VOLTDB

Guide to Performance and Customization

Abstract

This book explains how to optimize application performance and customize database features using VoltDB.

V9.3

Guide to Performance and Customization

V9.3

This document was generated on August 12, 2020.

Table of Contents

Preface viii
1. Organization of this Manual viii
2. Other Resources viii
1. Introduction 1
1.1. What Affects Performance? 1
1.2. How to Use This Book 1
2. Hello, World! Revisited 3
2.1. Optimizing your Application for VoltDB 3
2.2. Applying Hello World to a Practical Problem 3
2.3. Partitioned vs. Replicated Tables 4
2.3.1. Defining the Partitioning Column 4
2.3.2. Creating the Stored Procedures 5
2.4. Using Asynchronous Stored Procedure Calls 6
2.4.1. Understanding Asynchronous Programming 7
2.4.2. The Callback Procedure 8
2.4.3. Making an Asynchronous Procedure Call 9
2.5. Connecting to all Servers 9
2.6. Putting it All Together 10
2.7. Next Steps 12
3. Understanding VoltDB Execution Plans 13
3.1. How VoltDB Selects Execution Plans for Individual SQL Statements 13
3.2. Understanding VoltDB Execution Plans 13
3.3. Reading the Execution Plan and Optimizing Your SQL Statements 14
3.3.1. Evaluating the Use of Indexes 15
3.3.2. Evaluating the Table Order for Joins 17
4. Using Indexes Effectively 19
4.1. Basic Principles for Effective Indexing 19
4.2. Defining Indexes 20
4.3. The Goals for Effective Indexing 20
4.4. How Indexes Work 21
4.5. Summary 22
5. Creating Flexible Schemas With JSON 24
5.1. Using JSON Data Structures as VoltDB Content 24
5.2. Querying JSON Data 25
5.3. Updating JSON Data 26
5.4. Indexing JSON Fields 26
5.5. Summary: Using JSON in VoltDB 27
6. Creating Geospatial Applications 28
6.1. The Geospatial Datatypes 28
6.1.1. The GEOGRAPHY_POINT Datatype 28
6.1.2. The GEOGRAPHY Datatype 28
6.1.3. Sizing GEOGRAPHY Columns 29
6.1.4. How Geospatial Values are Interpreted 30
6.2. Entering Geospatial Data 30
6.3. Working With Geospatial Data 31
6.3.1. Working With Locations 32
6.3.2. Working With Regions 33
7. Creating Custom Importers, Exporters, and Formatters 35
7.1. Writing a Custom Exporter 35
7.1.1. The Structure and Workflow of the Export Client 35
7.1.2. How to Use Custom Properties to Configure the Client 37

7.1.3. How to Compile and Install the Client 37
7.1.4. How to Configure the Export Client 38
7.2. Writing a Custom Importer 38
7.2.1. Designing and Coding a Custom Importer 39
7.2.2. Packaging and Installing a Custom Importer 40
7.2.3. Configuring and Running a Custom Importer 41
7.3. Writing a Custom Formatter 42
7.3.1. The Structure of the Custom Formatter 42
7.3.2. Compiling and Packaging Custom Formatter Bundles 44
7.3.3. Installing and Invoking Custom Formatters 45
7.3.4. Using Custom Formatters With the kafkaloader Utility 46
8. Creating Custom SQL Functions 48
8.1. Writing a User-Defined Scalar Function 48
8.2. Writing a User-Defined Aggregate Function 49
8.3. Loading User-Defined Functions into the Database 51
8.4. Declaring a User-Defined Function 51
8.5. Invoking User-Defined Functions in SQL Statements 52
9. Creating Custom Tasks 53
9.1. Overview of How Custom Tasks Work 53
9.2. Modifying the Procedure Call and Arguments 54
9.2.1. Designing a Java Class That Implements ActionGenerator 54
9.2.2. Compiling and Loading the Class into VoltDB 57
9.2.3. Declaring the Task 57
9.3. Modifying the Interval Between Invocations 57
9.3.1. Designing a Java Class That Implements IntervalGenerator 58
9.3.2. Compiling and Loading the Class into VoltDB 60
9.3.3. Declaring the Task 60
9.4. Modifying Both the Procedure and Interval 60
9.4.1. Designing a Java Class That Implements ActionScheduler 61
9.4.2. Compiling and Loading the Class into VoltDB 63
9.4.3. Declaring the Task 64
10. Understanding VoltDB Memory Usage 65
10.1. How VoltDB Uses Memory 65
10.2. Actions that Impact Memory Usage 66
10.3. How VoltDB Manages Memory 68
10.4. How Memory is Allocated and Deallocated 69
10.5. Controlling How Memory is Allocated 69
10.6. Understanding Memory Usage for Specific Applications 70
11. Managing Time 72
11.1. The Importance of Time 72
11.2. Using NTP to Manage Time 72
11.2.1. Basic Configuration 72
11.2.2. Troubleshooting Issues with Time 73
11.2.3. Correcting Common Problems with Time 73
11.2.4. Example NTP Configuration 74
11.3. Configuring NTP in a Hosted, Virtual, or Cloud Environment 75
11.3.1. Considerations for Hosted Environments 76
11.3.2. Considerations for Virtual and Cloud Environments 76

List of Figures

2.1. Synchronous Procedure Calls 7
2.2. Asynchronous Procedure Calls 7
7.1. Structure of the Custom Export Class 36
10.1. The Three Types of Memory in VoltDB 66
10.2. Details of Memory Usage During and After an SQL Statement 67
10.3. Controlling the Java Heap Size 70

List of Tables

7.1. Structure of the Custom Importer 38

List of Examples

9.1. Custom Task Implementing ActionGenerator 56
9.2. Custom Task Implementing IntervalGenerator 59
9.3. Custom Task Implementing ActionScheduler 62
11.1. Custom NTP Configuration File 75

Preface

This book provides details on using VoltDB to optimize the performance of your database application as well as customize selective features of the VoltDB product. Other books — specifically the *VoltDB Tutorial* and *Using VoltDB* — describe the basic features of VoltDB and how to use them. However, creating an optimized application requires using those features in the right combination and in the appropriate context. What features you use and how depends on your specific application needs. This manual provides advice on those decisions.

