does in other languages.

Example

The following example uses the DECODE() function to interpret a coded data column and replace it with the appropriate meaning for each code.

```
SELECT title, industry, DECODE(salary_range,
             'A', 'under $25,000',
             'B', '$25,000 - $34,999',
             'C', '$35,000 - $49,999',
             'D', '$50,000 - $74,999',
             'E', '$75,000 - $99,000',
             'F', '$100,000 and over',
             'unspecified') from survey_results
       order by industry, title;
```

The next example tests a value against three columns and returns the name of the column when a match is found, or a message indicating no match if none is found.

```
SELECT product_name, DECODE(?,product_name,'PRODUCT NAME',
                            part_name, 'PART NAME',
                            category, 'CATEGORY',
                            'NO MATCH FOUND')
     FROM product_list ORDER BY product_name;
```

DISTANCE()

DISTANCE() — Returns the distance between two points or a point and a polygon.

Syntax

DISTANCE(*point-or-polygon*, *point-or-polygon*)

Description

The DISTANCE() function returns the distance, measured in meters, between two points or a point and a polygon. The arguments to the function can be either two GEOGRAPHY_POINT values or a GEOGRAPHY_POINT and GEOGRAPHY value.

Examples

The following example finds the closest city to a specified user, using the GEOGRAPHY_POINT column user.location and the GEOGRAPHY column city.boundary.

```
SELECT TOP 1 user.name, city.name,
    DISTANCE(user.location, city.boundary)
       FROM user, city WHERE user.id = ?
       ORDER BY DISTANCE(user.location, city.boundary) ASC;
```

The next example finds the distance in kilometers from a truck to stores, listed in order with closest first, using the two GEOGRAPHY_POINT columns truck.loc and store.loc.

```
SELECT store.address,
    DISTANCE(store.loc,truck.loc) / 1000 AS distance
       FROM store, truck WHERE truck.id = ?
       ORDER BY DISTANCE(store.loc,truck.loc)/1000 ASC;
```

DWITHIN()

DWITHIN() — Returns true or false depending whether two geospatial entities are within a specified distance of each other.

Syntax

DWITHIN(*polygon-or-point*, *polygon-or-point*, *distance*)

Description

The DWITHIN() function determines if two geospatial values are within the specified distance of each other. The values can be two points (GEOGRAPHY_POINT) or a point and a polygon (GEOGRAPHY). The maximum distance is specified as a numeric value measured in meters. If the distance between the two geospatial values is less than or equal to the specified distance, the function returns true. If not, it returns false.

Examples

The following example finds all the cities within five kilometers of a given user, by evaluating the distance between the GEOGRAPHY_POINT column user.loc and the GEOGRAPHY column city.boundary.

```
SELECT user.name, city.name, DISTANCE(user.loc, city.boundary)
   FROM user, city WHERE user.id=?
   AND DWITHIN(user.loc, city.boundary, 5000)
   ORDER BY DISTANCE(user.loc, city.boundary) ASC;
```

The next is a more generalized example, where the query returns all delivery trucks within a specified distance of a store, where both the distance and the store ID are parameterized and can be input at runtime.

```
SELECT store.address, truck.license_number
    DISTANCE(store.loc, truck.loc)/1000 AS distance_in_km
       FROM store, truck WHERE DWITHIN(store.loc, truck.loc, ?) and store.id=?
       ORDER BY DISTANCE(store.loc,truck.loc)/1000 ASC;
```

EXP()

EXP() — Returns the exponential of the specified numeric expression.

Syntax

```
EXP( numeric-expression )
```

Description

The EXP() function returns the exponential of the specified numeric expression. In other words, EXP(x) is the equivalent of the mathematical expression e^x.

Example

The following example uses the EXP function to calculate the potential population of certain species of animal projecting out ten years.

```
SELECT species, population AS current,
       (population/2) * EXP(10*(gestation/365)*litter) AS future
       FROM animals
       WHERE species = 'rabbit'
       ORDER BY population;
```

EXTRACT()

EXTRACT() — Returns the value of a selected portion of a timestamp.

Syntax

EXTRACT(*selection-keyword* FROM *timestamp-expression*)

EXTRACT(*selection-keyword, timestamp-expression*)

Description

The EXTRACT() function returns the value of the selected portion of a timestamp. Table C.1, "Selectable Values for the EXTRACT Function" lists the supported keywords, the datatype of the value returned by the function, and a description of its contents.

Table C.1. Selectable Values for the EXTRACT Function

Keyword	Datatype	Description
YEAR	INTEGER	The year as a numeric value.
QUARTER	TINYINT	The quarter of the year as a single numeric value between 1 and 4.
MONTH	TINYINT	The month of the year as a numeric value between 1 and 12.
DAY	TINYINT	The day of the month as a numeric value between 1 and 31.
DAY_OF_MONTH	TINYINT	The day of the month as a numeric value between 1 and 31 (same as DAY).
DAY_OF_WEEK	TINYINT	The day of the week as a numeric value between 1 and 7, starting with Sunday.
DAY_OF_YEAR	SMALLINT	The day of the year as a numeric value between 1 and 366.
WEEK	TINYINT	The week of the year as a numeric value between 1 and 52.
WEEK_OF_YEAR	TINYINT	The week of the year as a numeric value between 1 and 52 (same as WEEK).
WEEKDAY	TINYINT	The day of the week as a numeric value between 0 and 6, starting with Monday.
HOUR	TINYINT	The hour of the day as a numeric value between 0 and 23.
MINUTE	TINYINT	The minute of the hour as a numeric value between 0 and 59.
SECOND	DECIMAL	The whole and fractional part of the number of seconds within the minute as a floating point value between 0 and 60.

The timestamp expression is interpreted as a VoltDB timestamp; That is, time measured in microseconds.

Example

The following example lists all the contacts by name and birthday, listing the birthday as three separate fields for month, day, and year.

```
SELECT Last_name, first_name, EXTRACT(MONTH FROM dateofbirth),
```

```
EXTRACT(DAY FROM dateofbirth), EXTRACT(YEAR FROM dateofbirth)
FROM contact_list
ORDER BY last_name, first_name;
```

FIELD()

FIELD() — Extracts a field value from a JSON-encoded string column.

Syntax

FIELD(*column*, *field-name-path*)

Description

The FIELD() function extracts a field value from a JSON-encoded string. For example, assume the VARCHAR column Profile contains the following JSON string:

```
{"first":"Charles","last":"Dickens","birth":1812,
 "description":{"genre":"fiction",
                "period":"Victorian",
                "output":"prolific",
                "children":["Charles","Mary","Kate","Walter","Francis",
                            "Alfred","Sydney","Henry","Dora","Edward"]
               }
}
```

It is possible to extract individual field values using the FIELD() function, as in the following SELECT statement:

```
SELECT FIELD(profile,'first') AS firstname,
       FIELD(profile,'last') AS lastname FROM Authors;
```

It is also possible to find records based on individual JSON fields by using the FIELD() function in the WHERE clause. For example, the following query retrieves all records from the Authors table where the JSON field *birth* is 1812. Note that the FIELD() function always returns a string, even if the JSON type is numeric. The comparison must match the string datatype, so the constant `'1812'` is in quotation marks:

```
SELECT * FROM Authors WHERE FIELD(profile,'birth') = '1812';
```

The second argument to the FIELD() function can be a simple field name, as in the previous examples. In which case the function returns a first-level field matching the specified name. Alternately, you can specify a path representing a hierarchy of names separated by periods. For example, you can specify the genre element of the description field by specifying "description.genre" as the second argument, like so

```
SELECT * FROM Authors WHERE
  FIELD(profile,'description.genre') = 'fiction';
```

You can also use array notation — with square brackets and an integer value — to identify array elements by their position. So, for example, the function can return "Kate", the third child, by using the path specifier "description.children[2]", where "[2]" identifies the third array element because JSON arrays are zero-based.

Two important points to note concerning input to the FIELD() function:

- If the requested field name does not exist, the function returns a null value.
- The first argument to the FIELD() function *must* be a valid JSON-encoded string. However, the content is not evaluated until the function is invoked at runtime. Therefore, it is the responsibility of the database

application to ensure the validity of the content. If the FIELD() function encounters invalid content, the query will fail.

Example

The following example uses the FIELD() function to both return specific JSON fields within a VARCHAR column and filter the results based on the value of a third JSON field:

```
SELECT product_name, sku,
       FIELD(specification,'color') AS color,
       FIELD(specification,'weight') AS weight FROM Inventory
   WHERE FIELD(specification, 'category') = 'housewares'
   ORDER BY product_name, sku;
```

FLOOR()

FLOOR() — Returns the largest integer value less than or equal to a numeric expression.

Syntax

FLOOR(*numeric-expression*)

Description

The FLOOR() function returns the largest integer less then or equal to the specified numeric expression. In other words, the FLOOR() function truncates fractional numeric values. For example:

```
FLOOR(3.1415) = 3
FLOOR(2.0) = 2
FLOOR(-5.32) = -6
```

Example

The following example uses the FLOOR function to calculate the whole number of stocks owned by a specific shareholder.

```
SELECT customer, company,
       FLOOR(num_of_stocks) AS stocks_available_for_sale
       FROM shareholders WHERE customer_id = ?
       ORDER BY company;
```

FORMAT_CURRENCY()

FORMAT_CURRENCY() — Converts a DECIMAL to a text string as a monetary value.

Syntax

```
FORMAT_CURRENCY( decimal-value, rounding-position )
```

Description

The FORMAT_CURRENCY() function converts a DECIMAL value to its string representation, rounding to the specified position. The resulting string is formatted with commas separating every three digits of the whole portion of the number (indicating thousands, millions, and so on) and a decimal point before the fractional portion, as needed.

The *rounding-position* argument must be an integer between 12 and -25 and indicates the place to which the numeric value should be rounded. Positive values indicate a decimal place; for example 2 means round to 2 decimal places. Negative values indicate rounding to a whole number position; for example, -2 indicates the number should be rounded to the nearest hundred. A zero indicates that the value should be rounded to the nearest whole number.

Rounding is performed using "banker's rounding", in that any fractional half is rounded to the nearest even number. So, for example, if the rounding-position is 2, the value 22.225 is rounded to 22.22, but the value 33.335 is rounded to 33.34. The following list demonstrates some sample results.

FORMAT_CURRENCY(.123456789, 4) = 0.1235
FORMAT_CURRENCY(123456789.123, 2) = 123,456,789.12
FORMAT_CURRENCY(123456789.123, 0) = 123,456,789
FORMAT_CURRENCY(123456789.123, -2) = 123,456,800
FORMAT_CURRENCY(123456789.123, -6) = 123,000,000
FORMAT_CURRENCY(123456789.123, 6) = 123,456,789.123000

Example

The following example uses the FORMAT_CURRENCY() function to return a DECIMAL column as a string representation of its monetary value, rounding to two decimal places and appending the appropriate currency symbol from a VARCHAR column.

```
SELECT country,
       currency_symbol || format_currency(budget,2) AS annual_budget
    FROM world_economy ORDER BY country;
```

FROM_UNIXTIME()

FROM_UNIXTIME() — Converts a UNIX time value to a VoltDB timestamp.

Syntax

```
FROM_UNIXTIME( integer-expression )
```

Description

The FROM_UNIXTIME() function converts an integer expression to a VoltDB timestamp, interpreting the integer value as a POSIX time value; that is the number of seconds since the epoch (00:00.00 on January 1, 1970 Consolidated Universal Time). This function is a synonym for TO_TIMESTAMP(second, *integer-expression*).

Example

The following example inserts a record using FROM_UNIXTIME to convert the first argument, a POSIX time value, into a VoltDB timestamp:

```
INSERT event (e_when, e_what, e_where)
       VALUES (FROM_UNIX_TIME(?),?,?);
```

HEX()

HEX() — Returns the hexadecimal representation of a BIGINT value as a string.

Syntax

HEX(*value*)

Description

The HEX() function returns the hexadecimal representation of a BIGINT value as a string. The function will return the shortest valid string representation, truncating any preceding zeros (except in the case of the value zero, which is returned as the string "0").

Examples

The following example use the HEX and BITAND functions to return the hexadecimal representations of two BIGINT values and their binary intersection.

```
$ sqlcmd
1> create table bits (a bigint, b bigint);
2> insert into bits values(555,999);
3> select hex(a) as int1, hex(b) as int2,
4>     hex(bitand(a,b)) as intersection from bits;
INT1     INT2      INTERSECTION
-------- --------- -------------
22B      3E7       223
```

HOUR()

HOUR() — Returns the hour of the day as an integer value.

Syntax

```
HOUR( timestamp-value )
```

Description

The HOUR() function returns an integer value between 0 and 23 representing the hour of the day in a timestamp value. This function produces the same result as using the HOUR keyword with the EXTRACT() function.

Examples

The following example uses the HOUR(), MINUTE(), and SECOND() functions to return the time portion of a TIMESTAMP value in a formatted string.

```
SELECT eventname,
   CAST(HOUR(starttime) AS VARCHAR) || ' hours, ' ||
   CAST(MINUTE(starttime) AS VARCHAR) || ' minutes, and ' ||
   CAST(SECOND(starttime) AS VARCHAR) || ' seconds.'
   AS timestring FROM event;
```

ISINVALIDREASON()

ISINVALIDREASON() — Explains why a GEOGRAPHY polygon is invalid

Syntax

```
ISINVALIDREASON( polygon )
```

Description

The ISINVALIDREASON() function returns a text string explaining if the specified GEOGRAPHY value is valid or not and, if not, why not. The argument to the ISINVALIDREASON() function must be a GEOGRAPHY value describing a polygon. This function is especially useful when validating geospatial data.

Example

The following example uses the ISVALID() and ISINVALIDREASON() functions to report on any invalid polygons in the *border* column of the *country* table.

```
SELECT country_name, ISINVALIDREASON(border)
  FROM Country WHERE NOT ISVALID(border);
```

ISVALID()

ISVALID() — Determines if the specified GEOGRAPHY value is a valid polygon.

Syntax

```
ISVALID( polygon )
```

Description

The ISVALID() function returns true or false depending on whether the specified GEOGRAPHY value is a valid polygon or not. Polygons must follow rules defined by the Open Geospatial Consortium (OGC) standard for Well Known Text (WKT). Specifically:

- A GEOGRAPHY polygon consists of one or more *rings*, where a ring is a closed boundary described by a sequence of vertices and the lines, or *edges*, between those vertices.
- The first ring must be the outer ring and the vertices must be listed in counter clockwise order.
- All subsequent rings represent "holes" in the outer ring. The inner rings must be wholly contained within the outer ring and their vertices must be listed in clockwise order.
- Rings cannot intersect or have adjacent edges.
- The edges of an individual ring cannot cross (for example, a figure "8" is invalid).
- For each ring, the first vertex is listed twice: as both the first and last vertex.

If the specified GEOGRAPHY value is a valid polygon, the function returns true. If not, it returns false.

To maximize performance, VoltDB does not validate the GEOGRAPHY values when they are inserted. However, if you are not sure the WKT strings are valid, you can use ISVALID() to validate the resulting GEOGRAPHY values before inserting them or after they are inserted into the database.

Examples

The first example shows an UPDATE statement that uses the ISVALID() function to remove the contents of a GEOGRAPHY column (by setting it to NULL), if the current contents are invalid.

```
UPDATE REGION SET border = NULL WHERE NOT ISVALID(border);
```

The next example shows part of a stored procedure that uses ISVALID() to conditionally set the value of a column, *mustbevalid*, that is defined as NOT NULL. By setting the column valid to NULL, the procedure ensures that the INSERT statement fails and the stored procedure rolls back if the WKT input is invalid.

```
public class ValidateBorders extends VoltProcedure {

  public final SQLStmt insertrec = new SQLStmt(
    "INSERT INTO REGION COLUMNS (name, border, mustbevalid)" +
    " SELECT ?, ?, CASE WHEN ISVALID(?) THEN 1 ELSE NULL END" +
    " FROM anothertable LIMIT 1;"
  );
```

```
    public VoltTable[] run( String name, String wkt)
       throws VoltAbortException {
           GeographyValue border = GeographyValue.fromWKT(wkt);
           voltQueueSQL( insertrec, name, border, border);
           return voltExecuteSQL();
       }
}
```

LATITUDE()

LATITUDE() — Returns the latitude of a GEOGRAPHY_POINT value.

Syntax

```
LATITUDE( point )
```

Description

The LATITUDE() function returns the latitude, as a floating point value, from a GEOGRAPHY_POINT expression.

Example

The following example returns all ships that are located in the northern hemisphere by examining the latitude of their current location.

```
SELECT ship.number, ship.country FROM ship
   WHERE LATITUDE(ship.location) > 0;
```

LEFT()

LEFT() — Returns a substring from the beginning of a string.

Syntax

```
LEFT( string-expression, numeric-expression )
```

Description

The LEFT() function returns the first *n* characters from a string expression, where *n* is the second argument to the function.

Example

The following example uses the LEFT function to return an abbreviation (the first three characters) of the product category as part of the SELECT expression.

```
SELECT LEFT(category,3), product_name, price FROM product_list
    ORDER BY category, product_name;
```

LN(), LOG()

LN(), LOG() — Returns the natural logarithm of a numeric value.

Syntax

LN(*numeric-value*)

LOG(*numeric-value*)

Description

The LN() function returns the natural logarithm of the specified input value. The log is returned as a floating point (FLOAT) value. LN() and LOG() are synonyms and perform the same function.

Example

The following example uses the LN() function to calculate the rate of population growth from census data.

```
SELECT  city, current_population,
    ( ( LN(current_population) - LN(base_population) )
       / (current_year - base_year)
    ) * 100.0 AS percent_growth
    FROM  census ORDER BY city;
```

LONGITUDE()

LONGITUDE() — Returns the longitude of a GEOGRAPHY_POINT value.

Syntax

```
LONGITUDE( point )
```

Description

The LONGITUDE() function returns the longitude, as a floating point value, from a GEOGRAPHY_POINT expression.

Example

The following example returns all ships that are located in the western hemisphere by examining the longitude of their current location.

```
SELECT ship.number, ship.country FROM ship
   WHERE LONGITUDE(ship.location) < 0
   AND  LONGITUDE(ship.location) > -180;
```

LOWER()

LOWER() — Returns a string converted to all lowercase characters.

Syntax

LOWER(*string-expression*)

Description

The LOWER() function returns a copy of the input string converted to all lowercase characters.

Example

The following example uses the LOWER function to perform a case-insensitive search of a VARCHAR field.

```
SELECT product_name, product_id  FROM product_list
    WHERE LOWER(product_name) LIKE 'acme%'
    ORDER BY product_name, product_id
```

MAX()

MAX() — Returns the maximum value from a range of column values.

Syntax

```
MAX( column-expression )
```

Description

The MAX() function returns the highest value from a range of column values. The range of values depends on the constraints defined by the WHERE and GROUP BY clauses.

Example

The following example returns the highest price in the product list.

```
SELECT MAX(price) FROM product_list;
```

The next example returns the highest price for each product category.

```
SELECT category, MAX(price) FROM product_list
   GROUP BY category
   ORDER BY category;
```

MIN()

MIN() — Returns the minimum value from a range of column values.

Syntax

```
MIN( column-expression )
```

Description

The MIN() function returns the lowest value from a range of column values. The range of values depends on the constraints defined by the WHERE and GROUP BY clauses.

Example

The following example returns the lowest price in the product list.

```
SELECT MIN(price) FROM product_list;
```

The next example returns the lowest price for each product category.

```
SELECT category, MIN(price) FROM product_list
   GROUP BY category
   ORDER BY category;
```

MINUTE()

MINUTE() — Returns the minute of the hour as an integer value.

Syntax

```
MINUTE( timestamp-value )
```

Description

The MINUTE() function returns an integer value between 0 and 59 representing the minute of the hour in a timestamp value. This function produces the same result as using the MINUTE keyword with the EXTRACT() function.

Examples

The following example uses the HOUR(), MINUTE(), and SECOND() functions to return the time portion of a TIMESTAMP value in a formatted string.

```
SELECT eventname,
   CAST(HOUR(starttime) AS VARCHAR) || ' hours, ' ||
   CAST(MINUTE(starttime) AS VARCHAR) || ' minutes, and ' ||
   CAST(SECOND(starttime) AS VARCHAR) || ' seconds.'
   AS timestring FROM event;
```

MOD()

MOD() — Returns the result of a modulo operation.

Syntax

MOD(*dividend*, *divisor*)

Description

The MOD() function performs a modulo operation. That is, it divides one integer value, the dividend, by another integer value, the divisor, and returns the remainder of the division operation as a new integer value. Both the dividend and the divisor must be integer values and the divisor must *not* be zero. Use of non-integer datatypes or a divisor of zero will result in a runtime error.

Example

The following example uses the HOUR() and MOD() functions to return the hour of a timestamp in 12 hour format

```
SELECT event,
       MOD(HOUR(eventtime)+11, 12)+1,
       CASE WHEN HOUR(eventtime)/12 < 1
          THEN 'AM'
          ELSE 'PM'
       END
   FROM schedule ORDER BY 3, 2;
```

MONTH()

MONTH() — Returns the month of the year as an integer value.

Syntax

```
MONTH( timestamp-value )
```

Description

The MONTH() function returns an integer value between 1 and 12 representing the timestamp's month of the year. The MONTH() function produces the same result as using the MONTH keyword with the EXTRACT() function.

Examples

The following example uses the DAY(), MONTH(), and YEAR() functions to return a timestamp column as a formatted date string.

```
SELECT  CAST( MONTH(starttime) AS VARCHAR) || '/' ||
        CAST( DAY(starttime)   AS VARCHAR) || '/' ||
        CAST( YEAR(starttime)  AS VARCHAR), title, description
        FROM event ORDER BY starttime;
```

NOW

NOW — Returns the current time as a timestamp value.

