Using VoltDB

Abstract

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

V2.2.2

Using VoltDB

V2.2.2

This document was generated on March 09, 2012.

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 Application*
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, *Exporting Live Data* • Chapter 13, *Logging and Analyzing Activity in a VoltDB Database* • Chapter 14, *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, *Project Definition File (project.xml)* • Appendix D, *Configuration File (deployment.xml)* • Appendix E, *System Procedures*

This book provides the most complete description of the VoltDB product. A companion book, *Getting Started with VoltDB*, provides a quick introduction to the product and is recommended for new users. Both 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

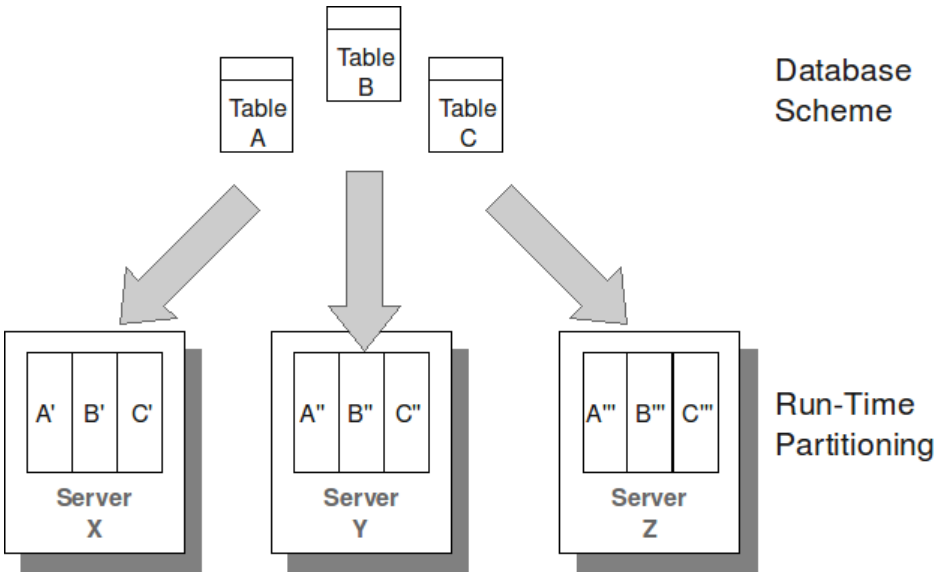
VoltDB is not like traditional database products. There is no such thing as a generic VoltDB "database". Each database is optimized for a specific application by compiling the schema, stored procedures, and partitioning information in to what is known as the VoltDB runtime *catalog*. The catalog is then 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



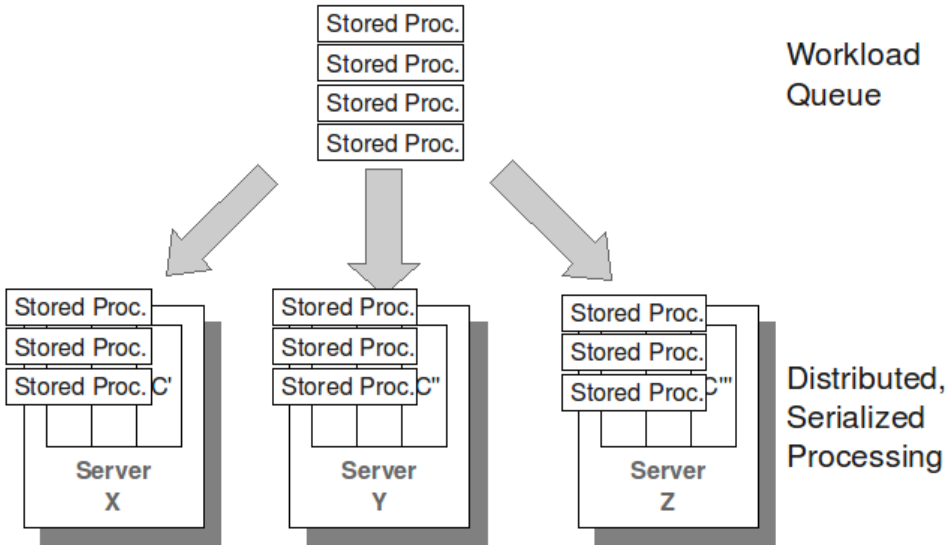
1.3.2. Serialized (Single-Threaded) Processing

At run-time, calls to the stored procedures are passed to the appropriate node of the cluster. When procedures are "single-sited" (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



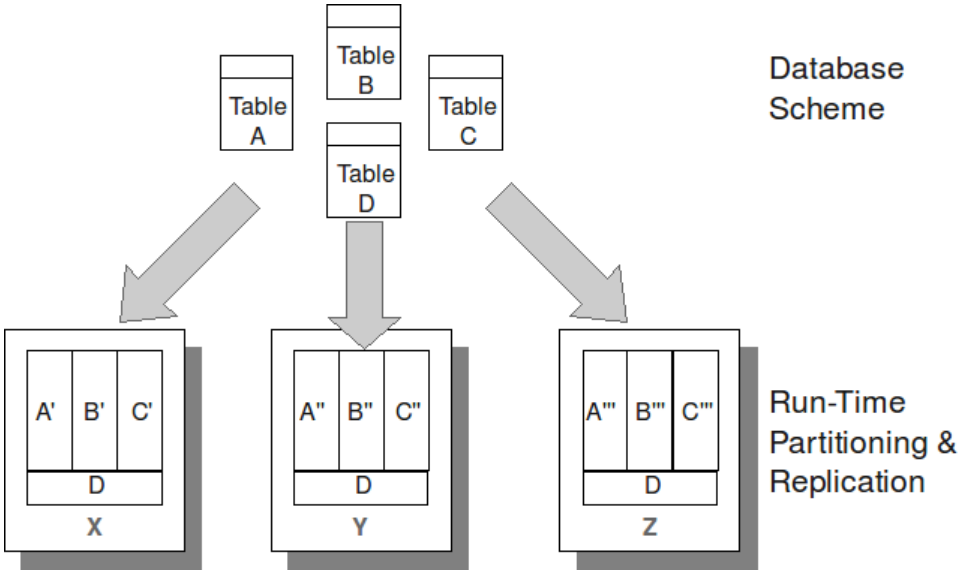
It is important to note that the VoltDB architecture is optimized for throughput over latency. The latency of any one transaction (the time from when the transaction begins until processing ends) is similar in VoltDB to other databases. However, the number of transactions that can be completed in a second (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-sited 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 doesn't require any changes to the database schema or application code. Nor does it require replacing existing hardware. With VoltDB, scaling up is a simple process:

1. Save the running database to disk using the @SnapshotSave system procedure.
2. Shut down the database.
3. Distribute a new copy of the VoltDB runtime catalog to all nodes (so they are aware of the changes to hardware configuration).
4. Restart the database (including the new nodes).
5. Restore and redistribute the partitions using the @SnapshotRestore system procedure.

Future versions of VoltDB are expected to support scaling (by adding nodes to the cluster) "on the fly", without having to save and restore. See Chapter 9, *Saving & Restoring a VoltDB Database* for details about using the Save and Restore procedures to reconfigure your database cluster.

Chapter 2. Installing VoltDB

VoltDB is available as both pre-built distributions and as source code. This chapter explains the system requirements, how to install VoltDB, 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 CentOS version 5.6 and Ubuntu versions 10.4 and 10.10. Development builds are also available for Macintosh OSX 10.6[1].
CPU	• Dual core[2] x86_64 processor • 64 bit • 1.6 GHz
Memory	4 Gbytes[3]
Java	Sun JDK 6 update 20 or later
Required Software	NTP[4]
Recommended Software	Ant 1.7 or later Eclipse 3.x (or other Java IDE)

Footnotes:

1. CentOS 5.6 and later and Ubuntu 10.4 and later 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 OSX 10.6.
2. Dual core processors are a minimum requirement. Four or eight physical cores are recommended for optimal performance.
3. Memory requirements are very specific to the storage needs of the application and the number of nodes in the cluster. However, 4 Gigabytes should be considered a minimum configuration.
4. 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 best way to install VoltDB is to unpack the distribution kit as a folder in the home directory of your personal account, like so:

```
$ tar -zxvf voltdb-2.2.2.tar.gz -C $HOME/
```

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

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

```
$ sudo tar -zxvf voltdb-2.2.2.tar.gz -C /opt
$ cd /opt
$ sudo mv voltdb-2.2.2 voltdb
```

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

2.2.1. Upgrading an Existing VoltDB Installation

If you are upgrading an existing installation of VoltDB, you have two choices:

- You can unpack the new version as a separate installation. VoltDB does this by default, since the tar file contains the version number in the folder name. Note that if you do install new versions alongside an existing installation, any existing Ant build files or shell scripts you have for building and running VoltDB applications will continue to use the older version.
- You can replace your existing installation with the new version. To do this, you need to delete the folder with your current installation and then follow the instructions for unpacking the new kit. For example, the following shell commands unpack the new version under the user's home directory, delete an old installation, and replace it:

  ```
  $ tar -zxvf voltdb-2.2.2.tar.gz -C $HOME
  $ cd $HOME
  $ rm -R voltdb/
  $ mv voltdb-2.2.2 voltdb
  ```

2.2.2. Building a New VoltDB Distribution Kit

If you want to build the 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 accessible from the VoltDB community web site.

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 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. 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.
Example Applications	VoltDB comes with several example applications that demonstrate VoltDB capabilities and performance. They can be found in the `examples/` subfolder.
Demo Dashboard	The demo dashboard provides a quick and easy way to become familiar with VoltDB. The dashboard helps you start, view, and interact with the example applications. Start the dashboard by invoking the `Click Here to Start.html` file in the top level installation folder.
VoltDB Studio Web	Studio Web is a browser-based tool for visualizing and querying a running VoltDB database that is bundled with the VoltDB server software. You can start Studio Web by connecting to the HTTP port of a running VoltDB database server. For example, `http://voltsvr:8080/studio`. Note that the httpd server and JSON interface must be enabled to access Studio Web from the server.
Shell Commands	The bin/ subfolder contains executable scripts to perform common VoltDB tasks, such as compiling application catalogs and starting the VoltDB server. Add the bin/ subfolder to your PATH environment variable to use the following shell commands: voltcompiler voltdb sqlcmd exporttofile
Generate Script	The Generate script helps you get started developing with VoltDB by creating a template application, complete with source files, folder structure, and Ant build file. The Generate script is in the `tools/` subfolder. (See *Getting Started with VoltDB* for more information about the Generate script.)
Documentation	Online documentation, including the full manuals and javadoc describing the Java programming interface, is available in the `doc/` subfolder.