1. Organization of this Manual

This book is divided into 11 chapters:

- Chapter 1, *Introduction*
- Chapter 2, *Hello, World! Revisited*
- Chapter 3, *Understanding VoltDB Execution Plans*
- Chapter 4, *Using Indexes Effectively*
- Chapter 5, *Creating Flexible Schemas With JSON*
- Chapter 6, *Creating Geospatial Applications*
- Chapter 7, *Creating Custom Importers, Exporters, and Formatters*
- Chapter 8, *Creating Custom SQL Functions*
- Chapter 9, *Creating Custom Tasks*
- Chapter 10, *Understanding VoltDB Memory Usage*
- Chapter 11, *Managing Time*

2. Other Resources

This book provides recommendations for optimizing VoltDB applications and customizing database features. It assumes you are already familiar with VoltDB and its features. If you are new to VoltDB, we suggest you read the following books first:

- *VoltDB Tutorial* provides a quick introduction to the product and is recommended for new users.
- *VoltDB Planning Guide* provides guidance for evaluating and sizing VoltDB implementations.
- *Using VoltDB* provides a complete reference to the features and functions of the VoltDB product.
- *VoltDB Administrator's Guide* provides information for system operators on setting up and managing VoltDB databases and the clusters that host them.

These books and more resources are available on the web from http://docs.voltdb.com/.

Chapter 1. Introduction

VoltDB is a best-in-class database designed specifically for high volume transactional applications. Other books describe the individual features and functions of VoltDB. However, getting the most out of any technology is not just a matter of features; it is using the features effectively, in the right combination, and in the right context.

The goal of this book is to explain how to achieve maximum performance using VoltDB. Performance is affected by many different factors, including:

- The design of your database and its stored procedures
- The client applications
- The configuration of the servers that run the database
- The network that connects the servers

Understanding the impact of each factor and the relationship between them can help you both design better solutions and detect and correct problems in a running system. However, first you must understand the product itself. If you are new to VoltDB, it is strongly recommended that you read *VoltDB Tutorial* and *Using VoltDB* before reading this book.

1.1. What Affects Performance?

There is no single factor that drives performance or even a single definition for what constitutes "good" performance. VoltDB is designed to provide exceptional throughput and much of this book is dedicated to an explanation of how you can maximize throughput in your application design and hardware configuration.

However, another aspect of performance that is equally important to database applications is durability: resilience against — and ability to recover from — hardware failures and other error conditions. VoltDB has features that enhance database durability. However, these features have their own requirements, particularly on system sizing and configuration.

All applications are different. There is no single combination of application design, hardware configuration, or database features that can satisfy them all. Your specific requirements drive the trade offs that need to be made concerning how you configure the database system as a whole. The goal of this book is to provide you with the facts you need to make an informed decision about those trade offs.

1.2. How to Use This Book

This book is divided into ten more chapters:

- The beginning of the book (chapters 2 and 4) explains how to design your database schema, stored procedures, and client applications for maximum performance.
- Chapter 5 explains how to accommodate flexibility in the schema design through the use of JSON columns and indexes.
- Chapter 6 explains how to use VoltDB's geospatial capabilities for applications that need to combine standard database content with location-specific information such geographic points and shapes.

- Chapter 7 explains how to create custom connectors for import and export, as well as formatters to be used by importers.
- Chapters 8 explains how to define your own scalar and aggregate functions for use in SQL statements.
- Chapters 9 explains how to implement custom tasks that execute stored procedures on a specified schedule.
- Chapter 10 explains in detail how memory is used by the VoltDB server process.
- Chapter 11 provides guidelines for configuring time services when running a VoltDB cluster.

Chapter 2. Hello, World! Revisited

The VoltDB software kit includes a Hello World example in the directory `/doc/tutorials/helloworld` that shows you how to create a simple VoltDB application, including a schema, stored procedures, and a client application. However, storing five records and doing a single SELECT is not a terribly interesting database application.

VoltDB is designed to process hundreds of thousands of transactions a second, providing unparalleled throughput. Hello World does little to demonstrate that. But perhaps we can change it a bit to better emulate real world situations and, in the process, learn how to write applications that maximize the power of VoltDB.

2.1. Optimizing your Application for VoltDB

VoltDB can be used generically like any other database to insert, select, and update records. But VoltDB also specializes in:

- Scalability
- Throughput performance
- Durability

Durability is built into the VoltDB database server software through several different functions, including snapshots, K-Safety, and command logging, features that are described in more detail in the *Using VoltDB* manual. Scalability and throughput are related to server configuration (e.g. number of servers, memory capacity, etc.). However, there are several things that can be done in the design of the database and the client application to maximize the throughput on any cluster. In particular, this update to the Hello World tutorial focuses on designing your application to take advantage of:

- Partitioned and replicated tables
- Asynchronous stored procedure calls
- Client connections to all nodes in the database cluster

2.2. Applying Hello World to a Practical Problem

The problem with Hello World is that it doesn't match any real problem, and certainly not one that VoltDB is designed to solve. However, it is not too hard to think of a practical problem where saying hello could be useful.

Let's assume we run a system (a website, for example) where users register and log in to use services. We want to acknowledge when a user logs in by saying hello. Let's further assume that our system is global in nature. It would be nice if we could say hello in the user's native language.

To support our new user sign in process, we need to store the different ways of saying hello in each language and we need to record the native language of the user. Then, when they sign in, we can retrieve their name and the appropriate word for hello.

This means we need two tables, one for the word "hello" in different languages and one for the users. We can reuse the HELLOWORLD table from our original application for the first table. But we need to add a

table for user data, including a unique identifier, the user's name, and their language. Often, the best and easiest unique identifier for an online account is the user's email address. So that is what we will use. Our schema now looks like this:

```
CREATE TABLE HELLOWORLD (
   HELLO VARCHAR(15),
   WORLD VARCHAR(15),
   DIALECT VARCHAR(15) NOT NULL,
   PRIMARY KEY (DIALECT)
);

CREATE TABLE USERACCOUNT (
   EMAIL VARCHAR(128) UNIQUE NOT NULL,
   FIRSTNAME VARCHAR(15),
   LASTNAME VARCHAR(15),
   LASTLOGIN TIMESTAMP,
   DIALECT VARCHAR(15) NOT NULL,
   PRIMARY KEY (EMAIL)
);
```

Oh, by the way, now that you know how to write a basic VoltDB application, you don't need to type in the sample code yourself anymore. You can concentrate on understanding the nuances that make VoltDB applications exceptional. The complete sources for the updated Hello World example are available in the `doc/tutorials/helloworldrevisited` subfolder when you install the VoltDB software.