Syntax

```
NOW
```

Description

The NOW function returns the current time as a VoltDB timestamp. The value of the timestamp is determined when the query or stored procedure is invoked. Several important aspects of how the NOW function operates are:

- The value returned is guaranteed to be identical for all partitions that execute the query.
- The value returned is measured in milliseconds then padded to create a timestamp value in microseconds.
- During command logging, the returned value is stored as part of the log, so when the command log is replayed, the same value is used during the replay of the query.
- Similarly, for database replication (DR) the value returned is passed and reused by the replica database when replaying the query.
- You can specify NOW as a default value in the CREATE TABLE statement when defining the schema of a VoltDB database.
- The NOW function *cannot* be used in the CREATE INDEX or CREATE VIEW statements.

The NOW and CURRENT_TIMESTAMP functions are synonyms and perform an identical function.

Example

The following example uses NOW in the WHERE clause to delete alert events that occurred in the past:

```
DELETE FROM Alert_event WHERE event_timestamp < NOW;
```

NUMINTERIORRINGS()

NUMINTERIORRINGS() — Returns the number of interior rings within a polygon GEOGRAPHY value.

Syntax

```
NUMINTERIORRINGS( polygon )
```

Description

The NUMINTERIORRINGS() function returns the number of interior rings within a polygon GEOGRAPHY value. Polygon GEOGRAPHY values can contain multiple polygons: one and only one outer polygon and one or more optional inner polygons that define "holes" in the outer polygon. The NUMINTERIORRINGS() function counts the number of inner polygons and returns the result as an integer value.

Example

The following example lists the countries of the world based on the number of interior polygons within the outline GEOGRAPHY column.

```
SELECT NUMINTERIORRINGS(outline), name, capital FROM country
   ORDER BY NUMINTERIORRINGS(outline);
```

NUMPOINTS()

NUMPOINTS() — Returns the number of points within a polygon GEOGRAPHY value.

Syntax

```
NUMPOINTS( polygon )
```

Description

The NUMPOINTS() function returns the total number of points that comprise a polygon GEOGRAPHY value. The number of points includes the points from both the outer polygon and any inner polygons. It also includes all of the points defining the polygon. Which means the starting point for each polygon is counted twice — once as the starting point and once and the ending point — because this is required in the WKT representation of a polygon.

Example

The following example lists the countries of the world based on the number of points in their outlines.

```
SELECT NUMPOINTS(outline), name, capital FROM country
   ORDER BY NUMPOINTS(outline);
```

OCTET_LENGTH()

OCTET_LENGTH() — Returns the number of bytes in a string.

Syntax

```
OCTET_LENGTH( string-expression )
```

Description

The OCTET_LENGTH() function returns the number of bytes of data in a string.

Note that the number of bytes required to store a string and the actual characters that make up the string can differ. To count the number of characters in the string use the CHAR_LENGTH() function.

Example

The following example returns the string in the column LastName as well as the number of characters and length in bytes of that string.

```
SELECT LastName, CHAR_LENGTH(LastName), OCTET_LENGTH(LastName)
    FROM Customers ORDER BY LastName, FirstName;
```

OVERLAY()

OVERLAY() — Returns a string overwriting a portion of the original string with the specified replacement.

Syntax

```
OVERLAY( string PLACING replacement-string FROM position [FOR length] )
```

Description

The OVERLAY() function overwrites a portion of the original string with the replacement string and returns the result. The replacement starts at the specified position in the original string and either replaces the characters one-for-one for the length of the replacement string or, if a FOR *length* is specified, replaces the specified number of characters.

For example, if the original string is 12 characters in length, the replacement string is 3 characters in length and starts at position 4, and the FOR clause is left off, the resulting string consists of the first 3 characters of the original string, the replacement string, and the last 6 characters of the original string:

OVERLAY('abcdefghijkl' PLACING 'XYZ' FROM 4) = 'abcXYZghijkl'

If the FOR clause is included specifying that the replacement string replaces 6 characters, the result is the first 3 characters of the original string, the replacement string, and the last 3 characters of the original string:

OVERLAY('abcdefghijkl' PLACING 'XYZ' FROM 4 FOR 6) = 'abcXYZjkl'

If the combination of the starting position and the replacement length exceed the length of the original string, the resulting output is extended as necessary to include all of the replacement string:

OVERLAY('abcdef' PLACING 'XYZ' FROM 5) = 'abcdXYZ'

If the starting position is greater than the length of the original string, the replacement string is appended to the original string:

OVERLAY('abcdef' PLACING 'XYZ' FROM 20) = 'abcdefXYZ'

Similarly, if the combination of the starting position and the FOR length is greater than the length of the original string, the replacement string simply overwrites the remainder of the original string:

OVERLAY('abcdef' PLACING 'XYZ' FROM 2 FOR 20) = 'aXYZ'

The starting position and length must be specified as non-negative integers. The starting position must be greater than zero and the length can be zero or greater.

Example

The following example uses the OVERLAY function to redact part of a name.

```
SELECT OVERLAY( fullname PLACING '****' FROM 2
                FOR CHAR_LENGTH(fullname)-2
       ) FROM users ORDER BY fullname;
```

PI()

PI() — Returns the value of the mathematical constant pi (π) as a FLOAT value.

Syntax

```
PI()
```

Description

The PI() function returns the value of the mathematical constant pi (π) as a double floating point (FLOAT) value.

Example

The following example uses the PI() function to return the surface area of a sphere.

```
SELECT radius, 4*PI()*POWER(radius, 2) FROM Sphere ORDER BY radius;
```

POINTFROMTEXT()

POINTFROMTEXT() — Returns a GEOGRAPHY_POINT value from the corresponding WKT

Syntax

```
POINTFROMTEXT( string )
```

Description

The POINTFROMTEXT() function generates a GEOGRAPHY_POINT value from a string containing a well known text (WKT) representation of a geographic point. The WKT string must be in the form 'POINT(*longitude latitude*)' where longitude and latitude are floating point values.

if the argument is not a valid WKT representation of a point, the function generates an error.

Example

The following example uses the POINTFROMTEXT() function to update a record containing a GEOGRAPHY_POINT column using two floating point input values (representing longitude and latitude).

```
UPDATE user SET location =
   POINTFROMTEXT( CONCAT('POINT(',CAST(? AS VARCHAR),' ',CAST(? AS VARCHAR),')') )
   WHERE id = ?;
```

POLYGONFROMTEXT()

POLYGONFROMTEXT() — Returns a GEOGRAPHY value from the corresponding WKT

Syntax

```
POLYGONFROMTEXT( text )
```

Description

The POLYGONFROMTEXT() function generates a GEOGRAPHY value from a string containing a well known text (WKT) representation of a geographic polygon. The WKT string must be a valid representation of a polygon with only one outer polygon and, optionally, one or more inner polygons.

if the argument is not a valid WKT representation of a polygon, the function generates an error.

Example

The following example uses the POLYGONFROMTEXT() function to insert a record containing a GEOGRAPHY column using a text input value containing the WKT representation of a geographic polygon.

```
INSERT INTO city (name, state, boundary) VALUES(?, ?, POLYGONFROMTEXT(?));
```

POSITION()

POSITION() — Returns the starting position of a substring in another string.

Syntax

```
POSITION( substring-expression IN string-expression )
```

Description

The POSITION() function returns the starting position of a substring in another string. The position, if a match is found, is an integer number between one and the length of the string being searched. If no match is found, the function returns zero.

Example

The following example selects all books where the title contains the word "poodle" and returns the book's title and the position of the substring "poodle" in the title.

```
SELECT Title, POSITION('poodle' IN Title) FROM Books
    WHERE Title LIKE '%poodle%'  ORDER BY Title;
```

POWER()

POWER() — Returns the value of the first argument raised to the power of the second argument.

Syntax

POWER(*numeric-expression*, *numeric-expression*)

Description

The POWER() function takes two numeric expressions and returns the value of the first raised to the power of the second. In other words, POWER(x,y) is the equivalent of the mathematical expression x^y.

Example

The following example uses the POWER function to return the surface area and volume of a cube.

```
SELECT length, 6 * POWER(length,2) AS surface,
       POWER(length,3) AS volume FROM Cube
      ORDER BY length;
```

QUARTER()

QUARTER() — Returns the quarter of the year as an integer value

Syntax

QUARTER(*timestamp-value*)

Description

The QUARTER() function returns an integer value between 1 and 4 representing the quarter of the year in a TIMESTAMP value. The QUARTER() function produces the same result as using the QUARTER keyword with the EXTRACT() function.

Examples

The following example uses the QUARTER() and YEAR() functions to group and sort records containing a timestamp.

```
SELECT year(starttime), quarter(starttime),
    count(*) as eventsperquarter
    FROM event
    GROUP BY year(starttime), quarter(starttime)
    ORDER BY year(starttime), quarter(starttime);
```

REGEXP_POSITION()

REGEXP_POSITION() — Returns the starting position of a regular expression within a text string.

Syntax

REGEXP_POSITION(*string*, *pattern* [, *flag*])

Description

The REGEXP_POSITION() function returns the starting position of the first instance of the specified regular expression within a text string. The position value starts at one (1) for the first position in the string and the functions returns zero (0) if the regular expression is not found.

The first argument to the function is the VARCHAR character string to be searched. The second argument is the regular expression pattern to look for. The third argument is an optional flag that specifies whether the search is case sensitive or not. The flag must be single character VARCHAR with one of the following values:

Flag	Description
c	Case-sensitive matching (default)
i	Case-insensitive matching

There are several different formats for regular expressions. The REGEXP_POSITION() uses the revised Perl compatible regular expression (PCRE2) syntax, which is described in detail on the PCRE website.

Examples

The following example uses the REGEXP_POSITION() to filter all records where the column description matches a specific pattern. The examples uses the optional *flag* argument to make the pattern match text regardless of case.

```
SELECT incident, description FROM securityLog
    WHERE REGXP_POSITION(description,
          'host:\s*10\.186\.[0-9]+\.[0-9]+',
          'i') > 0
    ORDER BY incident;
```

REPEAT()

REPEAT() — Returns a string composed of a substring repeated the specified number of times.

Syntax

REPEAT(*string-expression*, *numeric-expression*)

Description

The REPEAT() function returns a string composed of the substring *string-expression* repeated *n* times where *n* is defined by the second argument to the function.

Example

The following example uses the REPEAT and the CHAR_LENGTH functions to replace a column's actual contents with a mask composed of the letter "X" the same length as the original column value.

```
SELECT username, REPEAT('X', CHAR_LENGTH(password)) as Password
    FROM accounts ORDER BY username;
```

REPLACE()

REPLACE() — Returns a string replacing the specified substring of the original string with new text.

Syntax

```
REPLACE( string, substring, replacement-string )
```

Description

The REPLACE() function returns a copy of the first argument, replacing all instances of the substring identified by the second argument with the third argument. If the substring is not found, no changes are made and a copy of the original string is returned.

Example

The following example uses the REPLACE function to update the Address column, replacing the string "Ceylon" with "Sri Lanka".

```
UPDATE Customers SET address=REPLACE( address,'Ceylon', 'Sri Lanka')
        WHERE address LIKE '%Ceylon%';
```

RIGHT()

RIGHT() — Returns a substring from the end of a string.

Syntax

RIGHT(*string-expression*, *numeric-expression*)

Description

The RIGHT() function returns the last *n* characters from a string expression, where *n* is the second argument to the function.

Example

The following example uses the LEFT() and RIGHT() functions to return an abbreviated summary of the Description column, ensuring the result fits within 20 characters.

```
SELECT product_name,
       LEFT(description,10) || '...' || RIGHT(description,7)
       FROM product_list ORDER BY product_name;
```

SECOND()

SECOND() — Returns the seconds of the minute as a floating point value.

Syntax

```
SECOND( timestamp-value )
```

Description

The SECOND() function returns an floating point value between 0 and 60 representing the whole and fractional part of the number of seconds in the minute of a timestamp value. This function produces the same result as using the SECOND keyword with the EXTRACT() function.

Examples

The following example uses the HOUR(), MINUTE(), and SECOND() functions to return the time portion of a TIMESTAMP value in a formatted string.

```
SELECT eventname,
   CAST(HOUR(starttime) AS VARCHAR) || ' hours, ' ||
   CAST(MINUTE(starttime) AS VARCHAR) || ' minutes, and ' ||
   CAST(SECOND(starttime) AS VARCHAR) || ' seconds.'
   AS timestring FROM event;
```

SET_FIELD()

SET_FIELD() — Returns a copy of a JSON-encoded string, replacing the specified field value.

Syntax

SET_FIELD(*column*, *field-name-path, string-value*)

Description

The SET_FIELD() function finds the specified field within a JSON-encoded string and returns a copy of the string with the new value replacing that field's previous value. Note that the SET_FIELD() function returns an altered copy of the JSON-encoded string — it does not change any column values in place. So to change existing database columns, you must use SET_FIELD() with an UPDATE statement.

For example, assume the Product table contains a VARCHAR column Productinfo which for one row contains the following JSON string:

```
{"product":"Acme widget",
 "availability":"plenty",
 "info": { "description": "A fancy widget.",
           "sku":"ABCXYZ",
           "part_number":1234},
 "warehouse":[{"location":"Dallas","units":25},
              {"location":"Chicago","units":14},
              {"location":"Troy","units":67}]
}
```

It is possible to change the value of the `availability` field using the SET_FIELD function, like so:

```
UPDATE Product SET Productinfo =
    SET_FIELD(Productinfo,'availability','"limited"')
    WHERE FIELD(Productinfo,'product') = 'Acme widget';
```

The second argument to the SET_FIELD() function can be a simple field name, as in the previous example, In which case the function replaces the value of the top field matching the specified name. Alternately, you can specify a path representing a hierarchy of names separated by periods. For example, you can replace the SKU number by specifying "info.sku" as the second argument, or you can replace the number of units in the second warehouse by specifying the field path "warehouse[1].units". For example, the following UPDATE statement does both by nesting SET_FIELD commands:

```
UPDATE Product SET Productinfo =
    SET_FIELD(
       SET_FIELD(Productinfo,'info.sku','"DEFGHI"'),
          'warehouse[1].units', '128')
    WHERE FIELD(Productinfo,'product') = 'Acme widget';
```

Note that the third argument is the string value that will be inserted into the JSON-encoded string. To insert a numeric value, you enclose the value in single quotation marks, as in the preceding example where '128' is used as the replacement value for the `warehouse[1].units` field. To insert a string value, you must include the string quotation marks within the replacement string itself. For example, the preceding code uses the SQL string constant '"DEFGHI"' to specify the replacement value for the text field `info.sku`.

Finally, the replacement string value can be any valid JSON value, including another JSON-encoded object or array. It does not have to be a scalar string or numeric value.

Example

The following example uses the SET_FIELD() function to add a new array element to the warehouse field.

```
UPDATE Product SET Productinfo =
    SET_FIELD(Productinfo,'warehouse',
        '[{"location":"Dallas","units":25},
          {"location":"Chicago","units":14},
          {"location":"Troy","units":67},
          {"location":"Phoenix","units":23}]')
    WHERE FIELD(Productinfo,'product') = 'Acme widget';
```

SINCE_EPOCH()

SINCE_EPOCH() — Converts a VoltDB timestamp to an integer number of time units since the POSIX epoch.

Syntax

SINCE_EPOCH(*time-unit*, *timestamp-expression*)

Description

The SINCE_EPOCH() function converts a VoltDB timestamp into an 64-bit integer value (BIGINT) representing the equivalent number since the POSIX epoch in a specified time unit. POSIX time is usually represented as the number of seconds since the epoch; that is, since 00:00.00 on January 1, 1970 Consolidated Universal Time (UTC). So the function SINCE_EPOCH(SECONDS, timestamp) returns the POSIX time equivalent for the value of timestamp. However, you can also request the number of milliseconds or microseconds since the epoch. The valid keywords for specifying the time units are:

- SECOND — Seconds since the epoch
- MILLISECOND, MILLIS — Milliseconds since the epoch
- MICROSECOND, MICROS — Microseconds since the epoch

You cannot perform arithmetic on timestamps directly. So SINCE_EPOCH() is useful for performing calculations on timestamp values in SQL expressions. For example, the following SQL statement looks for events that are less than a minute in length, based on the timestamp columns STARTTIME and ENDTIME:

```
SELECT * FROM Event
  WHERE ( SINCE_EPOCH(Second, endtime)
        - SINCE_EPOCH(Second, starttime) ) < 60;
```

The TO_TIMESTAMP() function performs the inverse of SINCE_EPOCH(), by converting an integer value to a VoltDB timestamp based on the specified time units.

Example

The following example returns a timestamp column as the equivalent POSIX time value.

```
SELECT event_id, event_name,
    SINCE_EPOCH(Second, starttime) as posix_time FROM Event
    ORDER BY event_id;
```

The next example uses SINCE_EPOCH() to return the length of an event, in microseconds, by calculating the difference between two timestamp columns.

```
SELECT event_id, event_type
    SINCE_EPOCH(Microsecond, endtime)
    -SINCE_EPOCH(Microsecond, starttime) AS delta
    FROM Event ORDER BY event_id;
```

SPACE()

SPACE() — Returns a string of spaces of the specified length.

Syntax

SPACE(*numeric-expression*)

Description

The SPACE() function returns a string composed of *n* spaces where the string length *n* is specified by the function's argument. SPACE(n) is a synonym for REPEAT(' ', n).

Example

The following example uses the SPACE and CHAR_LENGTH functions to ensure the result is a fixed length, padded with blank spaces.

```
SELECT product_name || SPACE(80 - CHAR_LENGTH(product_name))
   FROM product_list ORDER BY product_name;
```

SQRT()

SQRT() — Returns the square root of a numeric expression.

Syntax

SQRT(*numeric-expression*)

Description

The SQRT() function returns the square root of the specified numeric expression.

Example

The following example uses the SQRT and POWER functions to return the distance of a graph point from the origin.

```
SELECT location, x, y,
       SQRT(POWER(x,2) + POWER(y,2)) AS distance
       FROM points ORDER BY location;
```

SUBSTRING()

SUBSTRING() — Returns the specified portion of a string expression.

Syntax

SUBSTRING(*string-expression* FROM *position* [FOR *length*])

SUBSTRING(*string-expression*, *position* [, *length*])

Description

The SUBSTRING() function returns a specified portion of the string expression, where *position* specifies the starting position of the substring (starting at position 1) and *length* specifies the maximum length of the substring. The length of the returned substring is the lower of the remaining characters in the string expression or the value specified by *length*.

For example, if the string expression is "ABCDEF" and position is specified as 3, the substring starts with the character "C". If length is also specified as 3, the return value is "CDE". If, however, the length is specified as 5, only the remaining four characters "CDEF" are returned.

If *length* is not specified, the remainder of the string, starting from the specified by *position*, is returned. For example, SUBSTRING("ABCDEF",3) and SUBSTRING("ABCDEF"3,4) return the same value.

Example

The following example uses the SUBSTRING function to return the month of the year, which is a VARCHAR column, as a three letter abbreviation.

```
SELECT event, SUBSTRING(month,1,3), day, year FROM calendar
    ORDER BY event ASC;
```

SUM()

SUM() — Returns the sum of a range of numeric column values.

Syntax

SUM(*column-expression*)

Description

The SUM() function returns the sum of a range of numeric column values. The values being added together depend on the constraints defined by the WHERE and GROUP BY clauses.

Example

The following example uses the SUM() function to determine how much inventory exists for each product type in the catalog.

```
SELECT category, SUM(quantity) AS inventory FROM product_list
    GROUP BY category ORDER BY category;
```

TO_TIMESTAMP()

TO_TIMESTAMP() — Converts an integer value to a VoltDB timestamp based on the time unit specified.

Syntax

TO_TIMESTAMP(*time-unit*, *integer-expression*)

Description

The TO_TIMESTAMP() function converts an integer expression to a VoltDB timestamp, interpreting the integer value as the number of specified time units since the POSIX epoch. POSIX time is usually represented as the number of seconds since the epoch; that is, since 00:00.00 on January 1, 1970 Consolidated Universal Time (UTC). So the function TO_TIMESTAMP(Second, timeinsecs) returns the VoltDB TIMESTAMP equivalent of timeinsecs as a POSIX time value. However, you can also request the integer value be interpreted as milliseconds or microseconds since the epoch. The valid keywords for specifying the time units are:

- SECOND — Seconds since the epoch
- MILLISECOND. MILLIS — Milliseconds since the epoch
- MICROSECOND, MICROS — Microseconds since the epoch

You cannot perform arithmetic on timestamps directly. So TO_TIMESTAMP() is useful for converting the results of arithmetic expressions to VoltDB TIMESTAMP values. For example, the following SQL statement uses TO_TIMESTAMP to convert a POSIX time value before inserting it into a VoltDB TIMESTAMP column:

```
INSERT INTO Event
   (event_id,event_name,event_type, starttime)
   VALUES(?,?,?,TO_TIMESTAMP(Second, ?));
```

The SINCE_EPOCH() function performs the inverse of TO_TIMESTAMP(), by converting a VoltDB TIMESTAMP to an integer value based on the specified time units.

Example

The following example updates a TIMESTAMP column, adding one hour (in seconds) to the current value using SINCE_EPOCH() and TO_TIMESTAMP() to perform the conversion and arithmetic:

```
UPDATE Contest
   SET deadline=TO_TIMESTAMP(Second, SINCE_EPOCH(Second,deadline) + 3600)
   WHERE expired=1;
```

TRIM()

TRIM() — Returns a string with leading and/or training spaces removed.

Syntax

TRIM([[LEADING | TRAILING | BOTH] [*character*] FROM] *string-expression*)

Description

The TRIM() function returns a string with leading and/or trailing spaces removed. By default, the TRIM function removes spaces from both the beginning and end of the string. If you specify the LEADING or TRAILING clause, spaces are removed from either the beginning or end of the string only.

You can also specify an alternate character to remove. By default only spaces (UTF-8 character code 32) are removed. If you specify a different character, only that character will be removed. For example, the following INSERT statement uses the TRIM function to remove any TAB characters from the beginning of the string input for the ADDRESS column:

```
INSERT INTO Customers (first, last, address)
      VALUES(?, ?,
         TRIM( LEADING CHAR(9) FROM CAST(? AS VARCHAR) )
      );
```

Example

The following example uses TRIM() to remove extraneous leading and trailing spaces from the output for three VARCHAR columns:

```
SELECT TRIM(first), TRIM(last), TRIM(address) FROM Customer
      ORDER BY last, first;
```

TRUNCATE()

TRUNCATE() — Truncates a VoltDB timestamp to the specified time unit.

