Using VoltDB

Abstract

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

V4.9

Using VoltDB

V4.9

This document was generated on November 24, 2014.

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

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 four sections:

Section 1: Introduction	Explains what VoltDB is, how it works, what problems it solves, and who should use it. The chapters in this section are: • Chapter 1, *Overview* • Chapter 2, *Installing VoltDB*
Section 2: Using VoltDB	Explains how to design and develop applications using VoltDB. The chapters in this section are: • Chapter 3, *Designing Your VoltDB Application* • Chapter 4, *Simplifying Application Development* • Chapter 5, *Building Your VoltDB Application* • Chapter 6, *Running Your VoltDB Application* • Chapter 7, *Updating Your VoltDB Database*
Section 3: Advanced Topics	Provides detailed information about advanced features of VoltDB. Topics covered in this section are: • Chapter 8, *Security* • Chapter 9, *Saving & Restoring a VoltDB Database* • Chapter 10, *Command Logging and Recovery* • Chapter 11, *Availability* • Chapter 12, *Database Replication* • Chapter 13, *Exporting Live Data* • Chapter 14, *Logging and Analyzing Activity in a VoltDB Database* • Chapter 15, *Using VoltDB with Other Programming Languages*
Section 4: 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 Section 2 — chapters 2 through 6 — are available in both versions of the product. Several features in Section 3, advanced topics — such as snapshots, command logging, database replication, and export — 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://www.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 a subset of 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 applications that require scalability, reliability, high availability, and outstanding throughput. In other words, VoltDB's target audience is what have traditionally been known as Online Transaction Processing (OLTP) applications. These applications have strict requirements for throughput to avoid bottlenecks. They also have a clearly architected workflow that predefines the allowed data access paths and critical interactions.

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.

VoltDB is *not* optimized for all types of queries, such as fetching and collating large data sets 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

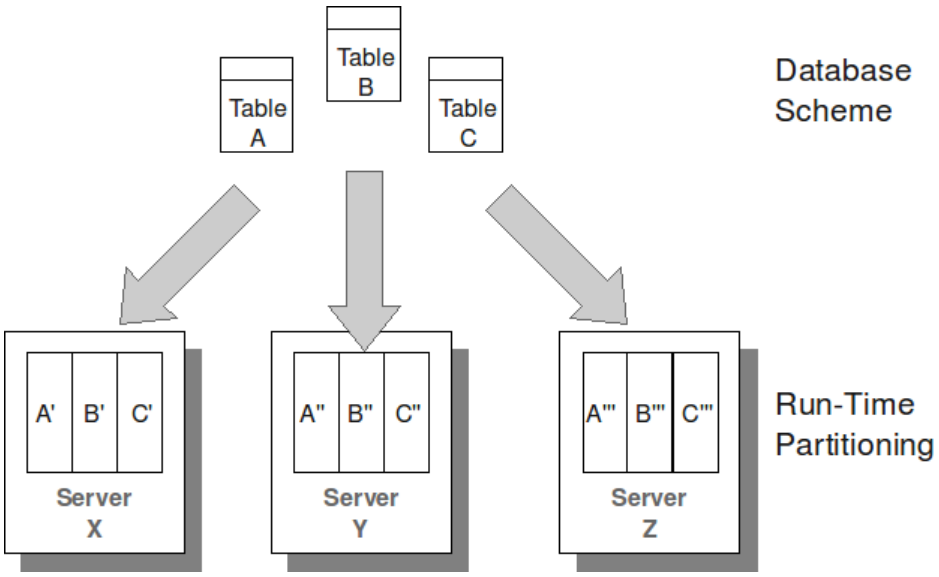
VoltDB is not like traditional database products. There is no such thing as a generic VoltDB "database". Each database is optimized for a specific application by compiling the schema, stored procedures, and partitioning information in to what is known as the VoltDB *application catalog*. The catalog is then loaded on one or more host machines to create the distributed database.

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 nodes on the cluster. In this way, each node of the cluster contains a unique "slice" of the data and the data processing.

Figure 1.1. Partitioning Tables



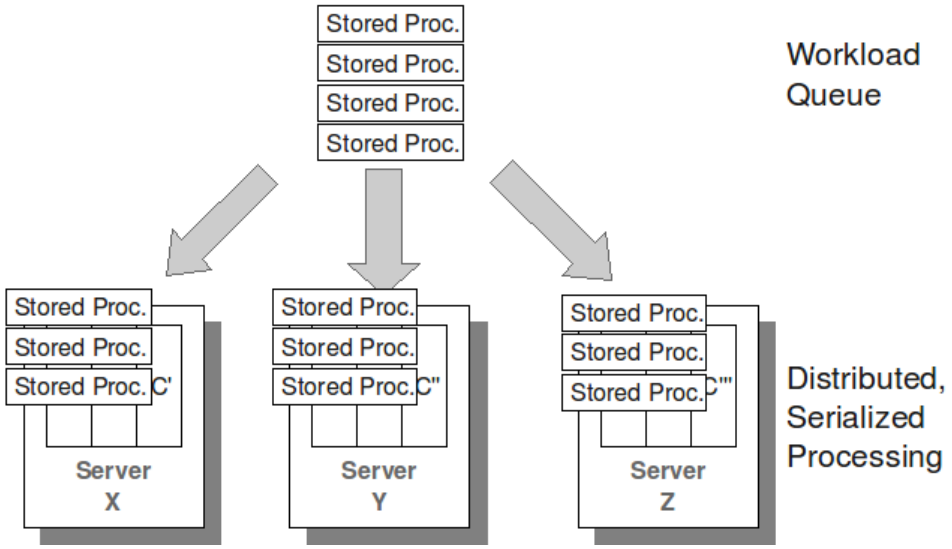
1.3.2. Serialized (Single-Threaded) Processing

At run-time, calls to the stored procedures are passed to the appropriate node of the cluster. When procedures are "single-partitioned" (meaning they operate on data within a single partition) the individual node 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 generally 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



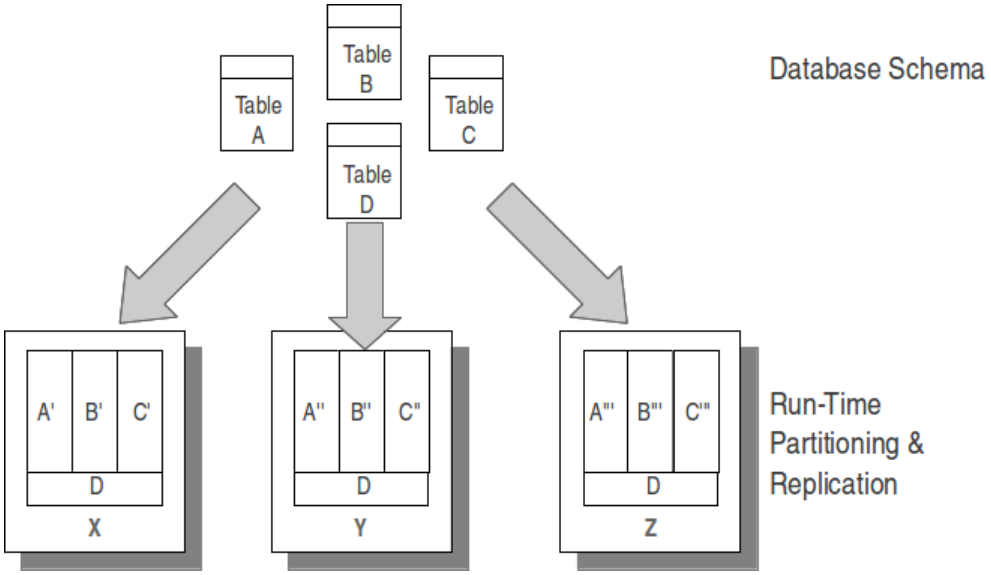
It is important to note that the VoltDB architecture is optimized for throughput over latency. The latency of any one transaction (the time from when the transaction begins until processing ends) is similar in VoltDB to other databases. However, the number of transactions that can be completed in a second (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 primary key that you, the developer or designer, specify. When you choose partitioning keys 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.

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 and schema management.

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 6.3 or later, including 7.0 • Red Hat (RHEL) version 6.3 or later, including 7.0 • Ubuntu versions 10.04[2], 12.04, and 14.04 Development builds are also available for Macintosh OS X 10.7 and later[1].
CPU	• Dual core[3] x86_64 processor • 64 bit • 1.6 GHz
Memory	4 Gbytes[4]
Java	Java 7 or 8 — VoltDB supports JDKs from OpenJDK or Oracle/Sun
Required Software	NTP[5] Python 2.5 or later release of 2.x
Recommended Software	Eclipse 3.x (or other Java IDE)

Footnotes:

1. CentOS 6.3, CentOS 7.0, RHEL 6.3, RHEL 7.0, and Ubuntu 10.04, 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 10.7.

2. Support for Ubuntu 10.04 is deprecated and will be removed in an upcoming release.

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

4. 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.

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 LINUX-voltdb-ent-4.9.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 LINUX-voltdb-ent-4.9.tar.gz -C /opt
$ cd /opt
$ sudo mv voltdb-ent-4.9 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. Alternately, you can use standard installation packages for Linux systems, as described in Section 2.2.2, “Installing Standard System Packages”.

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**).
3. Shutdown the database (**voltadmin shutdown**).
4. Upgrade VoltDB.
5. Start a new database using the **voltdb create** option, your existing application catalog, 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**).

Note that the **voltdb create** command automatically recompiles your catalog if the catalog was created by an older version. When using the Enterprise Manager, it is also recommended that you delete the Enterprise Manager configuration files (stored by default in the `.voltdb` subfolder in the home directory of the current account) when performing an upgrade.

2.2.2. Installing Standard System Packages

If you plan on making VoltDB available to all users of the system, you can use a common system package to install the VoltDB files in standard locations. Installation packages are available for both Debian-based

(deb) and Red Hat-based (rpm) systems. These packages simplify the installation process by placing the VoltDB files in standard system directories, making VoltDB available to all users of the system without their having to individually configure their PATH variable.

The advantages of using an install package are:

- The installation is completed in a single command. No additional set up is required.
- VoltDB becomes available to all system users.
- Upgrades are written to the same location. You do not need to modify your application scripts or move files after each upgrade.

However, there are a few changes to behavior that you should be aware of if you install VoltDB using a system package manager:

- The VoltDB libraries are installed in /usr/lib/voltdb. When compiling stored procedures, you must include this location in your Java classpath.
- The sample applications are installed into the directory `/usr/share/voltdb/examples/`. Because this is a system directory, users cannot run the samples directly in that location. Instead, first copy the folder containing the sample application you want to run and paste a copy into your home directory structure. Then run the sample from your copy. For example:

```
$ cp -r /usr/share/voltdb/examples/voter ~/
$ cd ~/voter
$ ./run.sh
```

2.2.2.1. Installing the Debian Package

To install the Debian package on Ubuntu or other Debian-based systems, download the package from the VoltDB web site. Then, from an account with root access issue the following commands to install Open JDK 7 and VoltDB:

```
$ sudo apt-get install openjdk-7-jdk
$ sudo dpkg -i voltdb_4.9-1_amd64.deb
```

2.2.2.2. Installing the RPM Package

To install the rpm package on compatible systems such as Red Hat or CentOS, download the package from the VoltDB web site. Then, from an account with root access issue the following command:

```
$ sudo yum localinstall voltdb-4.9-1.x86_64.rpm
```

2.2.3. 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 VoltDB as `/voltdb-n.n` in your $HOME directory:

```
$ export PATH="$PATH:$HOME/voltdb-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 Console	VoltDb Management Console is a browser-based management tool for monitoring, examining, and querying a running VoltDB database. The management Console is bundled with the VoltDB server software. You can start the Management Console by connecting to the HTTP port of a running VoltDB database server. For example, `http://voltsvr:8080/`. Note that the httpd server and JSON interface must be enabled on the server to be able to access the Management Console.
Shell Commands	The `/bin` subfolder contains executable scripts to perform common VoltDB tasks, such as compiling application catalogs and starting the VoltDB serv-

Component	Description
	er. 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 default to the `/examples` directory 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. Designing Your VoltDB Application

VoltDB produces ACID-compliant, relational databases using a subset of ANSI-standard SQL for defining the schema and accessing the data. So designing a VoltDB application is very much like designing any other database application.

The difference is that VoltDB requires you to be more organized and planful in your design:

- All data access should be done through stored procedures. Although ad hoc queries are possible, they do not take advantage of the optimizations that make VoltDB's exceptional performance possible.
- The schema and workflow should be designed to promote single-partitioned procedures wherever possible.

These are not unreasonable requirements for high-performance applications. In fact, for 20 years or more OLTP application designers have used these design principles to get the most out of commercial database products. The difference is that VoltDB actually takes advantage of these principles to provide exponentially better throughput without sacrificing any of the value of a fully-transactional database.

The following sections provide guidelines for designing VoltDB applications.

3.1. Designing the Database

VoltDB is a relational database product. Relational databases consist of tables and columns, with constraints, index keys, and aggregated 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.

For example, 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. Your database schema might look like the following:

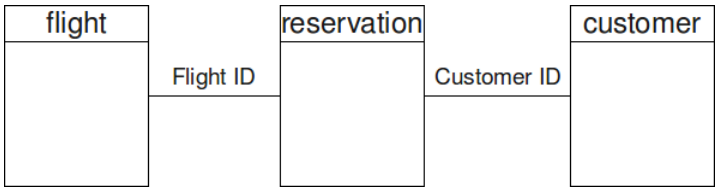


Figure 3.1 shows how the schema looks as defined in standard SQL DDL.

Figure 3.1. Example 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 UNIQUE NOT NULL,
   FlightID INTEGER NOT NULL,
   CustomerID INTEGER NOT NULL,
   Seat VARCHAR(5) DEFAULT NULL,
   Confirmed TINYINT DEFAULT '0',
   PRIMARY KEY(ReserveID)
);

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

But a schema is not all you need to define the database (or the application) effectively. You also need to know the expected volume and workload. 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

We can also define a set of functions the application must perform and the expected frequency. Again, for the sake of our example, let's assume the following is the estimated workload.

Table 3.1. 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

This additional information about the volume and workload affects the design of both the database and the application, because it impacts what SQL queries need to be written and what keys to use for accessing the data.

In the case of VoltDB, you use this additional information to configure the database and optimize performance. Specifically, you want to partition the individual tables to ensure that the most frequent transactions are single-partitioned.

The following sections discuss how to partition a database to maximize throughput, using the flight reservation case study as an example.

3.1.1. Partitioning Database Tables

The goal of partitioning the database tables is to ensure that the most frequent transactions are single-partitioned. 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, the key transactions to focus on are looking up a flight, seeing if a flight is available (in other words, has sufficient space), looking up a reservation, and making a reservation. Of these transactions, only the last modifies the database.

3.1.1.1. Choosing a Partition Column

We will discuss the Flight table later. But first let's look at the Reservation table. Reservation has a primary key, ReserveID, which is a unique identifier for the reservation. Looking at the schema alone, ReserveID might look like a good column to use to partition the table.

However, looking at the workload, there are only two transactions that are keyed to the reservation ID (looking up a reservation by ID and canceling a reservation), which occur only 200 times a second. Whereas, seeing if a flight has available seats, which requires looking up reservations by the Flight ID, occurs 5,000 times a second, or 25 times as frequently. Therefore, the Reservation table needs to be partitioned on the FlightID column.

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

Once you choose the columns to use for partitioning your database tables, you can 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 the `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:

```
PARTITION TABLE Reservation ON COLUMN FlightID;
PARTITION TABLE Customer ON COLUMN CustomerID;
```

3.1.1.2. Rules for Partitioning Tables

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

- **Any integer or string column can be a partition column.** VoltDB can partition on any column that is an integer (TINYINT, SMALLINT, INTEGER, or BIGINT) or string (VARCHAR) datatype.

- **There is 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.

- **Partition columns do not need to have unique values, but they cannot be null.** Numeric fields can be zero and string or character fields can be empty, but the column cannot contain a null value. You must specify NOT NULL in the schema, or VoltDB will report it as an error when you compile the schema.

3.1.2. Replicating Tables

The previous section describes how to choose a partitioning column for database tables, using 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.

Small, mostly read-only tables can be replicated across all of the partitions of a VoltDB database. This is particularly useful when a table is not accessed by a single column primarily.

3.1.2.1. 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 point of origin, the destination, and the departure time. Because of the nature of this transaction, it makes it hard to partition the table in a way that would make it single-partitioned.

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, all 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.

3.1.2.2. 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.

3.2. Designing the Data Access (Stored Procedures)

As you can see from the previous discussion of designing the database, defining the database schema — and particularly the partitioning plan — goes hand in hand with understanding how the data is accessed. The two must be coordinated to ensure optimum performance.

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 the preceding sections when discussing how to design the data access.

3.2.1. Writing VoltDB Stored Procedures

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.

In VoltDB, a stored procedure and a transaction are one and the same. The stored procedure succeeds or rolls back as a whole. Also, because the transaction is defined in advance as a stored procedure, there is no need for specific BEGIN TRANSACTION or END TRANSACTION commands.[1]

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 any other functions your applications need.

3.2.2. VoltDB Stored Procedures and Determinism

To ensure data consistency and durability, VoltDB procedures must be *deterministic*. That is, given specific input values, the outcome of the procedure is 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 11, *Availability* for more information on redundancy and availability.)

One key to deterministic behavior is avoiding ambiguous SQL queries. 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.

Another key to deterministic behavior is avoiding 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 outside the stored procedure and pass it in as input parameters or generate it in a deterministic way.

[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.

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 a method for doing 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.

Finally, even seemingly harmless programming techniques, such as static variables can introduce unpredictable 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 procedure parameter.

3.2.3. The Anatomy of a VoltDB Stored Procedure

The stored procedures themselves are written as Java classes, each procedure being a separate class. Example 3.1, “Components of a VoltDB 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 3.1. Components of a VoltDB 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)
        throws VoltAbortException {                               ❹

        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);
        seatsinuse = result.fetchRow(0).getLong(1);

        numofseats = numofseats - seatsinuse;                     ❽
        return numofseats; // Return available seats
    }
}
```

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

❷ Each stored procedure extends the generic class VoltProcedure.

❸ Within the stored procedure you access the database using a subset of 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. (See Appendix B, *Supported SQL Statements* for details on the supported SQL statements.)

❹ The bulk of the stored procedure is the run method. Note that the run method throws the exception VoltAbortException if any exceptions are not caught. VoltAbortException causes the stored procedure to rollback. (See Section 3.2.3.6, “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 to use) using the voltQueueSQL method.

❻ Once 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.

❽ In addition to queueing and executing queries, stored procedures can contain custom code. Note, 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 the following transactions in the queue.

The following sections describe these components in more detail.

3.2.3.1. The Structure of the Stored Procedure

VoltDB stored procedures are Java classes. The key points to remember are to:

- Import the VoltDB classes in org.voltdb.*
- Include the class definition, which extends the abstract class VoltProcedure
- Define the method *run*, that performs the SQL queries and processing that make up the transaction

The following diagram illustrates the basic structure if a VoltDB 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 ...


    }
}
```

3.2.3.2. 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 the *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, and Double
- Fixed decimal point: BigDecimal
- Timestamp types: VoltDB timestamp (org.voltdb.types.TimestampType), java.util.Date, java.sql.Date, and java.sql.Timestamp
- String and binary types: String and byte[]
- VoltDB types: 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 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 3.3.2, “Invoking Stored Procedures” for information on the callProcedure() method.)

3.2.3.3. 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 voltExecuteSQL

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);
```

Once you have queued all of the SQL statements you want to execute together, you can then process the queue using the voltExecuteSQL function:

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

Note that you can queue multiple SQL statements before calling voltExecuteSQL. This improves performance when executing multiple SQL queries because it minimizes the amount of network traffic within the cluster.

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

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

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

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

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

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

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

3.2.3.4. Interpreting the Results of SQL Queries

When you call voltExecuteSQL, 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 VoltTables are returned in the same order as the respective SQL statements in the queue.

The VoltTable itself consists of rows. Each row contains columns. Each column has a label and a value of a fixed datatype. The number of rows and columns per row depends on the specific query.

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) with 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.

VoltDB provides a set of convenience routines for accessing the contents of the VoltTable array. Table 3.2, "Methods of the VoltTable Classes" lists some of the most common methods.

Table 3.2. 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 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 the `get`*`Datatype`* methods passing 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 3.2, "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 3.2. 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.

3.2.3.5. Returning Results from a Stored Procedure

Stored procedures can return a single VoltTable, an array of VoltTables, or a long integer. 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 3.1, “Components of a VoltDB 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)
```

It is important to 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.

3.2.3.6. 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 (i.e. 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.

There may also be situations where a logical exception occurs. In other words, there is no programmatic issue that might be caught by Java or VoltDB, but a situation occurs where there is no practical way for the transaction to complete. In these conditions, the stored procedure 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 4.4, “Verifying Expected Query Results” for another way to roll back procedures when queries do not meet necessary conditions.

3.2.4. Partitioning Stored Procedures

To make your stored procedures accessible in the database, you must declare them in the DDL schema using the CREATE PROCEDURE statement. For example, the following statements declare five stored procedures, identifying them by their class name:

```
CREATE PROCEDURE FROM CLASS procedures.LookupFlight;
CREATE PROCEDURE FROM CLASS procedures.HowManySeats;
CREATE PROCEDURE FROM CLASS procedures.MakeReservation;
CREATE PROCEDURE FROM CLASS procedures.CancelReservation;
CREATE PROCEDURE FROM CLASS procedures.RemoveFlight;
```

You can also declare your stored procedures as single-partitioned or not. 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. However, the real focus of VoltDB, and the way to achieve maximum throughput for your OLTP application, is through the use of 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 (i.e. only the data within the specified partition). Most important of all *it is the responsibility of the application developer to ensure that the SQL queries within the stored procedure are actually single-partitioned*.

When you declare a stored procedure as single-partitioned, you must specify both the partitioning table and column using the PARTITION PROCEDURE statement in the schema DDL. For example, in our sample application the table RESERVATION is partitioned on FLIGHTID. Let's say you create a stored procedure with two arguments, *flight_id* and *reservation_id*. You declare the stored procedure as single-partitioned in the DDL schema using the FLIGHTID column as the partitioning column. By default, the first parameter to the procedure, *flight_id*, is used as the hash value. For example:

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

At this point, your stored procedure can operate on only those records in the RESERVATION with FLIGHTID=*flight_id*. What's more it can only operate on records in other partitioned tables *that are partitioned on the same hash value*.

In other words, the following rules apply:

- Any SELECT, UPDATE, or DELETE queries of the RESERVATION table must use the constraint WHERE FLIGHTID=? (where the question mark is replaced by the value of *flight_id*).
- SELECT statements can join the RESERVATION table to replicated tables, as long as the preceding constraint is also applied.
- SELECT statements can join the RESERVATION table to other partitioned tables as long as the following is true:
  - The two tables are partitioned on the same column (in this case, FLIGHTID).
  - The tables are joined on the shared partitioning column.
  - The preceding constraint (WHERE RESERVATION.FLIGHTID=?) is used.

For example, the RESERVATION table can be joined to 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=?`
- INVALID: `SELECT c.lastname FROM reservation AS r, customer AS c WHERE r.flightid=? AND c.customerid = r.customerid`

In the first example, the RESERVATION table is being constrained by a column (RESERVATIONID) which is *not* the partitioning column. In the second example, 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.

Warning

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

Finally, the PARTITION PROCEDURE statement assumes that the partitioning column value is the first parameter to the procedure. If you wish to partition on a different parameter value, say the third parameter, you must specify the partitioning parameter using the PARAMETER clause and a zero-based index for the parameter position. In other words, the index for the third parameter would be "2" and the PARTITION PROCEDURE statement would read as follows:

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

3.3. Designing the Application Logic

Once you design your database schema, partitioning, and stored procedures, you are ready to write the application logic. Most of the logic and code of the calling programs are specific to the application you are designing. The important aspect, with regards to using VoltDB, is understanding how to:

- Create a connection to the database
- Call stored procedures
- Close the client connection

The following sections explain how to perform these functions using the standard VoltDB Java client interface. 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 15, *Using VoltDB with Other Programming Languages* for more information about using VoltDB from applications written in other languages.

3.3.1. Connecting to the VoltDB Database

The first step for the calling program is to create a connection to the VoltDB database. You do this by:

1. Defining the configuration for your connections
2. Creating an instance of the VoltDB Client class
3. Calling the createConnection method

```
org.voltdb.client.Client client = null;
ClientConfig config = null;
try {
       config = new ClientConfig("advent","xyzzy");
       client = ClientFactory.createClient(config);
       client.createConnection("myserver.xyz.net");
} catch (java.io.IOException e) {
       e.printStackTrace();
       System.exit(-1);
```

```
}
```

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 3.4, “Handling Errors” for details.)

Once 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 do not match a user defined in the deployment file, the call to createConnection will throw an exception. See Chapter 8, *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.

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

3.3.1.1. 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 distributes 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 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.

3.3.1.2. 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 3.4.3, "Interpreting Other Errors". 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.setReconnecOnConnectionLoss(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.

3.3.2. Invoking Stored Procedures

Once you create the connection, you are ready to call the stored procedures. You invoke a stored procedure using the callProcedure method, passing the procedure name and variables as arguments to callProcedure. For example, to invoke the LookupFlight stored procedure that requires three values (the originating airport, the destination, and the departure time), the call to callProcedure might look like this:

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

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

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, as in the preceding example. See Section 3.2.3.4, "Interpreting the Results of SQL Queries" for more information about interpreting the results of VoltDB stored procedures.

3.3.3. Invoking Stored Procedures Asynchronously

Calling stored procedures synchronously can be useful because it simplifies the program logic; 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.

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

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

The callback procedure (MyCallback in this example) is invoked once the stored procedure completes. It is passed the same structure, ClientResponse, that is returned by 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 following is an example of a callback procedure:

```
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 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 async callback is recommended.

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. Two ways to handle this situation programmatically are to:
  - Let the client pause momentarily to let the queue subside. The asynchronous clent interface does this automatically for you.

- Create multiple connections to the cluster to better distribute asynchronous calls across the database nodes.

- 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.

3.3.4. 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. There are two steps to closing the connection:

1. Call drain() to make sure all asynchronous calls have completed.

2. Call close() to close all of the connections and release any resources associated with the client.

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.

The following example demonstrates how to close the client connection:

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

3.4. Handling Errors

One 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 executing stored procedures and in asynchronous callbacks.

3.4.1. Interpreting Execution Errors

If an error occurs in a stored procedure (such as an SQL contraint 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().

The getStatus() method tells you whether the stored procedure completed successfully and, if not, what type of error occurred. 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 3.4.2, “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.

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. In addition to identifying the type of error, for any values other than SUCCESS, the getStatusString() method returns a text message providing more information about the specific error that occurred.

If your stored procedure wants 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() getStatusString(), but rather than returning information set by VoltDB, getAppStatus() and getAppStatusString() return information set by 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. For example:

Stored Procedure

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

Client Application

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

    if (clientResponse.getStatus() != ClientResponse.SUCCESS) {
        System.err.println(clientResponse.getStatusString());
```

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

3.4.2. 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.

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, 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.

It is important to note that CONNECTION_TIMEOUT does not necessarily mean the 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, CONNECTION_LOST does not necessarily mean a pending 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 is 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, Section 3.4.3, “Interpreting Other Errors”, for an example of creating a status listener for delayed procedure responses.

3.4.3. Interpreting Other Errors

Certain types of errors can occur that the ClientResponse class cannot notify you about immediately. These errors include:

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.
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.
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.
Delayed 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.

In each of 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.

You must define the listener before you define the VoltDB client or open a connection. The ClientStatusListenerExt interface has four methods that you can implement — one for each type of error situation — *connectionLost*, *backpressure*, *uncaughtException*, and *lateProcedureResponse*. Once you declare your ClientStatusListenerExt, you add it to a ClientConfig object that is then used to define the client. The configuration class also defines the username and password to use for all connections.

By performing the operations in this order, 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.

The following example illustrates:

❶ Declaring a ClientStatusListenerExt
❷ Defining the client configuration, including authentication credentials and the status listener
❸ Creating a client with the specified configuration

For the sake of example, this 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,
                               int connectionsLeft,
                               DisconnectCause cause)
      {
        System.out.printf("A connection to the database 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 occured 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",       ❷
                                         "password",
                                         mylistener);

        /*
        *  Create the client using the specified configuration.
        */
Client myclient = ClientFactory.createClient(myconfig);    ❸
```

Chapter 4. Simplifying Application Development

The previous chapter (Chapter 3, *Designing Your VoltDB Application*) 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:

1. Default procedures
2. Shortcuts for defining simple stored procedures
3. Verifying expected SQL query results

The following sections describe each of these features separately.

4.1. 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 default stored procedures to perform these queries for any table with a primary key index. It also defines a default insert stored procedure for tables without a primary key. When you compile the application catalog, these default procedures are added to the catalog automatically.

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. The parameters to the procedures differ based on the procedure. For the insert procedure, the parameters are the columns of the table, in the same order as defined in the schema. For the select and delete procedures, only the primary key column values are required (listed in the order they appear in the primary key definition). For the update and upsert procedures, the columns 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.)

For example, the Hello World tutorial contains a single table, HELLOWORLD, with three columns and the partitioning column, DIALECT, as the primary key. As a result, the application catalog includes six default stored procedures, in addition to any user-defined procedures declared in the schema. Those default procedures are:

- HELLOWORLD.insert
- HELLOWORLD.select
- HELLOWORLD.update
- HELLOWORLD.upsert
- HELLOWORLD.delete

The following code example uses the default procedures for the HELLOWORLD table to insert, retrieve, 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");
```

4.2. 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. Normally, the schema contains entries that identify each of the stored procedures, like so:

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

The CREATE PROCEDURE statement specifies the class name of the Java procedure you write. 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;
```

When you include the SQL query in the CREATE PROCEDURE AS statement, VoltDB creates the procedure when you build your application (as described in Section 5.3, “Building the Application Catalog”). 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.

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 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 is single-partitioned, you can specify the partitioning information 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;
```

4.3. Writing Stored Procedures Inline Using Groovy

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 stored procedures directly in the schema definition file (DDL) by embedding the procedure code using the Groovy programming language. Groovy is an object-oriented language that dynamically compiles to Java Virtual Machine (JVM) bytecode. Groovy is not as strict as Java and promotes simpler coding through implicit typing and other shortcuts.

You embed a Groovy stored procedure in the schema file 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 CREATE PROCEDURE statement implements the Insert stored procedure from the Hello World example using Groovy:

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

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

- The definitions for VoltTypes, VoltProcedure, and VoltAbortException are automatically included and can be used without explicit import statements.
- As with Java stored procedures, you must declare all SQL queries as SQLStmt objects at the beginning of the Groovy procedure.
- 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 and can return a VoltTable, an array of VoltTables, or a long value.

In addition, VoltDB provides special wrappers, tuplerator and buildTable, that help you access VoltTable results and construct VoltTables from scratch. For example, the following code fragment shows the ContestantWinningStates stored procedure from the Voter sample application 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
        }
    }
}
```

Finally, 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.