2.4. 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 use the Demo Dashboard, which walks

you through two of the samples. Open the file named `Click Here to Start.html` in the top level VoltDB folder to start the dashboard.

See the README file in the `examples/` subfolder for a complete list of the applications and brief instructions. The accompanying book *Getting Started with VoltDB* provides more detail about running the example applications and using them to learn how VoltDB works.

Getting Started also contains a tutorial, Hello World, that walks you through the process of writing a VoltDB application from scratch. The sources for the Hello World tutorial are also included in the `tutorial/` subfolder.

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-sited 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:

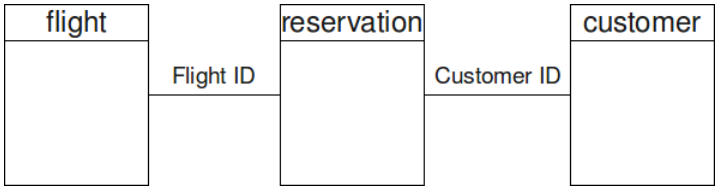


Figure 3.1 shows how the schema looks as defined in standard SQL DDL.

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-sited.

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-sited. 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.

3.1.1.2. Specifying the Partition Columns in the Project Definition File

Once you choose the columns to use for partitioning your database tables, you specify them in the project definition file. The project definition file is an XML file that includes additional information (beyond just the schema) that VoltDB uses to configure the database when you compile your project.

You specify the partition columns in the project definition file using the <partition> tag and the table and column attributes. For example, to specify FlightID and CustomerID as the partitioning columns for the Reservation and Customer tables, respectively, your project definition file would look like the following:

```
<?xml version="1.0"?>
<project>
   <database name='database'>
         . . .
      <partitions>
         <partition table="Reservation" column="FlightID">
```

```
            <partition table="Customer" column="CustomerID">
        </partitions>
            . . .
    </database>
</project>
```

3.1.1.3. 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 project definition file.

3.1.2. Replicating Lookup 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-sited.

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 project definition file, 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 all access to the database must be done through stored procedures. It is possible to perform ad hoc queries on a VoltDB database (as described later) and they can be useful for debugging an application or other administrative actions. However, ad hoc queries do not take advantage of the performance optimizations VoltDB specializes in and therefore should not be used for frequent or repetitive queries.

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 when these queries are read-only, VoltDB may detect inconsistency in the results of 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 may not match, a condition that VoltDB detects and reports as corruption. It is possible one or more nodes of the cluster may crash in this situation. Consequently, when returning multiple rows, use of an ORDER BY clause or a tree index in a WHERE constraint is strongly recommended for all SELECT statements.

[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.

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 anyway 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.

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.

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

@ProcInfo(                                                              ❷
    singlePartition = true,
    partitionInfo = "Reservation.FlightID: 0"
)

public class HowManySeats extends VoltProcedure {                      ❸

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

    public long run( int flightid)
        throws VoltAbortException {                                     ❺

        long numofseats;
        long seatsinuse;
        VoltTable[] queryresults;

        voltQueueSQL( GetSeatCount, flightid);                          ❻
        queryresults = voltExecuteSQL();                                ❼

        VoltTable result = queryresults[0];                             ❽
        if (result.getRowCount() < 1) { return -1; }
        numofseats = result.fetchRow(0).getLong(0);
        seatsinuse = result.fetchRow(0).getLong(1);

        numofseats = numofseats - seatsinuse;                           ❾
        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 The @ProcInfo annotation tells VoltDB if the procedure is single-sited or not and, if so, which column identifies the specific partition to use. If you do not specify that singlePartition is true, by default VoltDB assumes the procedure is multi-sited.

❸ Each stored procedure extends the generic class VoltProcedure.

❹ 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.)

❺ 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.)

❻ To perform database queries, you queue SQL statements (specifying both the SQL statement and the variables to use) using the voltQueueSQL method.
❼ Once you queue all of the SQL statements you want to perform, use voltExecuteSQL to execute the statements in the queue.
❽ 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.
❾ 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 @ProcInfo annotation to indicate whether the procedure is single-sited or not
- 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.*;

@ProcInfo( singlePartition = true|false,
           partitionInfo = "Table-name.Column-name: 0" )

public class Procedure-name extends VoltProcedure {

                // Declare SQL statements ...

    public datatype run ( arguments ) throws VoltAbortException {


               // Body of the Stored Procedure ...


    }
}
```

3.2.3.2. Using @ProcInfo

As described in the introduction, one of the ways to achieve optimal throughput is to use single-sited stored procedures wherever possible. When you are writing a single-sited stored procedure, it is important that you tell VoltDB which partitioning column you are using and what value to use to find the appropriate partition. You do this using the @ProcInfo annotation.

The @ProcInfo annotation has two components:

- **partitionInfo** specifies which column and what value is being used for the partition.
- **singlePartition** specifies whether the procedure is single-sited or not.

The partitionInfo annotation has three parts. The first two specify the names of the table being partitioned and the partitioning column, separated by a period. The third argument specifies which parameter of the run method is the value being used for that column. Note that the third argument is zero-based, so if the first parameter to run is the column value, then the argument to partitionInfo is 0. If it is the second parameter, then the argument is 1, and so on. You separate the third argument from the first and second with a colon (:).

For example, when looking up a reservation by FlightID (the partitioning column for the Reservation table) and passing in the Flight ID as the first parameter to run, the @ProcInfo annotation would like this:

```
@ProcInfo(
    singlePartition = true,
    partitionInfo = "Reservation.FlightID: 0"
)
```

Of course, if you state that a stored procedure is single-sited in the @ProcInfo, you must make sure that the SQL queries themselves operate in a single-sited fashion. See Section 3.2.4, “Writing Single-Partitioned Stored Procedures” for more information about how to write single-sited stored procedures.

If the stored procedure is not single-sited, you do not need to include the @ProcInfo annotation. However, it is a good coding practice to include it (and specify `singlePartition=false`) to indicate that the procedure is intentionally multi-sited.

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: FlightID, 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 4.3, “Verifying Expected Query Results” for another way to roll back procedures when queries do not meet necessary conditions.

3.2.4. Writing Single-Partitioned Stored Procedures

You can declare your stored procedures as either multi-partitioned or single-partitioned. 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 and the parameter that is used as the hash value for the partition. 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 @ProcInfo annotation) using the FLIGHTID column and the *flight_id* parameter as the hash value.

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.

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 14, *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 project 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.

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.

Creating multiple connections has little effect when you are making synchronous requests. However, for asynchronous requests, multiple connections can help balance the load across the cluster.

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();
}
```

3.3.2. 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());
    }
  }
}
```

Several important points to note about 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.
- 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. Specify the timeout in milliseconds, although the actual timeouts will only be accurate to within approximately a second of the specified value. Set the value to zero to disable timeouts. 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.setConnectionResponseTimeout(90 * 1000);
  config.setProcedureCallTimeout(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 for partitioned tables

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 for Partitioned Tables

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.

To simplify these operations, VoltDB defines default stored procedures to perform these queries for any partitioned table where the partitioning column is part of the primary key index. 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 procedure, 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 four default stored procedures, in addition to any defined in the project definition file. Those default procedures are:

- HELLOWORLD.insert
- HELLOWORLD.select
- HELLOWORLD.update
- 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",
                     "Howdy","Earth","American");
results = client.callProcedure("HELLOWORLD.select",
                     "American").getResults();
client.callProcedure("HELLOWORLD.update",
                     "Yo","Biosphere","American",
                     "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 *project definition file*. Chapter 5, *Building Your VoltDB Application* describes the project definition file in more detail, but in brief, it is an XML file that describes all of the components of the database and is used to compile an application catalog.

Normally, the project definition file contains entries that identify each of the stored procedures, like so:

```
<procedure class="procedures.MakeReservation"/>
<procedure class="procedures.CancelReservation"/>
```

The <procedure> tag 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 as a child of the <procedure> tag:

```
<procedure class="procedures.simple.CountReservations">
   <sql>SELECT COUNT(*) FROM RESERVATION</sql>
</procedure>
```

When you include the SQL query in the procedure definition, VoltDB generates the necessary Java code for you and compiles it when you build your application (as described in Section 5.3, “Building the Runtime Catalog”). Note that you must still provide a unique class name for the procedure. It is a good idea to put these simplified procedures into a separate package (*procedures.simple*, in the preceding example) from those written by hand.

Since the SQL statement is part of an XML file, it must obey the rules for XML syntax. In particular, if you need to use the greater than, less than, or ampersand symbols you should use the appropriate entity to represent them. For example, the following example shows a SELECT statement where the AGE column must be less than 21.

```
<procedure class="procedures.simple.CountChildren">
   <sql>SELECT COUNT(*) FROM CUSTOMER WHERE AGE &lt; 21</sql>
</procedure>
```

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:

```
<procedure class="procedures.simple.MyReservationsByTrip">
   <sql>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=?
   </sql>
</procedure>
```

Finally, you can also specify whether the simple procedure is single-partitioned or not. By default, simple stored procedures defined in the project definition file are assumed to be multi-partitioned. But if your procedure is single-partitioned, you can specify the partitioning information as an attribute to the <procedure> tag. 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.

```
<procedure class="procedures.simple.FetchReservations"
           partitioninfo="Reservation.flightid:0">
   <sql>SELECT * FROM RESERVATION
        WHERE FLIGHTID=?
   </sql>
</procedure>
```

Note that the syntax of the `partitioninfo` attribute is identical to the syntax of `partitionInfo` in the `@ProcInfo` annotation if you were writing the stored procedure as a Java class.

4.3. Verifying Expected Query Results

The automated default and simple stored procedures reduce the coding needed to perform simple queries. However, another substantial chunk of stored procedure and 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 three steps to building a VoltDB application:

1. Compiling the client application and stored procedures

2. Creating a project definition file

3. Compiling the VoltDB runtime catalog

This chapter explains the steps, with a particular focus on creating the project definition file to describe the database structure.

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, like so:

```
$ javac -classpath "./:/opt/voltdb/voltdb/*" *.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.

Obviously, it is much easier to use an automated build procedure to keep track of all of the application dependencies rather than typing the necessary commands by hand. VoltDB provides a tool to create a template application, including an automated Ant build script. The Generate script (which is provided in the /tools subfolder where you install VoltDB) creates a folder structure, build script, project definition file and sample Java source files to build a complete VoltDB application, including the necessary links to the VoltDB libraries.

We strongly recommend using the Generate script, or using one of the sample applications as a template, to simplify the building and running of your VoltDB applications. More information about the Generate script and the sample applications can be found in *Getting Started with VoltDB*.