Syntax

```
TRUNCATE( time-unit, timestamp )
```

Description

The TRUNCATE() function truncates a timestamp value to the specified time unit. For example, if the timestamp column Apollo has the value July 20, 1969 4:17:40 P.M, then using the function TRUNCATE(hour,apollo) would return the value July 20, 1969 4:00:00 P.M. Allowable time units for truncation include the following:

- YEAR
- QUARTER
- MONTH
- DAY
- HOUR
- MINUTE
- SECOND
- MILLISECOND, MILLIS

Example

The following example uses the TRUNCATE function to find records where the timestamp column, incident, falls within a specific day, entered as a POSIX time value.

```
SELECT incident, description FROM securitylog
       WHERE TRUNCATE(DAY, incident) = TRUNCATE(DAY,FROM_UNIXTIME(?))
       ORDER BY incident, description;
```

UPPER()

UPPER() — Returns a string converted to all uppercase characters.

Syntax

UPPER(*string-expression*)

Description

The UPPER() function returns a copy of the input string converted to all uppercase characters.

Example

The following example uses the UPPER function to return results alphabetically regardless of case.

```
SELECT UPPER(product_name), product_id  FROM product_list
    ORDER BY UPPER(product_name)
```

VALIDPOLYGONFROMTEXT()

VALIDPOLYGONFROMTEXT() — Returns a validated GEOGRAPHY value from the corresponding WKT

Syntax

```
VALIDPOLYGONFROMTEXT( text )
```

Description

The VALIDPOLYGONFROMTEXT() function generates a valid GEOGRAPHY value from a string containing a well known text (WKT) representation of a geographic polygon. If the GEOGRAPHY value resulting from the WKT string is not a valid representation of a polygon, the function returns an error. The error message includes an explanation of why the WKT is not valid.

The difference between the POLYGONFROMTEXT() function and the VALIDPOLYGONFROMTEXT() function is that the VALIDPOLYGONFROMTEXT() verifies that the resulting polygon meets all of the requirements for use by VoltDB. If not, the function returns an error. The POLYGONFROMTEXT() function simply constructs a GEOGRAPHY value without validating all of the requirements of a VoltDB polygon and may need separate validation (using the ISVALID() function) before it can be used effectively with other geospatial functions. See the description of the ISVALID() function for a description of the requirements for a valid polygon.

Example

The following example uses the VALIDPOLYGONFROMTEXT() function to insert a record containing a GEOGRAPHY column using a text input value containing the WKT representation of a geographic polygon. Note that if

```
INSERT INTO city (name, state, boundary) VALUES(?, ?, VALIDPOLYGONFROMTEXT(?));
```

WEEK(), WEEKOFYEAR()

WEEK(), WEEKOFYEAR() — Returns the week of the year as an integer value.

Syntax

```
WEEK( timestamp-value )

WEEKOFYEAR( timestamp-value )
```

Description

The WEEK() and WEEKOFYEAR() functions are synonyms and return an integer value between 1 and 52 representing the timestamp's week of the year. These functions produce the same result as using the WEEK_OF_YEAR keyword with the EXTRACT() fucntion.

Examples

The following example uses the WEEK() function to group and sort records containing a timestamp.

```
SELECT week(starttime), count(*) as eventsperweek
  FROM event GROUP BY week(starttime) ORDER BY week(starttime);
```

WEEKDAY()

WEEKDAY() — Returns the day of the week as an integer between 0 and 6.

Syntax

```
WEEKDAY( timestamp-value )
```

Description

The WEEKDAY() function returns an integer value between 0 and 6 representing the day of the week in a timestamp value. For the WEEKDAY() function, the week starts (0) on Monday and ends (6) on Sunday.

This function is provided for compatibility with MySQL and produces the same result as using the WEEKDAY keyword with the EXTRACT() function.

Examples

The following example uses WEEKDAY() and the DECODE() function to return a string value representing the day of the week for the specified TIMESTAMP value.

```
SELECT eventtime,
   DECODE(WEEKDAY(eventtime),
       0, 'Monday',
       1, 'Tuesday',
       2, 'Wednesday',
       3, 'Thursday',
       4, 'Friday',
       5, 'Saturday',
       6, 'Sunday') AS eventday
   FROM event ORDER BY eventtime;
```

YEAR()

YEAR() — Returns the year as an integer value.

Syntax

```
YEAR( timestamp-value )
```

Description

The YEAR() function returns an integer value representing the year of a TIMESTAMP value. The YEAR() function produces the same result as using the YEAR keyword with the EXTRACT() function.

Examples

The following example uses the DAY(), MONTH(), and YEAR() functions to return a timestamp column as a formatted date string.

```
SELECT  CAST( MONTH(starttime) AS VARCHAR) || '/' ||
        CAST( DAY(starttime)   AS VARCHAR) || '/' ||
        CAST( YEAR(starttime)  AS VARCHAR), title, description
        FROM event ORDER BY starttime;
```

Appendix D. VoltDB CLI Commands

VoltDB provides shell or CLI (command line interpreter) commands to perform common functions for developing, starting, and managing VoltDB applications and databases. This appendix describes those shell commands in detail.

The commands are listed in alphabetical order.

- csvloader
- jdbcloader
- kafkaloader
- sqlcmd
- voltadmin
- voltdb

csvloader

csvloader — Imports the contents of a CSV file and inserts it into a VoltDB table.

Syntax

csvloader *table-name* [arguments]

csvloader -p *procedure-name* [arguments]

Description

The csvloader command reads comma-separated values and inserts each valid line of data into the specified table in a VoltDB database. The most common way to use csvloader is to specify the database table to be loaded and a CSV file containing the data, like so:

```
$ csvloader employees -f acme_employees.csv
```

Alternately, you can use standard input as the source of the data:

```
$ csvloader employees < acme_employees.csv
```

In addition to inserting all valid content into the specified database table, csvloader creates three output files:

- **Error log** — The error log provides details concerning any errors that occur while processing the input file. This includes errors in the format of the input as well as errors that occur attempting the insert into VoltDB. For example, if two rows contain the same value for a column that is declared as unique, the error log indicates that the second insert fails due to a constraint violation.
- **Failed input** — A separate file contains the contents of each line that failed to load. This file is useful because it allows you to correct any formatting issues and retry just the failed content, rather than having to restart and reload the entire table.
- **Summary report** — Once all input lines are processed, csvloader generates a summary report listing how many lines were read, how many were successfully loaded and how long the operation took.

All three files are created, by default, in the current working directory using "csvloader" and the table name as prefixes. For example, using csvloader to insert contestants into the sample voter database creates the following files:

csvloader_contestants_insert_log.log
csvloader_contestants_invalidrows.csv
csvloader_contestants_insert_report.log

It is possible to use csvloader to load text files other than CSV files, using the `--separator`, `--quotechar`, and `--escape` flags. Note that csvloader uses Python to process the command line arguments. So to enter certain non-alphanumeric characters, you must use the appropriate escaping mechanism for Python command lines. For example, to use a tab-delimited file as input, you need to use the `--separator` flag, escaping the tab character like so:

```
$ csvloader --separator=$'\t'  \
            -f employees.tab  employees
```

Arguments

--batch *{integer}*
: Specifies the number of rows to submit in a batch. If you do not specify an insert procedure, rows of input are sent in batches to maximize overall throughput. You can specify how many rows are sent in each batch using the `--batch` flag. The default batch size is 200. If you use the `--procedure` flag, no batching occurs and each row is sent separately.

--blank {error | null | empty }
: Specifies what to do with missing values in the input. By default, if a line contains a missing value, it is interpreted as a null value in the appropriate datatype. If you do not want missing values to be interpreted as nulls, you can use the --blank argument to specify other behaviors. Specifying `--blank error` results in an error if a line contains any missing values and the line is not inserted. Specifying `--blank empty` returns the corresponding "empty" value in the appropriate datatype. An empty value is interpreted as the following:

  - Zero for all numeric columns
  - Zero, or the Unix epoch value, for timestamp columns
  - An empty or zero-length string for VARCHAR and VARBINARY columns

--columnsizelimit *{integer}*
: Specifies the maximum size of quoted column input, in bytes. Mismatched quotation marks in the input can cause csvloader to read all subsequent input — including line breaks — as part of the column. To avoid excessive memory use in this situation, the flag sets a limit on the maximum number of bytes that will be accepted as input for a column that is enclosed in quotation marks and spans multiple lines. The default is 16777216 (that is, 16MB).

--escape *{character}*
: Specifies the escape character that must precede a separator or quotation character that is supposed to be interpreted as a literal character in the CSV input. The default escape character is the backslash (\).

-f, --file *{file-specification}*
: Specifies the location of a CSV file to read as input. If you do not specify an input file, csvloader reads input from standard input.

--limitrows *{integer}*
: Specifies the maximum number of rows to be read from the input stream. This argument (along with --skip) lets you load a subset of a larger CSV file.

-m, --maxerrors *{integer}*
: Specifies the target number of errors before csvloader stops processing input. Once csvloader encounters the specified number of errors while trying to insert rows, it will stop reading input and end the process. Note that, since csvloader performs inserts asynchronously, it often attempts more inserts before the target number of exceptions are returned from the database. So it is possible more errors could be returned after the target is met. This argument lets you conditionally stop a large loading process if more than an acceptable number of errors occur.

--noquotechar
: Disables the interpretation of quotation characters in the CSV input. All input other than the separator character and line break will be treated as literal input characters.

--nowhitespace
: Specifies that the CSV input must not contain any whitespace between data values and separators. By default, csvloader ignores extra space between values, quotation marks, and the value separators. If

you use this argument, any input lines containing whitespace will generate an error and not be inserted into the database.

--password *{text}*
: Specifies the password to use when connecting to the database. You must specify a username and password if security is enabled for the database. If you specify a username with the --user argument but not the --password argument, VoltDB prompts for the password. This is useful when writing shell scripts because it avoids having to hardcode passwords as plain text in the script.

--port *{port-number}*
: Specifies the network port to use when connecting to the database. If you do not specify a port, csvloader uses the default client port 21212.

-p, --procedure *{procedure-name}*
: Specifies a stored procedure to use for loading each record from the data file. The named procedure must exist in the database schema and must accept the fields of the data record as input parameters. By default, csvloader uses a custom procedure to batch multiple rows into a single insert operation. If you explicitly name a procedure, batching does not occur.

--quotechar *{character}*
: Specifies the quotation character that is used to enclose values. By default, the quotation character is the double quotation mark (").

-r, --reportdir *{directory}*
: Specifies the directory where csvloader writes the three output files. By default, csvloader writes output files to the current working directory. This argument lets you redirect output to an alternative location.

--s, --servers=*server-id[,...]*
: Specifies the network address of one or more nodes of a database cluster. By default, csvloader attempts to insert the CSV data into a database on the local system (localhost). To load data into a remote database, use the --servers argument to specify the database nodes the loader should connect to.

--separator *{charactor}*
: Specifies the character used to separate individual values in the input. By default, the separator character is the comma (,).

--skip *{integer}*
: Specifies the number of lines from the input stream to skip before inserting rows into the database. This argument (along with --limitrows) lets you load a subset of a larger CSV file.

--strictquotes
: Specifies that all values in the CSV input must be enclosed in quotation marks. If you use this argument, any input lines containing unquoted values will generate an error and not be inserted into the database.

--update
: Specifies that existing records with a matching primary key are updated, rather than being rejected. By default, csvloader attempts to create new records. The --update flag lets you load updates to existing records — and create new records where the primary key does not already exist. To use --update, the table *must* have a primary key.

--user *{text}*
: Specifies the username to use when connecting to the database. You must specify a username and password if security is enabled for the database.

Examples

The following example loads the data from a CSV file, `languages.csv`, into the helloworld table from the Hello World example database and redirects the output files to the ./logs subfolder.

```
$ csvloader helloworld -f languages.csv -r ./logs
```

The following example performs the same function, providing the input interactively.

```
$ csvloader helloworld -r ./logs
"Hello", "World", "English"
"Bonjour", "Monde", "French"
"Hola", "Mundo", "Spanish"
"Hej", "Verden", "Danish"
"Ciao", "Mondo", "Italian"
CTRL-D
```

jdbcloader

jdbcloader — Extracts a table from another database via JDBC and inserts it into a VoltDB table.

Syntax

jdbcloader *table-name* [arguments]

jdbcloader -p *procedure-name* [arguments]

Description

The jdbcloader command uses the JDBC interface to fetch all records from the specified table in a remote database and then insert those records into a matching table in VoltDB. The most common way to use jdbcloader is to copy matching tables from another database to VoltDB. In this case, you specify the name of the table, plus any JDBC-specific arguments that are needed. Usually, the required arguments are the JDBC connection URL, the source table, the username, password, and local JDBC driver. For example:

```
$ jdbcloader employees \
     --jdbcurl=jdbc:postgresql://remotesvr/corphr \
     --jdbctable=employees \
     --jdbcuser=charlesdickens \
     --jdbcpassword=bleakhouse \
     --jdbcdriver=org.postgresql.Driver
```

In addition to inserting all valid content into the specified database table, jdbcloader creates three output files:

- **Error log** — The error log provides details concerning any errors that occur while processing the input file. This includes errors that occur attempting the insert into VoltDB. For example, if two rows contain the same value for a column that is declared as unique, the error log indicates that the second insert fails due to a constraint violation.
- **Failed input** — A separate file contains the contents of each record that failed to load. The records are stored in CSV (comma-separated value) format. This file is useful because it allows you to correct any formatting issues and retry just the failed content using the csvloader.
- **Summary report** — Once all input records are processed, jdbcloader generates a summary report listing how many records were read, how many were successfully loaded and how long the operation took.

All three files are created, by default, in the current working directory using "jdbcloader" and the table name as prefixes. For example, using jdbcloader to insert contestants into the sample voter database creates the following files:

jdbcloader_contestants_insert_log.log
jdbcloader_contestants_insert_invalidrows.csv
jdbcloader_contestants_insert_report.log

It is possible to use jdbcloader to perform other input operations. For example, if the source table does not have the same structure as the target table, you can use a custom stored procedure to perform the necessary translation from one to the other by specifying the procedure name on the command line with the --procedure flag:

```
$ jdbcloader --procedure translateEmpRecords \
```

```
 --jdbcurl=jdbc:postgresql://remotesvr/corphr \
  --jdbctable=employees \
  --jdbcuser=charlesdickens \
  --jdbcpassword=bleakhouse \
  --jdbcdriver=org.postgresql.Driver
```

Arguments

--batch *{integer}*
: Specifies the number of rows to submit in a batch to the target VoltDB database. If you do not specify an insert procedure, rows of input are sent in batches to maximize overall throughput. You can specify how many rows are sent in each batch using the `--batch` flag. The default batch size is 200. If you use the `--procedure` flag, no batching occurs and each row is sent separately.

--fetchsize *{integer}*
: Specifies the number of records to fetch in each JDBC call to the source database. The default fetch size is 100 records,

--jdbcdriver *{class-name}*
: Specifies the class name of the JDBC driver to invoke. The driver must exist locally and be accessible either from the CLASSPATH environment variable or in the `lib/extension` directory where VoltDB is installed.

--jdbcpassword *{text}*
: Specifies the password to use when connecting to the source database via JDBC. You must specify a username and password if security is enabled on the source database.

--jdbctable *{table-name}*
: Specifies the name of source table on the remote database. By default, jdbcloader assumes the source table has the same name as the target VoltDB table.

--jdbcurl *{connection-URL}*
: Specifies the JDBC connection URL for the source database. This argument is required.

--jdbcuser *{text}*
: Specifies the username to use when connecting to the source database via JDBC. You must specify a username and password if security is enabled on the source database.

--limitrows *{integer}*
: Specifies the maximum number of rows to be read from the input stream. This argument lets you load a subset of a remote database table.

-m, --maxerrors *{integer}*
: Specifies the target number of errors before jdbcloader stops processing input. Once jdbcloader encounters the specified number of errors while trying to insert rows, it will stop reading input and end the process. Note that, since jdbcloader performs inserts asynchronously, it often attempts more inserts before the target number of exceptions are returned from the database. So it is possible more errors could be returned after the target is met. This argument lets you conditionally stop a large loading process if more than an acceptable number of errors occur.

--password *{text}*
: Specifies the password to use when connecting to the database. You must specify a username and password if security is enabled for the database. If you specify a username with the --user argument but not the --password argument, VoltDB prompts for the password. This is useful when writing shell scripts because it avoids having to hardcode passwords as plain text in the script.

--port *{port-number}*
: Specifies the network port to use when connecting to the VoltDB database. If you do not specify a port, jdbcloader uses the default client port 21212.

-p, --procedure *{procedure-name}*
: Specifies a stored procedure to use for loading each record from the input source. The named procedure must exist in the VoltDB database schema and must accept the fields of the data record as input parameters. By default, jdbcloader uses a custom procedure to batch multiple rows into a single insert operation. If you explicitly name a procedure, batching does not occur.

-r, --reportdir *{directory}*
: Specifies the directory where jdbcloader writes the three output files. By default, jdbcloader writes output files to the current working directory. This argument lets you redirect output to an alternative location.

--s, --servers=*server-id[,...]*
: Specifies the network address of one or more nodes of a VoltDB cluster. By default, jdbcloader attempts to insert the data into a VoltDB database on the local system (localhost). To load data into a remote database, use the --servers argument to specify the VoltDB database nodes the loader should connect to.

--user *{text}*
: Specifies the username to use when connecting to the VoltDB database. You must specify a username and password if security is enabled on the target database.

Example

The following example loads records from the Products table of the Warehouse database on server hq.mycompany.com and writes the records into the Products table of the VotlDB database on servers svrA, svrB, and svrC, using the MySQL JDBC driver to access to source database. Note that the --jdbctable flag is not needed since the source and target tables have the same name.

```
$ jdbcloader Products --servers="svrA,svrB,svrC" \
    --jdbcurl="jdbc:mysql://hq.mycompany.com/warehouse" \
    --jdbcdriver="com.mysql.jdbc.Driver" \
    --jdbcuser="ceo" \
    --jdbcpassword="headhoncho"
```

kafkaloader

kafkaloader — Imports data from a Kafka message queue into the specified database table.

Syntax

kafkaloader *table-name* [arguments]

Description

The kafkaloader utility loads data from a Kafka message queue and inserts each message as a separate record into the specified database table. Apache Kafka is a distributed messaging service that lets you set up message queues which are written to and read from by "producers" and "consumers", respectively. In the Apache Kafka model, the kafkaloader acts as a "consumer".

When you start the kafkaloader, you must specify at least three arguments:

- The database table
- The Kafka server to read messages from, specified using the --zookeeper flag
- The Kafka "topic" where the messages are stored, specified using the --topic flag

For example:

```
$ kafkaloader --zookeeper=quesvr:2181 --topic=voltdb_customer customer
```

Note that Kafka does not impose any specific format on the messages it manages. The format of the messages are application specific. In the case of kafkaloader, VoltDB assumes the messages are encoded as standard comma-separated value (CSV) strings, with the values representing the columns of the table in the order listed in the schema definition. Each Kafka message contains a single row to be inserted into the database table.

It is also important to note that, unlike the csvloader which reads a static file, the kafkaloader is reading from a queue where messages can be written at any time, on an ongoing basis. Therefore, the kafkaloader process does not stop when it reads the last message on the queue; instead it continues to monitor the queue and process any new messages it receives. The kafkaloader process will continue to read from the queue until one of the following events occur:

- The connection to all of the VoltDB servers is broken and so kafkaloader can no longer access the VoltDB database.
- The maximum number of errors (specified by --maxerrors) is reached.
- The user explicitly stops the process.

The kafkaloader will *not* terminate if it loses its connection to the Kafka zookeeper. Therefore, it is important to monitor the Kafka service and restart the kafkaloader if and when the Kafka service is interrupted.

Finally, kafkaloader acks, or acknowledges, receipt of the messages from Kafka as soon as they are read from the queue. The messages are then batched for insert into the VoltDB database. This means that the queue messages are acked regardless of whether they are successfully inserted into the database or not. It is also possible messages may be lost if the loader process stops between when the messages are read and the insert transaction is sent to the VoltDB database.

Arguments

--batch *{integer}*
: Specifies the number of rows to submit in a batch. By default, rows of input are sent in batches to maximize overall throughput. You can specify how many rows are sent in each batch using the `--batch` flag. The default batch size is 200.

  Note that --batch and --flush work together. Whichever limit is reached first triggers an insert to the database.

--flush *{integer}*
: Specifies the maximum number of seconds before pending data is written to the database. The default flush period is 10 seconds.

  If data is inserted into the kafka queue intermittently, there could be a long delay between when data is read from the queue and when enough records have been read to meet the `--batch` limit. The flush value avoids unnecessary delays in this situation by periodically writing all pending data. If the flush limit is reached, all pending records are written to the database, even if the `--batch` limit has not been satisfied.

-m, --maxerrors *{integer}*
: Specifies the target number of input errors before kafkaloader stops processing input. Once kafkaloader encounters the specified number of errors while trying to insert rows, it will stop reading input and end the process.

  The default maximum error count is 100. Since kafka import can be an persistent process, you can avoid having input errors cancel ongoing import by setting the maximum error count to zero, which means that the loader will continue to run no matter how many input errors are generated.

--password *{text}*
: Specifies the password to use when connecting to the database. You must specify a username and password if security is enabled for the database. If you specify a username with the --user argument but not the --password argument, VoltDB prompts for the password. This is useful when writing shell scripts because it avoids having to hardcode passwords as plain text in the script.

--port *{port-number}*
: Specifies the network port to use when connecting to the database. If you do not specify a port, kafkaloader uses the default client port 21212.