2.3. Partitioned vs. Replicated Tables

In the original Hello World example, we partitioned the HELLOWORLD table on dialect to demonstrate partitioning, which is a key concept for VoltDB. However, there are only so many languages in the world, and the words for "hello" and "world" are not likely to change frequently. In other words, the HELLOWORLD table is both small and primarily read-only.

Not all tables need to be partitioned. If a table is small and updated infrequently, it can be *replicated*. Copies of a replicated table are stored on every server. This means that the tables can only be updated with a multi-partition procedure (which is why you shouldn't replicate write-intensive tables). However, replicated tables can be read from any single-partitioned procedure since there is a copy available to every partition.

HELLOWORLD is an ideal candidate for replication, so we will replicate it in this iteration of the Hello World application.

USERACCOUNT, on the other hand, is write-intensive. The table is updated every time a user signs in and the record count increases as new users register with the system. Therefore, it is important that we partition this table.

2.3.1. Defining the Partitioning Column

The partitioning column needs to support the key access methods for the table. In the case of registered users, the table is accessed via the user's unique ID, their email address, when the user signs in. So we will define the EMAIL column as the partitioning column for the table.

The choice of partitioning column is defined in the database schema. If a table is not listed as being partitioned, it becomes a replicated table by default. So for the updated Hello World example,

you can remove the PARTITION TABLE statement for the HELLOWORLD table and add one for USERACCOUNT. The updated schema contains the following PARTITION TABLE statement:

```
PARTITION TABLE USERACCOUNT ON COLUMN EMAIL;
```

2.3.2. Creating the Stored Procedures

For the sake of demonstration, we only need three stored procedures for our rewrite of Hello World:

- Insert Language — Loads the HELLOWORLD table, just as in the original Hello World tutorial.
- Register User — Creates a new USERACCOUNT record.
- Sign In — Performs the bulk of the work, looking up the user, recording their sign in, and looking up the correct word for saying hello.

2.3.2.1. Loading the Replicated Table

To load the HELLOWORLD table, we can reuse the Insert stored procedure from our original Hello World example. The only change we need to make is, because HELLOWORLD is now a replicated table, remove the PARTITION ON clause from the CREATE PROCEDURE statement that defines the Insert procedure in the schema DDL.

2.3.2.2. Registering New Users

To add a new user to the system, the RegisterUser stored procedure needs to add the user's name, language, and their email address as the unique identifier for the USERACCOUNT table.

Creating a new record can be done with a single INSERT statement. In this way, the RegisterUser procedure is very similar to the Insert procedure for the HELLOWORLD table. The RegisterUser procedure looks like this:

```
CREATE PROCEDURE RegisterUser
    AS INSERT INTO USERACCOUNT
      (Email, Firstname, Lastname, Dialect)
      VALUES (?,?,?,?);
```

The difference is that RegisterUser can and should be single-partitioned so it does not unnecessarily tie up multiple partitions. Since the table is partitioned on the EMAIL column, the CREATE PROCEDURE statement should include a PARTITION ON clause like so:

```
CREATE PROCEDURE RegisterUser
    PARTITION ON TABLE Useraccount COLUMN Email
    AS INSERT INTO USERACCOUNT
      (Email, Firstname, Lastname, Dialect)
      VALUES (?,?,?,?);
```

2.3.2.3. Signing In

Finally, we need a procedure to sign in the user and retrieve the word for "hello" in their native language. The key goal for this procedure, since it will be invoked more frequently than any other, is to be performant. To ensure the highest throughput, the procedure needs to be single-partitioned.

The user provides their email address as the unique ID when they log in, so we can make the procedure single-partitioned, specifying the email address as the partitioning value. Within the procedure itself we perform two actions:

- Join the USERACCOUNT and HELLOWORLD tables based on the Dialect column to retrieve both the user's name and the appropriate word for "hello"
- Update the user's record with the latest login timestamp.

Because this stored procedure uses two queries, we can write the stored procedure logic as a Java class. (See *Using VoltDB* for details on writing Java stored procedures.) We could write custom code to check the return values from the join of the two tables to ensure that an appropriate user record was found. However, VoltDB provides predefined *expectations* for many common query conditions. We can take advantage of one of these expectations, EXPECTS_ONE_ROW, to verify that we get the results we want. If the first query, getuser, does not return one row (for example, if no user record is found), VoltDB aborts the procedure and notifies the calling program that a rollback has occurred.

Expectations provide a way to simplify and standardize error handling in your stored procedures. See the chapter on simplifying application coding in the *Using VoltDB* manual for more information.

The resulting SignIn procedure is as follows:

```
import org.voltdb.*;

public class SignIn extends VoltProcedure {

  public final SQLStmt getuser = new SQLStmt(
      "SELECT H.HELLO, U.FIRSTNAME " +
      "FROM USERACCOUNT AS U, HELLOWORLD AS H " +
      "WHERE U.EMAIL = ? AND U.DIALECT = H.DIALECT;"
  );
  public final SQLStmt updatesignin = new SQLStmt(
      "UPDATE USERACCOUNT SET lastlogin=? " +
      "WHERE EMAIL = ?;"
  );

  public VoltTable[] run( String id, long signintime)
      throws VoltAbortException {
          voltQueueSQL( getuser, EXPECT_ONE_ROW, id );
          voltQueueSQL( updatesignin,  signintime, id );
          return voltExecuteSQL();
      }
}
```

We also want to declare the procedure and define how it is partitioned in the schema DDL. To do that, we add the following statement to our schema file:

```
CREATE PROCEDURE
   PARTITION ON TABLE Useraccount COLUMN Email
   FROM CLASS SignIn;
```

2.4. Using Asynchronous Stored Procedure Calls

Now we are ready to write the client application. There are two key aspects to taking full advantage of VoltDB in your client applications. One is make connections to all nodes on the cluster, which we will discuss shortly. The other is to use asynchronous stored procedure calls.