5.2. Creating the Project Definition File

Once you compile the application and stored procedure source files, the next step is to compile the runtime catalog. The runtime catalog contains all of the information needed to create the database on a server, including the database schema, security settings, and stored procedures.

[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 14, *Using VoltDB with Other Programming Languages* for more information on alternate client interfaces.

You tell the VoltDB compiler where to find this information in the *project definition file*. The project definition file identifies the main characteristics of your database, including the schema, the stored procedures, and partitioning information for the individual tables. The project definition is an XML file, so can be created using an XML editor (such as the XMLBuddy plugin for Eclipse) or a text editor.

The key elements of the project definition file are the root <project> element and the <database> element. Both are required.

```
<?xml version="1.0" ?>
<project>
  <database name="database">
       . . .
  </database>
</project>
```

Within the <database> element, you must include a <schema> element that points to the DDL file containing your database schema, as well as:

- <procedure> tags that identify all of the stored procedures within your database, specified by their class name
- <partition> tags that identify which tables within the database are partitioned and the column used for partitioning

Be sure to identify all of your stored procedures in the project definition file or they will not be included in the catalog and therefore will not be available to the client applications at runtime. Also, if you do not specify partitioning information for a table defined in the schema, that table will be replicated in all partitions. (See Section 3.1, “Designing the Database” for more information about partitioned and replicated tables.)

The following is an example of a complete project definition file:

```
<?xml version="1.0" ?>
<project>
  <database name="database">
    <schemas>
      <schema path="flight.ddl" />
    </schemas>
    <procedures>
      <procedure class="procedures.LookupFlight"/>
      <procedure class="procedures.HowManySeats"/>
      <procedure class="procedures.MakeReservation"/>
      <procedure class="procedures.CancelReservation"/>
      <procedure class="procedures.RemoveFlight"/>
    </procedures>
    <partitions>
      <partition table="Reservation" column="FlightID"/>
      <partition table="Customer" column="CustomerID"/>
    </partitions>
  </database>
</project>
```

You can also use the project definition file to define groups and permissions that are used for authenticating clients at runtime. See Chapter 8, *Security* for more information about defining database security.

5.3. Building the Runtime Catalog

You build the runtime catalog for your VoltDB database by compiling the project definition file using the VoltDB Compiler. The compiler comes as an executable class in the VoltDB library JAR file. So to run the compiler, you use the Java command, specifying the compiler class (VoltCompiler) followed by two arguments:

1. The name of the project definition file to use as input

2. The name of the runtime catalog to create as output

For example:

```
$ java -classpath .:/opt/voltdb/voltdb/* \
       org.voltdb.compiler.VoltCompiler  \
       project.xml catalog.jar
```

Note that, just as when compiling the application source files, you must include the VoltDB library in the classpath when running the VoltDB compiler. To make the process easier, the VoltDB installation includes executable scripts to simplify the command line and ensure all of the necessary libraries are included. If you add the `bin/` folder to your path, you can use the simplified command `voltcompiler`, which takes three arguments: your local classpath, the project definition file, and the target catalog file. For example, the following commands perform the same function as the preceding full Java command:

```
$ PATH="$PATH:/opt/voltdb/bin/"
$ voltcompiler "./" project.xml catalog.jar
```

Another alternative, especially for complex projects, is to use an automated build system (such as ant or make) to manage the build process.

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 project definition file that is used to compile the runtime 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, much like the project definition file, which is used 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. In general, this value is related to the number of processor cores per node. Section 6.1.1, “Determining How Many Partitions to Use” explains how to choose a value for this attribute,
- **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.)

The deployment file is used to enable and configure many other runtime options related to the database, many of which are described later in this book. For example, if the database project file enables security,

the deployment file 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

In general, the number of partitions 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, two important points to keep in mind are:

- It is never useful to specify more partitions than the number of CPUs reported by the operating system.
- 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 client 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. In the worst case, if the time shift is too large (greater than three seconds) VoltDB issues a fatal error and stops.

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:

- The name of the lead node in the cluster
- The location of the application catalog
- The location of the deployment file
- Optionally, a startup action (see Section 6.5, “Stopping and Restarting a VoltDB Database” for details)

The *lead node* can be any node in the cluster. However, that node plays a special role during startup; it hosts the application catalog and manages the cluster initiation process. Once startup is complete, the lead node'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 lead node you specify.

For example, the following Java command starts the cluster with the default startup action, specifying the location of the catalog and the deployment files, and naming voltsvr1 as the lead node:

```
java -Djava.library.path=/opt/voltdb/voltdb org.voltdb.VoltDB \
     leader voltsvr1 \
     catalog mycatalog.jar \
     deployment deployment.xml
```

If you are using the VoltDB Enterprise Edition, you must also specify the location of the license file. The license file is only required by the lead node when starting the cluster; the license argument is ignored in all other cases (including when using the community edition). This way, you can use the same command on all nodes.

The command to start a cluster using the Enterprise Edition looks like this:

```
java -Djava.library.path=/opt/voltdb/voltdb org.voltdb.VoltDB \
     leader voltsvr1 \
     catalog mycatalog.jar \
     deployment deployment.xml \
     license /opt/voltdb/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 lead 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 (verified using checksums) 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 lead node, it waits for initialization messages from the remaining nodes.

2. If you are starting the database on a non-lead node, it sends an initialization message to the lead node indicating that it is ready.

3. Once all the nodes have sent initialization messages, the lead node sends out a message to the other nodes that the cluster is complete. The lead node 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 lead node'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 either the Generate script or one of the sample applications as a template), the build.xml 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 programmatically. That is what the @Shutdown system procedure is for.

Calling the @Shutdown system procedure (from any node) will shutdown the database on the entire cluster. You call @Shutdown the same way you call stored procedures, using the callProcedure method:

```
try { client.callProcedure("@Shutdown");
} catch (Exception e) {
           // an exception is expected.
    System.out.print("Database shutdown request submitted.");
}
```

Note that if @Shutdown is successful, your client application will receive an exception from the callProcedure call. This is because as soon as the database server is shut down, it breaks the database connection to your client before the call is closed. This causes the VoltDB client to throw an exception indicating that the connection is lost. This exception does *not* indicate that the shutdown itself failed.

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 (Enterprise Edition only)

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 either perform a manual snapshot using the @SnapshotSave system procedure or you can 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. Command logging is a feature of the VoltDB Enterprise Edition and requires a commercial license.

When you enable command logging, 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 optional startup action is available on the command line when starting the server process. Those 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.
- **start** — starts a new database process and recovers the command log, if it exists. If no command log information is found, a new empty database is created. The `start` action allows for a single command to be used to both create the initial database and recover it on all subsequent sessions without having to change the database startup procedures or scripts.

The default action, if you do not specify one on the command line, is `start`.

Even if you are using the VoltDB community edition, rather than the Enterprise Edition, you can use the `create`, `recover`, and `start` actions. The community edition does not support creating or replaying command logs; however, for `start` and `recover`, it will attempt to restore the last snapshot found in the snapshot paths. Therefore, using automated snapshots and `recover` (or the default, `start`) action, it is possible with the community edition to automatically recover all of the data from the previous database session up until the last snapshot.

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

```
java -Djava.library.path=/opt/voltdb/voltdb org.voltdb.VoltDB \
     recover \
     leader voltsvr1 \
     catalog mycatalog.jar \
     deployment deployment.xml \
     license /opt/voltdb/voltdb/license.xml
```

The advantages of command logging 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 11, *Availability* for more information about enabling and 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 project definition 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 a call to @Resume.

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 restore a snapshot or prepopulate the database with data through a set of custom stored procedures. To do this, invoke @SnapshotRestore or your custom procedures through the admin port, then call @Resume once the initialization is complete.

Chapter 7. Updating Your VoltDB Application

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 or column 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 modify existing table columns or modify the actual parameters to or behavior of 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. You can add, remove, or update stored procedures "on the fly", while the database is running. You can also add or drop entire tables from the schema.

To make other changes to the database schema (such as modifying individual columns within a table) or to reconfigure the cluster hardware, you must first save and shutdown the database. However, even in this situation, VoltDB automates the process by transforming the data and redistributing partitions when you restart and reload the database in a new configuration.

This chapter explains the steps necessary to make various changes to your VoltDB database application, including:

- Updating the Stored Procedures
- Updating the Database Schema
- Updating the Hardware Configuration

7.2. Updating the Stored Procedures

Stored procedures, including the security permissions for accessing them, can be updated on the fly for a running VoltDB database. This is done by creating an updated application catalog and deployment file and telling the database process to use the new catalog with the @UpdateApplicationCatalog system procedure.

Specifically, the process for updating stored procedures is as follows:

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

For example:

```
String newcat = "mycatalog.jar";
String newdeploy = "mydeployment.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(); }
```

Note that the content of the new catalog and deployment files are passed to the cluster directly as parameters of the system procedure. You do not need to copy the files themselves to the cluster nodes, as you do when starting the cluster initially.

7.2.1. Validating the Updated Catalog

When you invoke @UpdateApplicationCatalog, 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 stored procedure call returns an error.

Changes that are not currently allowed to a running database include modifying columns within the database schema or changing the configuration of export. To make these more complex changes, you need to save and shutdown the database to change the catalog, as described in Section 7.3, “Updating the Database Schema”.

7.2.2. Managing the Update Process

For most simple changes, updating the application catalog lets you modify the database and its stored procedures without disrupting the normal operations. However, for certain changes, you should be aware of the impact on any client applications that use those procedures. For example, if you remove a stored procedure or change its parameters or permissions while client applications are still actively calling it, 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 stored procedure 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 Schema

It is also possible to modify the database schema. You can add or remove entire tables or views from a running database using the same procedure used for modifying stored procedures described in Section 7.2, “Updating the Stored Procedures”. That is, you can modify the schema definition, recompile the application catalog, and update the database on the fly using the @UpdateApplicationCatalog system procedure.

However, you cannot make smaller modifications, such as changing individual columns within a table or changing indexes on the fly. To make these sorts of changes to the schema you must:

1. Save the current data.
2. Shut down the database.
3. Replace the application catalog.
4. Restart the database with the new catalog.
5. Reload the data saved in Step #1.

Using this procedure, you can add or remove columns to existing tables. 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.

To change the number of nodes or the number of partitions per node, you must first save and shutdown the database. You can then edit the deployment file specifying the new number of nodes and partitions per node in the attributes of the <cluster> tag.

Once your have created a new deployment file with the desired hardware configuration, restart the database and reload the data using the @SnapshotRestore system procedure. See Section 9.1.3, “Changing the Database Schema or Cluster Configuration Using Save and Restore” for details on saving and restoring your database manually.

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 project definition 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 project definition file identifies the user as a member of a group 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 runtime catalog.

There are three steps to enabling security in the project definition and deployment files for a VoltDB application:

1. Add the `<security enabled="true"/>` tag to turn on authentication and authorization.
2. Define the users and groups you need to authenticate.
3. Define which groups 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 project definition file:

```
<project>
   <security enabled="true"/>
      . . .
</project>
```

Note that the <security> XML tag is a child of the <project> element, not the <database> element. The <security> tag must precede the <database> tag in the project definition file.

8.3. Defining Users and Groups

The key to security for VoltDB applications is the users and groups defined in the project definition and deployment files. You define users in the deployment file and groups in the project definition file.

This split is deliberate because it allows you to define the overall security structure globally in the project definition file, assigning permissions to generic groups (such as admin, dbuser, apps, and so on). You then define specific users and assign them to the generic groups as part of the deployment. This way you can create one configuration (including cluster information and users) for development and testing, then move the application catalog to a different configuration and 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"
            groups="group-name[,...]"  />
        [ ... ]
   </users>
   ...
</deployment>
```

Include a <user> tag for every username/password pair you want to define.