-p, --procedure *{procedure-name}*
: Specifies a stored procedure to use for loading each record from the data file. The named procedure must exist in the database schema and must accept the fields of the data record as input parameters. By default, kafkaloader uses a custom procedure to batch multiple rows into a single insert operation. If you explicitly name a procedure, batching does not occur.

--s, --servers=*server-id[,...]*
: Specifies the network address of one or more nodes of a database cluster. By default, kafkaloader attempts to insert the data into a database on the local system (localhost). To load data into a remote database, use the --servers argument to specify the database nodes the loader should connect to.

--update
: Specifies that existing records with a matching primary key are updated, rather than being rejected. By default, kafkaloader attempts to create new records. The --update flag lets you load updates to existing records — and create new records where the primary key does not already exist. To use --update, the table *must* have a primary key.

--user *{text}*
: Specifies the username to use when connecting to the database. You must specify a username and password if security is enabled for the database.

--zookeeper *{kafka-server[:port]}*
: Specifies the network address of the Kafka Zookeeper instance to connect to. The Kafka service must be running Kafka 0.8.

Examples

The following example starts the kafkaloader to read messages from the voltdb_customer topic on the Kafka server quesvr:2181, inserting the resulting records into the CUSTOMER table in the VoltDB cluster that includes the servers dbsvr1, dbsvr2, and dbsvr3. The process will continue, regardless of errors, until connection to the VoltDB database is lost or the user explicitly ends the process.

```
$ kafkaloader --maxerrors=0 customer \
   --zookeeper=quesvr:2181 --topic=voltdb_customer
   --servers=dbsvr1,dbsvr2,dbsvr3
```

sqlcmd

sqlcmd — Starts an interactive command prompt for issuing SQL queries to a running VoltDB database

Syntax

```
sqlcmd [args...]
```

Description

The sqlcmd command lets you query a VoltDB database interactively. You can execute SQL statements, invoke stored procedures, or use directives to examine the structure of the database. When sqlcmd starts it provides its own command line prompt until you exit the session. When you start the session, you can optionally specify one or more database servers to access. By default, sqlcmd accesses the database on the local system via localhost.

At the sqlcmd prompt, you have several options:

- **SQL queries** — You can enter ad hoc SQL queries that are run against the database and the results displayed. You must terminate the query with a semi-colon and carriage return.
- **Procedure calls** — You can have sqlcmd execute a stored procedure. You identify a procedure call with the **exec** directive, followed by the procedure class name, the procedure parameters, and a closing semi-colon. For example, the following sqlcmd directive executes the @SystemCatalog system procedure requesting information about the stored procedures.

  ```
  $ sqlcmd
  1> exec @SystemCatalog procedures;
  ```

  Note that string values can be entered as plain text or enclosed in single quotation marks. Also, the **exec** directive must be terminated by a semi-colon.
- **Show and Explain directives** — The **show** and **explain** directives let you examine the structure of the schema and user-defined stored procedures. Valid directives are:
  - SHOW CLASSES — Lists the user-defined classes in the database. Classes are grouped into procedures classes (those that can be invoked as a stored procedure) and non-procedure classes (shared classes that cannot themselves be called as stored procedures but can be invoked from within stored procedures).
  - SHOW PROCEDURES — Lists the user-defined, default, and system procedures for the current database, including the type and number of arguments for each.
  - SHOW TABLES — Lists the tables in the schema.
  - EXPLAIN *{sql-query}* — Displays the execution plan for the specified SQL statement.
  - EXPLAINPROC *{procedure-name}* — Displays the execution plan for the specified stored procedure.
- **Class management directives** — The load classes and remove classes directives let you add and remove Java classes from the database:
  - LOAD CLASSES —Loads any classes in the specified JAR file. If a class already exists in the database, it is replaced by the new class definition in the JAR file.

    - REMOVE CLASSES — Removes any classes that match the specified class name string. The class specification can include wildcards.

- **Command recall** — You can recall previous commands using the up and down arrow keys. Or you can recall a specific command by line number (the command prompt shows the line number) using the **recall** command. For example:

  ```
  $ sqlcmd
  1> select * from votes;
  2> show procedures;
  3> recall 1
  select * from votes;
  ```

  Once recalled, you can edit the command before reissuing it using typical editing keys, such as the left and right arrow keys and backspace and delete.

- **Script files** — You can run multiple queries or stored procedures in a single command using the **file** directive. The **file** directive takes a text file as an argument and executes all of the SQL queries and **exec** directives in the file as if they were entered interactively. Any **show**, **explain**, **recall**, or **exit** directives are ignored. For example, the following command processes all of the SQL queries and procedure invocations in the file `myscript.sql`:

  ```
  $ sqlcmd
  1> file myscript.sql;
  ```

  If the file contains only data definition language (DDL) statements, you can also have the entire file processed as a batch by including the `-batch` argument:

  ```
  $ sqlcmd
  1> file -batch myscript.sql;
  ```

  If a file or set of statements includes both DDL and DML statements, you can still batch process a group of DDL statements by enclosing the statements in a `file -inlinebatch` directive and the specified end marker. For example, in the following code the three CREATE PROCEDURE statements are processed as a batch:

  ```
  load classes myprocs.jar;
  file -inlinebatch END_OF_BATCH
  CREATE PROCEDURE FROM CLASS procs.AddEmployee;
  CREATE PROCEDURE FROM CLASS procs.ChangeDept;
  CREATE PROCEDURE FROM CLASS procs.PromoteEmployee;
  END_OF_BATCH
  ```

  Batch processing the DDL statements has two effects:

  - Batch processing can significantly improve performance since all of the schema changes are processed and distributed to the cluster nodes at one time, rather than individually for each statement.
  - The batch operates as a transaction, succeeding or failing as a unit. If any statement fails, all of the schema changes are rolled back.

- **Exit** — When you are done with your interactive session, enter the **exit** directive to end the session and return to the shell prompt.

To run a sqlcmd command without starting the interactive prompt, you can pipe the command through standard input to the sqlcmd command. For example:

```
$ echo "select * from contestants;" | sqlcmd
```

In general, the sqlcmd commands are not case sensitive and must be terminated by a semi-colon. However, the semi-colon is optional for the **exit**, **file**, and **recall** directives. Also, **list** and **quit** are supported as synonyms for the **show** and **exit** directives, respectively.

Arguments

--help
: Displays the sqlcmd help text then returns to the shell prompt.

--servers=*server-id[,...]*
: Specifies the network address of one or more nodes in the database cluster. By default, sqlcmd attempts to connect to a database on localhost.

--port=*port-num*
: Specifies the port number to use when connecting to the database servers. All servers must be using the same port number. By default, sqlcmd connects to the standard client port (21212).

--user=*user-id*
: Specifies the username to use for authenticating to the database. The username is required if the database has security enabled.

--password=*{text}*
: Specifies the password to use when connecting to the database. You must specify a username and password if security is enabled for the database. If you specify a username with the --user argument but not the --password argument, VoltDB prompts for the password. This is useful when writing shell scripts because it avoids having to hardcode passwords as plain text in the script.

--output-format={csv | fixed | tab}
: Specifies the format of the output of query results. Output can be formatted as comma-separated values (csv), fixed monospaced text (fixed), or tab-separated text fields (tab). By default, the output is in fixed monospaced text.

--output-skip-metadata
: Specifies that the column headings and other metadata associated with query results are not displayed. By default, the output includes such metadata. However, you can use this argument, along with the `--output-format` argument, to write just the data itself to an output file.

--query-timeout=*time-limit*
: Specifies a time limit for read-only queries. Any read-only queries that exceed the time limit are canceled and control returned to the user. Specify the time out as an integer number of milliseconds. The default timeout is set in the cluster deployment file (or set to 10 seconds by default, if not set by the deployment file). Only users with ADMIN privileges can set a sqlcmd timeout longer than the cluster-wide setting.

Example

The following example demonstrates an sqlcmd session, accessing the voter sample database running on node zeus.

```
$ sqlcmd --servers=zeus
SQL Command :: zeus:21212
1> select * from contestants;
 1 Edwina Burnam
```

```
 2 Tabatha Gehling
 3 Kelly Clauss
 4 Jessie Alloway
 5 Alana Bregman
 6 Jessie Eichman

(6 row(s) affected)
2> select sum(num_votes) as total, contestant_number from
v_votes_by_contestant_number_State group by contestant_number
order by total desc;
TOTAL   CONTESTANT_NUMBER
------- ------------------
 757240                  1
 630429                  6
 442962                  5
 390353                  4
 384743                  2
 375260                  3


(6 row(s) affected)
3> exit
$
```

voltadmin

voltadmin — Performs administrative functions on a VoltDB database.

Syntax

```
voltadmin {command} [args...]
```

Description

The voltadmin command allows you to perform administrative tasks on a VoltDB database. You specify the database server to access and, optionally, authentication credentials using arguments to the voltadmin command. Individual administrative commands may have they own unique arguments as well.

Arguments

The following global arguments are available for all voltadmin commands.

-h, --help
: Displays information about how to use a command. The --help flag and the help command perform the same function.

-H, --host=*server-id[:port]*
: Specifies which database server to connect to. You can specify the server as a network address or hostname. By default, voltadmin attempts to connect to a database on localhost. You can optionally specify the port number. If you do not specify a port, voltadmin uses the default admin port.

-p, --password=*{text}*
: Specifies the password to use when connecting to the database. You must specify a username and password if security is enabled for the database. If you specify a username with the --user argument but not the --password argument, VoltDB prompts for the password. This is useful when writing shell scripts because it avoids having to hardcode passwords as plain text in the script.

-u, --user=*user-id*
: Specifies the username to use for authenticating to the database. The username is required if the database has security enabled.

-v, -verbose
: Displays additional information about the specific commands being executed.

Commands

The following are the administrative functions that you can invoke using voltadmin.

help [*command*]
: Displays information about the usage of individual commands or, if you do not specify a command, summarizes usage information for all commands. The **help** command and **--help** qualifier are synonymous.

dr reset
: Resets the database replication (DR) connection on a master database. Performing a reset breaks the existing DR connection, deletes pending binary logs and stops the queuing of DR data. This command

is useful for eliminating unnecessary resource usage on a master database after the replica stops or is promoted. Note, however, after a reset DR must start over from scratch; it cannot be restarted where it left off.

pause [--wait]
: Pauses the database, stopping any additional activity on the client port. Normally, pause returns immediately. However, you can use the **--wait** flag to have the command wait until all pending transactions are processed and all database replication (DR) and export queues are flushed. Use of **--wait** is recommended if you are shutting down the database and do not intend to restart with recover, since **--wait** ensures all associated DR or export data is delivered prior to shutdown.

promote
: Promotes a replica database, stopping replication and enabling read/write queries on the client port.

resume
: Resumes normal database operation after a pause.

save {directory} {unique-ID}
: Creates a snapshot containing the current database contents. The contents are saved to disk on the server(s) using the unique ID as a file prefix and the directory specification as the file path. Additional arguments for the **save** command are:

  --format={ csv | native }
  : Specifies the format of the snapshot files. The allowable formats are CSV (comma-separated value) and native formats. Native format snapshots can be used for restoring the database. CSV files can be used by other utilities (such as spreadsheets or the VoltDB CSV loader) but cannot be restored using the **voltadmin restore** command.

  --blocking
  : Specifies that the snapshot will block all other transactions until the snapshot is complete. The advantage of blocking snapshots is that once the command completes you know the snapshot is finished. The disadvantage is that the snapshot blocks ongoing use of the database.

    By default, voltadmin performs non-blocking snapshots so as not to interfere with ongoing database operation. However, note that the non-blocking **save** command only starts the snapshot. You must use **show snapshots** to determine when the snapshot process is finished if you want to know when it is safe, for example, to shutdown the database.

  --skiptables={ *table-name [,...]* }
  : Specifies one or more tables to leave out of the snapshot. Separate multiple table names with commas.

  --tables={ *table-name [,...]* }
  : Specifies what table(s) to include in the snapshot. Only the specified tables will be included. Separate multiple table names with commas.

restore {directory} {unique-ID}
: Restores the data from a snapshot to the database. The data is read from a snapshot using the same unique ID and directory path that were used when the snapshot was created. If no tables exist in the database (that is, no schema has been defined) the restore command will also restore the original schema, including stored procedure classes, before restoring the data.

show snapshots
: Displays information about up to ten previous snapshots. This command is useful for determining the success or failure of snapshots started with the **save** command.

update [catalog] [deployment]
: Updates the deployment configuration or application catalog on a running database. (Note: use of application catalogs is deprecated and updating the catalog is only possible if the database was started with the `schema="catalog"` option in the deployment file.) There are limitations on what changes can be made on a live update. See the description of the @UpdateApplicationCatalog stored procedure for details.

stop {server-id}
: Stops an individual node in the cluster. The **voltadmin stop** command can only be used on a K-safe cluster and will not intentionally shutdown the database. That is, the command will only stop a node if there are enough nodes left for the cluster to remain viable.

shutdown
: Stops the database.

Example

The following example illustrates one way to perform an orderly shutdown of a VoltDB cluster, including pausing and saving the database contents.

```
$ voltadmin pause
$ voltadmin save --blocking ./ mydb
$ voltadmin shutdown
```

voltdb

voltdb — Performs management tasks on the current server, such as starting and recovering the database.

Syntax

```
voltdb collect [args] voltdbroot-directory

voltdb mask [args] source-deployment-file [new-deployment-file]

voltdb create [args]

voltdb recover [args]

voltdb add [args]

voltdb rejoin [args]
```

Description

The voltdb command performs local management functions on the current system, including:

- Starting the database process
- Adding or rejoining a node to a running database cluster
- Collecting log files into a single compressed file
- Hiding passwords in the deployment file

The action that is performed depends on which start action you specify to the voltdb command:

- **collect** — the collect option collects system and process logs related to the VoltDB database process on the current system and compresses them into a single file. This command is helpful when reporting problems to VoltDB support. The only required argument to the collect command is the path to the voltdbroot directory where the database was run. By default, the root directory is a subfolder, `voltdbroot`, in the current working directory where the database was started.
- **mask** — the mask option disguises the passwords associated with user accounts in the security section of the deployment file. The output of the voltdb mask command is either a new deployment file with hashed passwords or, if you do not specify an output file, the original input file is modified in place.
- **create** — the create option starts a new, empty database. This option is useful when starting a database for the first time or if you are updating the cluster configuration by performing a save, shutdown, startup, and restore. (See Chapter 9, *Using VoltDB in a Cluster* for information on updating the cluster.)
- **recover** — the recover option starts the database and restores a previous state from the last known snapshot or from command logs. VoltDB uses the snapshot and command log paths specified in the deployment file when looking for content to restore. If you specify recover as the startup action and no snapshots or command logs can be found, startup will fail.
- **add** — the add option adds the current node to an existing cluster. See Section 9.2, “Updating the Cluster Configuration” for details on elastic scaling.

- **rejoin** — If a node on a K-safe cluster fails, you can use the rejoin start action to have the node (or a replacement node) rejoin the cluster. The host-id you specify with the host argument can be any node still present in the database cluster; it does *not* have to be the host node specified when the cluster was started. You can also request a blocking rejoin by including the **--blocking** flag.

Finally, when creating a new database (**create**) or recovering an existing database (**recover**) you can include the **--replica** flag to create a recipient for database replication.

When starting the database, the voltdb command uses Java to instantiate the process. It is possible to customize the Java environment, if necessary, by passing command line arguments to Java through the following environment variables:

- **LOG4J_CONFIG_PATH** — Specifies an alternate Log4J configuration file.
- **VOLTDB_HEAPMAX** — Specifies the maximum heap size for the Java process. Specify the value as an integer number of megabytes. By default, the maximum heap size is set to 2048.
- **VOLTDB_OPTS** — Specifies all other Java command line arguments. You must include both the command line flag and argument. For example, this environment variable can be used to specify system properties using the -D flag:

```
export VOLTDB_OPTS="-DmyApp.DebugFlag=true"
```

Log Collection Arguments

The following arguments apply specifically to the **collect** action.

--days=*{integer}*
: Specifies the number of days of log files to collect. For example, using `--days=1` will collect data from the last 24 hours. By default, VoltDB collects 14 days (2 weeks) worth of logs.

--dry-run
: Lists the actions that will be taken, including the files that will be collected, but does not actually perform the collection or upload.

--no-prompt
: Specifies that the process will not prompt for input, such as whether to delete the output file after uploading is complete. This argument is useful when starting the collect action from within a script.

--prefix=*{file-prefix}*
: Specifies the prefix for the resulting output file. The default prefix is "voltdb_logs".

--skip-heap-dump
: Specifies that the heap dump not be included in the collection. The heap dump is usually significantly larger than the other log files and can be excluded to save space.

--upload=*{host}*
: Specifies a host server to which the output file will uploaded using SFTP.

--username=*{account-name}*
: Specifies the SFTP account to use when using the --upload option. If you specify --upload but not --username, you will be prompted for the account name.

--password=*{password}*
: Specifies the password to use when using the --upload option. If you specify --upload but not --password, you will be prompted for the password.

Database Startup Arguments

The following arguments apply to the **add**, **create**, **recover**, and **rejoin** start actions.

-H, --host=*{host-id}*
: Specifies the network address of the node that coordinates the starting of the database or the adding or rejoining of a node. When starting a database, all nodes must specify the same host address. Note that once the database starts and the cluster is complete, the role of the host node is complete and all nodes become peers.

  When rejoining or adding a node, you can specify any node still in the cluster as the host. The host for an add or rejoin operation does *not* have to be the same node as the host specified when the database started.

  The default if you do not specify a host when creating or recovering the database is `localhost`. In other words, a single node cluster running on the current system. You must specify a host on the command line when adding or rejoining a node.

  If the host node is using an internal port other than the default (3021), you must specify the port as part of the host string, in the format host:port.

-d, --deployment=*{deployment-file}*
: Specifies the location of the database configuration file. The configuration file is an XML file that defines the database configuration, including the initial size of the cluster and which options are enabled when the database is started. See Appendix E, *Deployment File (deployment.xml)* for a complete description of the syntax of the configuration file.

  The default, if you do not specify a deployment file, is a single node cluster without K-safety and with eight sites per host.

-f, --force
: Starts the server process, even if files (such as command logs or snapshots) already exist in the voltdbroot directory. Creating a new database after previously running a database in the same root directory could overwrite and therefore erase old command logs. Therefore, VoltDB will not by default start a new process with the **create** action if such files exist. If you do not need the files from the previous session, you can use the --force argument to overwrite these files. This argument is valid for the **create** action only.

-l, --license=*{license-file}*
: Specifies the location of the license file, which is required when using the VoltDB Enterprise Edition. The argument is ignored when using the community edition.

-B, --background
: Starts the server process in the background (as a daemon process).

-g, --placement-group=*{group-string}*
: Specifies the location of the server. When the K-safety value is greater than zero, VoltDB uses this argument to assist in rack-aware partitioning. The cluster will attempt to place multiple copies of each partition on different nodes to keep them physically as far apart as possible.

  The physical location is specified by the *group-string*, which is any set of alphanumeric names separated by periods. The names might represent physical servers, racks, switches, or anything meaningful to the user to avoid multiple copies failing at the same time. For example, the string might be "row6.rack5.server3", to ensure that in a virtualized environment copies of the same partition do not get placed on the same physical server or rack if possible. The group strings for all nodes of the

cluster are compared so matches of the rightmost name will be avoided first, then matches of the two rightmost names, and so on.

--ignore=thp
: For Linux systems, allows the database to start even if the server is configured to use Transparent Huge Pages (THP). THP is a known problem for memory-intense applications like VoltDB. So under normal conditions VoltDB will not start if the use of THP is enabled. This flag allows you to ignore that restriction for test purposes. *Do not use this flag on production systems.*

--blocking
: For the rejoin operation only, specifies that the database should block client transactions for the affected partitions until the rejoin is complete.

-r, --replica
: For the create and recover operations only, specifies that the database starts in read-only mode as a replica for database replication (DR). To create or recover a replica database, the deployment file must configure DR appropriately, including a `<connection>` tag identifying the source, or master database, for replication. See Chapter 11, *Database Replication* for more information.

Network Configuration Arguments

In addition to the arguments listed above, there are additional arguments that specify the network configuration for server ports and interfaces when starting a VoltDB database. In most cases, the default values can and should be accepted for these settings. The exceptions are the external and internal interfaces that should be specified whenever there are multiple network interfaces on a single machine.

You can also, optionally, specify a unique network interface for individual ports by preceding the port number with the interface's IP address (or hostname) followed by a colon. Specifying the network interface as part of an individual port setting overrides the default interface for that port set by --externalinterface or --internalinterface.

The network configuration arguments to the **voltdb** command are listed below. See the appendix on server configuration options in the *VoltDB Administrator's Guide* for more information about network configuration options.

--externalinterface=*{ip-address}*
: Specifies the default network interface to use for external ports, such as the admin and client ports.

--internalinterface =*{ip-address}*
: Specifies the default network interface to use for internal communication, such as the internal port.

--publicinterface=*{ip-address}*
: Specifies the public network interface. This argument is useful for hosted systems where the internal and external interfaces may not be generally reachable from the Internet. In which case, specifying the public interface helps the VoltDB Management Center provide publicly accessible links for the cluster nodes.

--admin=*[ip-address:]{port-number}*
: Specifies the admin port. The --admin flag overrides the admin port setting in the deployment file.

--client=*[ip-address:]{port-number}*
: Specifies the client port.

--http=*[ip-address:]{port-number}*
: Specifies the http port. The --http flag both sets the port number (and optionally the interface) and enables the http port, overriding the http setting, if any, in the deployment file.

--internal=*[ip-address:]{port-number}*
: Specifies the internal port used to communicate between cluster nodes.

--replication=*[ip-address:]{port-number}*
: Specifies the replication port used for database replication. The --replication flag overrides the replication port setting in the deployment file.

--zookeeper=*[ip-address:]{port-number}*
: Specifies the zookeeper port. By default, the zookeeper port is bound to the server's internal interface (127.0.0.1).