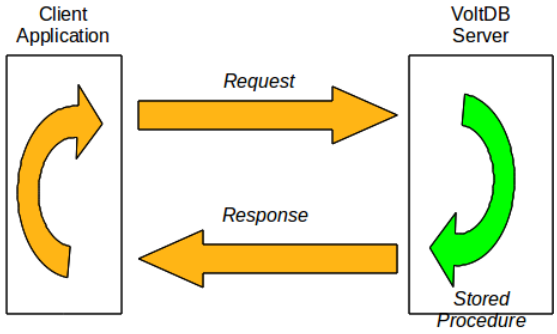
You can call VoltDB stored procedures either synchronously or asynchronously. When you call a stored procedure synchronously, your client application waits for the call to be processed before continuing. If you call a procedure asynchronously, your application continues processing once the call has been initiated. Once the procedure is complete, your application is notified through a callback procedure.

2.4.1. Understanding Asynchronous Programming

Synchronous calls are easy to understand because all processing is linear; your application waits for the query results. However, after VoltDB processes a transaction — between when VoltDB sends back the results, your application handles the results, initiates a new procedure call, and the call reaches the VoltDB server — the VoltDB database has no work to do (assuming there is only one client application). In this situation whether the stored procedures are single- or multi-partitioned doesn't matter, since you are only ever asking the cluster to process one procedure at a time.

As shown in Figure 2.1, “Synchronous Procedure Calls”, more time can be spent in the round trip between transactions (shown in yellow) than in processing the stored procedures themselves.

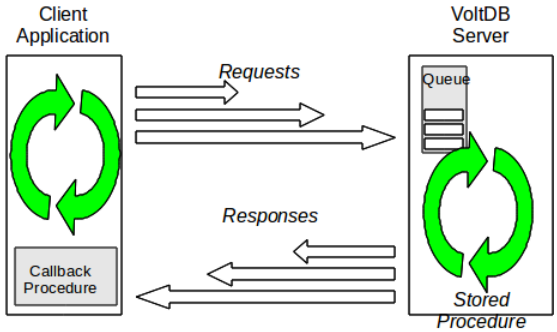
Figure 2.1. Synchronous Procedure Calls



What you would like to do is queue up as much work (i.e. transactions) as possible so the database always has work to do as soon as each transaction is complete. This is what asynchronous stored procedure calls do.

As soon as an asynchronous call is initiated, your application continues processing, including making additional asynchronous calls. These calls are queued up on the servers and processed in the order they are received. Once a stored procedure is processed, the results are returned to the calling application and the next queued transaction started. As Figure 2.2, “Asynchronous Procedure Calls” shows, the database does not need to wait for the next procedure request, it simply takes the next entry off the queue as soon as the current procedure is complete.

Figure 2.2. Asynchronous Procedure Calls



2.4.2. The Callback Procedure

For asynchronous procedures calls, you must provide a callback procedure that is invoked when the requested transaction is complete. Your callback procedure notifies the client application that the call is complete and performs the same logic your client application normally performs following a procedure call: interpreting the results of the procedure (if any) and making appropriate changes to client application variables.

For our new Hello World example, when the SignIn procedure completes, we want to display the return values in a welcome message to the user. So our callback procedure might look like this:

```
static class SignInCallback implements ProcedureCallback {    ❶
    @Override
    public void clientCallback(ClientResponse response) {     ❷

    // Make sure the procedure succeeded.
        if (response.getStatus() != ClientResponse.SUCCESS) {❸
            System.err.println(response.getStatusString());
            return;
         }

       VoltTable results[] = response.getResults();           ❹
       VoltTable recordset = results[0];

       System.out.printf("%s, %s!\n",
            recordset.fetchRow(0).getString("Hello"),
            recordset.fetchRow(0).getString("Firstname") );

   }
}
```

The following notes describe the individual components of the callback procedure.

❶ You define the callback procedure as a class that implements (and overrides) the VoltDB ProcedureCallback class.

❷ Whereas a synchronous procedure call returns the ClientResponse as a return value, an asynchronous call returns the same ClientResponse object as a parameter to the callback procedure.

❸ In the body of the callback, first we check to make sure the procedure completed successfully. (If the procedure fails for any reason, for example if a SQL query generates a constraint violation, the ClientResponse contains information about the failure.) In this case we are only looking for success.

❹ Once we know the procedure succeeded, we perform the same functions we would for a synchronous call. In this case, we retrieve the appropriate words from the response and use them to construct and display a greeting to the user.

Since we also want to call the RegisterUser procedure asynchronously, we need to create a callback for that procedure as well. In the case of registering the user, we do not need to provide feedback, so the callback procedure is simplified. All that is needed in the body of the callback is to detect and report any errors that might occur. The RegisterCallback looks like this:

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

                // Make sure the procedure succeeded. If not
                // (for example, account already exists),
                // report the error.
            if (response.getStatus() != ClientResponse.SUCCESS) {
                System.err.println(response.getStatusString());
              }

        }
    }
```

2.4.3. Making an Asynchronous Procedure Call

Once you define a callback, you are ready to initiate the procedure call. You make asynchronous procedure calls in the same way you make synchronous procedure calls. The only differences are that you specify the callback procedure as the first argument to the callProcedure method and you do not need to make an assignment to a client response, since the response is sent as a parameter to the callback procedure.

The following example illustrates both a synchronous and an asynchronous call to the SignIn procedure we defined earlier:

```
        // Synchronous procedure call
ClientResponse response = myApp.callProcedure("SignIn",
                    email, currenttime);

        // Asynchronous procedure call
myApp.callProcedure(new SignInCallback(), "SignIn",
                    email, currenttime);
```

If you do not need to verify the results of a transaction, you do not even need to create a unique callback procedure. Just as you can make a synchronous procedure call and not assign the results to a local object if they are not needed, you can make an asynchronous procedure call using the default callback procedure, which does no special processing. For example, the following code calls the Insert procedure to add a HELLOWORLD record using the default callback:

```
myApp.callProcedure(new ProcedureCallback(), "Insert",
                    "English", "Hello",  "World");
```

2.5. Connecting to all Servers

The final step, once you have optimized the partitioning and the procedure invocations, is to maximize the bandwidth between your client application and the cluster. You can create connections to any of the servers in the cluster and that server will coordinate the processing of your transactions with the other nodes.

Each node in the cluster has its own queue of pending transactions. That node is responsible for:

- Receiving the transaction request from the client and returning the results upon completion
- Routing the transaction to the appropriate partition, or *initiator*, based on the transaction's characteristics.

There is one initiator for each unique partition, and a separate initiator for multi-partition transactions within the cluster. Once the initiator receives the transaction, it is responsible for:

- Scheduling the transaction with the other nodes in the cluster
- Distributing the work items for the transaction to the appropriate nodes and partitions and collecting responses when it is time to execute the transaction

Any node in the cluster can receive transaction requests. However, for maximum performance it is best if all nodes do their share. Initiators are automatically distributed around the cluster. But if only one node is interacting with the client application, managing the queue can become a bottleneck and leave the other nodes in the cluster idle while they wait for work items.

This is why the recommendation is for client applications to create connections to as many nodes in the cluster as possible. When there are multiple connections, the Java client interface will direct each transaction to the most appropriate server, avoiding extra "hops" within the cluster as requests are redirected to the corresponding initiator. For other clients that support multiple connections, requests use a round-robin approach to distribute the procedure calls.

By default, the VoltDB sample applications assume a single server (localhost) and only create a single connection. This makes the examples easy to read and easy to run for anyone who downloads the kit. However, in real world examples your client application should create connections to all of the nodes in the cluster to evenly distribute the work load and avoid network bottlenecks.

The update to the Hello World example demonstrates one method for doing this. Since it is difficult to know in advance what nodes are used in the cluster, the revised application uses an argument on the command line to specify what nodes to connect to. (Avoiding hard-coded server names is also a good practice so you do not have to recode your application if you add or replace servers in the future.)

The first argument on the command line is assumed to be a comma-separated list of server names. This list is converted to an array, which is used to create connections to each node. If there is no command line argument, the default server "localhost" is used. The following is the applicable code from the beginning of the client application. Note that only one client is instantiated but multiple connections are made from that client object.

```
public static void main(String[] args) throws Exception {

    /*
     * Expect a comma-separated list of servers.
     * If not, use localhost.
     */
  String serverlist = "localhost";
  if (args.length > 0) { serverlist = args[0]; }
  String[] servers = serverlist.split(",");

    /*
     * Instantiate a client and connect to all servers
     */
  org.voltdb.client.Client myApp = ClientFactory.createClient();
  for (String server: servers) {
       myApp.createConnection(server);
  }
```

2.6. Putting it All Together

Now that we have defined the schema, created the stored procedures and the callback routines for asynchronous calls, and created connections to all of the nodes in the cluster, we can put together the new

and improved Hello World application. We start by loading the HELLOWORLD table just as we did in the previous version. Since this is only done once to initialize the run, we can make them synchronous calls. Note that we do not need to worry about constraint violations. If the client application is run two or more times, we can reuse the pre-loaded content.

```
/*
* Load the database.
*/
try {
      myApp.callProcedure("Insert", language[0], "Hello",   "World");
      myApp.callProcedure("Insert", language[1], "Bonjour", "Monde");
      myApp.callProcedure("Insert", language[2], "Hola",    "Mundo");
      myApp.callProcedure("Insert", language[3], "Hej",     "Verden");
      myApp.callProcedure("Insert", language[4], "Ciao",    "Mondo");
} catch (Exception e) {
      // Not to worry. Ignore constraint violations if we
      // load this table more than once.
}
```

To show off the performance, we then emulate the running system. We need some users. So, again, we initialize a few user records using the RegisterUser stored procedure. As a demonstration, we use a utility method for generating pseudo-random email addresses.

```
/*
* Start by making sure there are at least 5 accounts
*/
while (maxaccountID < 5) {
     String first = firstname[seed.nextInt(10)];
     String last = lastname[seed.nextInt(10)];
     String dialect = language[seed.nextInt(5)];
     String email = generateEmail(maxaccountID);
     myApp.callProcedure(new RegisterCallback(),"RegisterUser",
                         email,first,last,dialect );
     maxaccountID++;
}
```

Finally, we want to repeatedly call the SignIn stored procedure, while occasionally registering a new user (say, once every 100 sign ins).

```
/*
* Emulate a busy system: 100 signins for every 1 new registration.
* Run for 5 minutes.
*/
long countdowntimer = System.currentTimeMillis() + (60 * 1000 * 5);
while (countdowntimer > System.currentTimeMillis()) {

    for (int i=0; i<100; i++) {
          //int id = seed.nextInt(maxaccountID);
          String user = generateEmail(seed.nextInt(maxaccountID));
          myApp.callProcedure(new SignInCallback(), "SignIn",
                              user, System.currentTimeMillis());
   }

   String first = firstname[seed.nextInt(10)];
   String last = lastname[seed.nextInt(10)];
   String dialect = language[seed.nextInt(5)];
   String email = generateEmail(maxaccountID);

   myApp.callProcedure(new RegisterCallback(),"RegisterUser",
                       email,first,last,dialect);
   maxaccountID++;