4.4. 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 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 stored procedure 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 stored procedures.

Expectation	**Description**
EXPECT_EMPTY	The query must return no rows.
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.

Chapter 5. Building Your VoltDB Application

Once you have designed your application and created the source files, you are ready to build your application. There are four steps to building a VoltDB application:

1. Compiling the client application and stored procedures
2. Declaring the stored procedures in the schema
3. Compiling the VoltDB application catalog

This chapter explains these steps in more detail.

5.1. Compiling the Client Application and Stored Procedures

The VoltDB client application and stored procedures are written as Java classes[1], so you compile them using the Java compiler. To do this, 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 using the `-classpath` argument on the command line. You can also specify where to create the resulting class files using the `-o` flag to specify an output directory, as in the following example:

```
$ javac -classpath "./:/opt/voltdb/voltdb/*" \
        -o ./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 will need to include your installation path in the `-classpath` argument. Also, if your client application depends on other libraries, they will need to be included in the classpath as well.

5.2. Declaring the Stored Procedures

In addition to compiling the stored procedures, you must tell VoltDB which procedures to include in the runtime catalog. You do this by adding CREATE PROCEDURE statements to the database schema. For example:

```
CREATE PROCEDURE FROM CLASS procedures.LookupFlight;
CREATE PROCEDURE FROM CLASS procedures.HowManySeats;
CREATE PROCEDURE FROM CLASS procedures.MakeReservation;
CREATE PROCEDURE FROM CLASS procedures.CancelReservation;
CREATE PROCEDURE FROM CLASS procedures.RemoveFlight;
```

Be sure to identify all of your stored procedures or they will not be included in the catalog and therefore will not be available to the client applications at runtime.

[1]Although VoltDB stored procedures must be written in Java and the primary client interface is Java, it is possible to write client applications using other programming languages. See Chapter 15, *Using VoltDB with Other Programming Languages* for more information on alternate client interfaces.

You also specify the partitioning of the database tables and stored procedures in the schema. The important point is that if you do not specify partitioning information for a table, that table will be replicated in all partitions. See Section 3.1, "Designing the Database" for more information about partitioned and replicated tables.

5.3. Building the Application Catalog

You build the application catalog for your VoltDB database by compiling the database schema and stored procedures into the catalog. To run the compiler, use the **`voltdb compile`** command, specifying three arguments:

1. The path to your compiled stored procedure classes
2. The name of the schema file to use as input
3. The name of the application catalog to create as output

For example, if your stored procedure classes are in a subfolder called obj, the command might be:

```
$ voltdb compile --classpath="obj" -o flight.jar flightschema.sql
```

If you do not specify an output file, the catalog is created as `catalog.jar` in the current working directory.

Chapter 6. Running Your VoltDB Application

There are three steps to running a VoltDB application:

- Defining the cluster configuration
- Starting the VoltDB database
- Starting the client application or applications

The following sections describe the procedures for starting and stopping a VoltDB database in detail.

6.1. Defining the Cluster Configuration

The schema that is used to compile the application catalog defines how the database is logically structured: what tables to create, which tables are partitioned, and how they are accessed (i.e. what stored procedures to support). The other important aspect of a running database is the physical layout of the cluster that runs the database. This includes information such as:

- The number of nodes in the cluster
- The number of partitions (or "sites") per node
- The amount of K-safety to establish for durability

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"
            sitesperhost="n"
            kfactor="n"
   />
</deployment>
```

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

- **hostcount** — specifies the number of nodes in the cluster.
- **sitesperhost** — specifies the number of partitions (or "sites") per host. 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, Section 6.1.1, “Determining How Many Partitions to Use” explains how to determine the optimal value.
- **kfactor** — specifies the K-safety value to use when creating the database. This attribute is optional. If you do not specify a value, it defaults to zero. (See Chapter 11, *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 and specify only the catalog on the command line. If you do not specify a

deployment file or host, VoltDB defaults to one node, eight sites per host, a K-safety value of zero, and localhost as the host.

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 specifies whether security is enabled and defines the users and passwords that are used to authenticate clients at runtime. See Chapter 8, *Security* for more information about security and VoltDB databases.

6.1.1. Determining How Many Partitions to Use

There is very little penalty for allocating more partitions than needed (except for incremental memory usage). Consequently, VoltDB defaults to eight partitions 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 less than 8 processors and limited memory, you may wish to tune the sitesperhost attribute.

The number of partitions 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 partitions per node.

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 partitions 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 partitions per node is best calculated by actually benchmarking the application to see what combination of cores and partitions produces the best results. However, it is important to remember that all nodes in the cluster will use the same number of partitions. So the best performance is achieved by using a cluster with all nodes having the same physical architecture (i.e. cores).

6.1.2. Configuring Paths for Runtime Features

In addition to configuring the database process on each node of the cluster, the deployment file lets you enable and configure a number of features within VoltDB. Export, automatic snapshots, and network partition detection are all enabled through the deployment file. The later chapters of this book describe these features in detail.

An important aspect of these features is that some of them 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. To specify a common root, use the `<voltdbroot>` tag (as a child of `<paths>`) to specify where VoltDB will store disk files. For example, the following `<paths>` tag set specifies `/tmp` as the root directory:

```
<paths>
   <voltdbroot path="/tmp" />
</paths>
```

Of course, /tmp is appropriate for temporary files, such as export overflow. But /tmp is not a good location for files that must persist when the server reboots. So you can also identify specific locations for individual

features. For example, the following excerpt from a deployment file specifies `/tmp` as the default root but `/opt/voltdbsaves` as the directory for automatic snapshots:

```
<paths>
   <voltdbroot path="/tmp" />
   <snapshots path="/opt/voltdbsaves" />
</paths>
```

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.

On the other hand, 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. Similarly, if you name a specific feature path (such as the snapshots path) and it does not exist, VoltDB will attempt to create it for you.

6.1.3. 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 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 100 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.

6.2. Starting a VoltDB Database for the First Time

Once you define the configuration of your cluster, you start a VoltDB database by starting the VoltDB server process on each node of the cluster. You start the server process by invoking VoltDB and specifying:

- A startup action (see Section 6.5, “Stopping and Restarting a VoltDB Database” for details)
- The location of the application catalog
- The hostname or IP address of the host node in the cluster
- The location of the deployment file

The *host* can be any node in the cluster and plays a special role during startup; it hosts the application catalog and manages the cluster initiation process. Once startup is complete, the host's role is complete and it becomes a peer of all the other nodes. It is important that all nodes in the cluster can resolve the hostname or IP address of the host node you specify.

For example, the following `voltdb` command starts the cluster with the **create** startup action, specifying the location of the catalog and the deployment files, and naming voltsvr1 as the host node:

```
$ voltdb create mycatalog.jar \
     --deployment=deployment.xml \
     --host=voltsvr1
```

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

```
$  voltdb create mycatalog.jar \
     -d deployment.xml \
     -H voltsvr1
```

If you are using the VoltDB Enterprise Edition, you must also provide a license file. The license is only required by the host node when starting the cluster. To simplify startup, VoltDB looks for the license as a file named `license.xml` in three locations, in the following order:

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

So if you store the license file in any of these locations, you do not have to explicitly identify it on the command line. Otherwise, you can use the `--license` or `-l` flag to specify the license file location, For example:

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

When you are developing an application (where your cluster consists of a single node using localhost), this one command is sufficient to start the database. However, when starting a cluster, you must:

1. Copy the runtime catalog to the host node.
2. Copy the deployment file to all nodes of the cluster.
3. Log in and start the server process using the preceding command on each node.

The deployment file must be identical on all nodes for the cluster to start.

6.2.1. Simplifying Startup on a Cluster

Manually logging on to each node of the cluster every time you want to start the database can be tedious. There are several ways you can simplify the startup process:

- **Shared network drive** — By creating a network drive and mounting it (using NFS) on all nodes of the cluster, you can distribute the runtime catalog and deployment file (and the VoltDB software) by copying it once to a single location.
- **Remote access** — When starting the database, you can specify the location of either the runtime catalog or the deployment file as a URL rather than a file path (for example, `http://myserver.com/`

`mycatalog.jar`). This way you can publish the catalog and deployment file once to a web server and start all nodes of the server from those copies.

- **Remote shell scripts** — Rather than manually logging on to each cluster node, you can use secure shell (ssh) to execute shell commands remotely. By creating an ssh script (with the appropriate permissions) you can copy the files and/or start the database on each node in the cluster from a single script.

- **VoltDB Enterprise Manager** — The VoltDB Enterprise Edition includes a web-based management console, called the VoltDB Enterprise Manager, that helps you manage the configuration, initialization, and performance monitoring of VoltDB databases. The Enterprise Manager automates the startup process for you. See the *VoltDB Management Guide* for details.

6.2.2. How VoltDB Database Startup Works

When you are starting a VoltDB database, 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.

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.

3. Once all the nodes have sent initialization messages, the host sends out a message to the other nodes that the cluster is complete. The host then distributes the application catalog to all nodes.

At this point, the cluster is complete and the database is ready to receive requests from client applications. Several points to note:

- 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.

- The database is not operational until the correct number of nodes (as specified in the deployment file) have connected.

6.3. Starting VoltDB Client Applications

Client applications written in Java compile and run like other Java applications. Once again, when you start your client application, you must make sure that the VoltDB library JAR file is in the classpath. For example:

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

When developing your application (using one of the sample applications as a template), the `run.sh` file manages this dependency for you. However, 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.

6.4. Shutting Down 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 with a single command. You can do this either programmatically with the @Shutdown system procedure or interactively with the **voltadmin shutdown** command.

Either calling the @Shutdown system procedure (from any node) or invoking **voltadmin shutdown** will shutdown the database on the entire cluster. You do not have to issue commands on each node. Entering **voltadmin shutdown** without specify a host server assumes the current system is part of the database cluster. To shutdown a database running on different servers, you 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 --host=zeus shutdown
```

6.5. Stopping and Restarting a VoltDB Database

Because VoltDB is an in-memory database, once the database server process stops, the data itself is removed from memory. If you restart the database without taking any other action, the database starts fresh without any data. However, in many cases you want to retain the data across sessions. There are two ways to do this:

- Save and restore database snapshots
- Use command logging and recovery to reload the database automatically

6.5.1. Save and Restore

A database *snapshot* is exactly what it sounds like — a point-in-time copy the database contents written to disk. You can later use the snapshot to restore the data.

To save and restore data across sessions, you can perform a snapshot before shutting down the database and then restore the snapshot after the database restarts. You can perform a manual snapshot using the **voltadmin** command or using the @SnapshotSave system procedure. For example, the following commands pause the database, perform a manual snapshot, then do a shutdown on the current system:

```
$ voltadmin pause
$ voltadmin save 'voltdbroot/snapshots' 'MySnapshot'
$ voltdbadmin shutdown
```

You can also have the database automatically create periodic snapshots using the snapshot feature in the deployment file. See Chapter 9, *Saving & Restoring a VoltDB Database* for more information about using snapshots to save and restore the database.

6.5.2. Command Logging and Recovery

Another option for saving data across sessions is to use command logging and recovery. When command logging is enabled (which it is by default in VoltDB Enterprise Edition), the database not only performs periodic snapshots, it also keeps a log of all stored procedures that are initiated at each partition. If the database stops for any reason — either intentionally or due to system failure — when the server process restarts, the database restores the last snapshot and then "replays" the command log to recover all of the data committed prior to the cluster shutting down.

To support command logging, an alternative startup action is available on the command line when starting the server process. The valid startup actions are:

- **create** — explicitly creates a new, empty database and ignores any command log information, if it exists.
- **recover** — starts a new database process and recovers the command log from the last database session. The `recover` action is explicit; if the command log content is not found or is incomplete, the server initialization process stops and reports an error.

Even if you are not using command logging, you can still use the `create` and `recover` actions with automated snapshots. During the `recover` action, VoltDB attempts to restore the last snapshot found in the snapshot paths. Therefore, using automated snapshots and `the recover` action, it is possible to automatically recover all of the data from the previous database session up until the last snapshot.

The following example illustrates how to recover a database from a previous session.

```
$ voltdb recover --host=voltsvr1 \
     --deployment=deployment.xml \
     --license=/opt/voltdb/voltdb/license.xml
```

The advantages of command logging and recovery are that:

- The command log ensures that all data is recovered, including transactions between snapshots.
- The recovery is automated, ensuring no client activity occurs until the recovery is complete.

See Chapter 10, *Command Logging and Recovery* for more information about configuring command logging.

6.6. Modes of Operation

There are actually two modes of operation for a VoltDB database: normal operation and admin mode. During normal operation clients can connect to the cluster and invoke stored procedures (as allowed by the security permissions set in the application catalog and deployment files). In *admin mode*, only clients connected through a special admin port are allowed to initiate stored procedures. Requests received from any other clients are rejected.

6.6.1. Admin Mode

The goal of admin mode is to quell database activity prior to executing sensitive administrative functions. By entering admin mode, it is possible to ensure that no changes are made to the database contents during operations such as save, restore, or updating the runtime catalog.

You initiate admin mode by calling the @Pause system procedure through the admin port. The *admin port* works just like the regular client port and can be called through any of the standard VoltDB client interfaces (such as Java or JSON) by specifying the admin port number when you create the client connection.

Once the database enters admin mode, any requests received over the client port are rejected, returning a status of ClientResponse.SERVER_UNAVAILABLE. The client application can check for this response and resubmit the transaction after a suitable pause.

By default the admin port is 21211, but you can specify an alternate admin port using the <admin-mode> tag in the deployment file. For example:

```
<deployment>
```

```
    ...
   <admin-mode port="9999" />
</deployment>
```

Once admin mode is turned on, VoltDB processes requests received over the admin port only. Once you are ready to resume normal operation, you must call the system procedure @Resume through the admin port.

6.6.2. Starting the Database in Admin Mode

By default, a VoltDB database starts in normal operating mode. However, you can tell the database to start in admin mode by adding the adminstartup attribute to the <admin-mode> tag in the deployment file. For example:

```
<deployment>
    ...
   <admin-mode port="9999" adminstartup="true" />
</deployment>
```

When adminstartup is set to true, the database starts in admin mode. No activity is allowed over the standard client port until you explicitly stop admin mode with the **voltadmin resume** command or a call to the @Resume system procedure.

Starting in admin mode can be very useful, especially if you want to perform some initialization on the database prior to allowing client access. For example, it is recommended that you start in admin mode if you plan to manually restore a snapshot or prepopulate the database with data through a set of custom stored procedures. For example, the following commands restore a snapshot, then exit admin mode once the initialization is complete:

```
$ voltadmin restore 'voltdbroot/snapshots' 'MySnapshot'
$ voltdbadmin resume
```

Chapter 7. Updating Your VoltDB Database

Unlike traditional databases that allow interactive SQL statements for defining and modifying database tables, VoltDB requires you to pre-compile the schema and stored procedures into the application catalog. Pre-compiling lets VoltDB verify the structure of the database (including the partitioning) and optimize the stored procedures for maximum performance.

The down side of pre-compiling the database and stored procedures is that you cannot modify the database as easily as you can with more traditional relational database products. Of course, this constraint is both a blessing and a curse. It helps you avoid making rash or undocumented changes to the database without considering the consequences.

It is never a good idea to change the database structure or stored procedure logic arbitrarily. But VoltDB recognizes the need to make adjustments even on running systems. Therefore, the product provides mechanisms for updating your database and hardware configuration as needed, while still providing the structure and verification necessary to maintain optimal performance.

7.1. Planning Your Application Updates

Many small changes to the database application, such as bug fixes to the internal code of a stored procedure or adding a table to the database schema, do not have repercussions on other components of the system. It is nice to be able to make these changes with a minimal amount of disruption. Other changes can impact multiple aspects of your applications. For example if you add or remove an index from a table or modify the parameters to a stored procedure. Therefore, it is important to think through the consequences of any changes you make.

VoltDB tries to balance the trade offs of changing the database environment, making simple changes easy and automating as much as possible even complex changes. Using the VoltDB Enterprise Edition you can add, remove, or update stored procedures "on the fly", while the database is running. You can also add or drop tables and columns from the schema, as well as modify many indexes.

To make other changes to the database schema (such as adding unique indexes) you must first save and shutdown the database. However, even in this situation, VoltDB automates the process by transforming the data when you restart and reload the database in a new configuration.

This chapter explains different methods for making changes to your VoltDB database application, including:

- Updating the Database Schema on a Running Database
- Updating the Database Using Save and Restore
- Updating the Hardware Configuration

7.2. Updating the Database Schema on a Running Database

Many normal changes to the database schema and stored procedures can be made "on the fly", in other words while the database is running. These changes include:

- Adding, removing, or updating tables, columns, and indexes
- Adding or removing materialized views and export-only tables
- Adding, removing, or updating stored procedures and the security permissions for accessing them

Live schema updates are done by creating an updated application catalog and deployment file and telling the database process to use the new catalog. You do this with the @UpdateApplicationCatalog system procedure, or from the shell prompt using the **voltadmin update** command. The process is as follows:

1. Make the necessary changes to the source code for the stored procedures and the schema.
2. Recompile the class files and the application catalog as described in Chapter 5, *Building Your VoltDB Application*.
3. Use the @UpdateApplicationCatalog system procedure or **voltadmin update** command to pass the new catalog and deployment file to the cluster.

For example:

```
$ voltdb compile -o mycatalog.jar  myschema.sql
$ voltadmin update mycatalog.jar mydeployment.xml
```

7.2.1. Validating the Updated Catalog

When you submit a catalog update, the database nodes do a comparison of the new catalog and deployment configuration with the currently running catalog to ensure that only supported changes are included. If unsupported changes are included, the command returns an error.

Most schema changes are supported. The only changes that are not currently allowed are changes that add constraints to an existing index or column or that make changes to the contents of an existing view. To make these more complex changes, you need to save and restore the database to change the catalog, as described in Section 7.3, “Updating the Database Using Save and Restore”.

7.2.2. Managing the Update Process

Updating the application catalog lets you modify the database schema and its stored procedures without disrupting the normal operations. However, even when a change is allowed, you should be careful of the impact to client applications that use those procedures. For example, if you remove a table or change the parameters to a stored procedure while client applications are still active, you are likely to create an error condition for the calling applications.

In general, the catalog update operates like a transaction. Before the update, the original attributes, including permissions, are in effect. After the update completes, the new attributes and permissions are in effect. In either case, any individual call to the stored procedure will run to completion under a consistent set of rules.

For example, if a call to stored procedure A is submitted at approximately the same time as a catalog update that removes the stored procedure, the call to stored procedure A will either complete successfully or return an error indicating that the stored procedure no longer exists. If the stored procedure starts, it will not be interrupted by the catalog update.

In those cases where you need to make changes to a stored procedure that might negatively impact client applications, the following process is recommended:

1. Perform a catalog update that introduces a new stored procedure (with a new name) that implements the new function. Assuming the original stored procedure is A, let's call its replacement procedure B.
2. Update all client applications, replacing calls to procedure A with calls to procedure B, making the necessary code changes to accommodate any changed behavior or permissions.
3. Put the updated client applications into production.
4. Perform a second catalog update removing stored procedure A, now that all client application calls to the original procedure have been removed.

7.3. Updating the Database Using Save and Restore

If you need to make changes that are not supported by the **voltadmin update** command, it is still possible to modify the database schema using save and restore. You can modify the schema, including adding new constraints or modifying views, using the following steps:

1. Save the current data, using **voltadmin save**.
2. Shut down the database, using **voltadmin shutdown**.
3. Replace the application catalog.
4. Restart the database with the new catalog, using **voltdb create**.
5. Reload the data saved in Step #1 using **voltadmin restore**.

Using these steps, you can add or remove tables, columns and indexes. You can also change the datatype of existing columns, as long as you make sure the new type is compatible with the previous type (such as exchanging integer types or string types) and the new datatype has sufficient capacity for any values that currently exist within the database.

However, you cannot change the name of a column, add constraints to a column or change to a smaller datatype (such as changing from INTEGER to TINYINT) without the danger of losing data. To make these changes safely, it is better to add a new column with the desired settings and write a client application to move data from the original column to the new column, making sure to account for exceptions in data size or constraints.

See Section 9.1.3, “Changing the Database Schema or Cluster Configuration Using Save and Restore” for complete instructions for using save and restore to modify the database schema.

7.4. Updating the Hardware Configuration

Another change you are likely going to want to make at some point is changing the hardware configuration of your database cluster. Reasons for making these changes are:

- Increasing the number of nodes (and, as a consequence, capacity and throughput performance) of your database.
- Benchmarking the performance of your database application on different size clusters and with different numbers of partitions per node.

You can always change the number of nodes by saving the data (using a snapshot or command logging), editing the deployment file to specify the new number of nodes in the hostcount attribute of the <cluster>

tag, then stopping and restarting the database and using the **voltadmin restore** command to reload the data. 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.

7.4.1. 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, the DR agent stops replication. Once rebalancing is complete on the master database, you can:

  1. Restart the replica with additional nodes matching the new master cluster configuration.
  2. Restart the DR agent.

7.4.2. 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 re-

mains 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 8. 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.

8.1. 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 SHA-1 hashing rather than encryption when passing the username and password between the client and the server. The passwords are also hashed within the database. For an encrypted solution, you can consider implementing Kerberos security, described in Section 8.7, “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.

8.2. 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>
```

8.3. 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.

8.4. 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.

8.5. 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 8.1, “Named Security Permissions” describes the named functional permissions.

Table 8.1. 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[a]	Access to all ad hoc SQL queries, including data definition language	SQLREAD, DEFAULTPROC

Permission	Description	Inherits
	(DDL) statements and default procedures	
ALLPROC	Access to all user-defined stored procedures	
ADMIN[a]	Full access to all system procedures, all user-defined procedures, as well as default procedures and ad hoc SQL	ALLPROC, DEFAULTPROC, SQL

[a]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 (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;
```

8.6. 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, DDL, system procedures, and user-defined procedures. *User* has SQL and ALLPROC pemissions: access to ad hoc SQL, DDL, and all default and user-defined stored procedures.

These predefined roles are important when using the new dynamic DDL because if you start the database without a catalog, 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.

8.7. 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.

8.7.1. 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
```

8.7.2. 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;
};
```

8.7.3. 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 8.3, “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:

[1] On Macintosh systems, you must always specify the `java.security.krb5.conf` property.

```
import org.voltdb.client.ClientConfig;
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 9. 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.

9.1. 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 catalog and/or deployment file (if desired).
5. Restart the cluster in admin mode.
6. Restore the previous snapshot.
7. 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.

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.

9.1.1. 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.

9.1.2. How to Restore the Contents of a VoltDB Database

To 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 9.1.1.

```
$ 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.

9.1.3. Changing the Database Schema or Cluster Configuration Using Save and Restore

Between a save and a restore, it is possible to make selected changes to the database. You can:

- Add nodes to the cluster
- Modify the database schema
- Add, remove, or modify stored procedures

To make these changes, you must, as appropriate, edit the database schema, the procedure source files, or the deployment file. You can then recompile the application catalog and distribute the updated catalog and deployment file to the cluster nodes before restarting the cluster and performing the restore.

9.1.3.1. Adding Nodes to the Database

To add nodes to the cluster, use the following procedure:

- Save the database.
- Edit the deployment file, specifying the new number of nodes in the hostcount attribute of the <cluster> tag.
- Restart the cluster (including the new nodes).
- 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.

9.1.3.2. Modifying the Database Schema and Stored Procedures

To modify the database schema or stored procedures, make the appropriate changes to the source files (that is, the database DDL and the stored procedure Java source files), then recompile the application catalog. However, you can only make certain modifications to the database schema. Specifically, you can:

- Add or remove tables.
- Add or remove columns from tables.
- Change the datatypes of columns, assuming the two datatypes are compatible. (That is, the data can be converted from the old to the new type. For example, extending the length of VARCHAR columns or converting between two numeric datatypes.)

Note that you *cannot* rename tables or columns and retain the data. If you rename a table or column, it is equivalent to deleting the original table/column (and its data) and adding a new one. Two other important points to note when modifying the database structure 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.

9.2. 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 unique identifier to use as a prefix for the snapshot files.
- 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 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 6.1.2, “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>
```

9.3. 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.

9.4. 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 10. 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.

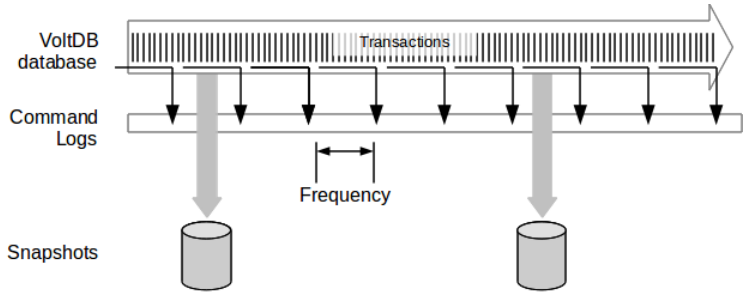
However, any additional processing can impact overall performance, especially when it involves disk I/O. So it is important to understand the tradeoffs concerning different aspects of command logging and how it interacts with the hardware and any other options you are utilizing. The following sections explain how command logging works and how to configure it to meet your specific needs.

10.1. 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 10.1. Command Logging in Action

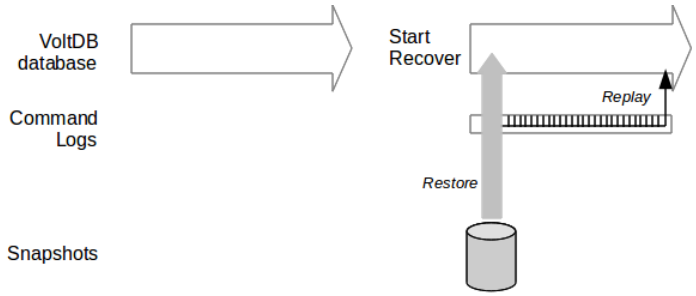


The frequency with which the transactions are written to the command log is configurable (as described in Section 10.3, “Configuring Command Logging for Optimal Performance”). By adjusting the frequency and

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 6.5.2, “Command Logging and Recovery”) 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 10.2. Recovery in Action



10.2. 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.

10.3. 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.

10.3.1. 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 10.1, “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.

10.3.2. 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 10.1, “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.