Then within the project definition file you define the groups the users can belong to. You define groups with a <groups> ... </groups> tag set.

```
<project>
   <database>
      <groups>
         <group name="group-name" />
           [ ... ]
      </groups>
      ...
   </database>
</project>
```

You specify which groups a user belongs to as part of the user definition in the deployment file using the groups attribute to the <user> tag. For example, the following code defines three users, adding operator and developer to the admin group and developer and clientapp to the dbuser group. When a user belongs to more than one group, you specify the group names as a comma-delimited list.

```
<deployment>
   <users>
      <user name="operator" password="mech" groups="admin" />
      <user name="developer" password="tech" groups="admin,dbuser" />
      <user name="clientapp" password="xyzzy" groups="dbuser" />
   </users>

</deployment>
```

Two important notes concerning the assignment of users to groups:

- Users must belong to at least one group, or else they cannot be assigned any permissions. (Permissions are assigned by group.)
- There must be a corresponding group defined in the project definition file for any groups listed in the deployment file.

8.4. Assigning Access to Stored Procedures

Once you define the users and groups you need, you assign them access to individual stored procedures using the groups attribute to the <procedure> tag. In the following example, users belonging to the groups admin and dbuser are permitted access to both the MyProc1 and MyProc2 procedures. Only members of the admin group have access to the MyProc3 procedure.

```
<database>
   ...
   <procedure class="MyProc1" groups="dbuser,admin" />
   <procedure class="MyProc2" groups="dbuser,admin" />
   <procedure class="MyProc3" groups="admin" />
   ...
</database>
```

When security is enabled, you must specify access rights for each stored procedure. If a procedure definition does not include a groups attribute, no access is allowed. In other words, calling applications will not be able to invoke that procedure.

8.5. Allowing Access to System Procedures and Ad Hoc Queries

There are several special procedures available within VoltDB that are not called out in the project definition file. These procedures, which all begin with an at sign (@), perform special functions such as saving and restoring snapshots of the database and performing ad hoc queries. (See Appendix E, *System Procedures* for more information about system procedures.)

By default, when security is not enabled, any calling application has access to these system procedures. However, when you enable security, you must explicitly assign access to these procedures as well.

Since there is no procedure definition in the project file for system procedures, you assign access to these functions through attributes of the <group> tag. There are two attributes for this purpose:

- sysproc -- allows access to all system procedures
- adhoc -- allows access to the @adhoc procedure only

In the <group> tags you enable or disable access to system procedures by assigning the sysproc and adhoc attributes values of true or false, respectively. (The default, if security is enabled and the attribute is not specified, is false.)

Note that the permissions are additive. So if a user belongs to one group that allows access to adhoc but not sysproc, but that user also belongs to another group that allows sysproc, the user has both permissions.

The following example assigns access to all system procedures to members of the admin group, access to the adhoc procedure to members of the dbuser group, and no access to system procedures for all other users.

```
<database>
   <groups>
      <group name="admin" sysproc="true" />
      <group name="dbuser" adhoc="true" sysproc="false" />
      <group name="apps" />
   </groups>
   ...
</database>
```

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 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. For example, if you need to upgrade the hardware or add a new node to the cluster. 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 by calling VoltDB system procedures. In other words, you can write Java code to perform these operations. If you are using the VoltDB Enterprise Edition, you can use the Enterprise Manager to perform many of these tasks from within the management console. See the *VoltDB Management Guide* for details.

When you issue a save command, you specify an absolute 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).

A third argument, an integer flag, indicates whether the save operation should block other transactions until it completes. In the case of manual saves, it is a good idea to set this flag to true (that is, any non-zero value) 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, you call the system procedure @SnapshotSave. The following example creates a snapshot at the path */tmp/voltdb/backup* using the unique identifier *TestSnapshot*.

```
static final String SAVEDIR = "/tmp/voltdb/backup";
static final String SAVEID = "TestSnapshot";
static final int BLOCKING = 1;

VoltTable[] results = null;
try {
      results = client.callProcedure("@SnapshotSave",
                        SAVEDIR,SAVEID,BLOCKING).getResults();
    }
catch (Exception ex)
    {
       ex.printStackTrace();
       fail("Save operation failed: " + ex.getMessage());
     }

for (VoltTable t: results) { System.out.println(t.toString()); }
```

In this example, the procedure call 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 the use of a for loop after calling *@SnapshotSave* to display the return values from the call. Since the save operation may succeed on some nodes of the cluster and not others, the system procedure does not throw an exception for all possible errors. Instead, it provides success or failure status information in the VoltTable array returned by callProcedure.

It is a good practice to examine the return value of the save operation 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 call the system procedure @SnapshotRestore. 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.

```
static final String SAVEDIR = "/tmp/voltdb/backup";
static final String SAVEID = "TestSnapshot";

VoltTable[] results = null;
try {
      results = client.callProcedure("@SnapshotRestore",
                       SAVEDIR,SAVEID).getResults();
    }
catch (Exception ex)
    {
       ex.printStackTrace();
       fail("Restore operation failed: " + ex.getMessage());
     }

for (VoltTable t: results) { System.out.println(t.toString()); }
```

As with save operations, it is always a good idea to check the status information returned by the restore procedure 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, the project definition file, or the deployment file. You can then recompile the runtime 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, the stored procedure Java source files, and the project definition file), 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 configuration 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 through the VoltTable array returned by the save operation. The return value contains detailed information from each of the nodes on the cluster.
- 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.

Important

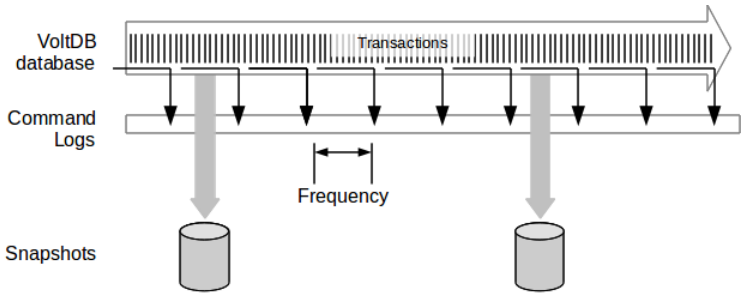
Command logging is a commercial feature that is included in the VoltDB Enterprise Edition only. Most of what is described in this chapter does not apply to the community edition of the product. The last section of the chapter describes what recovery options are available in the community edition.

10.1. How Command Logging Works

When you enable command logging, 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.

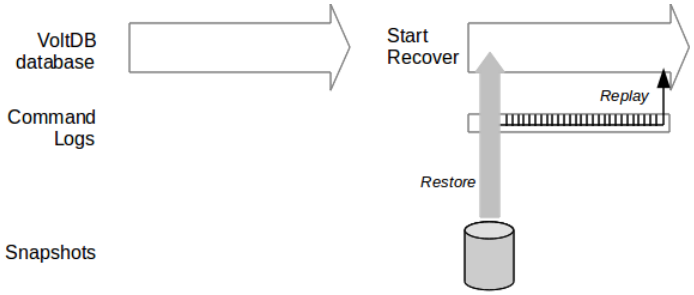
Figure 10.1. Command Logging in Action



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 a database starts with either the start or recover option (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. Enabling Command Logging

Command logging is not enabled by default. To enable command logging, you add the `<commandlog>` element to 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 command logging using the default configuration. You can use the `enabled` attribute to explicitly enable or disable the feature (as in the previous example), or you can add 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.

10.4. Recovery Options in the VoltDB Community Edition

Command logging, as described in this chapter, is a commercial feature available in the VoltDB Enterprise Edition only. However, you can still use the `start` and `recover` actions with the community edition of VoltDB.

The VoltDB community edition cannot create or replay command logs. It can, however, create and restore snapshots. If you specify the `start` or `recover` action when starting a VoltDB database using the community edition, the database will attempt to restore the most recent snapshot found in the appropriate snapshot paths. If used with the automated snapshots feature (as described in Section 9.2, “Scheduling Automated Snapshots”), this allows the community edition to provide a partially automated recovery from failure, in that it will restore the last known valid state of the database. The recovery is not as complete as with command logging, but in many cases (such as planned shutdowns), can be very useful.

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 three features of VoltDB: snapshots, command logging, and K-safety.

- *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.

Previous chapters described snapshots and command logging. 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.)

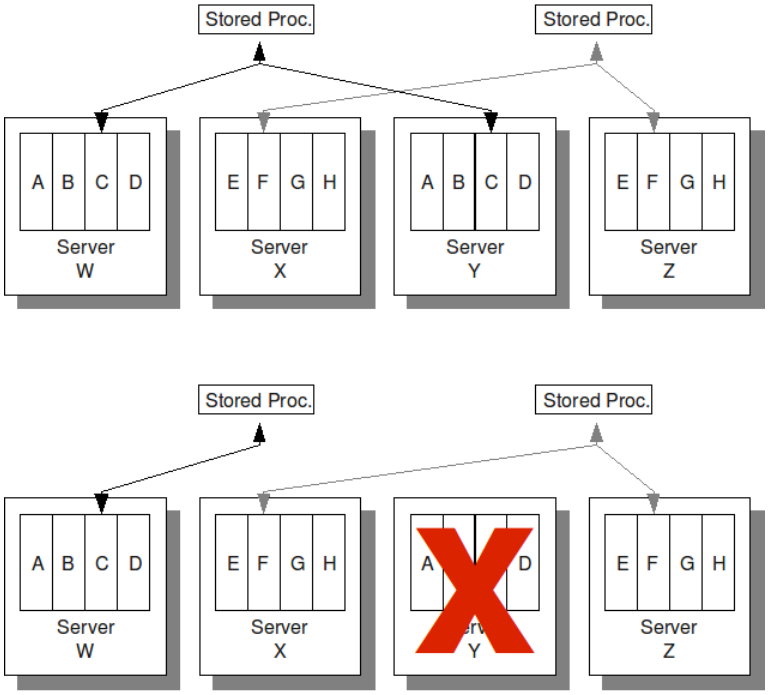
It is also important to note that K-safety is different than WAN replication. In replication the entire database cluster is replicated (usually at a remote location to provide for disaster recovery in case the entire cluster or site goes down due to catastrophic failure of some type).