Examples

The first example shows the command for creating a database using a custom configuration file, `2nodedeploy.xml`, and the node zeus as the host.

```
$ voltdb create --deployment=2nodedeploy.xml \
                --host=zeus
```

The second example takes advantage of the defaults for the host and deployment arguments to start a single-node database.

```
$ voltdb create
```

Appendix E. Deployment File (deployment.xml)

The deployment file describes the physical configuration of a VoltDB database cluster at runtime, including the number of hosts in the cluster and the number of sites per hosts, among other things. This appendix describes the syntax for each component within the deployment file.

The deployment file is a fully-conformant XML file. If you are unfamiliar with XML, see Section E.1, "Understanding XML Syntax" for a brief explanation of XML syntax.

E.1. Understanding XML Syntax

The deployment file is a fully-conformant XML file. 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 hostcount and sitesperhost attributes of the cluster element are assigned values of "2" and "4", respectively.

```
<deployment>
   <cluster hostcount="2" sitesperhost="4">
   </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 hostcount="2" sitesperhost="4"/>
   <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/.

E.2. The Structure of the Deployment File

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

Figure E.1, "Deployment XML Structure" shows the structure of the deployment file. The indentation indicates the hierarchical parent-child relationships of the elements and an ellipsis (...) shows where an element may appear multiple times.

Figure E.1. Deployment XML Structure

```
<deployment>
    <cluster/>
    <paths>
        <commandlog/>
        <commandlogsnapshot/>
        <exportoverflow/>
        <snapshots/>
        <voltdbroot/>
    </paths>
    <admin-mode/>
    <commandlog>
        <frequency/>
    <commandlog/>
    <dr>
         <connection/>
    </dr>
    <export>
        <configuration>
            <property/>...
        </configuration>...
    </export>
    <heartbeat/>
    <httpd>
        <jsonapi/>
    </httpd>
    <import>
        <configuration>
            <property/>...
        </configuration>...
    </import>
    <partition-detection>
        <snapshot/>
    </partition-detection>
    <security/>
    <snapshot/>
    <systemsettings>
        <elastic/>
        <resourcemonitor>
            <disklimit>
                <feature/>...
            </disklimit>
            <memorylimit/>
        </resourcemonitor>
        <snapshot/>
        <temptables/>
    </systemsettings>
    <users>
        <user/>...
    </users>
</deployment>
```

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

Table E.1. Deployment File Elements and Attributes

Element	Child of	Parent of	Attributes
deployment*	(root element)	admin-mode, commandlog, cluster, export, heartbeat, httpd, import, partition-detection, paths, security, snapshot, systemsettings, users	
cluster*	deployment		hostcount={int}* sitesperhost={int} kfactor={int}
admin-mode	deployment		port={int} adminstartup={true|false}
heartbeat	deployment		timeout={int}*
partition-detection	deployment	snapshot	enabled={true|false}
snapshot*	partition-detection		prefix={text}*
commandlog	deployment	frequency	enabled={true|false} synchronous={true|false} logsize={int}
frequency	commandlog		time={int} transactions={int}
dr	deployment	connection	id={int}* listen={true|false} port={int}
connection	dr		source={server[,..]}*
export	deployment	configuration	
configuration*	export	property	enabled={true|false} target={text}* type={file|http|jdbc|kafka|rabbitmq|custom} exportconnectorclass={class-name}
property	configuration		name={text}*
import	deployment	configuration	
configuration*	import	property	enabled={true|false} module={text} format={csv|tsv} type={kafka|custom}*
property	configuration		name={text}
httpd	deployment	jsonapi	port={int} enabled={true|false}
jsonapi	httpd		enabled={true|false}
paths	deployment	commandlog, commandlogsnapshot,	

Element	Child of	Parent of	Attributes
		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}*
security	deployment		enabled={true|false} provider={hash|kerberos}
snapshot	deployment		frequency={int}{s|m|h} prefix={text} retain={int} enabled={true|false}
systemsettings	deployment	elastic, query, resourcemonitor, snapshot, temptables	
elastic	systemsettings		duration={int} throughput={int}
query	systemsettings		timeout={int}*
resourcemonitor	systemsettings	disklimit, memorylimit	frequency={int}
disklimit	resourcemonitor	feature	
feature	diskllimit		name={text}* size={int[%]}*
memorylimit	systemsettings		size={int[%]}*
snapshot	systemsettings		priority={int}*
temptables	systemsettings		maxsize={int}*
users	deployment	user	
user	users		name={text}* password={text}* roles={role-name[,..]}

*Required

Appendix F. VoltDB Datatype Compatibility

VoltDB supports eleven datatypes. When invoking stored procedures from different programming languages or queuing SQL statements within a Java stored procedure, you must use an appropriate language-specific value and datatype for arguments corresponding to placeholders in the query. This appendix provides the mapping of language-specific datatypes to the corresponding VoltDB datatype.

In several cases, there are multiple possible language-specific datatypes that can be used. The following tables highlight the best possible matches in bold.

F.1. Java and VoltDB Datatype Compatibility

Table F.1, "Java and VoltDB Datatype Compatibility" shows the compatible Java datatypes for each VoltDB datatype when:

- Calling simple stored procedures defined using the CREATE PROCEDURE AS statement
- Calling default stored procedures created for each table in the schema

Note that when calling user-defined stored procedures written in Java, you can use additional datatypes, including arrays and the VoltTable object, as arguments to the stored procedure, as long as the actual query invocations within the stored procedure use the following datatypes. Within the stored procedure, when queuing SQL statements using the voltdbQueueSql method, implicit type casting is not guaranteed so using the highlighted Java type is recommended.

Another important point to be aware of is that VoltDB only accepts primitive numeric types (byte, short, int, and so on) and not their reference type equivalents (Byte, Short, Integer, etc.).

Table F.1. Java and VoltDB Datatype Compatibility

SQL Datatype	Compatible Java Datatypes	Notes
TINYINT	**byte** short int long String	Larger datatypes (short, int, and long) are valid input types. However, VoltDB throws a runtime error if the value exceeds the allowable range of a TINYINT. String input must be a properly formatted text representation of an integer value in the correct range.
SMALLINT	byte **short** int long String	Larger datatypes (int and long) are valid input types. However, VoltDB throws a runtime error if the value exceeds the allowable range of a SMALLINT. String input must be a properly formatted text representation of an integer value in the correct range.
INTEGER	byte short **int** long	A larger datatype (long) is a valid input type. However, VoltDB throws a runtime error if the value exceeds the allowable range of an INTEGER.

SQL Datatype	Compatible Java Datatypes	Notes
	String	String input must be a properly formatted text representation of an integer value in the correct range.
BIGINT	byte short int **long** String	String input must be a properly formatted text representation of an integer value in the correct range.
FLOAT	**double** float byte short int long String	String input must be a properly formatted text representation of a floating point value.
DECIMAL	**BigDecimal** double float byte short int long String	String input must be a properly formatted text representation of a decimal number.
GEOGRAPHY	(none)	Geospatial input should be converted from Well Known Text (WKT) to a VoltDB native format either using the GeographyValue.fromWKT() method or by passing a String and using the POLYGONFROMTEXT function within the SQL statement.
GEOGRAPHY_POINT	(none)	Geospatial input should be converted from Well Known Text (WKT) to a VoltDB native format either using the GeographyPointValue.fromWKT() method or by passing a String and using the POINTFROMTEXT function within the SQL statement.
VARCHAR()	**String** byte[] byte short int long float double BigDecimal VoltDB TimestampType	Byte arrays are interpreted as UTF-8 encoded string values. String objects can use other encodings. Numeric and timestamp values are converted to their string representation. For example, the double value 13.25 is interpreted as "13.25" when converted to a VARCHAR.
VARBINARY()	String **byte[]**	String input is interpreted as a hex-encoded binary value.

SQL Datatype	Compatible Java Datatypes	Notes
TIMESTAMP	**VoltDB TimestampType** int long String	For String variables, the text must be formatted as either `YYYY-MM-DD hh.mm.ss.nnnnnn` or just the date portion `YYYY-MM-DD`.

Appendix G. System Procedures

VoltDB provides system procedures that perform system-wide administrative functions. You can invoke system procedures interactively using the sqlcmd utility, or you can invoke them programmatically like other stored procedures, using the VoltDB client method callProcedure.

This appendix describes the following system procedures.

- @AdHoc
- @Explain
- @ExplainProc
- @GetPartitionKeys
- @Pause
- @Promote
- @Quiesce
- @Resume
- @Shutdown
- @SnapshotDelete
- @SnapshotRestore
- @SnapshotSave
- @SnapshotScan
- @SnapshotStatus
- @Statistics
- @StopNode
- @SystemCatalog
- @SystemInformation
- @UpdateApplicationCatalog
- @UpdateClasses
- @UpdateLogging

@AdHoc

@AdHoc — Executes an SQL statement specified at runtime.

Syntax

```
@AdHoc String SQL-statement
```

Description

The @AdHoc system procedure lets you perform arbitrary SQL statements on a running VoltDB database.

You can execute multiple SQL statements — either queries or data definition language (DDL) statements — in a single call to @AdHoc by separating the individual statements with semicolons. When you do this, the statements are performed as a single transaction. That is, the statements all succeed as a group or they all roll back if any of them fail. You cannot mix SQL queries and DDL in a single @AdHoc call.

Performance of ad hoc queries is optimized, where possible. However, it is important to note that ad hoc queries are not pre-compiled, like queries in stored procedures. Therefore, use of stored procedures is recommended over @AdHoc for frequent, repetitive, or performance-sensitive queries.

Return Values

Returns one VoltTable for each statement, with as many rows as there are records returned by the statement. The column names and datatypes match the names and datatypes of the fields returned by the query.

Examples

The following program example uses @AdHoc to execute an SQL SELECT statement and display the number of reservations for a specific customer in the flight reservation database.

```
try {
    VoltTable[] results = client.callProcedure("@AdHoc",
       "SELECT COUNT(*) FROM RESERVATION " +
       "WHERE CUSTOMERID=" + custid).getResults();
    System.out.printf("%d reservations found.\n",
        results[0].fetchRow(0).getLong(0));
}
catch (Exception e) {
    e.printStackTrace();
}
```

Note that you do not need to explicitly invoke @AdHoc when using sqlcmd. You can type your statement directly into the sqlcmd prompt, like so:

```
$ sqlcmd
1> SELECT COUNT(*) FROM RESERVATION WHERE CUSTOMERID=12345;
```

@Explain

@Explain — Returns the execution plan for the specified SQL query.

Syntax

```
@Explain String SQL-statement
```

Description

The @Explain system procedure evaluates the specified SQL query and returns the resulting execution plan. Execution, or explain, plans describe how VoltDB expects to execute the query at runtime, including what indexes are used, the order the tables are joined, and so on. Execution plans are useful for identifying performance issues in query design. See the chapter on execution plans in the *VoltDB Guide to Performance and Customization* for information on how to interpret the plans.

Return Values

Returns one VoltTable with one row and one column.

Name	Datatype	Description
EXECUTION_PLAN	VARCHAR	The execution plan as text.

Examples

The following program example uses @Explain to evaluate an ad hoc SQL SELECT statement against the voter sample application.

```
try {
    String query = "SELECT COUNT(*) FROM CONTESTANTS;";
    VoltTable[] results = client.callProcedure("@Explain",
       query ).getResults();
    System.out.printf("Query: %d\nPlan:\n%d",
        query, results[0].fetchRow(0).getString(0));
}
catch (Exception e) {
    e.printStackTrace();
}
```

In the sqlcmd utility, the "explain" command is a shortcut for "exec @Explain". So the following two commands are equivalent:

```
$ sqlcmd
1> exec @Explain 'SELECT COUNT(*) FROM CONTESTANTS';
2> explain SELECT COUNT(*) FROM CONTESTANTS;
```

@ExplainProc

@ExplainProc — Returns the execution plans for all SQL queries in the specified stored procedure.

Syntax

```
@ExplainProc String procedure-name
```

Description

The @ExplainProc system procedure returns the execution plans for all of the SQL queries within the specified stored procedure. Execution, or explain, plans describe how VoltDB expects to execute the queries at runtime, including what indexes are used, the order the tables are joined, and so on. Execution plans are useful for identifying performance issues in query and stored procedure design. See the chapter on execution plans in the *VoltDB Guide to Performance and Customization* for information on how to interpret the plans.

Return Values

Returns one VoltTable with one row for each query in the stored procedure.

Name	Datatype	Description
SQL_STATEMENT	VARCHAR	The SQL query.
EXECUTION_PLAN	VARCHAR	The execution plan as text.

Examples

The following example uses @ExplainProc to evaluate the execution plans associated with the ContestantWinningStates stored procedure in the voter sample application.

```
try {
    VoltTable[] results = client.callProcedure("@ExplainProc",
       "ContestantWinningStates" ).getResults();
    results[0].resetRowPosition();
    while (results[0].advanceRow()) {
         System.out.printf("Query: %d\nPlan:\n%d",
         results[0].getString(0),results[0].getString(1));
    }
}
catch (Exception e) {
    e.printStackTrace();
}
```

In the sqlcmd utility, the "explainproc" command is a shortcut for "exec @ExplainProc". So the following two commands are equivalent:

```
$ sqlcmd
1> exec @ExplainProc 'ContestantWinningStates';
2> explainproc ContestantWinningStates;
```

@GetPartitionKeys

@GetPartitionKeys — Returns a list of partition values, one for every partition in the database.

Syntax

```
@GetPartitionKeys String datatype
```

Description

The @GetPartitionKeys system procedure returns a set of partition values that you can use to reach every partition in the database. This procedure is useful when you want to run a stored procedure in every partition but you do not want to use a multi-partition procedure. By running multiple single-partition procedures, you avoid the impact on latency and throughput that can result from a multi-partition procedure. This is particularly true for longer running procedures. Using multiple, smaller procedures can also help for queries that modify large volumes of data, such as large deletes.

When you call @GetPartitionKeys you specify the datatype of the keys to return as the second parameter. You specify the datatype as a case-insensitive string. Valid options are "INTEGER", "STRING", and "VARCHAR" (where "STRING" and "VARCHAR" are synonyms).

Note that the results of the system procedure are valid at the time they are generated. If the cluster is static (that is, no nodes are being added and any rebalancing is complete), the results remain valid until the next elastic event. However, during rebalancing, the distribution of partitions is likely to change. So it is a good idea to call @GetPartitionKeys once to get the keys, act on them, then call the system procedure again to verify that the partitions have not changed.

Return Values

Returns one VoltTable with a row for every unique partition in the cluster.

Name	Datatype	Description
PARTITION_ID	INTEGER	The numeric ID of the partition.
PARTITION_KEY	INTEGER or STRING	A valid partition key for the partition. The datatype of the key matches the type requested in the procedure call.

Examples

The following example shows the use of sqlcmd to get integer key values from @GetPartitionKeys:

```
$sqlcmd
1> exec @GetPartitionKeys integer;
```

The next example shows a Java program using @GetPartitionKeys to execute a stored procedure to clear out old records, one partition at a time.

```
VoltTable[] results = client.callProcedure("@GetPartitionKeys",
    "INTEGER").getResults();
VoltTable keys = results[0];
for (int k=0;k<keys.getRowCount();k++) {
    long key = keys.fetchRow(k).getLong(1);
```

```
            client.callProcedure("PurgeOldData", key);
        }
```

@Pause

@Pause — Initiates read-only mode on the cluster.

Syntax

```
@Pause
```

Description

The @Pause system procedure initiates admin mode on the cluster. Admin mode puts the database into read-only mode and ensures no further changes to the database can be made through the client port when performing sensitive administrative operations, such as taking a snapshot before shutting down.

While in admin mode, any write transactions on the client port are rejected and return an error status. Read-only transactions, including system procedures, are allowed. However, write transactions such as inserts, deletes, or schema changes are only allowed through the admin port.

Several important points to consider concerning @Pause are:

- @Pause must be called through the admin port, not the standard client port.
- Although write transactions on the client port are rejected in admin mode, existing connections from client applications are not removed.
- To return to normal database operation, you must call the system procedure @Resume on the admin port.

Return Values

Returns one VoltTable with one row.

Name	Datatype	Description
STATUS	BIGINT	Always returns the value zero (0) indicating success.

Examples

It is possible to call @Pause using the **sqlcmd** utility. However, you must explicitly connect to the admin port when starting **sqlcmd** to do this. Also, it is often easier to use the **voltadmin** utility, which connects to the admin port by default. For example, the following commands demonstrate pausing and resuming the database using both **sqlcmd** and **voltadmin**:

```
$ sqlcmd --port=21211
1> exec @Pause;
2> exec @Resume;

$ voltadmin pause
$ voltadmin resume
```

The following program example, if called through the admin port, initiates admin mode on the database cluster.

```
client.callProcedure("@Pause");
```

@Promote

@Promote — Promotes a replica database to normal operation.

Syntax

```
@Promote
```

Description

The @Promote system procedure promotes a replica database to normal operation. During database replication, the replica database only accepts input from the master database. If, for any reason, the master database fails and replication stops, you can use @Promote to change the replica database from a replica to a normal database. When you invoke the @Promote system procedure, the replica exits read-only mode and becomes a fully operational VoltDB database that can receive and execute both read-only and read/write queries.

Note that once a database is promoted, it cannot return to its original role as the receiving end of database replication without first stopping and reinitializing the database as a replica. If the database is *not* a replica, invoking @Promote returns an error.

Return Values

Returns one VoltTable with one row.

Name	Datatype	Description
STATUS	BIGINT	Always returns the value zero (0) indicating success.

Examples

The following programming example promotes a database cluster.

```
client.callProcedure("@Promote");
```

It is also possible to promote a replica database using **sqlcmd** or the **voltadmin promote** command. The following commands are equivalent:

```
$ sqlcmd
1> exec @Promote;

$ voltadmin promote
```

@Quiesce

@Quiesce — Waits for all queued export data to be written to the connector.

Syntax

```
@Quiesce
```

Description

The @Quiesce system procedure waits for any queued export data to be written to the export connector before returning to the calling application. @Quiesce also does an fsync to ensure any pending export overflow is written to disk. This system procedure should be called after stopping client applications and before calling @Shutdown to ensure that all export activity is concluded before shutting down the database.

If export is not enabled, the procedure returns immediately.

Return Values

Returns one VoltTable with one row.

Name	Datatype	Description
STATUS	BIGINT	Always returns the value zero (0) indicating success.

Examples

The following example calls @Quiesce using sqlcmd:

```
$ sqlcmd
1> exec @Quiesce;
```

The following program example uses drain and @Quiesce to complete any asynchronous transactions and clear the export queues before shutting down the database.

```
     // Complete all outstanding activities
try {
    client.drain();
    client.callProcedure("@Quiesce");
}
catch (Exception e) {
    e.printStackTrace();
}

     // Shutdown the database.
try {
    client.callProcedure("@Shutdown");
}
     // We expect an exception when the connection drops.
     // Report any other exception.
catch (org.voltdb.client.ProcCallException e) { }
catch (Exception e) { e.printStackTrace(); }
```

@Resume

@Resume — Returns a paused database to normal operating mode.

Syntax

```
@Resume
```

Description

The @Resume system procedure switches all nodes in a database cluster from admin mode to normal operating mode. In other words, @Resume is the opposite of @Pause.

After calling this procedure, the cluster returns to accepting read/write ad hoc queries and stored procedure invocations from clients connected to the standard client port.

@Resume must be invoked from a connection to the admin port.

Return Values

Returns one VoltTable with one row.

Name	Datatype	Description
STATUS	BIGINT	Always returns the value zero (0) indicating success.

Examples

You can call @Resume using the **sqlcmd** utility. However, you must explicitly connect to the admin port when starting **sqlcmd** to do this. It is often easier to use the **voltadmin resume** command, which connects to the admin port by default. For example, the following commands are equivalent:

```
$ sqlcmd --port=21211
1> exec @Resume;

$ voltadmin resume
```

The following program example uses @Resume to return the cluster to normal operation.

```
client.callProcedure("@Resume");
```

@Shutdown

@Shutdown — Shuts down the database.

Syntax

```
@Shutdown
```

Description

The @Shutdown system procedure performs an orderly shut down of a VoltDB database on all nodes of the cluster.

VoltDB is an in-memory database. By default, data is not saved when you shut down the database. If you want to save the data between sessions, you can enable command logging or save a snapshot (either manually or using automated snapshots) before the shutdown. See Chapter 14, *Command Logging and Recovery* and Chapter 13, *Saving & Restoring a VoltDB Database* for more information.

Note that once the database shuts down, the client connection is lost and the calling program cannot make any further requests to the server.

Examples

The following examples show calling @Shutdown from **sqlcmd** and using the **voltadmin shutdown** command. These two commands are equivalent:

```
$ sqlcmd
1> exec @Shutdown;

$ voltadmin shutdown
```

The following program example uses @Shutdown to stop the database cluster. Note the use of catch to separate out a VoltDB call procedure exception (which is expected) from any other exception.

```
try {
    client.callProcedure("@Shutdown");
}

        // we expect an exception when the connection drops.
catch (org.voltdb.client.ProcCallException e) {
    System.out.println("Database shutdown initiated.");
}
        // report any other exception.
catch (Exception e) {
    e.printStackTrace();
}
```

@SnapshotDelete

@SnapshotDelete — Deletes one or more native snapshots.

Syntax

@SnapshotDelete String[] *directory-path*s, String[] *Unique-IDs*

Description

The @SnapshotDelete system procedure deletes native snapshots from the database cluster. This is a cluster-wide operation and a single invocation will remove the snapshot files from all of the nodes.

The procedure takes two parameters: a String array of directory paths and a String array of unique IDs (prefixes).

The two arrays are read as a series of value pairs, so that the first element of the directory path array and the first element of the unique ID array will be used to identify the first snapshot to delete. The second element of each array will identify the second snapshot to delete. And so on.

@SnapshotDelete can delete native format snapshots only. The procedure cannot delete CSV format snapshots.

Return Values

Returns one VoltTable with a row for every snapshot file affected by the operation.

Name	Datatype	Description
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
PATH	STRING	The directory path where the snapshot file resides.
NONCE	STRING	The unique identifier for the snapshot.
NAME	STRING	The file name.
SIZE	BIGINT	The total size, in bytes, of the file.
DELETED	STRING	String value indicating whether the file was successfully deleted ("TRUE") or not ("FALSE").
RESULT	STRING	String value indicating the success ("SUCCESS") or failure ("FAILURE") of the request.
ERR_MSG	STRING	If the result is FAILURE, this column contains a message explaining the cause of the failure.

Example

The following example uses @SnapshotScan to identify all of the snapshots in the directory `/tmp/voltdb/backup/`. This information is then used by @SnapshotDelete to delete those snapshots.