10.3.3. 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 10.3.4, “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>
```

10.3.4. 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 11. 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 9, *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 10, *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.

Previous chapters described 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.

11.1. 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.)

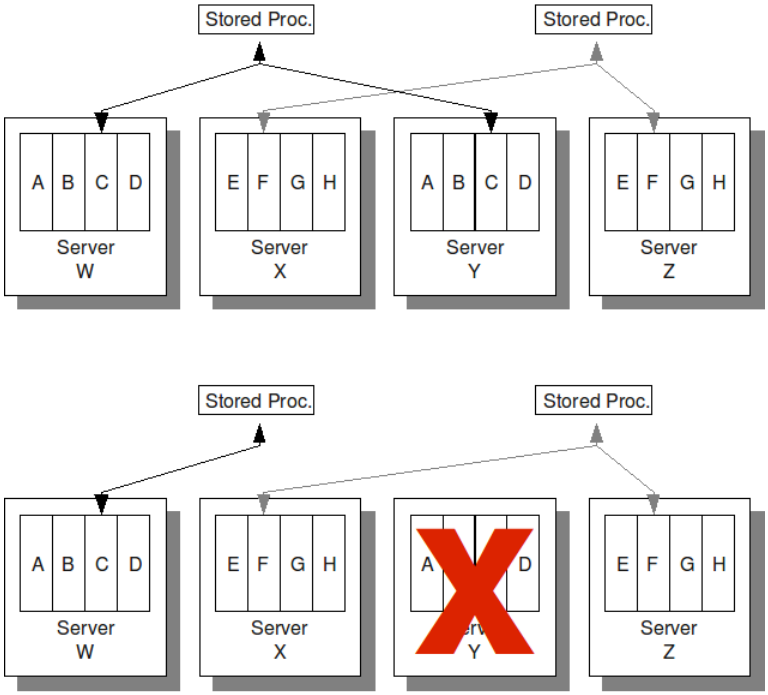
It is also important to note that K-safety is different than WAN replication. In replication the entire database cluster is replicated (usually at a remote location to provide for disaster recovery in case the entire cluster or site goes down due to catastrophic failure of some type).

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 11.1, “K-Safety in Action”). If a node fails, the database continues to function sending the work to the unaffected copies of the partition.

Figure 11.1. K-Safety in Action



11.2. 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.

11.2.1. 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.

11.2.2. 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.

11.3. 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 Chapter 14, *Logging and Analyzing Activity in a VoltDB Database* 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.

11.3.1. 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 application catalog 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 catalog 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.

11.3.2. 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 with the appropriate catalog and deployment file 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.

11.4. 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*.

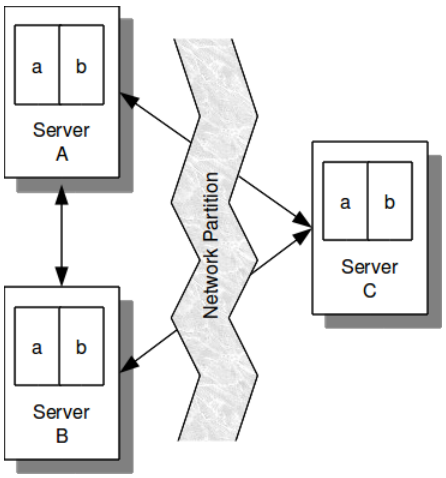
11.4.1. K-Safety and Network Partitions

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 11.2, “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 11.2. 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.

11.4.2. 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 cluster by 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 6.1.2, “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/`.

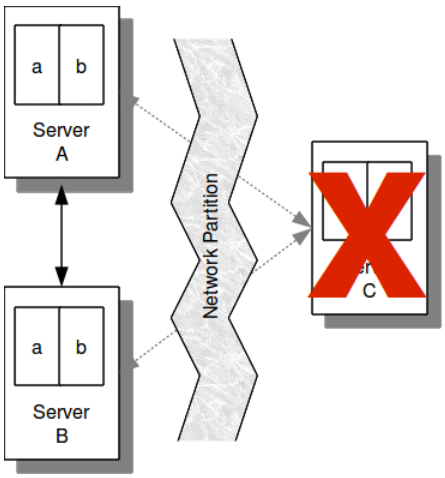
```
   <partition-detection enabled="true">
       <snapshot prefix="netfaultsave"/>
   </partition-detection>
   <paths>
       <snapshots path="/opt/voltdb/snapshots/" />
   </paths>
```

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 11.2, “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 11.3, “Network Fault Protection in Action”).

Figure 11.3. 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 11.3, “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 12. Database Replication

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

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 a separate and distinct copy of the entire database.

Database replication can be used for:

- Offloading read-only workloads, such as reporting
- Maintaining a "hot standby" in case of failure
- Protecting against catastrophic events, often called disaster recovery

The next section, Section 12.1, “How Database Replication Works”, explains the principles behind database replication in VoltDB. Section 12.2, “Database Replication in Action” provides step-by-step instructions for establishing and managing database replication using the functions and features of VoltDB, including:

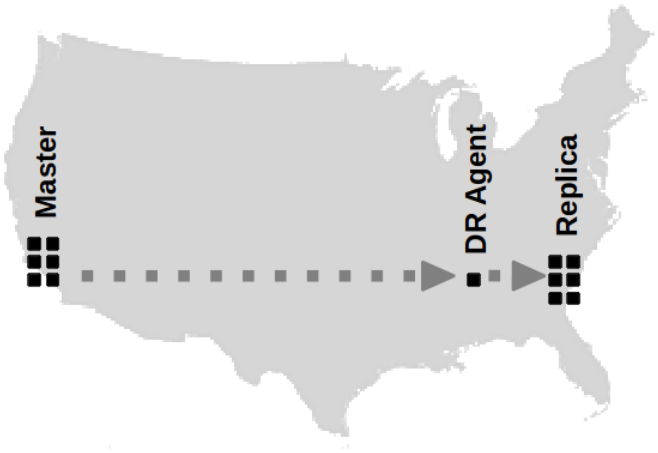
- Starting Replication
- Stopping Replication
- What to Do in Case of a Disaster
- Monitoring and Managing Replication

12.1. How Database Replication Works

Database replication involves duplicating the contents of one database cluster (known as the *master*) to another database cluster (known as the *replica*). The contents of the replica cluster are completely controlled by the master, which is why this arrangement is sometimes referred to as a *master/slave* relationship.

The replica database can be in the rack next to the master, in the next room, the next building, or another city entirely. The location depends upon your goals for replication. For example, if you are using replication for disaster recovery, geographic separation of the master and replica is required. If you are using replication for hot standby or offloading read-only queries, the physical location may not be important.

Figure 12.1. The Components of Database Replication



The process of retrieving completed transactions from the master and applying them to the replica is managed by a separate process called the Data Replication (DR) agent. The DR agent is critical to the replication process. It performs the following tasks:

- Initiates the replication, telling the master database to start queuing completed transactions and establishing a special client connection to the replica.
- POLLs and ACKs the completed transactions from the master database and recreates the transactions on the replica.
- Monitors the replication process, detects possible errors in the replica or delays in synchronizing the two clusters, and — when necessary — reports error conditions and cancels replication.

12.1.1. Starting Replication

Database Replication is easy to establish:

1. Any normal VoltDB database can be the master; you simply start the database as usual and the DR agent tells the master when it should start queuing completed transactions.
2. Next, you create the replica database. You do this by starting the database with the **`create`** action and the **--replica** flag. This creates a read-only database that waits for the DR agent to contact it.
3. Finally, you start the DR agent, specifying the location of the master and replica databases.

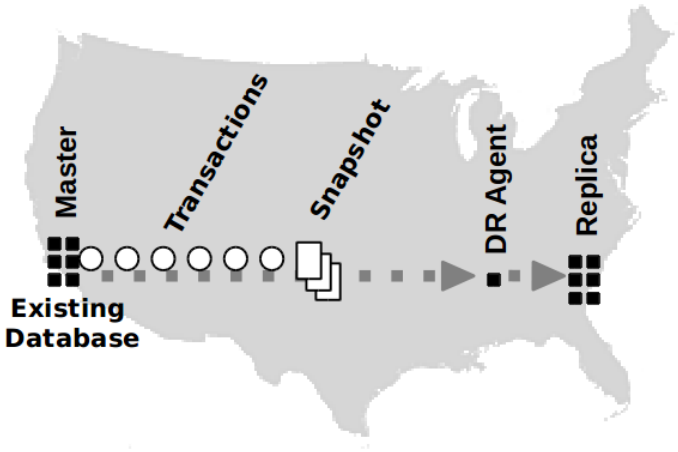
Note that the DR agent can be located anywhere. However, the replication process is optimized for the DR agent to be co-located with the replica database (as shown in Figure 12.1, “The Components of Database Replication”). Communication between the DR agent and the master database is kept to a minimum to avoid bottlenecks; only write transactions are replicated and the messages between the master and the agent are compressed. Whereas the DR agent sends transactions to the replica using standard client invocations. Therefore, when distributing the database across a wide-area network (WAN), locating the DR agent near the replica is recommended.

12.1.2. Replication and Existing Databases

If data already exists in the master database when the DR agent starts replication, the master first creates a snapshot of the current contents and passes the snapshot to the DR agent so the master and the replica can

start from the same point. The master then queues and transmits all subsequent transactions to the agent, as shown in Figure 12.2, “Replicating an Existing Database”.

Figure 12.2. Replicating an Existing Database



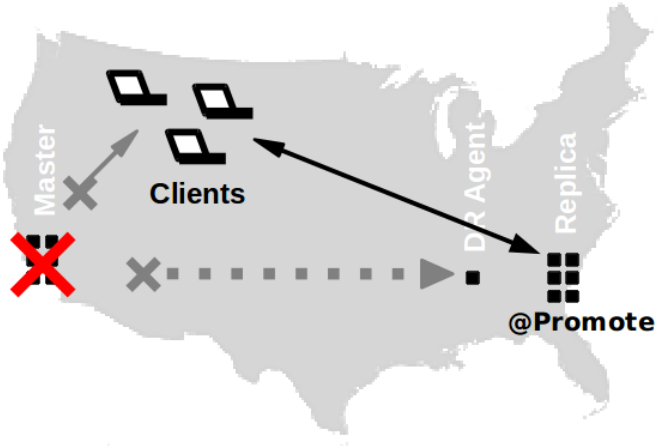
12.1.3. Database Replication and Disaster Recovery

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 a master itself. 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 shut it down.
2. Stop the DR agent, if it has not stopped already.
3. Promote the replica to a master using the **voltadmin promote** command.
4. Redirect the client applications to the new master database.

Figure 12.3, “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.

Figure 12.3. Promoting the Replica



Once the master is offline and the replica is promoted to a master itself, 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, use the original master hardware to create a new replica — essentially switching the roles of the master and replica databases.
- 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.

12.1.4. 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 completed transactions from the master database to the replica. Because replication happens after the fact, there is no guarantee that the contents of the master and replica cluster are identical at any given point in time. Instead, the replica database 'catches up" with the master after the transactions are received and processed by the DR agent.

If the master cluster crashes, there is no guarantee that the DR agent has managed to retrieve all transactions that were queued on the master. Therefore, it is possible that some transactions that completed on the master are not reproduced on the replica.

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. 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.

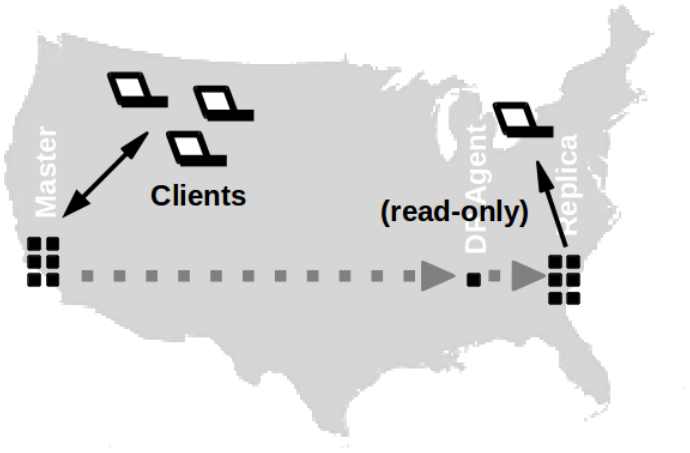
Your own business requirements and the specific situation that caused the outage will determine which choice to make. However, database replication makes the choice possible and significantly eases the dangers of unforeseen events.

12.1.5. Database Replication and Read-only Clients

While database replication is occurring, the replica responds to write transactions (INSERT, UPDATE, and DELETE) from the DR agent only. Other clients can connect to the replica and use it for read-only transactions, including read-only ad hoc queries and system procedures. Any attempt to perform a write transaction from a client other than the DR agent returns an error.

There will always be some delay between a transaction completing on the master and being replayed 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 12.4. Read-Only Access to the Replica



12.2. Database Replication in Action

The previous section explains the principles behind database replication. The following sections provide step-by-step instructions for setting up and managing replication using VoltDB.

All of the following examples use the same fictional servers to describe the replication process. The server used for the master cluster is called serverA; the server for the replica is serverB.

12.2.1. Starting Replication

It is easy to establish database replication with VoltDB. You can replicate any VoltDB database — there are no special requirements or configuration needed for the master database. It is also possible to begin replication of a new (empty) database or an existing database that already has content in it.

The steps to start replication are:

1. **Start the master database.**

   You can either create a new database or use an existing database as the master. When starting the database, you can use either of the standard startup arguments: **create** or **recover**. For example:

   ```
   $ voltdb create catalog.jar \
       -d deployment.xml \
       -H serverA \
       -l license.xml
   ```

   If any of the servers in the master database cluster have two or more network interface cards (and therefore multiple network addresses), you must explicitly identify which interface the server uses for both internal and external communication when you start VoltDB. For example:

   ```
   $ voltdb create catalog.jar \
       -d deployment.xml \
       -H serverA \
       -l license.xml \
       --externalinterface=10.11.169.10 \
       --internalinterface=10.12.171.14
   ```

If you do not specify which interface to use for multi-homed servers, replication will fail when the DR agent attempts to connect to those servers of the master database.

2. **Create a replica database.**

   You create a replica database just as you would any other VoltDB database, except instead of specifying **create** as the startup action, you specify **replica**. For example:

   ```
   $ voltdb create --replica catalog.jar \
       -d deployment.xml \
       -H serverB \
       -l license.xml
   ```

   Note that the replica database must:

   - Use the same version of the VoltDB server software.
   - Start with the same catalog as the master database.
   - Have the same configuration (that is, the same number of servers, sites per host, and K-safety value) as the master database.

   If these settings do not match, the DR agent will report an error and fail to start in the next step.

3. **Start the DR agent.**

   The DR agent is a separate process that can be run on any server that meets the hardware and software requirements for VoltDB. It is possible to run the agent on the same node as one of the master or replica cluster nodes. However, for best performance, it is recommended that the DR agent run on a separate, dedicated server located near the replica database.

   To start the DR agent, use the `dragent` command specifying the IP address or hostname of a node from the master database and a node from the replica database as arguments to the command. For example:

   ```
   $ dragent master serverA replica serverB
   ```

   If the master or replica use ports other than the default, you can specify which port the DR agent should use as part of the server name. For example, the following command tells the agent to connect to the master starting at port 6666 and the replica on port 23232:

   ```
   $ dragent master serverA:6666 replica serverB:23232
   ```

If you are using the Enterprise Manager to manage your databases, you can start the master database (Step 1) as you would normally, using the create, restore, or recover action. There is also a **replica** option on the Start Database dialog for creating a replica database (Step 2). The DR agent must be started by hand.

When the DR agent starts, it performs the following actions:

- Contacts both the master and replica databases.
- Verifies that the application catalogs match for the two databases.
- Verifies that the two clusters have the same number of unique partitions.
- Requests a snapshot from the master database. If data exists, the agent replays the snapshot on the replica.
- Begins to POLL and ACK the master database for completed transactions to be replayed on the replica.

12.2.2. Stopping Replication

If, for any reason, you wish to stop replication of a database, all you need to do is stop the DR agent process or the replica database. If either the agent or the replica database is not capable of processing the stream of transactions, the master will continue to queue completed transactions until the queue is full. At which point the master will abandon replication, delete the queue, and resume normal operation.

In other words, except for logging error messages explaining that replication has stopped, there is no outward change to the master cluster and no interruption of client activity. If you wish to shutdown replication in a more orderly fashion, you can:

1. Pause the master cluster, using the **voltadmin pause** command, to put the database in admin mode and stop client activity.

2. Once all transactions have passed through the DR agent to the replica (see Section 12.2.4.1, "Monitoring the Replication Process"), stop the DR agent process.

3. Stop the replica database, using **voltadmin shutdown** to perform an orderly shutdown.

4. Resume normal client operations on the master database, using **voltadmin resume**.

12.2.3. Promoting the Replica When the Master Becomes Unavailable

If the master database becomes unreachable for whatever reason (such as catastrophic system or network failure) and you choose to "turn on" the replica as a live database in its place, you use the **voltadmin promote** command to promote the replica to a fully active (writable) database. Specifically:

1. Stop the DR agent process. If not, the agent will report an error and stop after the following step.

2. Issue the **voltadmin promote** command on the replica database.

When you invoke **voltadmin promote**, the replica 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
```

12.2.4. Managing Database Replication

Database replication runs silently in the background, providing security against unexpected disruptions. Ideally, the replica will never be needed. But it is there just in case and the replication process is designed to withstand normal operational glitches. However, there are some conditions that can interrupt replication and it is important to be able to recognize and be able to respond to those situations, in order to ensure ongoing protection.

Both the master database and the DR agent maintain queues to handle fluctuations in the transmission of transactions. Network hiccups or a sudden increase of load on the master database can cause delays. Nodes on the master cluster may fail and rejoin (assuming K-safety). The queues help the replication process survive such interruptions.

In the case of the master database, replication initially queues data in memory. If the pending data exceeds the allocated queue size, data then overflows to disk in the directory `voltdbroot/dr_overflow`.

If the problem persists for too long, it is possible for the queues to fill up, resulting in either the master or the DR agent (or both) canceling replication. When this happens, it is necessary to restart the replication process. The following sections explain how to monitor the replication process and how to respond to error conditions.

12.2.4.1. Monitoring the Replication Process

There are two ways to monitor the replication process:

- The DR agent provides a stream of informational messages concerning its status as part of its logs (displayed on the console by default).
- You can query the master database about its current replication queue using the @Statistics system procedure and the "DR" component type.

The DR agent logs information about the ongoing transmissions with the master and the replica. It also reports any issues communicating with the master and continues to retry until communication is re-established. If the agent encounters a problem it cannot recover from, it logs the error and the process stops. In this situation, you must restart replication from the beginning. (See Section 12.2.4.2, “Restarting Replication if an Error Occurs” for details.)

If you do not want the log messages displayed on the console, you can redirect them by providing an alternate Log4J configuration file. You specify the alternate configuration file with the environment variable LOG4J_CONFIG_PATH. For example, the following commands start the DR agent and specify an alternate log configuration file mylogconfig.xml in the current working directory:

```
$ export LOG4J_CONFIG_PATH="mylogconfig.xml"
$ dragent master serverA replica serverB
```

In addition to the DR agent logs, you can query the master database to determine the current state of its replication queues using the @Statistics system procedure. The "DR" keyword returns information about the amount of replication data currently in memory (waiting to be sent to the agent). One VoltTable reports the amount of memory used for queuing transactions and another reports on the current status of any snapshots (if any) waiting to be sent.

12.2.4.2. Restarting Replication if an Error Occurs

If an error does occur that causes replication to fail, you must restart replication from the beginning. In other words:

1. Stop the DR agent process, if it is not already stopped.
2. Shutdown and restart the replica database.
3. If the master database is not running, restart it.
4. Restart the DR agent.

Note that, if the master is still running, it does not need to be stopped and restarted. However, both the DR agent and the replica database *must* be restarted if any condition causes replication to fail. Situations that will require restarting replication include the following:

- If the replica database stops.
- If the master database stops.
- If the DR agent stops.

- If a snapshot is restored to the master database. (Consequently, if restoring or recovering data when restarting the master database, be sure the restore completes on the master before beginning replication.)
- If communication between the master and the DR agent is delayed to the point where the master cluster's replication queues overflow.
- If any transaction replayed on the replica fails. Note that only successfully completed transactions are sent to the replica. So if a transaction fails, the replica is no longer in sync with the master.
- If any transaction replayed on the replica returns a different result than received on the master. The results are hashed and compared. Just as all replicated transactions must succeed, they must produce the same results or the two databases are out of sync.