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 less than or equal to the total number of partitions divided by the needed number of copies:

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

Therefore, if 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.

If neither the number of nodes nor the total number of partitions is divisible by K+1, then VoltDB will assign sufficient duplicates to achieve the desired K-safety value and then assign any remaining partitions to additional duplicates. For example, if you specify a cluster size of 4 nodes with 4 partitions each and request a value of K=2 (3 total copies of each partition), VoltDB will create 3 sets of 5 unique partitions. However, there is a total of 16 available partitions. VoltDB will create a fourth copy of one of the partitions to use the last available slot.

This approach ensures that your database achieves the desired K-safety. However, it also essentially "wastes" a partition. Therefore, to maximize your hardware investment, it is strongly recommended that when you specify a K value, you ensure that the number of total partitions (number of nodes times partitions per node) is a multiple of the K+1.

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 13, *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 and adding the argument "rejoinhost" with the address of one of the remaining nodes of the cluster. For example, to rejoin a node to the VoltDB cluster where myclusternode5 is one of the current member nodes, you use the following command:

```
$ java -Djava.library.path=/opt/voltdb org.voltdb.VoltDB \
      deployment mydeployment.xml \
      rejoinhost myclusternode5
```

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

If security is enabled for the cluster, you must also specify a username and, optionally, a password on the command line. (If you specify a username but not a password, you will be prompted for the password.) The full syntax for specifying the node to reconnect to is as follows. You only need to specify the port number if the server was started using a different port number than the default.

```
username:password@nodename:port
```

For example, the following command attempts to rejoin the current system to the cluster that includes the node *voltserver2* using the username *operator*. VoltDB will prompt for the password.

```
$ java -Djava.library.path=/opt/voltdb org.voltdb.VoltDB \
      deployment mydeployment.xml \
      rejoinhost operator@voltserver2
```

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. What's more, 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 call 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.

More importantly, 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 node with the appropriate catalog and deployment file 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 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.

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 eliminating 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*.

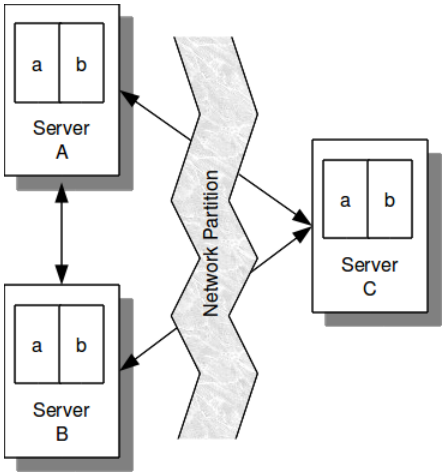
11.4.1. K-Safety and Network Partitions

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



Now, it is important to remember that network partitions are rare, and rarer still the chance that a partition leaves two viable segments of a durable VoltDB cluster. But rare as it may be, it is a possibility.

The problem is that you do not want two separate copies of the database continuing to operate and accept requests thinking they are the only viable copy. If the cluster is physically on a single network switch, a

network partition is for all intents and purposes impossible. But if the cluster is on multiple switches the possibility of a network partition is a risk you must consider and account 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.

You enable network fault protection in the deployment file, by adding 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="netfaultsave"/>
   </partition-detection>
</deployment>
```

Note that you must specify the <snapshot> tag and its attributes as a child of <partition-detection>. If a partition is detected, the affected nodes automatically do a snapshot of the current database before shutting down. The <snapshot> attribute `prefix` specifies what file prefix to use.

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 you enable network fault protection, 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...

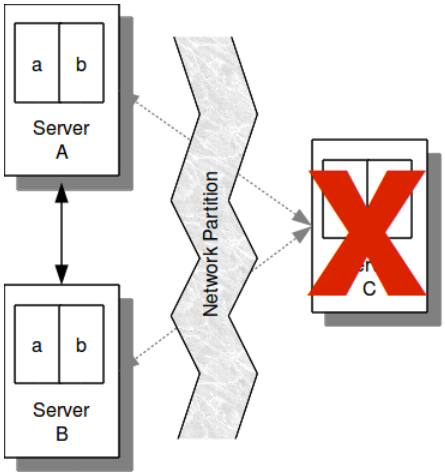
3. Determine which segment is the larger segment (that is, contains more nodes).

   - 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.

This is why network fault protection is not enabled by default. It is up to you, as the database administrator, to decide how much risk network partitioning is to your specific configuration. If your cluster is connected to a single switch, you should **not** enable network fault protection. If your cluster is separated by several switches, or the configuration is not known, you need to decide whether the threat of a network partition is sufficient to protect against it by enabling fault protection.

Chapter 12. 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:

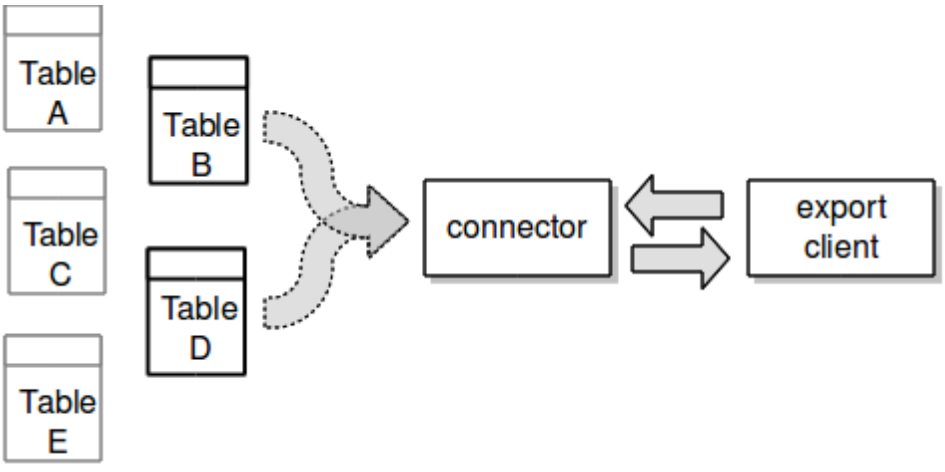
- You only export selected data (as required by the business process)
- Export is an ongoing process rather than a one-time event
- The outcome of exporting data is that information is used by other business processes, not as a backup copy for restoring the database

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.

12.1. Understanding Export

VoltDB lets you automate the export process by specifying certain tables in the schema as sources for export as part of the project definition file. At runtime, any data written to the specified tables is sent to the export *connector*, which manages the exchange of the updated information to a separate receiving application. Figure 12.1, "Overview of Export Process" illustrates the basic structure of the export process, where Tables B and D are specified as export tables.

Figure 12.1. Overview of Export Process



Note that you, as the application developer, 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.

12.2. Planning your Export Strategy

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

- What data to export
- When to export the data

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

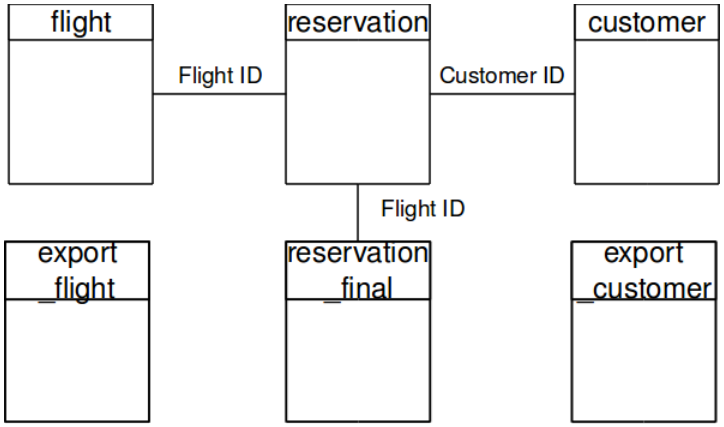
[1]There is no guarantee on the latency of export between the connector and the export client. The export function is transactionally correct; no export occurs if the stored procedure rolls back and the export data is in the appropriate transaction order. But the flow of export data from the connector to the client is not synchronous with the completion of the transaction. There may be several seconds delay before the export data is available to the client.

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 12.2, “Flight Schema with Export Table” shows the modified database schema with the added export-only tables, EXPORT_FLIGHT, EXPORT_CUSTOMER, and RESERVATION_FINAL.

Figure 12.2. 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.

12.3. Identifying Export Tables in the Project Definition File

Once you have decide what data to export and define the appropriate tables in the schema, you are ready to identify them as export-only tables in your project definition file. 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 in the project definition file. You do this with the <export> and <tables> tags. For example:

```
<project>
   <database name="database">
       . . .

      <export>
         <tables>
            <table name="export_customer" />
            <table name="export_flight" />
            <table name="reservation_final"/>
         </tables>
      </export>
   </database>
</project>
```

You specify which tables to export within the <tables> ... </tables> tag group. If a table is not specified in the <tables> group, 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. Since they are listed for export in the project definition file, these are export-only tables even if export is disabled in the deployment file.

You can also specify whether the export-only tables are partitioned or not using the <partition> tag in the project definition file. 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 and only one tuple is written to the export stream.

The <export> tag also lets you restrict what clients are allowed to connect to the export connector. By adding the groups attribute to the <export> tag, you specify what groups have permission to create a connection and read from the export queue. By default, security is not enabled in the project definition file, so to restrict access you must explicitly enable security. In reverse, if security is enabled, you must make sure you specify groups as part of the export definition to ensure the client can connect to the export queue.

It is a good practice to put the export definition at the end of the project definition file, both for readability and ease of editing.

12.4. Configuring Export in the Deployment File

The VoltDB connector creates packets of the serialized export data and then waits for an export client application to connect and request the packets. To enable the connector (and the export function) at runtime, you include the <export> tag in the deployment file.