```
try {
     results = client.callProcedure("@SnapshotScan",
```

```
                                        "/tmp/voltdb/backup/").getResults();
}
catch (Exception e) { e.printStackTrace(); }

VoltTable table = results[0];
int numofsnapshots = table.getRowCount();
int i = 0;

if (numofsnapshots > 0) {
   String[] paths  = new String[numofsnapshots];
   String[] nonces = new String[numofsnapshots];
   for (i=0;i<numofsnapshots;i++) { paths[i] = "/etc/voltdb/backup/"; }
   table.resetRowPosition();
   i = 0;
   while (table.advanceRow()) {
       nonces[i] = table.getString("NONCE");
       i++;
   }

   try {
       client.callProcedure("@SnapshotDelete",paths,nonces);
   }
  catch (Exception e) { e.printStackTrace(); }
}
```

@SnapshotRestore

@SnapshotRestore — Restores a database from disk using a native format snapshot.

Syntax

```
@SnapshotRestore String directory-path, String unique-ID
```

Description

The @SnapshotRestore system procedure restores a previously saved database from disk to memory. The snapshot must be in native format. (You cannot restore a CSV format snapshot using @SnapshotRestore.) The restore request is propagated to all nodes of the cluster, so a single call to @SnapshotRestore will restore the entire database cluster.

The first parameter, *directory-path*, specifies where VoltDB looks for the snapshot files.

The second parameter, *unique-ID*, is a unique identifier that is used as a filename prefix to distinguish between multiple snapshots.

You can perform only one restore operation on a running VoltDB database. Subsequent attempts to call @SnapshotRestore result in an error. Note that this limitation applies to both manual and automated restores. Since command logging often includes snapshots, you should never perform a manual @SnapshotRestore after recovering a database using command logs.

See Chapter 13, *Saving & Restoring a VoltDB Database* for more information about saving and restoring VoltDB databases.

Return Values

Returns one VoltTable with a row for every table restored at each execution site.

Name	Datatype	Description
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
TABLE	STRING	The name of the table being restored.
PARTITION_ID	INTEGER	The numeric ID for the logical partition that this site represents. When using a K value greater than zero, there are multiple copies of each logical partition.
RESULT	STRING	String value indicating the success ("SUCCESS") or failure ("FAILURE") of the request.
ERR_MSG	STRING	If the result is FAILURE, this column contains a message explaining the cause of the failure.

Examples

The following example uses @SnapshotRestore to restore previously saved database content from the path `/tmp/voltdb/backup/` using the unique identifier *flight*.

```
$ sqlcmd
1> exec @SnapshotRestore '/tmp/voltdb/backup/', 'flight';
```

Alternately, you can use the **voltadmin restore** command to perform the same function:

```
$ voltadmin restore /tmp/voltdb/backup/ flight
```

Since there are a number of situations that impact what data is restored, it is a good idea to review the return values to see what tables and partitions were affected. In the following program example, the contents of the VoltTable array is written to standard output so the operator can confirm that the restore completed as expected.

```
VoltTable[] results = null;

try {
     results = client.callProcedure("@SnapshotRestore",
                                    "/tmp/voltdb/backup/",
                                    "flight").getResults();
}
catch (Exception e) {
    e.printStackTrace();
}

for (int t=0; t<results.length; t++) {
    VoltTable table = results[t];
    for (int r=0;r<table.getRowCount();r++) {
        VoltTableRow row = table.fetchRow(r);
        System.out.printf("Node %d Site %d restoring " +
              "table %s partition %d.\n",
              row.getLong("HOST_ID"), row.getLong("SITE_ID"),
              row.getString("TABLE"),row.getLong("PARTITION"));
     }
}
```

@SnapshotSave

@SnapshotSave — Saves the current database contents to disk.

Syntax

@SnapshotSave String *directory-path*, String *unique-ID*, Integer *blocking-flag*

@SnapshotSave String *json-encoded-options*

Description

The @SnapshotSave system procedure saves the contents of the current in-memory database to disk. Each node of the database cluster saves its portion of the database locally.

There are two forms of the @SnapshotSave stored procedure: a procedure call with individual argument parameters and a procedure call with all arguments in a single JSON-encoded string. When you specify the arguments as individual parameters, VoltDB creates a native mode snapshot that can be used to recover or restore the database. When you specify the arguments as a JSON-encoded string, you can request a different format for the snapshot, including CSV (comma-separated value) files that can be used for import into other databases or utilities.

Individual Arguments

When you specify the arguments as individual parameters, you must specify three arguments:

1. The directory path where the snapshot files are stored
2. An identifier that is included in the file names to uniquely identify the files that make up a single snapshot
3. A flag value indicating whether the snapshot should block other transactions until it is complete or not

The resulting snapshot consists of multiple files saved to the directory specified by *directory-path* using *unique-ID* as a filename prefix. The third argument, *blocking-flag*, specifies whether the save is performed synchronously (thereby blocking any following transactions until the save completes) or asynchronously. If this parameter is set to any non-zero value, the save operation will block any following transactions. If it is zero, others transactions will be executed in parallel.

The files created using this invocation are in native VoltDB snapshot format and can be used to restore or recover the database at some later time. This is the same format used for automatic snapshots. See Chapter 13, *Saving & Restoring a VoltDB Database* for more information about saving and restoring VoltDB databases.

JSON-Encoded Arguments

When you specify the arguments as a JSON-encoded string, you can specify what snapshot format you want to create. Table G.1, "@SnapshotSave Options" describes all possible options when creating a snapshot using JSON-encoded arguments.

Table G.1. @SnapshotSave Options

Option	Description

uripath	Specifies the path where the snapshot files are created. Note that, as a JSON-encoded argument, the path must be specified as a URI, not just a system directory path. Therefore, a local directory must be specified using the `file://` identifier, such as "`file:///tmp`", and the path must exist on all nodes of the cluster.
nonce	Specifies the unique identifier for the snapshot.
block	Specifies whether the snapshot should be synchronous (true) and block other transactions or asynchronous (false).
format	Specifies the format of the snapshot. Valid formats are "csv" and "native". When you save a snapshot in CSV format, the resulting files are in standard comma-separated value format, with only one file for each table. In other words, duplicates (from replicated tables or duplicate partitions due to K-safety) are eliminated. CSV formatted snapshots are useful for import or reuse by other databases or utilities. However, they *cannot* be used to restore or recover a VoltDB database. When you save a snapshot in native format, each node and partition saves its contents to separate files. These files can then be used to restore or recover the database. It is also possible to later convert native format snapshots to CSV using the snapshot utilities described in the *VoltDB Administrator's Guide*.
skiptables	Specifies tables to leave out of the snapshot. Use of tables or skiptables allows you to create a partial snapshot of the larger database. Specify the list of tables as a JSON array. For example, the following JSON argument excludes the Areacode and Country tables from the snapshot: `"skiptables":["areacode","country"]`
tables	Specifies tables to include in the snapshot. Use of tables or skiptables allows you to create a partial snapshot of the larger database. Specify the list of tables as a JSON array. For example, the following JSON argument includes only the Employee and Company tables in the snapshot: `"tables":["employee","company"]`

For example, the JSON-encoded arguments to synchronously save a CSV formatted snapshot to /tmp using the unique identifier "mydb" is the following:

```
{uripath:"file:///tmp",nonce:"mydb",block:true,format:"csv"}
```

The block and format arguments are optional. If you do not specify them they default to `block:false` and `format:"native"`. The arguments `uripath` and `nonce` are required. The tables and skiptables arguments are mutually exclusive.

Because the unique identifier is used in the resulting filenames, the identifier can contain only characters that are valid for Linux file names. In addition, hyphens ("-") and commas (",") are not permitted.

Note that it is normal to perform manual saves synchronously, to ensure the snapshot represents a known state of the database. However, automatic snapshots are performed asynchronously to reduce the impact on ongoing database activity.

Return Values

The @SnapshotSave system procedure returns two different VoltTables, depending on the outcome of the request.

Option #1: one VoltTable with a row for every execution site. (That is, the number of hosts multiplied by the number of sites per host.).

Name	Datatype	Description
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
RESULT	STRING	String value indicating the success ("SUCCESS") or failure ("FAILURE") of the request.
ERR_MSG	STRING	If the result is FAILURE, this column contains a message explaining the cause of the failure.

Option #2: one VoltTable with a variable number of rows.

Name	Datatype	Description
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
TABLE	STRING	The name of the database table. The contents of each table is saved to a separate file. Therefore it is possible for the snapshot of each table to succeed or fail independently.
RESULT	STRING	String value indicating the success ("SUCCESS") or failure ("FAILURE") of the request.
ERR_MSG	STRING	If the result is FAILURE, this column contains a message explaining the cause of the failure.

Examples

The following example uses @SnapshotSave to save the current database content in native snapshot format to the path `/tmp/voltdb/backup/` using the unique identifier *flight* on each node of the cluster.

```
$ sqlcmd
1> exec @SnapshotSave '/tmp/voltdb/backup/', 'flight', 1;
```

Alternately, you can use the **voltadmin save** command to perform the same function. When using the **voltadmin save** command, you use the `--blocking` flag instead of a third parameter to request a blocking save:

```
$ voltadmin save --blocking /tmp/voltdb/backup/ flight
```

Note that the procedure call will return successfully even if the save was not entirely successful. The information returned in the VoltTable array tells you what parts of the operation were successful or not. For example, save may succeed on one node but not on another.

The following code sample performs the same function, but also checks the return values and notifies the operator when portions of the save operation are not successful.

```
VoltTable[] results = null;

try { results = client.callProcedure("@SnapshotSave",
                                     "/tmp/voltdb/backup/",
                                     "flight", 1).getResults(); }
```

```
catch (Exception e) { e.printStackTrace(); }

for (int table=0; table<results.length; table++) {
    for (int r=0;r<results[table].getRowCount();r++) {
        VoltTableRow row = results[table].fetchRow(r);
        if (row.getString("RESULT").compareTo("SUCCESS") != 0) {
            System.out.printf("Site %s failed to write " +
                   "table %s because %s.\n",
                   row.getString("HOSTNAME"), row.getString("TABLE"),
                   row.getString("ERR_MSG"));
        }
    }
}
```

@SnapshotScan

@SnapshotScan — Lists information about existing native snapshots in a given directory path.

Syntax

```
@SnapshotScan String directory-path
```

Description

The @SnapshotScan system procedure provides information about any native snapshots that exist within the specified directory path for all nodes on the cluster. The procedure reports the name (prefix) of the snapshot, when it was created, how long it took to create, and the size of the individual files that make up the snapshot(s).

@SnapshotScan does not include CSV format snapshots in its output. Only native format snapshots are listed.

Return Values

On successful completion, this system procedure returns three VoltTables providing the following information:

- A summary of the snapshots found
- Available space in the directories scanned
- Details concerning the Individual files that make up the snapshots

The first table contains one row for every snapshot found.

Name	Datatype	Description
PATH	STRING	The directory path where the snapshot resides.
NONCE	STRING	The unique identifier for the snapshot.
TXNID	BIGINT	The transaction ID of the snapshot.
CREATED	BIGINT	The timestamp when the snapshot was created (in milliseconds).
SIZE	BIGINT	The total size, in bytes, of all the snapshot data.
TABLES_REQUIRED	STRING	A comma-separated list of all the table names listed in the snapshot digest file. In other words, all of the tables that make up the snapshot.
TABLES_MISSING	STRING	A comma-separated list of database tables for which no data can be found. (That is, the corresponding files are missing or unreadable.)
TABLES_INCOMPLETE	STRING	A comma-separated list of database tables with only partial data saved in the snapshot. (That is, data from some partitions is missing.)
COMPLETE	STRING	A string value indicating whether the snapshot as a whole is complete ("TRUE") or incomplete ("FALSE"). If this col-

Name	Datatype	Description
		umn is "FALSE", the preceding two columns provide additional information concerning what is missing.

The second table contains one row for every host.

Name	Datatype	Description
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
PATH	STRING	The directory path specified in the call to the procedure.
TOTAL	BIGINT	The total space (in bytes) on the device.
FREE	BIGINT	The available free space (in bytes) on the device.
USED	BIGINT	The total space currently in use (in bytes) on the device.
RESULT	STRING	String value indicating the success ("SUCCESS") or failure ("FAILURE") of the request.
ERR_MSG	STRING	If the result is FAILURE, this column contains a message explaining the cause of the failure.

The third table contains one row for every file in the snapshot collection.

Name	Datatype	Description
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
PATH	STRING	The directory path where the snapshot file resides.
NAME	STRING	The file name.
TXNID	BIGINT	The transaction ID of the snapshot.
CREATED	BIGINT	The timestamp when the snapshot was created (in milliseconds).
TABLE	STRING	The name of the database table the data comes from.
COMPLETED	STRING	A string indicating whether all of the data was successfully written to the file ("TRUE") or not ("FALSE").
SIZE	BIGINT	The total size, in bytes, of the file.
IS_REPLICATED	STRING	A string indicating whether the table in question is replicated ("TRUE") or partitioned ("FALSE").
PARTITIONS	STRING	A comma-separated string of partition (or site) IDs from which data was taken during the snapshot. For partitioned tables where there are multiple sites per host, there can be data from multiple partitions in each snapshot file. For replicated tables, data from only one copy (and therefore one partition) is required.
TOTAL_PARTITIONS	BIGINT	The total number of partitions from which data was taken.
READABLE	STRING	A string indicating whether the file is accessible ("TRUE") or not ("FALSE").
RESULT	STRING	String value indicating the success ("SUCCESS") or failure ("FAILURE") of the request.

Name	Datatype	Description
ERR_MSG	STRING	If the result is FAILURE, this column contains a message explaining the cause of the failure.

If the system procedure fails because it cannot access the specified path, it returns a single VoltTable with one row and one column.

Name	Datatype	Description
ERR_MSG	STRING	A message explaining the cause of the failure.

Examples

The following example uses @SnapshotScan to list information about the snapshots in the directory `/tmp/voltdb/backup/`.

```
$ sqlcmd
1> exec @SnapshotScan /tmp/voltdb/backup/;
```

The following program example performs the same function, using the VoltTable `toString()` method to display the results of the procedure call:

```
VoltTable[] results = null;

try { results = client.callProcedure("@SnapshotScan",
                        "/tmp/voltdb/backup/").getResults();
}
catch (Exception e) { e.printStackTrace(); }

for (VoltTable t: results) {
   System.out.println(t.toString());
}
```

In the return value, the first VoltTable in the array lists the snapshots and certain status information. The second element of the array provides information about the directory itself (such as used, free, and total disk space). The third element of the array lists specific information about the individual files in the snapshot(s).

@SnapshotStatus

@SnapshotStatus — Lists information about the most recent snapshots created from the current database.

Syntax

```
@SnapshotStatus
```

Description

Warning

The @SnapshotStatus system procedure is being deprecated and may be removed in future versions. Please use the @Statistics "SNAPSHOTSTATUS" selector, which returns the same results, to retrieve information about recent snapshots.

The @SnapshotStatus system procedure provides information about up to ten of the most recent snapshots performed on the current database. The information provided includes the directory path and prefix for the snapshot, when it occurred and how long it took, as well as whether the snapshot was completed successfully or not.

@SnapshotStatus provides status of any snapshots, including both native and CSV snapshots, as well as manual, automated, and command log snapshots.

Note that @SnapshotStatus does not tell you whether the snapshot files still exist, only that the snapshot was performed. You can use the procedure @SnapshotScan to determine what snapshots are available.

Also, the status information is reset each time the database is restarted. In other words, @SnapshotStatus only provides information about the most recent snapshots since the current database instance was started.

Return Values

Returns one VoltTable with a row for every snapshot file in the recent snapshots performed on the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the snapshot was initiated (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
TABLE	STRING	The name of the database table whose data the file contains.
PATH	STRING	The directory path where the snapshot file resides.
FILENAME	STRING	The file name.
NONCE	STRING	The unique identifier for the snapshot.
TXNID	BIGINT	The transaction ID of the snapshot.
START_TIME	BIGINT	The timestamp when the snapshot began (in milliseconds).
END_TIME	BIGINT	The timestamp when the snapshot was completed (in milliseconds).

Name	Datatype	Description
SIZE	BIGINT	The total size, in bytes, of the file.
DURATION	BIGINT	The length of time (in milliseconds) it took to complete the snapshot.
THROUGHPUT	FLOAT	The average number of bytes per second written to the file during the snapshot process.
RESULT	STRING	String value indicating whether the writing of the snapshot file was successful ("SUCCESS") or not ("FAILURE").

Examples

The following example uses @SnapshotStatus to display information about the most recent snapshots performed on the current database:

```
$ sqlcmd
1> exec @SnapshotStatus;
```

The following code example demonstrates how to perform the same function programmatically:

```
VoltTable[] results = null;

try {
   results = client.callProcedure("@SnapshotStatus").getResults();
}
catch (Exception e) { e.printStackTrace(); }

for (VoltTable t: results) {
   System.out.println(t.toString());
}
```

@Statistics

@Statistics — Returns statistics about the usage of the VoltDB database.

Syntax