12.3. Using the Sample Applications to Demonstrate Replication

One way to familiarize yourself with replication is to try it with an existing application. VoltDB comes with several sample applications. You can use any of the samples to test or demonstrate replication.

The following sections show how to create a replicated database, using the voter application as an example. The first section explains how to use the Enterprise Manager for the demonstration and the second uses the VoltDB shell commands. Both examples assume you have three servers:

- ServerA as the master
- ServerB as the replica
- ServerC as the agent

It is also possible to perform this demonstration on two nodes by using ServerB for both the replica and the DR agent.

12.3.1. Replicating the Voter Sample Using the Enterprise Manager

First, using the command line, run the voter sample once to create the application catalog. Then, using the Enterprise Manager:

1. Create two new databases, *Voter Master* and *Voter Replica*, using the voter application catalog for both of them.
2. Add ServerA to the *Voter Master* database.
3. Add ServerB to the *Voter Replica* database.
4. Start both databases, using the **create** action for *Voter Master* and **create** and **replica** for Voter Replica.

From the command line on ServerC, start the DR agent using the following command:

```
$ dragent master serverA replica serverB
```

Finally, from the command line on ServerA, run the sample client application:

```
$ cd examples/voter
```

```
$ ./run.sh client
```

You should see the client inserts on the master database replicated on the replica. Note that you can also start the client application before the DR agent, to show that replication can be started on an existing, active database.

12.3.2. Replicating the Voter Sample Using the Command Line

In the current release, the scripts for running the sample applications do not add the necessary command line arguments for starting a master or replica database by default. However, you can use the **voltdb** convenience command to solve this problem:

1. On both ServerA and ServerB, run the voter sample once to build the application catalog:

   ```
   $ cd examples/voter
   $ ./run.sh catalog
   ```

2. On ServerA, use the **voltdb** command to start the master database:

   ```
   $ voltdb create voter.jar \
      -d deployment.xml -H localhost \
      -l ../../voltdb/license.xml
   ```

3. On ServerB, use the **voltdb** command to start the replica database:

   ```
   $ voltdb create --replica voter.jar \
      -d deployment.xml -H localhost \
      -l ../../voltdb/license.xml
   ```

4. On ServerC, use the **dragent** command to start the DR agent:

   ```
   $ dragent master serverA replica serverB
   ```

5. On ServerB, start the voter client application:

   ```
   $ ./run.sh client
   ```

Note that you can also start the client application (step #5) before the DR agent (step #4), to show that replication can be started on an existing, active database.

Chapter 13. 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.

Exporting differs from save and restore (as described in Chapter 9, *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

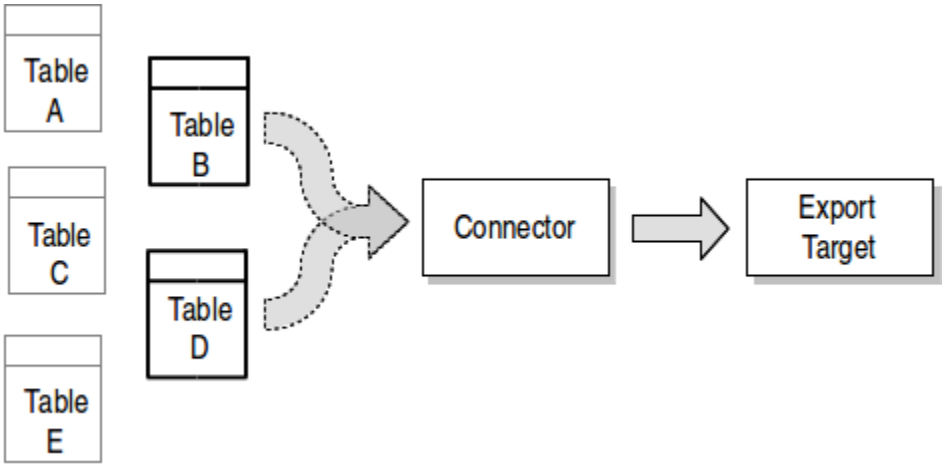
The target for exporting data from VoltDB may be another database, a repository (such as a sequential log file), or a process (such as a system monitor or accounting system). No matter what the target, VoltDB helps automate the process for you. This chapter explains how to plan for and implement the exporting of live data using VoltDB.

13.1. Understanding Export

VoltDB lets you automate the export process by specifying certain tables in the schema as sources for export. At runtime, any data written to the specified tables 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, 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 13.1, “Overview of the Export Process” illustrates the basic export procedure, where Tables B and D are specified as export tables.

Figure 13.1. 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 export-only tables; 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 an export-only table, 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-only tables have several important characteristics:

- Export-only tables let you limit the export to only the data that is required. For example, in the preceding example, Table 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 Table B for export to the remote database.
- Export-only tables let you combine fields from several existing tables into a single exported table. This technique is particularly useful if your VoltDB database and the target of the export have different schemas. The export-only table can act as a transformation of VoltDB data to a representation of the target schema.
- Export-only tables let you control *when* data is exported. Again, in the previous example, Table D might be an export-only table that is 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 Table 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-only tables. Since they have no storage associated with them, they are for INSERT only. Any attempt to SELECT, UPDATE, or DELETE export-only tables will result in an error when the project is compiled.

13.2. Planning your Export Strategy

The important point when planning to export data, is deciding:

- What data to export
- When to export the data

[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-only replicas of all tables in the schema and writing to the export-only table 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-only tables within VoltDB.

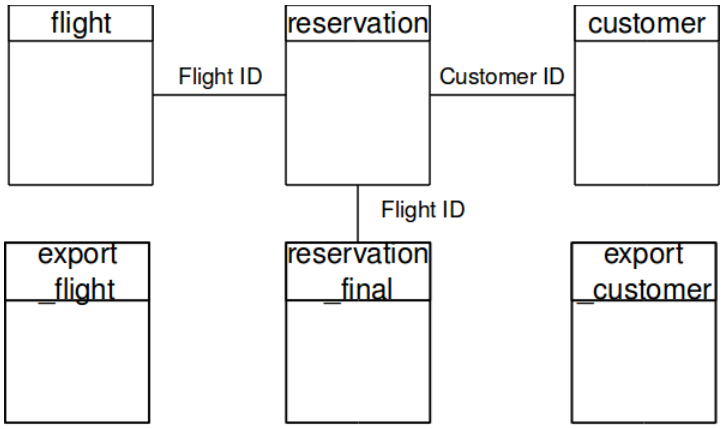
The second consideration is *when* to export the data. For tables that are not updated frequently, inserting the data to a complementary export-only table whenever data is inserted into the real table is the easiest and most practical approach. For tables that are updated frequently (hundreds or thousands of times a second) you should consider writing a copy of the data to an export-only table 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 3, *Designing Your VoltDB Application* 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 3.1, “Example Application Workload”, the customer and flight tables change infrequently. So data can be inserted into the export-only tables at the same time as the "live" flight and reservation tables. (It is a good idea to give the export-only copy of the table a meaningful name so its purpose is clear. In this example we identify the export-only tables with the export_ prefix or, in the case of the reservation table 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 export-only reservation table in real-time, a separate stored procedure is invoked when a flight takes off. This procedure copies the final reservation data to the export-only table and deletes the associated flight and reservation records from the VoltDB database. Figure 13.2, “Flight Schema with Export Table” shows the modified database schema with the added export-only tables, EXPORT_FLIGHT, EXPORT_CUSTOMER, and RESERVATION_FINAL.

Figure 13.2. Flight Schema with Export Table



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.

13.3. Identifying Export Tables in the Schema

Once you decide what data to export and define the appropriate tables in the schema, you are ready to identify them as export-only tables. As mentioned before, export-only tables are defined in the database schema just like any other table. So in the case of the flight application, we need to add the export tables to our schema. The following example illustrates (in bold) the addition of an export-only table 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 TABLE Reservation_final (
   ReserveID INTEGER UNIQUE NOT NULL,
   FlightID INTEGER NOT NULL,
   CustomerID INTEGER NOT NULL,
   Seat VARCHAR(5) DEFAULT NULL
);
     . . .
```

Again, it is a good idea to distinguish export-only tables by their table name, so anyone reading the schema understands their purpose. Once you add the necessary tables to the schema, you then need to define them as export-only tables. You do this by adding an EXPORT TABLE statement for each table to the schema tags. For example:

```
EXPORT TABLE export_customer;
EXPORT TABLE export_flight;
EXPORT TABLE reservation_final;
```

If a table is not listed in an EXPORT TABLE statement, it is not exported. In the preceding example, the *export_customer*, *export_flight*, and *reservation_final* tables are identified as the tables that will be included in the export. In addition, since they are export-only tables, inserting data into these tables will have no effect if export is disabled in the deployment file.

You can also specify whether the export-only tables are partitioned or not using the PARTITION TABLE statement in the schema. For example, if an export table is a copy of a normal data table, it can be partitioned on the same column. However, partitioning is not necessary for export-only tables. Whether they are partitioned or "replicated", since no storage is associated with the export table, you can INSERT into the table 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 stream.

13.4. Configuring Export in the Deployment File

To enable export at runtime, you include the `<export>` tag in the deployment file, specifying which export connector to use in with the `target` attribute. For example:

```
<export enabled="true" target="file">
   <configuration>
     . . .
   </configuration>
</export>
```

You must also configure the 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 enabled="true" target="file">
   <configuration>
     <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 five export connectors:

- Export to file
- Export to HTTP, including Hadoop
- Export to JDBC
- Export to Kafka
- Export to RabbitMQ

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.

13.5. The File Connector

The file connector receives the serialized data from the export tables and writes it out as text files (either comma or tab separated) to disk. The file connector writes the data out one file per database table, "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 catalog
- The table 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, Ohor 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 13.1, “File Export Properties” describes the supported properties for the file connector.

Table 13.1. 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.
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 skipinternals as "true", the output files contain only the exported table data.
with-schema	true, false	Specifies whether to write a JSON representation of each table'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 table, in the same order as they are listed in the database definition (DDL) file.

13.6. The HTTP Connector

The HTTP connector receives the serialized data from the export tables 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.

13.6.1. 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 13.2, “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 table. The table name is inserted into the endpoint in all uppercase.
%p	The VoltDB partition ID for the partition where the INSERT query to the export table 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.

Placeholder	Description
%g	The export generation. The generation is an identifier assigned by VoltDB. The generation increments each time the database starts or the application catalog 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 13.2, “HTTP Export Properties” describes the supported properties for the HTTP connector.

Table 13.2. HTTP Export Properties

Property	Allowable Values	Description
endpoint*	string	Specifies the target URL. The endpoint can contain placeholders for inserting the table 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.
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 table 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.
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 on-

Property	Allowable Values	Description
		ly (see Section 13.6.2, “Exporting to Hadoop via WebHDFS” for details.)

*Required

13.6.2. 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/1.0/"), 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 table 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 table data to WebHDFS using the HTTP connector and writing each table to a separate directory, with separate files based on the partition ID, generation, and period timestamp, rolling over every 2 hours:

```
<export enabled="true" target="http">
   <configuration>
     <property name="endpoint">
        http://myhadoopsvr/webhdfs/v1.0/%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 enabled="true" target="http">
   <configuration>
     <property name="endpoint">
       http://myhadoopsvr/webhdfs/v1.0/%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/v1.0/%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 table name and columns names are converted, removing underscores and changing the resulting words to lowercase with initial capital letters (sometimes called "camelcase"). The table name is given an initial capital letter, while columns names start with a lowercase letter. For example, the table 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.

13.7. The JDBC Connector

The JDBC connector receives the serialized data from the export tables 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-only tables 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 table names from the VoltDB schema or (if you do not enable the ignoregenerations property) the table name prefixed by the database generation ID.

If the target database has existing tables that match the VoltDB export-only tables 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 13.3, “JDBC Export Properties” describes the supported properties for the JDBC connector.

Table 13.3. 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 catalog 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 table data. The default is false.

*Required

13.8. The Kafka Connector

The Kafka connector receives serialized data from the export tables and writes it to a message queue using the Apache Kafka version 0.8 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 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 tables, it establishes a connection to the Kafka messaging service as a Kafka producer. It then writes records to the service using the VoltDB table name and a predetermined prefix as the Kafka "topic". How and when the data is transmitted to Kafka and the name of the topic prefix are controlled by the export connector properties.

The majority of the Kafka 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 `metadata.broker.list`, which lists the Kafka servers that the VoltDB export connector should connect to. You must specify this property so VoltDB knows where to send the export data.

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 `batch.mode` properties affect how and when the data is written to Kafka.

The `topic.prefix` property specifies the text that precedes the table name when constructing the Kafka topic. If you do not specify a prefix, it defaults to "voltdbexport". 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 table manually before starting the export process. Enabling auto-creation of topics when setting up the Kafka brokers is recommended.

The `batch.mode` property specifies whether messages are sent in batches, like the JDBC connector, or one message at a time. When configuring the export connector, it is important to understand the relationship between batch mode and synchronous versus asynchronous processing and their effect on database latency.

Using batch mode reduces the number of packets that must be sent to the Kafka servers, optimizing network bandwidth. If the export data is sent asynchronously, by setting the property `producer.type` to "async", the impact of export on the database is further reduced, since the export connector does not wait for the Kafka server 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 the Kafka server 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.

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.

To balance performance with durability of the exported data, the following are the two recommended configurations for producer type and batch mode:

- **Synchronous with batch mode** — Using synchronous mode ensures all packets are received by the Kafka system while batch mode reduces the possible latency impact by decreasing the number of packets that get sent.

```
<property name="producer.type">sync</property>
<property namr="batch.mode">true</property>
```

- **Asynchronous without batch mode** — Using asynchronous mode eliminates latency due to waiting for responses from the Kafka infrastructure while not using batch mode ensures that if a request fails, only one row of export data is affected, reducing the durability impact.

```
<property name="producer.type">async</property>
<property namr="batch.mode">false</property>
```

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 table.

Table 13.4, “Kafka Export Properties” lists the supported properties for the Kafka connector, including the standard Kafka producer properties and the VoltDB unique properties.

Table 13.4. Kafka Export Properties

Property	Allowable Values	Description
metadata.broker.list*	string	A comma-separated list of Kafka brokers.
batch.mode	true, false	Whether to submit multiple rows as a single request or send each export row separately. The default is true.
partition.key	{*table*}.{*column*} [,...]	Specifies which table column value to use as the Kafka partitioning key for each table. Kafka uses the partition key to distribute messages across multiple servers. By default, the value of the table's partitioning column is used as the Kafka partition key. Using this property you can specify a list of table column names, where the table name and column name are separated by a period and the list of table references is separated by commas. If the table 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 contains only the exported table 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.prefix	string	The prefix to use when constructing the topic name. Each row is sent to a topic identified by {prefix}{table-name}. The default prefix is "voltdbexport".
metadata.broker.list, request.required.acks, request.timeout.ms, producer.type, serializer.class, key.serializer.class, partitioner.class, compression.codec, compressed.topics,	various	Standard Kafka producer properties can be specified as properties to the VoltDB Kafka connector.