```
<export enabled="true"/>
```

If you include the <export> tag without any attributes, you implicitly enable export. You can use the *enabled* attribute to explicitly enable or disable export.

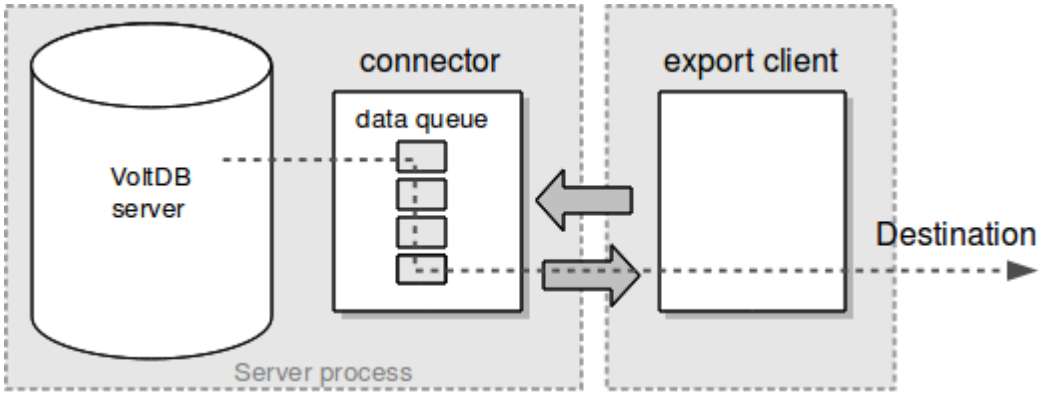
12.5. How Export Works

The export connector implements a loosely coupled approach to extracting export data from a running VoltDB database. When export is enabled at runtime:

1. Insert operations to the database tables identified as export-only in the project definition file are queued to the export connector.
2. An export client establishes a link to the connector through one of the standard TCP/IP ports (either the client or admin port). The client then issues POLL requests.
3. The connector responds to the POLL requests with the next queued data block (or an empty block if the queue is empty).
4. The client is then responsible for receiving the data and writing it to the appropriate destination.
5. Finally, the export client sends an ACK message acknowledging completion of the export (at which point the connector can remove it from the queue) before polling for the next data block.

Figure 12.3, “The Components of the Export Process” shows the interaction between the VoltDB database, the connector, and the export client.

Figure 12.3. The Components of the Export Process



The export function queues and passes data to the connector automatically. You do not need to do anything explicitly to start the connector; it starts and stops when the database starts and stops. The connector and the export client use a series of poll and ack requests to exchange the data over the TCP port.

The export client decides what is done with the data it receives from the connector. For example, one client writes the serialized data to a sequence of files while another could insert it into an analytic database. Only one export client can connect to the connector at a time. But it is possible to change the destination of the export by using different clients.

It is also important to note that the export client must create connections to all nodes in the database cluster, since each node creates its own instance of the connector.

12.5.1. Export Overflow

For the export process to work, it is important that the connector and client keep up with the queue of exported information. If too much data gets queued to the connector by the export function without being fetched by the client, the VoltDB server process consumes increasingly large amounts of memory.

If the export client 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 client fails, writing to disk helps VoltDB avoid consuming too much memory while waiting for the client to restart.
- If the database is stopped, the export data is retained across sessions. When the database restarts and the client reconnects, 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..

12.5.2. Persistence Across Database Sessions

It is important to note that VoltDB only uses the disk storage for overflow data. However, you can force VoltDB to write all queued export data to disk by either calling the @Quiesce system procedure or by requesting a blocking snapshot. (That is, calling @SnapshotSave with the blocking flag set.) This means it is possible to perform an orderly shutdown of a VoltDB database and ensure all data (including export data) is saved with the following procedure:

1. Put the database into admin mode with @Pause.
2. Perform a blocking snapshot with @SnapshotSave, saving both the database and any existing queued export data.
3. Shutdown the database with @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.

12.6. Using the Export Clients

VoltDB comes with two export clients:

- The export-to-file client
- The export-to-Hadoop client (Enterprise Edition only)

As the names imply, the export-to-file client writes the exported data to local files, while the export-to-Hadoop client uses Sqoop, the SQL importer for Hadoop from Cloudera, to write the exported data to a Hadoop distributed file system. Despite the different targets, the two export clients operate in much the same way.

12.6.1. How the Export Clients Work

When you start an export client, you specify the cluster nodes for the client to query for information (using the `--servers` argument). The client queries these nodes, one at a time, until it receives a response. Part of the response it receives is a description of the cluster, including a list of nodes and available ports. The client then creates connections to every node in the cluster using the port specified in the `--connect` argument (either the client port or the admin port).

Note that you don't have to specify all of the nodes of the cluster on the command line. You only have to specify one. The client then discovers the cluster configuration from the first node it reaches. However, you can specify multiple nodes in case one or more of the nodes is unavailable when the client starts.

Once the client connects to the cluster, it starts to poll and ack for export data. The client "decodes" the export stream from its serialized form into the appropriate datatypes for each column in the table. It then writes the data out to its appropriate target data form, whether files or a Hadoop distributed file system.

If the client loses connection to any of the nodes in the cluster (either because of a node failure or a shutdown), it disconnects from the cluster and repeats the initial discovery process, using the information it collected from the original connection. In other words, it will query every node in the cluster, one at a time, until it determines the new configuration. If the client cannot reach any of the nodes (for example, if the cluster is temporarily shut down) it will periodically retry the discovery process until the database cluster comes back online.

Once the cluster comes back, the client resumes export operations, picking up with the last data packet it received prior to the interruption This allows the export process to continue without operator intervention even across network disruptions, node failures, and database sessions.

All export clients use the preceding process. The following sections describe each export client and its operation in more detail.

12.7. The Export-to-File Client

The export-to-file client fetches the serialized data from the export connector and writes it out as text files (either comma or tab separated) to disk.

You start the export-to-file client using the Java command. In its simplest form, the command looks something like the following:

```
$ java -classpath "/home/me/voltdb/voltdb/*" \
       org.voltdb.exportclient.ExportToFileClient \
       --connect client \
       --servers myserver \
       --nonce ExportData \
       --type csv
```

12.7.1. Understanding the Export-to-File Client Output

When the export-to-file client receives export data, it de-serializes the content and writes it out to disk, 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 `--nonce`)
- 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, or post-processed as desired.

There are two main options when running the export-to-file client:

- The `--type` option lets you choose between comma-separated files (csv) or tab-delimited files (tsv).
- The `--batched` option tells the export client to group all of the files for one time period into a subfolder, rather than have all of the files in a single directory. In this case, when the export client "rolls" the files, it creates a new subfolder and it is the folder rather than the files that has the "active-" prefix appended to it.

Whatever options you choose, the order and representation of the content within the output files is the same. The export client 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.

12.7.2. The Export-to-File Client Command Line

The export-to-file client has a number of command line options that let you customize the export process to meet your needs. The complete syntax of the command line is as follows:

```
$ java   -classpath {path}   \
           org.voltdb.exportclient.ExportToFileClient   \
          {arguments...}

$ java   -classpath {path}   \
           org.voltdb.exportclient.ExportToFileClient   \
           --help
```

The supported arguments are:

--servers {host-name[:port]} [,...] A comma separated list of host names or IP addresses to query.

--nonce {text}	The prefix to use for the files that the client creates. The client creates a separate file for every table that is exported, constructing a file name that includes a transaction ID, the nonce, the name of the table, a timestamp, and a file type specified by the `--type` argument.
--connect {client|admin}	The port to connect to. You specify the type of port (client or admin), not the port number.
--type {csv|tsv}	The type of files to create. You can specify either csv (for comma-separated files) or tsv (for tab-delimited files).
--user {text}	The username to use for authenticating to the VoltDB server(s). Required only if security is enabled for the database.
--password {text}	The password to use for authenticating to the VoltDB server(s). Required only if security is enabled for the database. If you specify a username but not a password, the export client prompts you for the password.
--outdir {path}	(Optional.) The directory where the output files are created. If you do not specify an output path, the client writes the output files to the current default directory.
--period {integer}	(Optional.) The frequency, in minutes, for "rolling" the output file. The default frequency is 60 minutes.
--batched	(Optional.) Store the output files in subfolders that are "rolled" according to the frequency specified by `--period`. The subfolders are named according to the nonce and the timestamp, with "active-" prefixed to the subfolder currently being written.
--with-schema	(Optional.) Writes a JSON representation of each table's schema as part of the export. The primary output files of the export-to-file client contain the exported data in rows, but do not identify the datatype of each column. 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.
--delimiters {text}	(Optional.) Alternate delimiter characters for the 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.
--dateformat {format-string}	(Optional.) 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".
--skipinternals	(Optional.) Eliminates the six columns of VoltDB metadata (such as transaction ID and timestamp) from the output. If you specify `--skipinternals` the output files contain only the exported table data.

12.8. The Export-to-Hadoop Client (Enterprise Edition Only)

The export-to-Hadoop client fetches the serialized data from the export connector and — rather than writing it out as text files — exports it to the Hadoop distributed file system (hdfs) using the Sqoop import application from Cloudera. The export-to-Hadoop client is available in the VoltDB Enterprise Edition only.

To use the export-to-Hadoop client, you must have already installed and configured both Hadoop and Sqoop and started the Hadoop file system. The client uses the variable HADOOP_HOME to determine where Hadoop is installed and what file system to use as the target of the export. It then extracts and formats the exported data from the VoltDB connector and runs the Sqoop importer to read that data into Hadoop.

You start the export-to-Hadoop client using the Java command. However, it is important that all of the necessary JAR files for VoltDB, Hadoop, and Sqoop are in your class path. The easiest way to do this is using an Ant build script. But for demonstration purposes, the following example uses the export command to define CLASSPATH.

The command to start the export-to-Hadoop client looks something like the following:

```
$ V_PATH="/opt/voltdb/voltdb/*"
$ H_PATH="$HADOOP_HOME/*:$HADOOP_HOME/conf:$HADOOP_HOME/lib/*"
$ S_PATH="$SQOOP_HOME/*:$SQOOP_HOME/lib/*"
$ export CLASSPATH="$V_PATH:$H_PATH:$S_PATH"
$ java org.voltdb.hadoop.VoltDBSqoopExportClient \
       --connect client \
       --servers myserver \
       --nonce ExportData
```

12.8.1. The Export-to-Hadoop Client Command Line

The export-to-Hadoop client has a number of command line options that let you customize the export process to meet your needs. Many of the options are the same as for the export-to-file client. However, some options are specific to this client and allow you to control the Sqoop importer. In the following description the generic export client options are listed separately from the Sqoop-specific options.

The complete syntax of the command line is as follows:

```
$ java   -classpath {path}   \
          org.voltdb.hadoop.VoltDBSqoopExportClient   \
         {arguments...}