```
@Statistics String component, Integer delta-flag
```

Description

The @Statistics system procedure returns information about the VoltDB database. The second argument, *component*, specifies what aspect of VoltDB to return statistics about. The third argument, *delta-flag*, specifies whether statistics are reported from when the database started or since the last call to @Statistics where the flag was set.

If the delta-flag is set to zero, the system procedure returns statistics since the database started. If the delta-flag is non-zero, the system procedure returns statistics for the interval since the last time @Statistics was called with a non-zero flag. (If @Statistics has not been called with a non-zero flag before, the first call with the flag set returns statistics since startup.)

Note that in a cluster with K-safety, if a node fails, the statistics reported by this procedure are reset to zero for the node when it rejoins the cluster.

The following are the allowable values of *component*:

"COMMANDLOG [335]"	Returns information about the progress of command logging, including the number of segment files in use and the amount of command log data waiting to be written to disk.
"CPU [336]"	Returns information about the amount of CPU used by each VoltDB server process. CPU usage is returned as a number between 0 and 100 representing the amount of CPU used by the VoltDB process out of the total CPU available for that server.
"DRCONSUMER [336]"	Returns information about the status of database replication on a replica database, including the status and data replication rate of each partition. This information is available only if the database is licensed for database replication and started as a replica.
"DRPRODUCER [337]"	Returns information about the status of database replication on a master database, including how much data is waiting to be sent to the replica. This information is available only if the database is licensed for database replication.
"IMPORTER [338]"	Returns statistics on the import streams, including how many import transactions have succeeded, failed, and been retried and how many rows have been read but not applied yet.
"INDEX [339]"	Returns information about the indexes in the database, including the number of keys for each index and the estimated amount of memory used to store those keys. Separate information is returned for each partition in the database.

"INITIATOR [339]"	Returns information on the number of procedure invocations for each stored procedure (including system and import procedures). The count of invocations is reported for each connection to the database.
"IOSTATS [340]"	Returns information on the number of messages and amount of data (in bytes) sent to and from each connection to the database.
"LIVECLIENTS [340]"	Returns information about the number of outstanding requests per client. You can use this information to determine how much work is waiting in the execution queues.
"MANAGEMENT"	Returns the same information as INDEX [339], INITIATOR [339], IOSTATS [340], MEMORY [341], PROCEDURE [343], and TABLE [346], except all in a single procedure call.
"MEMORY [341]"	Returns statistics on the use of memory for each node in the cluster. MEMORY statistics include the current resident set size (RSS) of the VoltDB server process; the amount of memory used for Java temporary storage, database tables, indexes, and string (including varbinary) storage; as well as other information.
"PARTITIONCOUNT [342]"	Returns information on the number of unique partitions in the cluster. The VoltDB cluster creates multiple partitions based on the number of servers and the number of sites per host requested. So, for example, a 2 node cluster with 4 sites per host will have 8 partitions. However, when you define a cluster with K-safety, there are duplicate partitions. PARTITIONCOUNT only reports the number of unique partitions available in the cluster.
"PLANNER [342]"	Returns information on the use of cached plans within each partition. Queries in stored procedures are planned when the procedure is declared in the schema. However, ad hoc queries must be planned at runtime. To improve performance, VoltDB caches plans for ad hoc queries so they can be reused when a similar query is encountered later. There are two caches: the level 1 cache performs exact matches on queries and the level 2 cache parameterizes constants so it can match queries with the same plan but different input. The planner statistics provide information about the size of each cache, how frequently it is used, and the minimum, maximum, and average execution time of ad hoc queries as a result.
"PROCEDURE [343]"	Returns information on the usage of stored procedures for each site within the database cluster sorted by partition. The information includes the name of the procedure, the number of invocations (for each site), and selected performance information on minimum, maximum, and average execution time.
"PROCEDUREINPUT [343]"	Returns summary information on the size of the input data submitted with stored procedure invocations. PROCEDUREINPUT uses information from PROCEDURE, except it focuses on the input parameters and aggregates data for the entire cluster.
"PROCEDUREOUTPUT [344]"	Returns summary information on the size of the result sets returned by stored procedure invocations. PROCEDUREOUTPUT uses information from PROCEDURE, except it focuses on the result sets and aggregates data for the entire cluster.

"PROCEDURE-PROFILE [344]"	Returns summary information on the usage of stored procedures averaged across all partitions in the cluster. The information from PROCEDUREPROFILE is similar to the information from PROCEDURE, except it focuses on the performance of the individual procedures rather than on procedures by partition. The weighted average across partitions is helpful for determining which stored procedures the application is spending most of its time in.
"REBALANCE [345]"	Returns information on the current progress of rebalancing on the cluster. Rebalancing occurs when one or more nodes are added "on the fly" to an elastic cluster. If no rebalancing is occurring, no data is returned. During a rebalance, this selector returns information about the speed of migration of the data, the latency of rebalance tasks, and the estimated time until completion. For rebalance, the delta flag to the system procedure is ignored. All rebalance statistics are cumulative for the current rebalance activity.
"SNAPSHOTSTATUS [345]"	Returns information about up to ten of the most recent snapshots performed by the database. The results include the directory path and prefix for the snapshot, when it occurred, how long it took, and whether the snapshot was completed successfully or not. The results report on both native and CSV snapshots, as well as manual, automated, and command log snapshots. Note that this selector does not tell you whether the snapshot files still exist, only that the snapshot was performed. Use the @SnapshotScan procedure to determine what snapshots are available.
"TABLE [346]"	Returns information about the database tables, including the number of rows per site for each table. This information can be useful for seeing how well the rows are distributed across the cluster for partitioned tables.

Note that INITIATOR and PROCEDURE report information on both user-declared stored procedures and system procedures. These include certain system procedures that are used internally by VoltDB and are not intended to be called by client applications. Only the system procedures documented in this appendix are intended for client invocation.

Return Values

Returns different VoltTables depending on which component is requested. The following tables identify the structure of the return values for each component. (Note that the MANAGEMENT component returns seven VoltTables.)

COMMANDLOG — Returns a row for every server in the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
OUTSTANDING_BYTES	BIGINT	The size, in bytes, of pending command log data. That is, data for transactions that have been initiated but the log has

Name	Datatype	Description
		yet to be written to disk. For synchronous logging, this value is always zero.
OUTSTANDING_TXNS	BIGINT	The size, in number of transactions, of pending command log data. That is, the number of transactions that have been initiated for which the log has yet to be written to disk. For synchronous logging, this value is always zero.
IN_USE_SEGMENT_COUNT	INTEGER	The total number of segment files currently in use for command logging.
SEGMENT_COUNT	INTEGER	The number of segment files allocated, including currently unused segments.
FSYNC_INTERVAL	INTEGER	The average interval, in milliseconds, between the last 10 fsync system calls.

CPU — Returns a row for every server in the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
PERCENT_USED	BIGINT	The percentage of total CPU available used by the database server process.

DRCONSUMER — Returns two VoltTables. The first table returns a row for every host in the cluster, showing whether a replication snapshot is in progress and if it is, the status of transmission to the replica.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
CLUSTER_ID	INTEGER	The numeric ID assigned to the cluster in the deployment file.
STATE	STRING	A text string indicating the current state of replication. Possible values are: • UNINITIALIZED — DR has not begun yet or has stopped • INITIALIZE — DR is enabled and the replica is attempting to contact the master • SYNC — DR has started and the replica is synchronizing by receiving snapshots of existing data from the master • RECEIVE — DR is underway and the replica is receiving binary logs from the master • END — DR has been canceled for some reason and the replica is stopping DR
REPLICATION_RATE_1M	BIGINT	The average rate of replication over the past minute. The data rate is measured in bytes per second.

Name	Datatype	Description
REPLICATION_RATE_5M	BIGINT	The average rate of replication over the past five minutes. The data rate is measured in bytes per second.

The second table contains information about the replication streams, which consist of a row per partition for each server. The data shows the current state of replication and how much data has been received by the replica.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
CLUSTER_ID	INTEGER	The numeric ID assigned to the cluster in the deployment file.
PARTITION_ID	INTEGER	The numeric ID for the logical partition.
IS_COVERED	STRING	A text string of "true" or "false" indicating whether this partition is currently connected to and receiving data from a matching partition on the master cluster.
COVERING_HOST	STRING	The host name of the server in the master cluster that is providing DR data to this partition. If IS_COVERED is "false", this field is empty.
LAST_RECEIVED_TIMESTAMP	TIMESTAMP	The timestamp of the last transaction received from the master.
LAST_APPLIED_TIMESTAMP	TIMESTAMP	The timestamp of the last transaction successfully applied to this partition on the replica.

DRPRODUCER — Returns two VoltTables. The first table contains information about the replication streams, which consist of a row per partition for each server. The data shows the current state of replication and how much data is currently queued for the replica.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
PARTITION_ID	INTEGER	The numeric ID for the logical partition.
STREAMTYPE	STRING	The type of stream, which can either be "TRANSACTIONS" or "SNAPSHOT".
TOTALBYTES	BIGINT	The total number of bytes currently queued for transmission to the replica.
TOTALBYTESINMEMORY	BIGINT	The total number of bytes of queued data currently held in memory. If the amount of total bytes is larger than the amount in memory, the remainder is kept in overflow storage on disk.
TOTALBUFFERS	BIGINT	The total number of buffers in this partition currently waiting for acknowledgement from the replica. The partitions

Name	Datatype	Description
		buffer the binary logs to reduce overhead and optimize network transfers.
LASTQUEUEDDRID	BIGINT	The ID of the last transaction queued for transmission to the replica.
LASTACKDRID	BIGINT	The ID of the last transaction acknowledged by the replica.
LASTQUEUEDTIMESTAMP	TIMESTAMP	The timestamp of the last transaction queued for transmission to the replica.
LASTACKTIMESTAMP	TIMESTAMP	The timestamp of the last transaction acknowledged by the replica.
ISSYNCED	STRING	A text string indicating whether the database is currently being replicated. If replication has not started, or the overflow capacity has been exceeded (that is, replication has failed), the value of ISSYNCED is "false". If replication is currently in progress, the value is "true".
MODE	STRING	A text string indicating whether this particular partition is replicating data for the replica ("NORMAL") or not ("PAUSED"). Only one copy of each logical partition actually sends data during replication. So for clusters with a K-safety value greater than zero, not all physical partitions will report "NORMAL" even when replication is in progress.

The second table returns a row for every host in the cluster, showing whether a replication snapshot is in progress and if it is, the status of transmission to the replica.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
STATE	STRING	A text string indicating the current state of replication. Possible values are "OFF" (replication is not enabled), 'PENDING" (replication is enabled but not occurring), and "ACTIVE" (replication is enabled and a replica database has initiated DR).
SYNCSNAPSHOTSTATE	STRING	A text string indicating the current state of the synchronization snapshot that begins replication. During normal operation, this value is "NONE" indicating either that replication is not active or that transactions are actively being replicated. If a synchronization snapshot is in progress, this value provides additional information about the specific activity underway.
ROWSINSYNCSNAPSHOT	BIGINT	Reserved for future use.
ROWSACKEDFORSYNCSNAPSHOT	BIGINT	Reserved for future use.

IMPORTER — Returns a separate row for each import stream and each server.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
IMPORTER_NAME	STRING	The name of the import stream.
PROCEDURE_NAME	STRING	The name of the stored procedure invoked by the import stream to insert the incoming data.
SUCCESSES	BIGINT	The number of import transactions that succeeded.
FAILURES	BIGINT	The number of import transactions that failed.
OUTSTANDING_REQUESTS	BIGINT	The number of records read from the import stream and waiting to be inserted into the database.
RETRIES	BIGINT	The number of attempts to replay failed transactions.

INDEX — Returns a row for every index in every execution site.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	BIGINT	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	BIGINT	Numeric ID of the execution site on the host node.
PARTITION_ID	BIGINT	The numeric ID for the logical partition that this site represents. When using a K value greater than zero, there are multiple copies of each logical partition.
INDEX_NAME	STRING	The name of the index.
TABLE_NAME	STRING	The name of the database table to which the index applies.
INDEX_TYPE	STRING	A text string identifying the type of the index as either a hash or tree index and whether it is unique or not. Possible values include the following: CompactingHashMultiMapIndex CompactingHashUniqueIndex CompactingTreeMultiMapIndex CompactingTreeUniqueIndex
IS_UNIQUE	TINYINT	A byte value specifying whether the index is unique (1) or not (0).
IS_COUNTABLE	TINYINT	A byte value specifying whether the index maintains a counter to optimize COUNT(*) queries.
ENTRY_COUNT	BIGINT	The number of index entries currently in the partition.
MEMORY_ESTIMATE	BIGINT	The estimated amount of memory (in kilobytes) consumed by the current index entries.

INITIATOR — Returns a separate row for each connection and the stored procedures initiated by that connection.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
CONNECTION_ID	BIGINT	Numeric ID of the client connection invoking the procedure.
CONNECTION_HOST NAME	STRING	The server name of the node from which the client connection originates. In the case of import procedures, the name of the importer is reported here.
PROCEDURE_NAME	STRING	The name of the stored procedure. If import is enabled, import procedures are included as well.
INVOCATIONS	BIGINT	The number of times the stored procedure has been invoked by this connection on this host node.
AVG_EXECUTION_TIME	INTEGER	The average length of time (in milliseconds) it took to execute the stored procedure.
MIN_EXECUTION_TIME	INTEGER	The minimum length of time (in milliseconds) it took to execute the stored procedure.
MAX_EXECUTION_TIME	INTEGER	The maximum length of time (in milliseconds) it took to execute the stored procedure.
ABORTS	BIGINT	The number of times the procedure was aborted.
FAILURES	BIGINT	The number of times the procedure failed unexpectedly. (As opposed to user aborts or expected errors, such as constraint violations.)

IOSTATS — Returns one row for every client connection on the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
CONNECTION_ID	BIGINT	Numeric ID of the client connection invoking the procedure.
CONNECTION_HOST NAME	STRING	The server name of the node from which the client connection originates.
BYTES_READ	BIGINT	The number of bytes of data sent from the client to the host.
MESSAGES_READ	BIGINT	The number of individual messages sent from the client to the host.
BYTES_WRITTEN	BIGINT	The number of bytes of data sent from the host to the client.
MESSAGES_WRITTEN	BIGINT	The number of individual messages sent from the host to the client.

LIVECLIENTS — Returns a row for every client connection currently active on the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
CONNECTION_ID	BIGINT	Numeric ID of the client connection invoking the procedure.
CLIENT_HOSTNAME	STRING	The server name of the node from which the client connection originates.
ADMIN	TINYINT	A byte value specifying whether the connection is to the client port (0) or the admin port (1).
OUTSTANDING_ REQUEST_BYTES	BIGINT	The number of bytes of data sent from the client currently pending on the host.
OUTSTANDING_ RESPONSE_MESSAGES	BIGINT	The number of messages on the host queue waiting to be retrieved by the client.
OUTSTANDING_ TRANSACTIONS	BIGINT	The number of transactions (that is, stored procedures) initiated on behalf of the client that have yet to be completed.

MEMORY — Returns a row for every server in the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
RSS	INTEGER	The current resident set size. That is, the total amount of memory allocated to the VoltDB processes on the server.
JAVAUSED	INTEGER	The amount of memory (in kilobytes) allocated by Java and currently in use by VoltDB.
JAVAUNUSED	INTEGER	The amount of memory (in kilobytes) allocated by Java but unused. (In other words, free space in the Java heap.)
TUPLEDATA	BIGINT	The amount of memory (in kilobytes) currently in use for storing database records.
TUPLEALLOCATED	BIGINT	The amount of memory (in kilobytes) allocated for the storage of database records (including free space).
INDEXMEMORY	BIGINT	The amount of memory (in kilobytes) currently in use for storing database indexes.
STRINGMEMORY	BIGINT	The amount of memory (in kilobytes) currently in use for storing string, binary, and geospatial data that is not stored "in-line" in the database record.
TUPLECOUNT	BIGINT	The total number of database records currently in memory.
POOLEDMEMORY	BIGINT	The total size of memory (in kilobytes) allocated for tasks other than database records, indexes, and strings. (For example, pooled memory is used for temporary tables while processing stored procedures.)

Name	Datatype	Description
PHYSICALMEMORY	BIGINT	The total size of physical memory (in kilobytes) on the server.
JAVAMAXHEAP	INTEGER	The maximum heap size (in kilobytes) of the Java runtime environment.

PARTITIONCOUNT — Returns one row identifying the total number of partitions and the host that provided that information.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
PARTITION_COUNT	INTEGER	The number of unique or logical partitions on the cluster. When using a K value greater than zero, there are multiple copies of each logical partition.

PLANNER — Returns a row for every planner cache. That is, one cache per execution site, plus one global cache per server. (The global cache is identified by a site and partition ID of minus one.)

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
PARTITION_ID	INTEGER	The numeric ID for the logical partition that this site represents. When using a K value greater than zero, there are multiple copies of each logical partition.
CACHE1_LEVEL	INTEGER	The number of query plans in the level 1 cache.
CACHE2_LEVEL	INTEGER	The number of query plans in the level 2 cache.
CACHE1_HITS	INTEGER	The number of queries that matched and reused a plan in the level 1 cache.
CACHE2_HITS	INTEGER	The number of queries that matched and reused a plan in the level 2 cache.
CACHE_MISSES	INTEGER	The number of queries that had no match in the cache and had to be planned from scratch
PLAN_TIME_MIN	BIGINT	The minimum length of time (in nanoseconds) it took to complete the planning of ad hoc queries.
PLAN_TIME_MAX	BIGINT	The maximum length of time (in nanoseconds) it took to complete the planning of ad hoc queries.
PLAN_TIME_AVG	BIGINT	The average length of time (in nanoseconds) it took to complete the planning of ad hoc queries.
FAILURES	BIGINT	The number of times planning for an ad hoc query failed.

PROCEDURE — Returns a row for every stored procedure that has been executed on the cluster, grouped by execution site.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
PARTITION_ID	INTEGER	The numeric ID for the logical partition that this site represents. When using a K value greater than zero, there are multiple copies of each logical partition.
PROCEDURE	STRING	The class name of the stored procedure.
INVOCATIONS	BIGINT	The total number of invocations of this procedure at this site.
TIMED_INVOCATIONS	BIGINT	The number of invocations used to measure the minimum, maximum, and average execution time.
MIN_EXECUTION_TIME	BIGINT	The minimum length of time (in nanoseconds) it took to execute the stored procedure.
MAX_EXECUTION_TIME	BIGINT	The maximum length of time (in nanoseconds) it took to execute the stored procedure.
AVG_EXECUTION_TIME	BIGINT	The average length of time (in nanoseconds) it took to execute the stored procedure.
MIN_RESULT_SIZE	INTEGER	The minimum size (in bytes) of the results returned by the procedure.
MAX_RESULT_SIZE	INTEGER	The maximum size (in bytes) of the results returned by the procedure.
AVG_RESULT_SIZE	INTEGER	The average size (in bytes) of the results returned by the procedure.
MIN_PARAMETER_SET_SIZE	INTEGER	The minimum size (in bytes) of the parameters passed as input to the procedure.
MAX_PARAMETER_SET_SIZE	INTEGER	The maximum size (in bytes) of the parameters passed as input to the procedure.
AVG_PARAMETER_SET_SIZE	INTEGER	The average size (in bytes) of the parameters passed as input to the procedure.
ABORTS	BIGINT	The number of times the procedure was aborted.
FAILURES	BIGINT	The number of times the procedure failed unexpectedly. (As opposed to user aborts or expected errors, such as constraint violations.)

PROCEDUREINPUT — Returns a row for every stored procedure that has been executed on the cluster, summarized across the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).

Name	Datatype	Description
PROCEDURE	STRING	The class name of the stored procedure.
WEIGHTED_PERC	BIGINT	A weighted average expressed as a percentage of the parameter set size for invocations of this stored procedure compared to all stored procedure invocations.
INVOCATIONS	BIGINT	The total number of invocations of this procedure.
MIN_PARAMETER _SET_SIZE	BIGINT	The minimum parameter set size in bytes.
MAX_PARAMETER _SET_SIZE	BIGINT	The maximum parameter set size in bytes.
AVG_PARAMETER _SET_SIZE	BIGINT	The average parameter set size in bytes.
TOTAL_PARAMETER _SET_SIZE_MB	BIGINT	The total input for all invocations of this stored procedure measured in megabytes.

PROCEDUREOUTPUT — Returns a row for every stored procedure that has been executed on the cluster, summarized across the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
PROCEDURE	STRING	The class name of the stored procedure.
WEIGHTED_PERC	BIGINT	A weighted average expressed as a percentage of the result set size returned by invocations of this stored procedure compared to all stored procedure invocations.
INVOCATIONS	BIGINT	The total number of invocations of this procedure.
MIN_RESULT_SIZE	BIGINT	The minimum result set size in bytes.
MAX_RESULT_SIZE	BIGINT	The maximum result set size in bytes.
AVG_RESULT_SIZE	BIGINT	The average result set size in bytes.
TOTAL_RESULT _SIZE_MB	BIGINT	The total output returned by all invocations of this stored procedure measured in megabytes.

PROCEDUREPROFILE — Returns a row for every stored procedure that has been executed on the cluster, summarized across the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
PROCEDURE	STRING	The class name of the stored procedure.
WEIGHTED_PERC	BIGINT	A weighted average expressed as a percentage of the execution time for this stored procedure compared to all stored procedure invocations.
INVOCATIONS	BIGINT	The total number of invocations of this procedure.
AVG	BIGINT	The average length of time (in nanoseconds) it took to execute the stored procedure.

Name	Datatype	Description
MIN	BIGINT	The minimum length of time (in nanoseconds) it took to execute the stored procedure.
MAX	BIGINT	The maximum length of time (in nanoseconds) it took to execute the stored procedure.
ABORTS	BIGINT	The number of times the procedure was aborted.
FAILURES	BIGINT	The number of times the procedure failed unexpectedly. (As opposed to user aborts or expected errors, such as constraint violations.)

REBALANCE — Returns one row if the cluster is rebalancing. No data is returned if the cluster is not rebalancing.

Warning

The rebalance selector is still under development. The return values are likely to change in upcoming releases.

Name	Datatype	Description
TOTAL_RANGES	BIGINT	The total number of partition segments to be migrated.
PERCENTAGE_MOVED	FLOAT	The percentage of the total segments that have already been moved.
MOVED_ROWS	BIGINT	The number of rows of data that have been moved.
ROWS_PER_SECOND	FLOAT	The average number of rows moved per second.
ESTIMATED_REMAINING	BIGINT	The estimated time remaining until the rebalance is complete, in milliseconds.
MEGABYTES_PER_SECOND	FLOAT	The average volume of data moved per second, measured in megabytes.
CALLS_PER_SECOND	FLOAT	The average number of rebalance work units, or transactions, executed per second.
CALLS_LATENCY	FLOAT	The average execution time for rebalance transactions, in milliseconds.

SNAPSHOTSTATUS — Returns a row for every snapshot file in the recent snapshots performed on the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the snapshot was initiated (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
TABLE	STRING	The name of the database table whose data the file contains.
PATH	STRING	The directory path where the snapshot file resides.
FILENAME	STRING	The file name.
NONCE	STRING	The unique identifier for the snapshot.
TXNID	BIGINT	The transaction ID of the snapshot.
START_TIME	BIGINT	The timestamp when the snapshot began (in milliseconds).

Name	Datatype	Description
END_TIME	BIGINT	The timestamp when the snapshot was completed (in milliseconds).
SIZE	BIGINT	The total size, in bytes, of the file.
DURATION	BIGINT	The length of time (in milliseconds) it took to complete the snapshot.
THROUGHPUT	FLOAT	The average number of bytes per second written to the file during the snapshot process.
RESULT	STRING	String value indicating whether the writing of the snapshot file was successful ("SUCCESS") or not ("FAILURE").
TYPE	STRING	String value indicating how the snapshot was initiated. Possible values are: • AUTO — an automated snapshot as defined by the deployment file • COMMANDLOG — a command log snapshot • MANUAL — a manual snapshot initiated by a user

TABLE — Returns a row for every table, per partition. In other words, the number of tables, multiplied by the number of sites per host and the number of hosts.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	BIGINT	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	BIGINT	Numeric ID of the execution site on the host node.
PARTITION_ID	BIGINT	The numeric ID for the logical partition that this site represents. When using a K value greater than zero, there are multiple copies of each logical partition.
TABLE_NAME	STRING	The name of the database table.
TABLE_TYPE	STRING	The type of the table. Values returned include "PersistentTable" for normal data tables and views and "StreamedTable" for export-only tables.
TUPLE_COUNT	BIGINT	The number of rows currently stored for this table in the current partition. For export-only tables, the cumulative total number of rows inserted into the table.
TUPLE_ALLOCATED_MEMORY	BIGINT	The total size of memory, in kilobytes, allocated for storing inline data associated with this table in this partition. The allocated memory can exceed the currently used memory (TUPLE_DATA_MEMORY). For export-only tables, this field identifies the amount of memory currently in use to queue export data (both in memory and as export overflow) prior to its being passed to the export target.
TUPLE_DATA_MEMORY	BIGINT	The total memory, in kilobytes, used for storing inline data associated with this table in this partition. The total memory used for storing data for this table is the combination of memory used for inline (tuple) and non-inline (string) data.

Name	Datatype	Description
STRING_DATA _MEMORY	BIGINT	The total memory, in kilobytes, used for storing non-inline variable length data (VARCHAR, VARBINARY, and GEOGRAPHY) associated with this table in this partition. The total memory used for storing data for this table is the combination of memory used for inline (tuple) and non-inline (string) data.
TUPLE_LIMIT	INTEGER	The row limit for this table. Row limits are optional and are defined in the schema as a maximum number of rows that any partition can contain. If no row limit is set, this value is null.
PERCENT_FULL	INTEGER	The percentage of the row limit currently in use by table rows in this partition. If no row limit is set, this value is zero.

Examples

The following example uses @Statistics to gather information about the distribution of table rows within the cluster:

```
$ sqlcmd
1> exec @Statistics TABLE, 0;
```

The next program example shows a procedure that collects and displays the number of transactions (i.e. stored procedures) during a given interval, by setting the delta-flag to a non-zero value. By calling this procedure iteratively (for example, every five minutes), it is possible to identify fluctuations in the database workload over time (as measured by the number of transactions processed).