Property	Allowable Values	Description
message.send.max.retries, retry.backoff.ms, topic.metadata.refresh.interval.ms, queue.buffering.max.ms, queue.buffering.max.messages, queue.enqueue.timeout.ms, batch.num.messages, send.buffer.bytes, client.id		

*Required

13.9. The RabbitMQ Connector

The RabbitMQ connector fetches serialized data from the export tables 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 tables, 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 table. You can further refine the routing by appending a suffix to the table name, based on the contents of individual table columns. By default, the value of the export table's partitioning column is used as a suffix for the routing key. Alternately, you can specify a different column for each table by declaring the routing.key.suffix property as a list of table and column name pairs, separating the table from the column name with a period and separating the pairs with commas. For example:

```
<export enabled="true" target="rabbitmq">
   <configuration>
     <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 tables names. This can be achieved by using a filter for each export table. For example, using the flight example in Section 13.2, "Planning your Export Strategy", you can create filters for EXPORT_FLIGHT.*, EXPORT_CUSTOMER.*, and RESERVATION_FINAL.*.

Table 13.5, "RabbitMQ Export Properties" lists the supported properties for the RabbitMQ connector.

Table 13.5. 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	{*table*}.{*column*}[,...]	Specifies which table columns to use as a suffix for the RabbitMQ routing key. The routing key always starts with the table name, in uppercase. A suffix is then appended to the table name, separated by a period. By default, the value of the table's partitioning column is used as the suffix. Using this property you can specify a list of table column names, where the table name and column name are separated by a period and the list of table references is separated by commas. This syntax allows you to specify a different routing key suffix for each table.
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 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 skipinternals as true, the output contains only the exported table 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

13.10. How Export Works

Two important aspects of export to keep in mind are:

- Export is automatic. When you enable export 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.
- 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 table.

13.10.1. 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 6.1.2, “Configuring Paths for Runtime Features” for more information about configuring paths in the deployment file.

13.10.2. 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.

Chapter 14. Logging and Analyzing Activity in a VoltDB Database

VoltDB uses Log4J, an open source logging service available from the Apache Software Foundation, to provide access to information about database events. By default, when using the VoltDB shell commands, the console display is limited to warnings, errors, and messages concerning the status of the current process. A more complete listing of messages (of severity INFO and above) is written to log files in the subfolder `/log`, relative to the user's current default location.

The advantages of using Log4J are:

- Logging is compiled into the code and can be enabled and configured at run-time.
- Log4J provides flexibility in configuring what events are logged, where, and the format of the output.
- By using Log4J in your client applications, you can integrate the logging and analysis of both the database and the application into a single consistent output stream.
- By using an open source logging service with standardized output, there are a number of different applications, such as Chainsaw, available for filtering and presenting the results.

Logging is important because it can help you understand the performance characteristics of your application, check for abnormal events, and ensure that the application is working as expected.

Of course, any additional processing and I/O will have an incremental impact on the overall database performance. To counteract any negative impact, Log4J gives you the ability to customize the logging to support only those events and servers you are interested in. In addition, when logging is not enabled, there is no impact to VoltDB performance. With VoltDB, you can even change the logging profile on the fly without having to shutdown or restart the database.

The following sections describe how to enable and customize logging of VoltDB using Log4J. This chapter is **not** intended as a tutorial or complete documentation of the Log4J logging service. For general information about Log4J, see the Log4J web site at http://wiki.apache.org/logging-log4j/.

14.1. Introduction to Logging

Logging is the process of writing information about application events to a log file, console, or other destination. Log4J uses XML files to define the configuration of logging, including three key attributes:

- **Where** events are logged. The destinations are referred to as *appenders* in Log4J (because events are appended to the destinations in sequential order).
- **What** events are logged. VoltDB defines named classes of events (referred to as *loggers*) that can be enabled as well as the severity of the events to report.
- **How** the logging messages are formatted (known as the *layout*),

14.2. Creating the Logging Configuration File

VoltDB ships with a default Log4J configuration file, voltdb/log4j.xml, in the installation directory. The sample applications and the VoltDB shell commands use this file to configure logging and it is recommended for new application development. This default Log4J file lists all of the VoltDB-specific logging

categories and can be used as a template for any modifications you wish to make. Or you can create a new file from scratch.

The following is an example of a Log4J configuration file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

<appender name="Async" class="org.apache.log4j.AsyncAppender">
  <param name="Blocking" value="true" />
  <appender-ref ref="Console" />
  <appender-ref ref="File" />
</appender>

<appender name="Console" class="org.apache.log4j.ConsoleAppender">
  <param name="Target" value="System.out" />
  <layout class="org.apache.log4j.TTCCLayout" />
</appender>

<appender name="File" class="org.apache.log4j.FileAppender">
  <param name="File" value="/tmp/voltdb.log" />
  <param name="Append" value="true" />
  <layout class="org.apache.log4j.TTCCLayout" />
</appender>

<logger name="AUTH">
<!-- Print all VoltDB authentication messages -->
  <level value="trace" />
</logger>

<root>
      <priority value="debug" />
      <appender-ref ref="Async" />
</root>
</log4j:configuration>
```

The preceding configuration file defines three destinations, or appenders, called *Async*, *Console*, and *File*. The appenders define the type of output (whether to the console, to a file, or somewhere else), the location (such as the file name), as well as the layout of the messages sent to the appender. See the log4J documentation for more information about layout.

Note that the appender Async is a superset of Console and File. So any messages sent to Async are routed to both Console and File. This is important because for logging of VoltDB, you should always use an asynchronous appender as the primary target to avoid the processing of the logging messages from blocking other execution threads.

More importantly, you should not use any appenders that are susceptible to extended delays, blockages, or slow throughput, This is particularly true for network-based appenders such as SocketAppender and third-party log infrastructures including logstash and JMS. If there is any prolonged delay in writing to the appenders, messages can end up being held in memory causing performance degradation and, ultimately, possible out of memory errors.

The configuration file also defines a root class. The root class is the default logger and all loggers inherit the root definition. So, in this case, any messages of severity "debug" or higher are sent to the Async appender.

Finally, the configuration file defines a logger specifically for VoltDB authentication messages. The logger identifies the class of messages to log (in this case "AUTH"), as well as the severity ("trace"). VoltDB defines several different classes of messages you can log. Table 14.1, “VoltDB Components for Logging” lists the loggers you can invoke.

Table 14.1. VoltDB Components for Logging

Logger	Description
ADHOC	Execution of ad hoc queries
AUTH	Authentication and authorization of clients
COMPILER	Interpretation of SQL in ad hoc queries
CONSOLE	Informational messages intended for display on the console
EXPORT	Exporting data
GC	Java garbage collection
HOST	Host specific events
NETWORK	Network events related to the database cluster
REJOIN	Node recovery and rejoin
SNAPSHOT	Snapshot activity
SQL	Execution of SQL statements
TM	Transaction management

.

14.3. Enabling Logging for VoltDB

Once you create your Log4J configuration file, you specify which configuration file to use by defining the variable LOG4J_CONFIG_PATH before starting the VoltDB database. For example:

```
$ LOG4J_CONFIG_PATH="$HOME/MyLog4jConfig.xml"
$ voltdb create mycatalog.jar \
     -H localhost -d mydeployment.xml
```

14.4. Customizing Logging in the VoltDB Enterprise Manager

When using the VoltDB Enterprise Manager to manage your databases, the startup process is automated for you. There is no command line for specifying a Log4J configuration file.

Instead, the Enterprise Manager provides a Log4J properties file that is used to start each node in the cluster. You can change the logging configuration by modifying the properties file `server_log4j.properties` included in the `/management` subfolder of the VoltDB installation. The Enterprise Manager copies and uses this file to enable logging on all servers when it starts the database.

Note that the properties file used by the VoltDB Enterprise Manager is in a different format than the XML file used when configuring Log4J on the command line. However, both files let you configure the same logging attributes. In the case of the properties file, be sure to add your modifications to the end of the file so as not to interfere with the logging required by the Enterprise Manager itself.

14.5. Changing the Timezone of Log Messages

By default all VoltDB logging is reported in GMT (Greenwich Mean Time). If you want the logging to be reported using a different timezone, you can use extensions to the Log4J service to achieve this.

To change the timezone of log messages:

1. Download the extras kit from the Apache Extras for Apache Log4J website, http://logging.apache.org/log4j/extras/.

2. Unpack the kit and place the included JAR file in the `/lib/extension` folder of the VoltDB installation directory.

3. Update your Log4J configuration file to enable the Log4J extras and specify the desired timezone for logging for each appender.

You enable the Log4J extras by specifying `EnhancedPatternLayout` as the layout class for the appenders you wish to change. You then identify the desired timezone as part of the layout pattern. For example, the following XML fragment changes the timezone of messages written to the file appender to GMT minus four hours:

```
<appender name="file" class="org.apache.log4j.DailyRollingFileAppender">
    <param name="file" value="log/volt.log"/>
    <param name="DatePattern" value="'.'yyyy-MM-dd" />
    <layout class="org.apache.log4j.EnhancedPatternLayout">
        <param name="ConversionPattern" value="%d{ISO8601}{GMT-4}   %-5p [%t]
    </layout>
</appender>
```

You can use any valid ISO-8601 timezone specification, including named timezones, such as EST.

To customize the timezone when using the VoltDB Enterprise Manager, you must specify the layout class and pattern using the Log4J properties file in the `/management` folder. For example, the following code appended to the file `server_log4j.properties` would make the same change to the timezone for clusters started with the Enterprise Manager as the preceding example would when starting a server manually:

```
# Add your own log4j configurations below this line

log4j.appender.file.layout=org.apache.log4j.EnhancedPatternLayout
log4j.appender.file.layout.ConversionPattern=%d{ISO8601}{GMT-4}   %-5p [%t] %c: %m
```

14.6. Changing the Configuration on the Fly

Once the database has started, you can still start or reconfigure the logging without having to stop and restart the database. By calling the system procedure @UpdateLogging you can pass the configuration XML to the servers as a text string. For any appenders defined in the new updated configuration, the existing appender is removed and the new configuration applied. Other existing appenders (those not mentioned in the updated configuration XML) remain unchanged.

Chapter 15. 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
- ODBC
- 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.

15.1. 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++.

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

```
#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 <vector>
```

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.

15.1.2. 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.

15.1.3. 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 */
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 */
boost::shared_ptr<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);
```

15.1.4. 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
         (boost::shared_ptr<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()) {}
```

15.1.5. 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())
{
    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 */
vector<boost::shared_ptr<voltdb::Table> > results = response->results();
voltdb::TableIterator iterator = results[0]->iterator();

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

15.2. 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`.

15.2.1. 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 a SHA-1 encoded string. (The hashed password must be a 40-byte hex-encoding of the 20-byte SHA-1 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.

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`). If the database is in admin mode, you must submit requests over the admin port or else the request is rejected by the server.

The admin port should be used for administrative tasks only. Although all stored procedures can be invoked through the admin port, using the admin port through JSON is far less efficient than using the client port. All admin mode requests to JSON are separate synchronous requests; whereas calls to the normal client port are asynchronous through a shared session.

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.

15.2.2. 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();

        }
      }
    }
  }
}
```

15.2.3. 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 15.1, “Datatypes in the JSON Interface” summarizes how to pass different datatypes in the JSON interface.

Table 15.1. Datatypes in the JSON Interface

Datatype	How to Pass	Example
Integers (Byte, Short, Integer, Long)	An integer value	12345
DOUBLE	A value with a decimal point	123.45

Datatype	How to Pass	Example
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"

15.2.4. 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 15.1, “The Structure of the VoltDB JSON Response” shows the structure of the object returned by the JSON interface.

Figure 15.1. 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 — Indicates the success or failure of the stored procedure. If *appstatus* is false, *appstatusstring* contains the text of the status message.

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 VoltDB server in its attempt to execute the stored procedure. The difference between *appstatus* and *status* is that if the server cannot execute the stored procedure, the status is returned in *status*, whereas if the stored procedure can be invoked, but a failure occurs within the stored procedure itself (such as a SQL constraint violation), the status is returned in *appstatus*.

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]
```

15.2.5. 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.)

15.3. 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.

15.3.1. 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");
```

15.3.2. 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();
        }
    }
}
```

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 in a VoltDB schema definition and any exceptions, extensions, or limitations that application developers should be aware of.

The supported standard SQL DDL statements are:

- CREATE INDEX
- CREATE TABLE
- CREATE VIEW

The supported VoltDB-specific extensions for declaring stored procedures and partitioning are:

- CREATE PROCEDURE AS
- CREATE PROCEDURE FROM CLASS
- CREATE ROLE
- EXPORT TABLE
- IMPORT CLASS
- PARTITION PROCEDURE
- PARTITION TABLE

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 [,...])
```

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) );
```

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
);
```

CREATE PROCEDURE AS

CREATE PROCEDURE AS — Defines a stored procedure composed of a SQL query.

Syntax

```
CREATE PROCEDURE procedure-name [ALLOW role-name [,...]] AS sql-statement

CREATE PROCEDURE procedure-name [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. The declared procedures are evaluated and included in the application catalog when you compile the database schema.

Use CREATE PROCEDURE AS when declaring stored procedures directly within the schema definition. 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.

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.

If security is enabled at runtime, only those roles named in the ALLOW clause 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 *admin* role:

```
CREATE PROCEDURE ChangeAdminPassword ALLOW admin AS
    UPDATE Accounts SET (HashedPassword=?) WHERE userID='root';
```

CREATE PROCEDURE FROM CLASS

CREATE PROCEDURE FROM CLASS — Defines a stored procedure associated with a Java class.

Syntax

```
CREATE PROCEDURE [ALLOW role-name [,...]] FROM CLASS class-name
```

Description

You must declare stored procedures to make them accessible at runtime. The declared procedures are evaluated and included in the application catalog when you compile the database schema.

If security is enabled at runtime, only those roles named in the ALLOW clause 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.

Use CREATE PROCEDURE FROM CLASS when adding user-defined stored procedures written in Java. The *class-name* is the name of the Java class. This class must be accessible from the classpath argument used when compiling the application catalog.

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 catalog 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 permissions that can be assigned by the WITH clause are:

ADHOC	Allows access to ad hoc queries (through the @AdHoc system procedure and **sqlcmd** command)
DEFAULTPROC	Allows access to the default procedures for all tables
SYSPROC	Allows access to all system procedures

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 adhoc;
```

Also note that the permissions are additive. So if a user is assigned to one role that allows access to adhoc but not sysproc, but that user also is assigned to another role that allows sysproc, 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 sysproc, defaultproc;
CREATE ROLE developer WITH adhoc, defaultproc;
CREATE ROLE batch WITH defaultproc;
```

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

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
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).

SQL Datatype	Equivalent Java Datatype	Description
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, updating the application catalog, 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 that 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.

- 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. However, VARBINARY columns cannot be used in indexes or in conditional comparisons (such as in SELECT ... WHERE statements).

- 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, used to optimize access to specific columns within a table.

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 obey the following constraints:
  - There must be a GROUP BY clause in the SELECT statement.
  - All of the columns and selection expressions listed in the GROUP BY must be listed in the same order at the start of the SELECT statement.
  - SELECT must include a field specified as COUNT(*). Other aggregate functions (COUNT, MAX, MIN, and SUM) are allowed following the COUNT(*).

Example

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;
```

EXPORT TABLE

EXPORT TABLE — Specifies that a table is for export only.

Syntax

```
EXPORT TABLE table-name
```

Description

At runtime, any records written to an export-only table are queued to the export connector, as described in Chapter 13, *Exporting Live Data*. If export is enabled, this data is then passed to the export connector that manages the export process.

The EXPORT TABLE statement lets you specify which tables in the schema are export-only tables. These tables become write-only. That is, they can be used in INSERT statements, but not SELECT, UPDATE, or DELETE statements.

If export is not enabled at runtime, writing to export-only tables has no effect.

Example

The following example defines two tables — *User* and *User_Export* — with similar columns. The second table is then defined as an export table. By inserting into the *User_Export* table every time a row is inserted into the *User* table, an automated list of users can be maintained external to the active VoltDB database.