$ java   -classpath {path}   \
          org.voltdb.hadoop.VoltDBSqoopExportClient   \
          --help
```

The supported export client arguments are:

--servers {host-name[:port]} [,...]	A comma separated list of host names or IP addresses to query.
--nonce {text}	The prefix to use for the directories that the client creates in Hadoop.
--connect {client|admin}	The port to connect to. You specify the type of port (client or admin), not the port number.

--user {text}	The username to use for authenticating to the VoltDB server(s). Required only if security is enabled for the database.
--password {text}	The password to use for authenticating to the VoltDB server(s). Required only if security is enabled for the database. If you specify a username but not a password, the export client prompts you for the password.
--period {integer}	(Optional.) The frequency, in minutes, for "rolling" the output file. The default frequency is 60 minutes.
--outdir {path}	(Optional.) The directory where temporary files are written as part of the export/import process.
--delimiters {text}	(Optional.) Alternate delimiter characters for the 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.
--require-enclosed-output	(Optional.) Specifies that all CSV fields, even numeric and null fields, are enclosed (in quotation marks, by default).
--http-port {integer}	(Optional.) The http port for connecting to the Hadoop file system (port 8099 by default).
--skipinternals	(Optional.) Eliminates the six columns of VoltDB metadata (such as transaction ID and timestamp) from the output. If you specify `--skipinternals` the output files contain only the exported table data.

The following are supported Sqoop-specific arguments. These arguments are passed directly to the Sqoop importer interface and are documented in detail in the Sqoop documentation:

--hadoop-home {path}	(Optional.) Overrides the value of the HADOOP_HOME environment variable. You must provide a valid Hadoop home directory, either by defining HADOOP_HOME or specifying the location with --hadoop-home.
--verbose	(Optional.) Displays additional information during Sqoop processing.
--target-dir {path}	(Optional.) The destination directory in HDFS.
--warehouse-dir {path}	(Optional.) A parent directory in HDFS where separate folders are created for each table. The options `--target-dir` and `--warehouse-dir` are mutually exclusive.
--null-string {text}	(Optional.) The string to use for null string values in the output.
--null-non-string {text}	(Optional.) The string to use for null values in the output for all datatypes except strings.

Chapter 13. 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. 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/`.

13.1. 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*),

13.2. Creating the Logging Configuration File

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 docmentation 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.

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 13.1, “VoltDB Components for Logging” lists the loggers you can invoke.

Table 13.1. VoltDB Components for Logging

Logger	Description
ADHOCPLANNERTHREAD	Execution of ad hoc queries
AUTH	Authentication and authorization of clients
COMPILER	Interpretation of SQL in ad hoc queries
EXPORT	Exporting data

Logger	Description
HOST	Host specific events
NETWORK	Network events related to the database cluster
RECOVERY	Node recovery and rejoin
SQL	Execution of SQL statements

.

13.3. Enabling Logging for VoltDB

Once you create your Log4J configuration file, you start logging by specifying the configuration file on the command line when you start the VoltDB database. For example:

```
java -Djava.library.path=/opt/voltdb org.voltdb.VoltDB \
     -Dlog4j.configuration=./myLog4jConfig.xml \
     catalog mycatalog.jar
```

13.4. 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 14. 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
- Java (packaged with VoltDB)
- JDBC (packaged with VoltDB)
- JSON (packaged with VoltDB)
- 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.

14.1. C++ Client Interface

VoltDB provides a client interface for programs written in C++. The C++ client interface is available pre-compiled 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++.

14.1.1. 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.

14.1.2. 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.

14.1.3. 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);
```

14.1.4. 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()) {}
```

14.1.5. 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;
}
```

14.2. 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`.

14.2.1. 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);
```

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

[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.

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.

14.2.2. 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 words for "Hello" and "World" and the name of the language) 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("Pozdrav","Svijet","Croatian");
   $params = json_encode($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(["Pozdrav","Svijet","Croatian"])
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 = '["Pozdrav","Svijet","Croatian"]';
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 = "[\"Pozdrav\",\"Svijet\",\"Croatian\"]";
      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();

        }
      }
    }
  }
}
```

14.2.3. 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 milliseconds from the epoch.) Otherwise, quoted strings are interpreted as a string datatype.

Table 14.1, “Datatypes in the JSON Interface” summarizes how to pass different datatypes in the JSON interface.

Table 14.1. 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"

14.2.4. 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 14.1, “The Structure of the VoltDB JSON Response” shows the structure of the object returned by the JSON interface.

Figure 14.1. 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]
```

14.2.5. 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 stated 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.)

14.3. 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.

14.3.1. 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}".

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

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

14.3.2. 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 the VoltDB software jar is in the Java classpath or the application will not be able to find the driver class.

The following is a complete example that uses JDBC to access the Hello World tutorial described in *Getting Started With VoltDB* and 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.

This is not intended as a complete description of the SQL DDL. Instead, it summarizes the subset of standard SQL DDL statements that are allowed in a VoltDB schema definition and any exceptions or limitations that application developers should be aware of.

The supported SQL DDL statements are:

- CREATE INDEX
- CREATE TABLE
- CREATE VIEW

CREATE INDEX

CREATE INDEX — Creates an index for faster access to a table.

Syntax

```
CREATE INDEX index-name ON table-name ( column-name [,...])
```

Description

Creating an index on a table makes read access to the table faster when using the index as a key. Note that VoltDB creates an index automatically when you specify a primary key in the CREATE TABLE statement.

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.

CREATE TABLE

CREATE TABLE — Creates a table in the database.

Syntax

```
CREATE TABLE table-name (
        column-definition [,...]
        [, key-definition [,...]]

);

column-definition: column-name datatype [UNIQUE] [DEFAULT value ] [ NULL | NOT NULL ]

key-definition: [CONSTRAINT constraint-name] PRIMARY KEY (column-name [,...])
```

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
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
VARBINARY()	byte array	Variable length binary string (sometimes referred to as a "blob")
TIMESTAMP	long, VoltDB TimestampType	Time in microseconds

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.

- 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 project definition file and schema.
- When you specify a primary key, 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.
- For integer and floating-point datatypes, the largest possible negative value is reserved by VoltDB and interpreted as null.
- The VARBINARY datatype provides variable storage for arbitrary strings of binary data and operates similarly to VARCHAR 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.

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 table-column-name [,...] COUNT(*) AS alias [, aggregate [ ,... ]]
        FROM table-name
        GROUP BY column-name [,...]
```

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 updated. 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.
  - The first columns listed in the GROUP BY must match the initial columns in the SELECT statement in both name and order.
  - SELECT must include a field specified as COUNT(*). Other aggregates (SUM or COUNT) are allowed following the COUNT(*).

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
- UPDATE

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 a new row in the database, using the specified values for the columns.

Syntax

INSERT INTO *table-name* [(*column-name* [,...])] VALUES (*value-expression* [,...])

Description

The INSERT statement creates a new row in the database. If you specify the column names, 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.

You can specify a subset of the columns in the table by specifying the column names and their desired values. However, 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.

VoltDB supports the following arithmetic operators in expressions: addition (+), subtraction (-), multiplication (*), and division (*).

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 [ TOP integer-value ]
        { * | [ ALL | DISTINCT ] { column-name | selection-expression } [AS alias] [,...] }
        FROM { table-name | view-name } [AS alias] [,...]
        [WHERE [NOT] boolean-expression [ {AND | OR} [NOT] boolean-expression]...]
        [clause...]

clause:
        ORDER BY { column-name | alias } [ ASC | DESC ] [,...]
        GROUP BY { column-name | alias } [,...]
        LIMIT { integer-value [OFFSET row-count] | ALL }
```

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 limitations are important to note when using the SELECT statement with VoltDB:

- VoltDB supports the following functions in the selection expression: AVG, COUNT, MAX, MIN, and SUM.
- VoltDB supports the following arithmetic operators in expressions: addition (+), subtraction (-), multiplication (*), and division (*).
- `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 (<=), 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.
- VoltDB supports inner joins only.
- 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.

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 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 project definition file 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).

Project Definition File:

```
<procedures>
   <procedure class="EmpByLimit"
              partitioninfo="Employee.company:0">
       <sql>SELECT lastname, firstname FROM employee
            WHERE company = ?
            ORDER BY lastname ASC, firstname ASC
            LIMIT 500 OFFSET ?;</sql>
   </procedure>
       . . .
</procedures>
```

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;
}
```

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;
```

Appendix C. Project Definition File (project.xml)

The project definition file (frequently referred to as project.xml), describes the structure of a VoltDB database, including the location of the database schema and stored procedures, the partitioning keys for individual tables, and other characteristics. This appendix describes the syntax for each of these components within the project definition file.

C.1. Understanding XML Syntax

The project definition 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:

```
<project>
   <database>
   </database>
</project>
```

Elements can be nested. In the preceding example `database` is a child of the element `project`.

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 database element is assigned a value of "database" for its name attribute:

```
<project>
   <database name="database">
   </database>
</project>
```

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 `procedure` tags use this form of shorthand:

```
<project>
   <database name="database">
      <procedures>
         <procedure class="procs.myproc1" />
         <procedure class="procs.myproc2" />
      </procedures>
   </database>
</project>
```

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

C.2. The Structure of the Project Definition File

Because the project definition file is a standard XML file, you should always start the file with the XML declaration. The root element of the definition file is the project element, so the next line of the definition

file should start the project element. The remainder of the definition consists of elements that are children of the project element.

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

Figure C.1. Project Definition XML Structure

```
<project>
    <info>
        <name>
        <description>
        <version>
    </info>
    <security/>
    <database>
        <schemas>
            <schema/>...
        </schemas>
       <groups>
            <group/>...
        </groups>
        <procedures>
            <procedure/>...
            <procedure>
                <sql>
            </procedure>...
        </procedures>
        <partitions>
            <partition/>...
        </partitions>
        <export>
            <tables>
                <table>...
            </tables>
        </export>
    </database>
</project>
```

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

Table C.1. Project Definition File Elements and Attributes

Element	Child of	Parent of	Attributes
project*	(root element)	security, database	
info	project	name, description, version	
name*	info		
description*	info		
version*	info		
security	project		enabled={true|false}

Element	Child of	Parent of	Attributes
database*	project	groups, procedure, partition, snapshot	name={text}
schemas*	database	schema	
schema*	schemas		path={file-spec}*
groups	database	group	
group	groups		name={text}* adhoc={true|false} sysproc={true|false}
procedures	database	procedure	
procedure	procedures	sql	class={text}* groups={group-name[,..]} partitioninfo={text}
sql	procedure		joinorder={table-name[,...]}
partitions	database	partition	
partition	partitions		table={text}* column={text}*
export	database	tables	groups={group-name[,..]}
tables	export	table	
table	tables		name={text}*

*Required

Appendix D. Configuration File (deployment.xml)

The configuration file (frequently referred to as deployment), describes the configuration of a VoltDB database cluster at runtime, including the number of hosts in the cluster, the number of sites per hosts, the lead node when starting the cluster, as well as other configuration information such as the identification of allowed users and passwords. This appendix describes the syntax for each of these components within the configuration file.

The configuration file is a fully-conformant XML file. If you are unfamiliar with XML, see Section C.1, “Understanding XML Syntax” for a brief explanation of XML syntax.

D.1. The Structure of the Configuration File

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

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

Figure D.1. Configuration XML Structure