```
void measureWorkload() {
  VoltTable[] results = null;
  String procName;
  int procCount = 0;
  int sysprocCount = 0;

  try { results = client.callProcedure("@Statistics",
                  "INITIATOR",1).getResults(); }
  catch (Exception e) { e.printStackTrace(); }

  for (VoltTable t: results) {
     for (int r=0;r<t.getRowCount();r++) {
         VoltTableRow row = t.fetchRow(r);
         procName = row.getString("PROCEDURE_NAME");
                /* Count system procedures separately */
         if (procName.substring(0,1).compareTo("@") == 0)
           { sysprocCount += row.getLong("INVOCATIONS"); }
         else
           { procCount += row.getLong("INVOCATIONS"); }
     }
  }
  System.out.printf("System procedures: %d\n" +
                    "User-defined procedures: %d\n",+
                    sysprocCount,procCount);
}
```

@StopNode

@StopNode — Stops a VoltDB server process, removing the node from the cluster.

Syntax

```
@StopNode Integer host-ID
```

Description

The @StopNode system procedure lets you stop a specific server in a K-safe cluster. You specify which node to stop using the host ID, which is the unique identifier for the node assigned by VoltDB when the server joins the cluster.

Note that by calling the @StopNode procedure on a node other than the node being stopped, you will receive a return status indicating the success or failure of the call. If you call the procedure on the node that you are requesting to stop, the return status can only indicate that the call was interrupted (by the VoltDB process on the node stopping), not whether it was successfully completed or not.

If you call @StopNode on a node or cluster that is not K-safe — either because it was started with a K-safety value of zero or one or more nodes have failed so any further failure could crash the database — the @StopNode procedure will not be executed. You can only stop nodes on a cluster that will remain viable after the node stops. To stop the entire cluster, please use the @Shutdown system procedure.

Return Values

Returns one VoltTable with one row.

Name	Datatype	Description
STATUS	BIGINT	Always returns the value zero (0) indicating success.

Examples

The following program example uses **grep**, **sqlcmd**, and the @SystemInformation stored procedure to identify the host ID for a specific node (doodah) of the cluster. The example then uses that host ID (2) to call @StopNode and stop the desired node.

```
$ echo "exec @SystemInformation overview;" | sqlcmd | grep "doodah"
       2 HOSTNAME                doodah
$ sqlcmd
1> exec @StopNode 2;
```

The following Java code fragment performs the same function.

```
try {
     results = client.callProcedure("@SystemInformation",
                                    "overview").getResults();
}
catch (Exception e) { e.printStackTrace(); }

VoltTable table = results[0];
```

```
table.resetRowPosition();
int targetHostID = -1;

while (table.advanceRow() && targetHostId < 0) {
    if ( (table.getString("KEY") == "HOSTNAME") &&
            (table.getString("VALUE") == targetHostName) ) {
         targetHostId = (int) table.getLong("HOST_ID");
    }
}

try {
      client.callProcedure("@SStopNode",
                           targetHostId).getResults();
   }
  catch (Exception e) { e.printStackTrace(); }
```

@SystemCatalog

@SystemCatalog — Returns metadata about the database schema.

Syntax

```
@SystemCatalog String component
```

Description

The @SystemCatalog system procedure returns information about the schema of the VoltDB database, depending upon the component keyword you specify. The following are the allowable values of *component*:

"TABLES"	Returns information about the tables in the database.
"COLUMNS"	Returns a list of columns for all of the tables in the database.
"INDEXINFO"	Returns information about the indexes in the database schema. Note that the procedure returns information for each column in the index. In other words, if an index is composed of three columns, the result set will include three separate entries for the index, one for each column.
"PRIMARYKEYS"	Returns information about the primary keys in the database schema. Note that the procedure returns information for each column in the primary key. If an primary key is composed of three columns, the result set will include three separate entries.
"PROCEDURES"	Returns information about the stored procedures defined in the database, including system procedures.
"PROCEDURECOLUMNS"	Returns information about the arguments to the stored procedures.

Return Values

Returns a different VoltTable for each component. The layout of the VoltTables is designed to match the corresponding JDBC data structures. Columns are provided for all JDBC properties, but where VoltDB has no corresponding element the column is unused and a null value is returned.

For the TABLES component, the VoltTable has the following columns:

Name	Datatype	Description
TABLE_CAT	STRING	Unused.
TABLE_SCHEM	STRING	Unused.
TABLE_NAME	STRING	The name of the database table.
TABLE_TYPE	STRING	Specifies whether the table is a data table ("TABLE"), a materialized view ("VIEW"), or an export-only table ('EXPORT").
REMARKS	STRING	Unused.
TYPE_CAT	STRING	Unused.

Name	Datatype	Description
TYPE_SCHEM	STRING	Unused.
TYPE_NAME	STRING	Unused.
SELF_REFERENCING _COL_NAME	STRING	Unused.
REF_GENERATION	STRING	Unused.

For the COLUMNS component, the VoltTable has the following columns:

Name	Datatype	Description
TABLE_CAT	STRING	Unused.
TABLE_SCHEM	STRING	Unused.
TABLE_NAME	STRING	The name of the database table the column belongs to.
COLUMN_NAME	STRING	The name of the column.
DATA_TYPE	INTEGER	An enumerated value specifying the corresponding Java SQL datatype of the column.
TYPE_NAME	STRING	A string value specifying the datatype of the column.
COLUMN_SIZE	INTEGER	The length of the column in bits, characters, or digits, depending on the datatype.
BUFFER_LENGTH	INTEGER	Unused.
DECIMAL_DIGITS	INTEGER	The number of fractional digits in a DECIMAL datatype column. (Null for all other datatypes.)
NUM_PREC_RADIX	INTEGER	Specifies the radix, or numeric base, for calculating the column size. A radix of 2 indicates the column size is measured in bits while a radix of 10 indicates a measurement in bytes or digits.
NULLABLE	INTEGER	Indicates whether the column value can be null (1) or not (0).
REMARKS	STRING	Contains the string "PARTITION_COLUMN" if the column is the partitioning key for a partitioned table. Otherwise null.
COLUMN_DEF	STRING	The default value for the column.
SQL_DATA_TYPE	INTEGER	Unused.
SQL_DATETIME_SUB	INTEGER	Unused.
CHAR_OCTET_LENGTH	INTEGER	For variable length columns (VARCHAR and VARBINARY), the maximum length of the column. Null for all other datatypes.
ORDINAL_POSITION	INTEGER	An index specifying the position of the column in the list of columns for the table, starting at 1.
IS_NULLABLE	STRING	Specifies whether the column can contain a null value ("YES") or not ("NO").
SCOPE_CATALOG	STRING	Unused.
SCOPE_SCHEMA	STRING	Unused.
SCOPE_TABLE	STRING	Unused.

Name	Datatype	Description
SOURCE_DATE_TYPE	SMALLINT	Unused.
IS_AUTOINCREMENT	STRING	Specifies whether the column is auto-incrementing or not. (Always returns "NO").

For the INDEXINFO component, the VoltTable has the following columns:

Name	Datatype	Description
TABLE_CAT	STRING	Unused.
TABLE_SCHEM	STRING	Unused.
TABLE_NAME	STRING	The name of the database table the index applies to.
NON_UNIQUE	TINYINT	Value specifying whether the index is unique (0) or not (1).
INDEX_QUALIFIER	STRING	Unused.
INDEX_NAME	STRING	The name of the index that includes the current column.
TYPE	SMALLINT	An enumerated value indicating the type of index as either a hash (2) or other type (3) of index.
ORDINAL_POSITION	SMALLINT	An index specifying the position of the column in the index, starting at 1.
COLUMN_NAME	STRING	The name of the column.
ASC_OR_DESC	STRING	A string value specifying the sort order of the index. Possible values are "A" for ascending or null for unsorted indexes.
CARDINALITY	INTEGER	Unused.
PAGES	INTEGER	Unused.
FILTER_CONDITION	STRING	Unused.

For the PRIMARYKEYS component, the VoltTable has the following columns:

Name	Datatype	Description
TABLE_CAT	STRING	Unused.
TABLE_SCHEM	STRING	Unused.
TABLE_NAME	STRING	The name of the database table.
COLUMN_NAME	STRING	The name of the column in the primary key.
KEY_SEQ	SMALLINT	An index specifying the position of the column in the primary key, starting at 1.
PK_NAME	STRING	The name of the primary key.

For the PROCEDURES component, the VoltTable has the following columns:

Name	Datatype	Description
PROCEDURE_CAT	STRING	Unused.
PROCEDURE_SCHEM	STRING	Unused.
PROCEDURE_NAME	STRING	The name of the stored procedure.
RESERVED1	STRING	Unused.

Name	Datatype	Description
RESERVED2	STRING	Unused.
RESERVED3	STRING	Unused.
REMARKS	STRING	Unused.
PROCEDURE_TYPE	SMALLINT	An enumerated value that specifies the type of procedure. Always returns zero (0), indicating "unknown".
SPECIFIC_NAME	STRING	Same as PROCEDURE_NAME.

For the PROCEDURECOLUMNS component, the VoltTable has the following columns:

Name	Datatype	Description
PROCEDURE_CAT	STRING	Unused.
PROCEDURE_SCHEM	STRING	Unused.
PROCEDURE_NAME	STRING	The name of the stored procedure.
COLUMN_NAME	STRING	The name of the procedure parameter.
COLUMN_TYPE	SMALLINT	An enumerated value specifying the parameter type. Always returns 1, corresponding to procedureColumnIn.
DATA_TYPE	INTEGER	An enumerated value specifying the corresponding Java SQL datatype of the column.
TYPE_NAME	STRING	A string value specifying the datatype of the parameter.
PRECISION	INTEGER	The length of the parameter in bits, characters, or digits, depending on the datatype.
LENGTH	INTEGER	The length of the parameter in bytes. For variable length datatypes (VARCHAR and VARBINARY), this value specifies the maximum possible length.
SCALE	SMALLINT	The number of fractional digits in a DECIMAL datatype parameter. (Null for all other datatypes.)
RADIX	SMALLINT	Specifies the radix, or numeric base, for calculating the precision. A radix of 2 indicates the precision is measured in bits while a radix of 10 indicates a measurement in bytes or digits.
NULLABLE	SMALLINT	Unused.
REMARKS	STRING	If this column contains the string "PARTITION_PARAMETER", the parameter is the partitioning key for a single-partitioned procedure. If the column contains the string "ARRAY_PARAMETER" the parameter is a native Java array. Otherwise this column is null.
COLUMN_DEF	STRING	Unused.
SQL_DATA_TYPE	INTEGER	Unused.
SQL_DATETIME_SUB	INTEGER	Unused.
CHAR_OCTET_LENGTH	INTEGER	For variable length columns (VARCHAR and VARBINARY), the maximum length of the column. Null for all other datatypes.
ORDINAL_POSITION	INTEGER	An index specifying the position in the parameter list for the procedure, starting at 1.

Name	Datatype	Description
IS_NULLABLE	STRING	Unused.
SPECIFIC_NAME	STRING	Same as COLUMN_NAME

Examples

The following example calls @SystemCatalog to list the stored procedures in the active database schema:

```
$ sqlcmd
1> exec @SystemCatalog procedures;
```

The next program example uses @SystemCatalog to display information about the tables in the database schema.

```
VoltTable[] results = null;
try {
     results = client.callProcedure("@SystemCatalog",
               "TABLES").getResults();
     System.out.println("Information about the database schema:");
     for (VoltTable node : results) System.out.println(node.toString());
}
catch (Exception e) {
     e.printStackTrace();
}
```

@SystemInformation

@SystemInformation — Returns configuration information about VoltDB and the individual nodes of the database cluster.

Syntax

```
@SystemInformation

@SystemInformation String component
```

Description

The @SystemInformation system procedure returns information about the configuration of the VoltDB database or the individual nodes of the database cluster, depending upon the component keyword you specify. The following are the allowable values of *component*:

"DEPLOYMENT" Returns information about the configuration of the database. In particular, this keyword returns information about the various features and settings enabled through the deployment file, such as export, snapshots, K-safety, and so on. These properties are returned in a single VoltTable of name/value pairs.

"OVERVIEW" Returns information about the individual servers in the database cluster, including the host name, the IP address, the version of VoltDB running on the server, as well as the path to the deployment file in use. The overview also includes entries for the start time of the server and length of time the server has been running.

If you do not specify a component, @SystemInformation returns the results of the OVERVIEW component (to provide compatibility with previous versions of the procedure).

Return Values

Returns one of two VoltTables depending upon which component is requested.

For the DEPLOYMENT component, the VoltTable has the columns specified in the following table.

Name	Datatype	Description
PROPERTY	STRING	The name of the deployment property being reported.
VALUE	STRING	The corresponding value of that property in the deployment file (either explicitly or by default).

For the OVERVIEW component, information is reported for each server in the cluster, so an additional column is provided identifying the host node.

Name	Datatype	Description
HOST_ID	INTEGER	A numeric identifier for the host node.
KEY	STRING	The name of the system attribute being reported.
VALUE	STRING	The corresponding value of that attribute for the specified host.

Examples

The first example displays information about the individual servers in the database cluster:

```
$ sqlcmd
1> exec @SystemInformation overview;
```

The following program example uses @SystemInformation to display information about the nodes in the cluster and then about the database itself.

```
VoltTable[] results = null;
try {
     results = client.callProcedure("@SystemInformation",
               "OVERVIEW").getResults();
     System.out.println("Information about the database cluster:");
     for (VoltTable node : results) System.out.println(node.toString());

     results = client.callProcedure("@SystemInformation",
               "DEPLOYMENT").getResults();
     System.out.println("Information about the database deployment:");
     for (VoltTable node : results) System.out.println(node.toString());

}
catch (Exception e) {
     e.printStackTrace();
}
```

@UpdateApplicationCatalog

@UpdateApplicationCatalog — Reconfigures the database by replacing the application catalog and/or deployment configuration.

Syntax

```
@UpdateApplicationCatalog byte[] catalog, String deployment
```

Description

The @UpdateApplicationCatalog system procedure lets you modify the configuration of a running database without having to shutdown and restart.

Note

The @UpdateApplicationCatalog system procedure is primarily for updating the deployment configuration. Updating an application catalog is only supported for databases that were started from a catalog and with the deployment setting `schema="catalog"`. In general, updating the database schema interactively is recommended and use of application catalogs is being phased out.

@UpdateApplicationCatalog supports the following changes to the database:

- Add, remove, or modify stored procedures
- Add, remove, or modify database tables and columns
- Add, remove, or modify indexes (except where new constraints are introduced)
- Add or remove views and export-only tables
- Modify the security permissions in the database schema

@UpdateApplicationCatalog supports the following changes to the deployment file:

- Modify the security settings in the database configuration
- Modify the settings for automated snapshots (whether they are enabled or not, their frequency, location, prefix, and number retained)
- Modify the export settings

In general, you can make any changes to the database schema as long as there is no data in the tables. However, if there is data in a table, the following changes are *not* allowed:

- Changing the partitioning of the table
- Changing columns to NOT NULL
- Reducing the datatype size of a column (for example, from INTEGER to SMALLINT) or changing to an incompatible datatype (for example, from VARCHAR to INTEGER)
- Adding or broadening constraints, such as indexes and primary keys, that could conflict with existing data in the table

The arguments to the system procedure are a byte array containing the contents of the new catalog jar and a string containing the contents of the deployment file. That is, you pass the actual contents of the catalog and deployment files, using a byte array for the binary catalog and a string for the text deployment file. You can use null for either argument to change just the catalog or the deployment.

The new catalog and the deployment file must not contain any changes other than the allowed modifications listed above. Currently, if there are any other changes from the original catalog and deployment file (such as changes to the export configuration or to the configuration of the cluster), the procedure returns an error indicating that an incompatible change has been found.

If you call @UpdateApplicationCatalog on a master database while database replication (DR) is active, the DR process automatically communicates any changes to the application catalog to the replica database to keep the two databases in sync. However, any changes to the deployment file apply to the master database only. To change the deployment settings on a replica database, you must stop and restart the replica (and database replication) using an updated deployment file.

To simplify the process of encoding the catalog contents, the Java client interface includes two helper methods (one synchronous and one asynchronous) to encode the files and issue the stored procedure request:

ClientResponse client.updateApplicationCatalog(File *catalog-file*, File *deployment-file*)

ClientResponse client.updateApplicationCatalog(clientCallback *callback*, File *catalog-file*, File *deployment-file*)

Similarly, the sqlcmd utility interprets both arguments as filenames.

Examples

The following example uses **sqlcmd** to update the application catalog using the files `mycatalog.jar` and `mydeploy.xml`:

```
$ sqlcmd
1> exec @UpdateApplicationCatalog mycatalog.jar, mydeploy.xml;
```

An alternative is to use the **voltadmin update** command. In which case, the following command performs the same function as the preceding **sqlcmd** example:

```
$ voltadmin update mycatalog.jar mydeploy.xml
```

The following program example uses the @UpdateApplicationCatalog procedure to update the current database catalog, using the catalog at `project/newcatalog.jar` and configuration file at `project/production.xml`.

```
String newcat = "project/newcatalog.jar";
String newdeploy = "project/production.xml";

try {
   File file = new File(newcat);
   FileInputStream fin = new FileInputStream(file);
   byte[] catalog = new byte[(int)file.length()];
   fin.read(catalog);
   fin.close();
   file  = new File(newdeploy);
   fin = new FileInputStream(file);
```

```
   byte[] deploybytes = new byte[(int)file.length()];
   fin.read(deploybytes);
   fin.close();
   String deployment = new String(deploybytes, "UTF-8");
   client.callProcedure("@UpdateApplicationCatalog",catalog, deployment);
}
catch (Exception e) { e.printStackTrace(); }
```

The following example uses the synchronous helper method to perform the same operation.

```
String newcat = "project/newcatalog.jar";
String newdeploy = "project/production.xml";
try {
   client.updateApplicationCatalog(new File(newcat), new File(newdeploy));
}
catch (Exception e) { e.printStackTrace(); }
```

@UpdateClasses

@UpdateClasses — Adds and removes Java classes from the database.

Syntax

```
@UpdateClasses byte[] JAR-file, String class-selector
```

Description

The @UpdateClasses system procedure performs two functions:

- Loads into the database any Java classes in the JAR file passed as the first parameter
- Removes any classes matching the class selector string passed as the second parameter

You need to compile and pack your stored procedure classes into a JAR file and load them into the database using @UpdateClasses before entering the CREATE PROCEDURE and PARTITION PROCEDURE statements that define those classes as VoltDB stored procedures. Note that, for interactive use, the sqlcmd utility has two directives, **load classes** and **remove classes**, that perform these actions in separate steps.

To remove classes, you specify the class names in the second parameter, the class selector. You can include multiple class selectors using a comma-separated list. You can also use Ant-style wildcards in the class specification to identify multiple classes. For example, the following command deletes all classes that are children of org.mycompany.utils as well as *.DebugHandler:

```
sqlcmd
1> exec @UpdateClasses NULL "org.mycompany.utils.*,*.DebugHandler";
```

You can also use the @UpdateClasses system procedure to include reusable code that is accessed by multiple stored procedures. Any classes and methods called by stored procedures must follow the same rules for deterministic behavior that stored procedures follow, as described in Section 5.1.2, “VoltDB Stored Procedures are Deterministic”.

However, use of @UpdateClasses is not recommended for large, established libraries of classes used by stored procedures. For larger, static libraries that do not need to be modified on the fly, the preferred approach is to include the code by placing JAR files in the /lib directory where VoltDB is installed on the database servers.

Examples

The following example compiles and packs Java stored procedures into the file myapp.jar. The example then uses @UpdateCLasses to load the classes from the JAR file, then defines and partitions a stored procedure based on the uploaded classes.

```
$ javac -cp "/opt/voltdb/voltdb/*" -d obj src/myapp/*.java
$ jar cvf myapp.jar -C obj .
$ sqlcmd
1> exec @UpdateClasses myapp.jar "";
2> CREATE PROCEDURE FROM CLASS myapp.procedures.AddCustomer;
3> PARTITION PROCEDURE AddCustomer ON TABLE Customer COLUMN CustomerID;
```

The second example removes the class added and defined in the preceding example. Note that you must drop the procedure definition first; you cannot delete classes that are referenced by defined stored procedures.

```
$ sqlcmd
1> DROP PROCEDURE AddCustomer;
2> exec @UpdateClasses NULL "myapp.procedures.AddCustomer";
```

As an alternative, the loading and removing of classes can be performed using native sqlcmd directives **load classes** and **remove classes**. So the previous tasks can be performed using the following commands:

```
$ sqlcmd
1> load classes myapp.jar "";
2> CREATE PROCEDURE FROM CLASS myapp.procedures.AddCustomer;
3> PARTITION PROCEDURE AddCustomer ON TABLE Customer COLUMN CustomerID;
1> DROP PROCEDURE AddCustomer;
2> remove classes "myapp.procedures.AddCustomer";
```

@UpdateLogging

@UpdateLogging — Changes the logging configuration for a running database.

Syntax

```
@UpdateLogging CString configuration
```

Description

The @UpdateLogging system procedure lets you change the logging configuration for VoltDB. The second argument, *configuration*, is a text string containing the Log4J XML configuration definition.

Return Values

Returns one VoltTable with one row.

Name	Datatype	Description
STATUS	BIGINT	Always returns the value zero (0) indicating success.

Examples

It is possible to use **sqlcmd** to update the logging configuration. However, the argument is interpreted as raw XML content rather than as a file specification. Consequently, it can be difficult to use interactively. But you can write the file contents to an input file and then pipe that to **sqlcmd**, like so:

```
$ echo "exec @UpdateLogging '" > sqlcmd.input
$ cat mylog4j.xml >> sqlcmd.input
$ echo "';" >> sqlcmd.input
$ cat sqlcmd.input | sqlcmd
```

The following program example demonstrates another way to update the logging, using the contents of an XML file (identified by the string *xmlfilename*).

```
try {
    Scanner scan = new Scanner(new File(xmlfilename));
    scan.useDelimiter("\\Z");
    String content = scan.next();
    client.callProcedure("@UpdateLogging",content);
}
catch (Exception e) {
    e.printStackTrace();
}
```