```
CREATE TABLE User (
   UserID VARCHAR(15) NOT NULL,
   EmailAddress VARCHAR(128) NOT NULL,
   Created TIMESTAMP,
   Password VARCHAR(14),
   LastLogin TIMESTAMP);

CREATE TABLE User_Export (
   UserID BIGINT NOT NULL,
   EmailAddress VARCHAR(128) NOT NULL,
   Created TIMESTAMP);

EXPORT TABLE User_Export;
```

IMPORT CLASS

IMPORT CLASS — Specifies additional Java classes to include in the application catalog.

Syntax

```
IMPORT CLASS class-name
```

Description

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 3.2.2, “VoltDB Stored Procedures and Determinism”.

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 PARTITION 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;
```

Appendix B. Supported SQL Statements

This appendix describes the SQL syntax that VoltDB supports in stored procedures.

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]...]
```

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.

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 BIDPRICE 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 the 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 3.2.2, “VoltDB Stored Procedures and Determinism” 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, 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.

- 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 = "Salery";
  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.

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.

Column Aggregation Functions

- AVG()
- COUNT()
- MAX()
- MIN()
- SUM()

Date and Time Functions

- CURRENT_TIMESTAMP
- DAY(), DAYOFMONTH()
- DAYOFWEEK()
- DAYOFYEAR()
- EXTRACT()
- FROM_UNIXTIME()
- HOUR()
- MINUTE()
- MONTH()
- NOW
- QUARTER()
- SECOND()
- SINCE_EPOCH()
- TO_TIMESTAMP()
- TRUNCATE()
- WEEK(), WEEKOFYEAR()
- WEEKDAY()
- YEAR()

JSON Functions

- ARRAY_ELEMENT()
- ARRAY_LENGTH()
- FIELD()
- SET_FIELD()

Logic and Conversion Functions

- CAST()
- COALESCE()
- DECODE()

Math Function

- ABS()

- CEILING()
- EXP()
- FLOOR()
- POWER()
- SQRT()

String Functions

- CHAR()
- CHAR_LENGTH()
- CONCAT()
- FORMAT_CURRENCY()
- LEFT()
- LOWER()
- OCTET_LENGTH()
- OVERLAY()
- 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;
```

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=?;
```

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;
```

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;
```

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;
```

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.

Example

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;
```

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  MONTH(starttime) || '/' ||
        DAY(starttime) || '/' ||
        YEAR(starttime), 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)
           || ' 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;
```

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 (when, what, where) VALUES (FROM_UNIX_TIME(?),?,?);
```

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;
```

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;
```

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;
```

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  MONTH(starttime) || '/' ||
        DAY(starttime) || '/' ||
        YEAR(starttime), 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;
```

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;
```

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);
```

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_type,
    SINCE_EPOCH(Microsecond, endtime)
    -SINCE_EPOCH(Microsecond, starttime) AS delta
    FROM Event GROUP BY event_type;
```

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* [TO *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 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 ?) );
```

Example

The following example uses TRIM() to remove extraneous leading and trailing spaces from the input for three VARCHAR columns:

```
INSERT INTO Customers (first, last, address)
      VALUES( TRIM(?), TRIM(?), TRIM(?) );
```

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)
```

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  MONTH(starttime) || '/' ||
        DAY(starttime) || '/' ||
        YEAR(starttime), 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
- dragent
- 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.

--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.

--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 catalog 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.

--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
```

dragent

dragent — Starts the database replication agent.

Syntax

dragent master *server-id*[:*port-num*] replica *server-id*[:*port-num*] [statsinterval *seconds*] [username *username-string* password *password-string*]

Description

The dragent command starts the database replication agent and begins replicating the master database to the replica. See Chapter 12, *Database Replication* for more information about the database replication process.

Arguments

master *server-id[:port-num]*

Specifies the network address of one node from the master database cluster. The server-id can be an IP address or hostname. The port number to connect to is optional. By default, the replication agent uses three ports to connect to the master database server, starting with the default replication port (5555). If a different replication port was specified when the database server was started, you must specify that port number when starting the DR agent.

replica *server-id[:port-num]*

Specifies the network address of one node from the replica database cluster. The server-id can be an IP address or hostname. The port number to connect to is optional. By default, the replication agent uses the standard client port.

If security is enabled for the replica database, you must also specify a username and password as additional arguments. For example, the following command connects to the replica database *antarctic* using the username *penguin* and password *wheretheylive*:

```
$ dragent master arctic replica antarctic \
          username penguin password wheretheylive
```

statsinterval *seconds*

Specifies the frequency with which the agent reports statistics concerning the replication throughput. These statistics are useful in determining if replication is keeping up with the throughput from the master database.

Example

The following example starts database replication between the master database cluster that includes the node zeus and the replica database cluster that includes the node apollo. The replication agent uses the admin port to connect to apollo.

```
$ dragent master zeus replica apollo:21211
```

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 VoltDB database. You must specify a username and password if security is enabled on the target database.

--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 catalog 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.

--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.

--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.

--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 commands 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** command, followed by the procedure class name, the procedure parameters, and a closing semi-colon. For example, the following sqlcmd command 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 or double quotation marks. Also, the exec command must be terminated by a semi-colon.
- **Show and Explain commands** — The **show** and **explain** commands let you examine the structure of the schema and user-defined stored procedures. Valid commands are:
  - SHOW CLASSES — Lists the user-defined classes in the catalog. 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.
- **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** command. The **file** command takes a text file as an argument and executes all of the SQL queries and **exec** commands in the file as if they were entered interactively. Any **show**, **explain**, **recall**, or **exit** commands 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;
  ```

- **Exit** — When you are done with your interactive session, enter the **exit** command 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** commands. Also, **list** and **quit** are supported as synonyms for the **show** and **exit** commands, 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=*password-string*
: Specifies the password to use for authenticating to the database. The password is required if the database has security enabled.

--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.

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 [args...] {command}
```

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=*password*
: Specifies the password to use for authenticating to the database. The password is required if the database has security enabled..

-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.

pause
: Pauses the database, stopping any additional activity on the client port.

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.

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.

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 catalog and deployment configuration on a running database. There are some limitations on what changes can be made on a live update. For example, you cannot rename a table or change its partitioning column. See the description of the @UpdateApplicationCatalog stored procedure for details.

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 compiling the application catalog and starting the database.

Syntax

voltdb collect [args] *voltdbroot-directory*

voltdb compile [args] [*DDL-file* ...]

voltdb create [args] *application-catalog*

voltdb recover [args]

voltdb add [args]

voltdb rejoin [args]

Description

The voltdb command performs local management functions on the current system, including:

- Compiling schema files and stored procedures into an application catalog
- Starting the database process
- Collecting log files into a single compressed 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.
- **compile** — the compile option compiles the database schema and stored procedures into an application catalog. You can specify one or more data definition language (DDL) files that describe the schema of the database, the stored procedures, and the partitioning columns. See Appendix A, *Supported SQL DDL Statements* for the SQL statements supported in the DDL files. The output of the compile action is an application catalog that can be used to start the VoltDB database. The default output filename is `catalog.jar`. However, you can use the `--output` argument to specify a different file name or location. See the next section for other arguments to the **compile** action.
- **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 catalog by performing a save, shutdown, startup, and restore. (See Chapter 7, *Updating Your VoltDB Database* for information on updating your application catalog.)
- **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 7.4, “Updating the Hardware 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 starting a new database 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.

Schema Compilation Arguments

The following arguments apply specifically to the **compile** action.

-c, --classpath=*{Java-classpath}*
Specifies additional classpath locations for the compilation process to search when looking for stored procedure class files. The classpath you specify with this argument is appended to any existing classpath definition.

-o, --output=*{application-catalog}*
Specifies the file and path name to use for the application catalog that is created as a result of the compilation.

Database Startup Arguments

The following arguments apply to the **add**, **create**, **recover**, and **rejoin** start actions.

{application-catalog}
Specifies the application catalog containing the schema and stored procedures to load when starting the database. Two special notes concerning the catalog:

- The catalog must be identical on all nodes when starting a cluster.
- The catalog specified on the command line is only used when creating a new database.

If you recover previous data using the **recover** start action, the catalog saved with the snapshot or command log is loaded and any catalog you specify on the command line is ignored.

-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 two sites per host.

-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).

--blocking
For the rejoin operation only, specifies that the database should block client transactions for the affected partitions until the rejoin is complete.

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.

--internal=*[ip-address:]{port-number}*
Specifies the internal port used to communicate between cluster nodes.

--client=*[ip-address:]{port-number}*
Specifies the client port.

--admin=*[ip-address:]{port-number}*
Specifies the admin port. The --admin flag overrides the admin port setting in the deployment file.

--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.

--replication=*[ip-address:]{port-number}*
Specifies the first of three replication ports 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 uses the compile action to create an application catalog from two DDL files. The `--classpath` argument specifies the location of the stored procedure class files.

```
$ voltdb compile --classpath=./obj employees.sql company.sql
```

The next example shows the command for creating a database running the voter sample application, using a custom configuration file, `2nodedeploy.xml`, and the node zeus as the host.

```
$ voltdb create voter.jar --deployment=2nodedeploy.xml \
        --host=zeus
```

The following example takes advantage of the defaults for the host and deployment arguments to start a single-node database on the current system using the voter catalog.

```
$ voltdb create voter.jar
```

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/>
    <export>
        <configuration>
            <property/>...
        </configuration>
    </export>
    <heartbeat/>
    <httpd>
        <jsonapi/>
    </httpd>
    <partition-detection>
        <snapshot/>
    </partition-detection>
    <replication/>
    <security/>
    <snapshot/>
    <systemsettings>
        <elastic/>
        <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, partition-detection, paths, security, snapshot, systemsettings, users	
cluster*	deployment		hostcount={int}* sitesperhost={int}

Element	Child of	Parent of	Attributes
			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}
export	deployment	configuration	enabled={true|false} target={file|jdbc|custom} exportconnectorclass={class-name}
configuration*	export	property	
property	configuration		name={text}*
httpd	deployment	jsonapi	port={int} enabled={true|false}
jsonapi	httpd		enabled={true|false}
paths	deployment	exportoverflow, snapshots, voltdbroot	
commandlog	paths		path={directory-path}*
commandlogsnapshot	paths		path={directory-path}*
exportoverflow	paths		path={directory-path}*
snapshots	paths		path={directory-path}*
voltdbroot	paths		path={directory-path}*
replication	deployment		port={int}
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, snapshot, temptables	
elastic	systemsettings		duration={int} throughput={int}
query	systemsettings		timeout={int}*
snapshot	systemsettings		priority={int}*
temptables	systemsettings		maxsize={int}*
users	deployment	user	
user	users		name={text}*

Element	Child of	Parent of	Attributes
			password={text}* roles={role-name[,..]}[1]

*Required

[1]The attribute "groups" can be used in place of "roles" for backwards compatibility.

Appendix F. VoltDB Datatype Compatibility

VoltDB supports nine 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:

- Queuing SQL statements using the voltdbQueueSql method
- 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. Another important distinction 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 Int, 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.
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.
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
- @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 queries on a running VoltDB database.

You can execute multiple SQL queries in a single call to @AdHoc by separating the individual queries with semicolons. When you do this, the queries are performed as a single transaction. That is, the queries all succeed as a group or they all roll back if any of them fail.

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 query, with as many rows as there are records returned by the query. 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 query 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 Performance Guide* 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 Performance Guide* 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 admin mode on the cluster.

Syntax

```
@Pause
```

Description

The @Pause system procedure initiates admin mode on the cluster. In admin mode, no further transaction requests are accepted from clients on the client port. All interactions with a database in admin mode must occur through the admin port specified in the deployment file.

There may be existing transactions still in the queue after admin mode is initiated. Until these transactions are completed, the database is not entirely paused. You can use the @Statistics system procedure with the "LIVECLIENTS" keyword to determine how many transactions are outstanding for each client connection.

The goal of admin mode is to pause the system and ensure no further changes to the database can occur when performing sensitive administrative operations, such as taking a snapshot before shutting down.

Several important points to consider concerning @Pause are:

- @Pause must be called through the admin port, not the standard client port.
- Although new stored procedure invocations received 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 database replication (DR) agent. 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 new connections 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 10, *Command Logging and Recovery* and Chapter 9, *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-paths, 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 @SnashotRestore 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 9, *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 9, *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*.

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.

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*:

"CPU"	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.
"DR"	Returns information about the status of database replication, including how much data is waiting to be sent to the DR agent. This information is available only if the database is licensed for database replication.
"INDEX"	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"	Returns information on the number of procedure invocations for each stored procedure (including system procedures). The count of invocations is reported for each connection to the database.
"IOSTATS"	Returns information on the number of messages and amount of data (in bytes) sent to and from each connection to the database.
"LIVECLIENTS"	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, INITIATOR, IOSTATS, MEMORY, PROCEDURE, and TABLE, except all in a single procedure call.

"MEMORY"	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"	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"	Returns information on the use of cached plans within each partition. Queries in stored procedures are planned when the application catalog is compiled. 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"	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"	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"	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"	Returns summary information on the usage of stored procedures averaged across all partitions in the cluster. The information from PROCEDURE-PROFILE 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"	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" 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" 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.)

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.

DR — 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 DR agent.

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 DR agent.
TOTALBYTESIN MEMORY	BIGINT	The total number of bytes of queued data currently held in memory. If the amount of total bytes is larger than the

Name	Datatype	Description
		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 DR agent. Partitions create a buffer every five milliseconds.
LASTACKTIMESTAMP	BIGINT	The timestamp of the last acknowledgement received from the DR agent.
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 DR agent ("NORMAL") or not ("PAUSED"). Only one copy of each logical partition actually sends data to the DR agent 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 DR agent.

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.
ENABLED	STRING	A text string indicating whether the database is currently being replicated. Possible values are "true" and "false".
SYNCSNAPSHOTSTATE	STRING	A text string indicating the current state of the synchronization snapshot that begins replication. During normal operation, this value is "NOT_SYNCING" 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 infomation about the specific activity underway.
ROWSINSYNC SNAPSHOT	BIGINT	Reserved for future use.
ROWSACKEDFORSYNC SNAPSHOT	BIGINT	Reserved for future use.

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.

Name	Datatype	Description
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 followiing: 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	INTEGER	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.
PROCEDURE_NAME	STRING	The name of the stored procedure.
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.

Name	Datatype	Description
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_HOSTNAME	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	INTEGER	The amount of memory (in kilobytes) currently in use for storing database records.
TUPLEALLOCATED	INTEGER	The amount of memory (in kilobytes) allocated for the storage of database records (including free space).
INDEXMEMORY	INTEGER	The amount of memory (in kilobytes) currently in use for storing database indexes.
STRINGMEMORY	INTEGER	The amount of memory (in kilobytes) currently in use for storing string and binary 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.)
PHYSICALMEMORY	BIGINT	The total size of physical memory (in kilobytes) on the server.

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).

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.
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.

Name	Datatype	Description
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).
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).

Name	Datatype	Description
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.
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.

Name	Datatype	Description
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).
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").

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.

Name	Datatype	Description
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	INTEGER	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	INTEGER	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.
STRING_DATA_MEMORY	INTEGER	The total memory, in kilobytes, used for storing non-inline variable length data (VARCHAR and VARBINARY) 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 application catalog, 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 catalog:

```
$ 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 catalog and deployment files 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 make modifications to a running database without having to shutdown and restart. @UpdateApplicationCatalog supports the following changes:

- 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 for the database
- Modify the settings for automated snapshots (whether they are enabled or not, their frequency, location, prefix, and number retained)

When modifying indexes, you can add, remove, or rename non-unique indexes, you can add or remove columns from a non-unique index, and you can rename, add columns to, or remove in its entirety a unique index. The only limitations are that you *cannot* add a unique index or remove a column from an existing unique index.

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(); }
```

@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();
}
```