```
<deployment>
    <cluster/>
    <paths>
        <commandlog/>
        <commandlogsnapshot/>
        <exportoverflow/>
        <snapshots/>
        <voltdbroot/>
    </path>
    <admin-mode/>
    <heartbeat/>
    <partition-detection>
        <snapshot/>
    </partition-detection>
    <httpd>
        <jsonapi/>
    </httpd>
    <commandlog>
        <frequency/>
    <commandlog/>
    <export/>
    <snapshot/>
    <users>
        <user/>...
    </users>
    <systemsettings>
        <temptables/>
        <snapshot/>
    </systemsettings>
</deployment>
```

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

Table D.1. Configuration File Elements and Attributes

Element	Child of	Parent of	Attributes
deployment*	(root element)	admin-mode, commandlog, cluster, export, heartbeat, httpd, partition-detection, paths, snapshot, systemsettings, users	
cluster*	deployment		hostcount={int} sitesperhost={int} kfactor={int}
admin-mode	deployment		port={int} adminstartup={true|false}
heartbeat	deployment		timeout={int}*
partition-detection	deployment	snapshot	enabled={true|false}
snapshot*	partition-detection		prefix={text}*

Element	Child of	Parent of	Attributes
commandlog	deployment	frequency	enabled={true|false} synchronous={true|false} logsize={int}
frequency	commandlog		time={int} transactions={int}
export	deployment		enabled={true|false}
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}*
snapshot	deployment		frequency={int}{s|m|h}* prefix={text}* retain={int}* enabled={true|false}
systemsettings	deployment	snapshot, temptables	
snapshot	systemsettings		priority={int}*
temptables	systemsettings		maxsize={int}*
users	deployment	user	
user	users		name={text}* password={text}* groups={group-name[,..]}

*Required

Appendix E. System Procedures

VoltDB provides system procedures that client applications can call to perform certain system-wide administrative functions. You invoke system procedures the same way you invoke other stored procedures, using the VoltDB client method callProcedure.

This appendix describes the following system procedures.

- @AdHoc
- @Pause
- @Quiesce
- @Resume
- @Shutdown
- @SnapshotDelete
- @SnapshotRestore
- @SnapshotSave
- @SnapshotScan
- @SnapshotStatus
- @Statistics
- @SystemCatalog
- @SystemInformation
- @UpdateApplicationCatalog
- @UpdateLogging

@AdHoc

@AdHoc — Executes an SQL statement specified at runtime.

Syntax

ClientResponse client.callProcedure("@AdHoc", String *SQL-statement*)

Description

The @AdHoc system procedure lets you perform arbitrary SQL queries on a running VoltDB database.

Note that ad hoc queries are not optimized — they are executed as multi-partition transactions — and may impact performance of other stored procedures. Therefore, use of @AdHoc is not recommended for frequent or repetitive queries.

Return Values

Returns one VoltTable 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.

Example

The following 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();
}
```

@Pause

@Pause — Initiates admin mode on the cluster.

Syntax

```
ClientResponse client.callProcedure("@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 application are not removed.
- However, if export is enabled when admin mode is invoked, any connection from an export client to the client port is terminated and further connections refused. The export client will attempt to reconnect until normal operations are resumed and the connection is re-established. Export clients connected to the admin port are not affected.
- 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.

Example

The following example, if called through the admin port, initiates admin mode on the database cluster.

```
client.callProcedure("@Pause");
```

@Quiesce

@Quiesce — Waits for all queued export data to be written to the connector.

Syntax

```
ClientResponse client.callProcedure("@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.

Example

The following 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

```
ClientResponse client.callProcedure("@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. If any export clients connected to the client port were disconnected by @Pause, they will automatically reconnect and restart export processing once @Resume restores normal operation.

@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.

Example

The following example uses @Resume to return the cluster to normal operation.

```
client.callProcedure("@Resume");
```

@Shutdown

@Shutdown — Shuts down the database.

Syntax

```
ClientResponse client.callProcedure("@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 (Enterprise Edition only) 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.

Example

The following 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 snapshots.

Syntax

```
ClientResponse client.callProcedure("@SnapshotDelete", String[] directory-paths, String[]
        Unique-IDs)
```

Description

The @SnapshotDelete system procedure deletes 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 additional 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.

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.

Syntax

```
ClientResponse client.callProcedure("@SnapshotRestore", String directory-path, String unique-ID)
```

Description

The @SnapshotRestore system procedure restores a previously saved database from disk to memory. This request is propagated to all nodes of the cluster, so a single call to @SnashotRestore will restore the entire database cluster.

The second parameter, *directory-path*, specifies where VoltDB looks for the snapshot files.

The third argument, *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.

Example

The following example uses @SnapshotRestore to restore previously saved database content from the path `/tmp/voltdb/backup/` using the unique identifier *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 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

```
ClientResponse client.callProcedure("@SnapshotSave", String directory-path, String unique-ID,
        Integer blocking-flag)
```

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. The snapshot consists of multiple files saved to the directory specified by *directory-path* using *unique-ID* as a filename prefix.

The fourth parameter, *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.

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.

See Chapter 9, *Saving & Restoring a VoltDB Database* for more information about saving and restoring VoltDB databases.

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.

Name	Datatype	Description
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.

Example

The following example uses @SnapshotSave to save the current database content to the path `/tmp/voltdb/backup/` using the unique identifier *flight* on each node of the cluster.

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 example 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 snapshots in a given directory path.

Syntax

```
ClientResponse client.callProcedure("@SnapshotScan", String directory-path)
```

Description

The @SnapshotScan system procedure provides information about any 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).

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 column 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.
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.

Example

The following example uses @SnapshotScan to list information about the snapshots in the directory /tmp/voltdb/backup/.

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).

```
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());
}
```

@SnapshotStatus

@SnapshotStatus — Lists information about the most recent snapshots created from the current database.

Syntax

```
ClientResponse client.callProcedure("@SnapshotStatus")
```

Description

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.

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).
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").

Example

The following example uses @SnapshotStatus to display information about most recent snapshots performed on the current database.

```
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

```
ClientResponse client.callProcedure("@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 following are the allowable values of *component*:

"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.
"PROCEDURE"	Returns information on the usage of stored procedures for each site within the database cluster. 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.
"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.

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.)

Return Values

Returns a different VoltTable 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 all seven VoltTables.)

INDEX — Returns a row for every index in every execution site.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	BIGINT	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	BIGINT	Numeric ID of the execution site on the host node.
PARTITION_ID	BIGINT	The numeric ID for the logical partition that this site represents. When using a K value greater than zero, there are multiple copies of each logical partition.
INDEX_NAME	STRING	The name of the index.
TABLE_NAME	STRING	The name of the database table to which the index applies.
INDEX_TYPE	STRING	A text string identifying the type of the index as either a hash or tree index and whether it is unique or not. Possible values include the followiing: CompactingHashMultiMapIndex CompactingHashUniqueIndex CompactingTreeMultiMapIndex CompactingTreeUniqueIndex
IS_UNIQUE	TINYINT	A byte value specifying whether the index is unique (1) or not (0).
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).

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.
CONNECTION_ID	INTEGER	Numeric ID of the client connection invoking the procedure.
CONNECTION_ HOSTNAME	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.
AVE_EXECUTION_TIME	INTEGER	The average length of time (in milliseconds) it took to execute the stored procedure.
MIN_EXECUTION_TIME	INTEGER	The minimum length of time (in milliseconds) it took to execute the stored procedure.
MAX_EXECUTION_TIME	INTEGER	The maximum length of time (in milliseconds) it took to execute the stored procedure.
ABORTS	BIGINT	The number of times the procedure was aborted.
FAILURES	BIGINT	The number of times the procedure failed unexpectedly. (As opposed to user aborts or expected errors, such as constraint violations.)

IOSTATS — Returns one row for every client connection on the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
CONNECTION_ID	BIGINT	Numeric ID of the client connection invoking the procedure.
CONNECTION_ 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).

Name	Datatype	Description
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 megabytes) allocated for tasks other than database records, indexes, and strings. (For example, pooled memory is used for temporary tables while processing stored procedures.)

PARTITIONCOUNT — Returns one row with one column.

Name	Datatype	Description
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.

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.
AVE_EXECUTION_TIME	BIGINT	The average length of time (in nanoseconds) it took to execute the stored procedure.
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.
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.)

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	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.
TABLE_NAME	STRING	The name of the database table.

Name	Datatype	Description
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.
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).
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.

Examples

The following example uses @Statistics to gather information about the distribution of table rows within the cluster. The example uses the toString() method of VoltTable to display the results of the procedure call.

```
try {
     VoltTable[] results = client.callProcedure("@Statistics",
                                    "TABLE",0).getResults();
     for (VoltTable t: results) { System.out.println(t.toString()); }
}
catch (Exception e) {
     e.printStackTrace();
}
```

The following 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);
}
```

@SystemCatalog

@SystemCatalog — Returns metadata about the database schema

Syntax

```
ClientResponse client.callProcedure("@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.

Name	Datatype	Description
TYPE_CAT	STRING	Unused.
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.

Name	Datatype	Description
SCOPE_TABLE	STRING	Unused.
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.

Name	Datatype	Description
RESERVED1	STRING	Unused.
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.

Name	Datatype	Description
ORDINAL_POSITION	INTEGER	An index specifying the position in the parameter list for the procedure, starting at 1.
IS_NULLABLE	STRING	Unused.
SPECIFIC_NAME	STRING	Same as COLUMN_NAME

Example

The following 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

```
ClientResponse client.callProcedure("@SystemInformation")

ClientResponse client.callProcedure("@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.

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.

Example

The following 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 currently in use.

Syntax

```
ClientResponse client.callProcedure("@UpdateApplicationCatalog", byte[] catalog, String
        deployment)
```

Description

The @UpdateApplicationCatalog system procedure lets make the following modifications to a running database without having to shutdown and restart:

- Add, remove, or modify stored procedures
- Add or remove database tables to the schema
- 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)

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.

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.

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*)

Example

The following example uses the @UpdateApplicationCatalog directly 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

```
ClientResponse client.callProcedure("@UpdateLogging", String 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.

Example

The following example shows one 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();
}
```