}
```

The completed source code can be found (and run) in the `doc/tutorials/helloworldrevisited/` folder where VoltDB is installed. Give it a try on a single system or on a multi-node cluster.

2.7. Next Steps

Updating the Hello World example demonstrates how to design applications that can maximize the value of the VoltDB software. However, even with these changes, Hello World is still a very simple application. Deciding how to partition the database for your specific needs and how to configure a cluster to support the VoltDB features you want to use requires careful consideration of capabilities and tradeoffs. The following chapters provide further guidance on this topics.

Chapter 3. Understanding VoltDB Execution Plans

This chapter explains how VoltDB plans for executing SQL statements, the information it generates about the plans, and how you can use that information to evaluate and optimize your SQL code.

3.1. How VoltDB Selects Execution Plans for Individual SQL Statements

When VoltDB parses a stored procedure definition or an ad hoc query, it reviews possible execution plans for the SQL statements involved. Based on the schema, the partition columns, and any implicit or explicit indexes for the tables, VoltDB chooses what it believes is the most efficient plan for executing each statement. The more complex the SQL statement, the more execution plans VoltDB considers.

As part of the compilation process, VoltDB generates *explain* or *execution* plans that you can use to understand what execution order was selected. You can also affect those plans by specifying the order in which tables are joined as part of your SQL statement declaration.

3.2. Understanding VoltDB Execution Plans

VoltDB stores the execution plans for stored procedures along with the schema in the database. There are three methods for reviewing these execution plans. You can:

- Call the @Explain or @ExplainProc system procedures
- Use the **explain** or **explainproc** directives in sqlcmd
- Review the execution plans in the VoltDB Management Center Schema tab

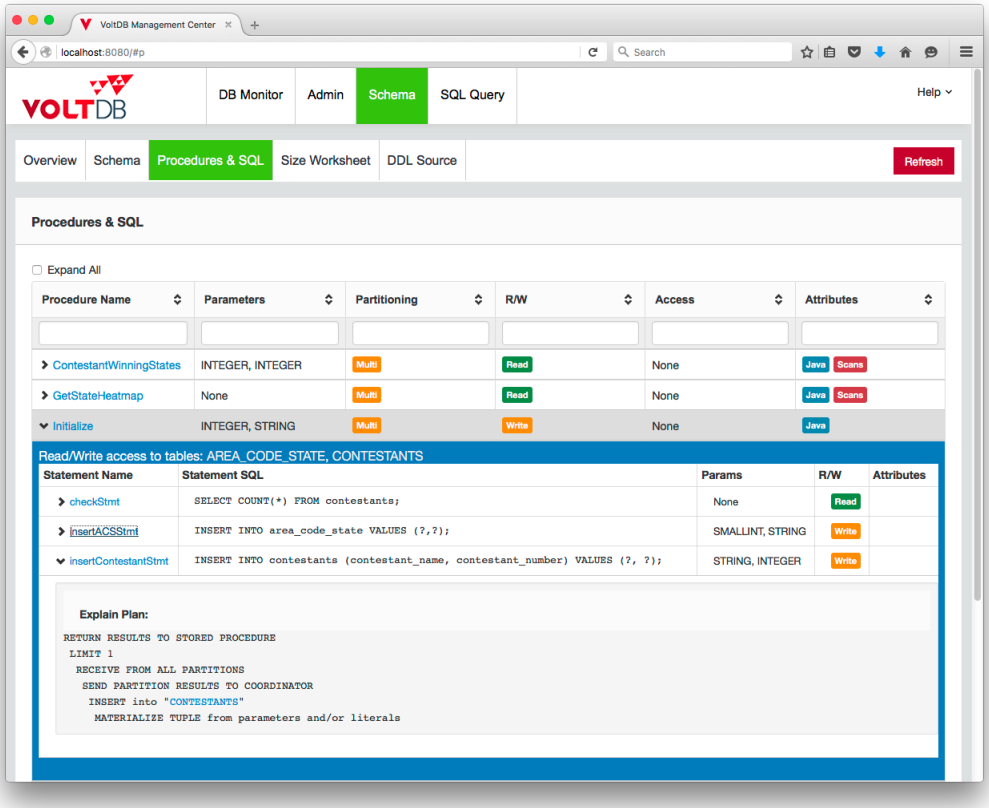
The system procedures and sqlcmd directives produce identical output. For example, if you enter the **explainproc** directive in sqlcmd with the name of a stored procedure, it displays the execution plan for every SQL statement defined in the stored procedure. You get the same results calling the @ExplainProc system procedure. You can see the same information by connecting to the VoltDB Management Center in a web browser. The explain directive and @Explain system procedure allow you to review the execution plan for an ad hoc SQL query by entering the text of the query.

Let's look at the voter sample program as an example. The voter sample has five stored procedures. The Initialize procedure declares three SQL statements. You can see the execution plans for all three statements by starting the sample application, connecting to the server using sqlcmd and using the **explainproc** directive. You can also get the execution plan for an ad hoc count of the votes table using the **explain** directive, like so:

```
$ sqlcmd
1> explainproc Initialize;
 . . .
2> explain select count(*) from votes;
 . . .
```

In the VoltDB Management Center, which is available from a web browser via http:://*{server-name}*:8080 by default, you can see the execution plans by navigating to the *Schema* tab, clicking on *Procedures &*

SQL, and expanding the stored procedure to see the individual statements. The execution plan is displayed in the expanded view. The following example shows the execution plan for *InsertContestantStmt* in the *Initialize* stored procedure.



3.3. Reading the Execution Plan and Optimizing Your SQL Statements

The execution plan is an ordered representation of how VoltDB will execute the statement. Read the plan from bottom up to understand the order in which the plan is executed. So, for example, looking at the *InsertACSStmt* SQL statement in the Voter application's *Initialize* stored procedure, we see the following execution plan for inserting an area code into the area_code_state table:

```
RETURN RESULTS TO STORED PROCEDURE
 LIMIT 1
  RECEIVE FROM ALL PARTITIONS
   SEND PARTITION RESULTS TO COORDINATOR
    INSERT into "AREA_CODE_STATE"
     MATERIALIZE TUPLE from parameters and/or literals
```

As mentioned before it is easiest to read the plans from the bottom up. So in this instance, how the SQL statement is executed is by:

- Constructing a record based on input parameters and/or literal values
- Inserting the record into the table

- Because this is a multi-partitioned procedure, each partition then sends its results to the coordinator
- The coordinator then rolls up the results, limits the results (that is, the status of the insert) to one row, and returns that value to the stored procedure

You will notice that the lines of the execution plan are indented to indicate precedence. For example, the construction of the tuple must happen before it is inserted into the table.

Let's look at another example from the Voter sample. The checkContestantStmt in the Vote stored procedure performs a read operation:

```
RETURN RESULTS TO STORED PROCEDURE
 INDEX SCAN of "CONTESTANTS" using its primary key index
  uniquely match (CONTESTANT_NUMBER = ?0)
```

You can see from the plan that the scan of the CONTESTANTS table uses the primary key index. It is also a partitioned table and procedure so the results can be sent directly back to the stored procedure.

Of course, planning for a SQL statement accessing one table with only one condition is not very difficult. The execution plan becomes far more interesting when evaluating more complex statements. For example, you can find a more complex execution plan in the GetStateHeatmap stored procedure:

```
RETURN RESULTS TO STORED PROCEDURE
 ORDER BY (SORT)
  Hash AGGREGATION ops: SUM(V_VOTES_BY_CONTESTANT_NUMBER_STATE.NUM_VOTES)
   RECEIVE FROM ALL PARTITIONS
    SEND PARTITION RESULTS TO COORDINATOR
     SEQUENTIAL SCAN of "V_VOTES_BY_CONTESTANT_NUMBER_STATE"
```

In this example you see an execution plan for a multi-partition stored procedure. Again, reading from the bottom up, the order of execution is:

- At each partition, perform a sequential scan of the votes-per-contestant-and-state table.
- Return the results from each partition to the initiator that is coordinating the multi-partition transaction.
- Use an aggregate function to sum the votes for all partitions by contestant.
- Sort the results
- And finally, return the results to the stored procedure.

3.3.1. Evaluating the Use of Indexes

What makes the execution plans important is that they can help you optimize your database application by pointing out where the data access can be improved, either by modifying indexes or by changing the join order of queries. Let's start by looking at indexes.

VoltDB uses information about the partitioning column to determine what partition needs to execute a single-partitioned stored procedure. However, it does not automatically create an index for accessing records in that column. So, for example, in the Hello World example, if we remove the primary key (DIALECT) on the HELLOWORLD table, the execution plan for the Select statement also changes.

Before:

```
RETURN RESULTS TO STORED PROCEDURE
```

```
 INDEX SCAN of "HELLOWORLD" using its primary key index
  uniquely match (DIALECT = ?0)
```

After:

```
RETURN RESULTS TO STORED PROCEDURE
 SEQUENTIAL SCAN of "HELLOWORLD"
  filter by (DIALECT = ?0)
```

Note that the first operation has changed to a sequential scan of the HELLOWORLD table, rather than a indexed scan. Since the Hello World example only has a few records, it does not take very long to look through five or six records looking for the right one. But imagine doing a sequential scan of an employee table containing tens of thousands of records. It quickly becomes apparent how important having an index can be when looking for individual records in large tables.

There is an incremental cost associated with inserts or updates to tables containing an index. But the improvement on read performance often far exceeds any cost associated with writes. For example, consider the flight application that is used as an example in the *Using VoltDB* manual. The FLIGHT table is a replicated table with an index on the FLIGHT_ID, which helps for transactions that join the FLIGHT and RESERVATION tables looking for a specific flight.

However, one of the most common transactions associated with the FLIGHT table is customers looking for flights during a specific time period; not by flight ID. In fact, looking up flights by time period is estimated to occur at least twice as often as looking for a specific flight.

The execution plan for the LookupFlight stored procedure using the original schema looks like this:

```
RETURN RESULTS TO STORED PROCEDURE
 SEQUENTIAL SCAN of "FLIGHT"
  filter by ((((ORIGIN = ?0) AND (DESTINATION = ?1))
     AND (DEPARTTIME > ?2)) AND (DEPARTTIME < ?3))
```

Clearly, looking through a table of 2,000 flights without an index 10,000 times a second will impact performance. So it makes sense to add another index to improve this transaction. For example:

```
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 INDEX flightTimeIdx ON FLIGHT ( departtime);
```

After adding the index, the execution plan changes to use the index:

```
RETURN RESULTS TO STORED PROCEDURE
 INDEX SCAN of "FLIGHT" using "FLIGHTTIMEIDX"
 range-scan covering from (DEPARTTIME > ?2) while (DEPARTTIME < ?3),
    filter by ((ORIGIN = ?0) AND (DESTINATION = ?1))
```

Indexes are not required for every database query. For very small tables or infrequent queries, an index could be unnecessary overhead. However, in most cases and especially frequent queries over large datasets, not having an applicable index can severely impact performance.

When tuning your VoltDB database application, one useful step is to review the schema for any unexpected sequential (non-indexed) scans. The VoltDB Management Center makes this easy because it puts the "Scans" icon in the attributes column for any stored procedures that contain sequential scans.

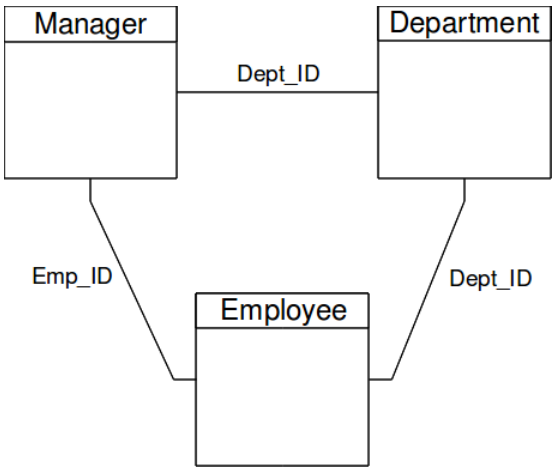
See the following chapter, Chapter 4, *Using Indexes Effectively*, for more information on tuning your database through effective use of indexes.

3.3.2. Evaluating the Table Order for Joins

The other information that the execution plans provides, in addition to the use of indexes, is the order in which the tables are joined.

Join order often impacts performance. Normally, when joining two or more tables, you want the database engine to scan the table that produces the smallest number of matching records first. That way, there are fewer comparisons to evaluate when considering the other conditions. However, at compile time, VoltDB does not have any information about the potential sizing of the individual tables and must make its best guess based solely on the table schema, query, and any indexes that are defined.

For example, assume we have a database that correlates employees to departments. There is a DEPARTMENT table and an EMPLOYEE table, with a DEPT_ID column that acts as a foreign key. But departments have managers, who are themselves employees. So there is a MANAGER table that also contains both a DEPT_ID and an EMP_ID column. The relationship of the tables looks like this:



Most transactions look up employees by their employee ID or their department ID. So indexes are created for those columns. However, say we want to look up all the employees that report to a specific manager. Now we need to join the MANAGER table (to get the department ID), the DEPARTMENT table (to get the department name), and the EMPLOYEE table (to get the employees' names). VoltDB does not know, in advance when compiling the schema, that there will be many more employees than departments or managers. As a result, the winning plan might look like the following:

```
RETURN RESULTS TO STORED PROCEDURE
 ORDER BY (SORT)
  NESTLOOP INDEX JOIN
  inline (INDEX SCAN of "DEPARTMENT" using "DEPTIDX" (unique-scan covering))
   NESTLOOP INDEX JOIN
   inline (INDEX SCAN of "MANAGER" using "MGRIDX" (unique-scan covering))
    RECEIVE FROM ALL PARTITIONS
```

```
     SEND PARTITION RESULTS TO COORDINATOR
      SEQUENTIAL SCAN of "EMPLOYEE"
```

Clearly, performing a sequential scan of the employees (since the department ID has not been identified yet) is not going to provide the best performance. What you really want to do is to join the MANAGER and DEPARTMENT tables first, to identify the department ID before joining the EMPLOYEE table so the last join can take advantage of the appropriate index.

For cases where you are joining multiple tables and know what the optimal join order would be, VoltDB lets you specify the join order as part of the SQL statement definition. Normally, you declare a new SQLstmt class by specifying the SQL query only. However, you can provide a second argument specifying the join order as a comma-separated list of table names and aliases. For example, the declaration of the preceding SQL query, including join order, would look like this:

```
public final SQLStmt FindEmpByMgr = new SQLStmt(
    "SELECT dept.dept_name, dept.dept_id, emp.emp_id, " +
    "emp.first_name, emp.last_name, manager.emp_id " +
    "FROM MANAGER, DEPARTMENT AS Dept, EMPLOYEE AS Emp " +
    "WHERE manager.emp_id=? AND manager.dept_id=dept.dept_id " +
    "AND manager.dept_id=emp.dept_id " +
    "ORDER BY emp.last_name, emp.first_name",
    "manager,dept,emp");
```

Note that where the query defines an alias for a table — as the preceding example does for the DEPARTMENT and EMPLOYEE tables — the join order must use the alias name rather than the original table name. Also, if a query joins six or more tables, you *must* specify the join order or VoltDB reports an error when it compiles the project.

Having specified the join order, the chosen execution plan changes to reflect the new sequence of operations:

```
RETURN RESULTS TO STORED PROCEDURE
 ORDER BY (SORT)
  RECEIVE FROM ALL PARTITIONS
   SEND PARTITION RESULTS TO COORDINATOR
    NESTLOOP INDEX JOIN
    inline (INDEX SCAN of "EMPLOYEE" using "EMPDEPTIDX" (unique-scan covering))
     NESTLOOP INDEX JOIN
     inline (INDEX SCAN of "DEPARTMENT" using "DEPTIDX" (unique-scan covering))
      SEQUENTIAL SCAN of "MANAGER"
```

The new execution plan has at least three advantages over the default plan:

- It starts with a sequential scan of the MANAGER table, a table with 10-20 times fewer rows than the EMPLOYEE table.
- Because MANAGER and DEPARTMENT are replicated tables, all of the initial table scanning and filtering can occur locally within each partition, rather than returning the full EMPLOYEE data from each partition to the initiator to do the later joins and sorting.
- Because the join order retrieves the department ID first, the execution plan can utilize the index on that column to improve the scanning of EMPLOYEE, the largest table.

Chapter 4. Using Indexes Effectively

Indexes provide a classic “space for speed” trade-off. They add to the persistent memory required by your application data but they make query filtering significantly faster. They also represent a trade-off that sacrifices incremental write performance for potentially significant read performance, on the assumption that indexed data is accessed by read queries more frequently than it is modified.

Using the best practices described in the chapter when defining indexes can maximize query performance in return for minimum investments in memory usage and computation overhead when writing data.

4.1. Basic Principles for Effective Indexing

Here are seven tips to creating effective indexes in VoltDB:

- Avoid indexes that have a column list that is simply a prefix of another index's column list. The index with the longer column list will usually serve the same queries as the shorter one. If the primary key of table X is (A, B, C), then an index on (A, B) is of little use. An index on (B, C) may be of use in this scenario or it may be more effective to define the primary key as (B, C, A) — if B is likely to be filtered in queries where A is not equality-filtered.

- Avoid "low-cardinality" indexes — An index defined solely on a column that only has a few distinct values is usually not very effective. Because of its large number of duplicate values, it does little to narrow the set of rows to be sequentially filtered by other filters in the query. Such an index can sometimes cause the planner to fail to select a more effective index for a query or even a more efficient sequential scan. One way to increase index effectiveness of low cardinality indexes is to add other filtered columns to the index, keeping in mind that the effectiveness of an index for a query "tops out" at the first column that has an inequality filter — or before the second column that has an IN filter.

- When deciding how to order columns within an index (or primary key or unique constraint) definition, columns that are more likely to be filtered with an exact equality (such as A = ?), should be listed before columns that tend to be range-filtered (B <= ?). Queries that are run the most often or that benefit the most from indexing (perhaps because they lack filters that can be covered by other indexes) should weigh more heavily in this decision.

- In some cases, with a roughly equal mix between queries using forms like "WHERE A = ? AND B <= ?" and other queries using forms like "WHERE A > ? AND B = ?", it may be worthwhile to define indexes on both permutations — on X(A, B ...) and on X(B, A ...). Otherwise, when two or more columns in an index tend to both get equality filtered in combination, it is generally better to list a column first if it also tends to be filtered (without the other) in other queries. A possible exception to this rule is when the column has low cardinality (to avoid the ineffective use of the index).

- Placing the low-cardinality column later in the index's list prevents the index from being applied as a low-cardinality indexed filter and favors the selection of a more effective index or sequential scan.

- Any non-unique filter that is listed in the schema report as having no procedures using it is a candidate for elimination. But first, it may make sense to look for queries that would be expected to use the index and determine what they are using instead for scans on the table. It may be that the index chosen by the planner is not actually as effective and that index may be the better candidate for elimination. Also, note that indexes that are only used to support recalculation of min and max values in materialized views may be erroneously reported as unused.

- Index optimization is best accomplished iteratively, eliminating or tuning an index on a table and seeing its effect on statements before making other changes to other competing indexes.

4.2. Defining Indexes

VoltDB indexes provide multiple benefits. They help to guard against unintended duplication of data. They help to optimize recalculation of min and max values in materialized views. Tree indexes in particular can also be used to replace memory- and processor-intensive sorting operations in queries that have ORDER BY and GROUP BY clauses. This discussion focuses on the benefits of indexes in implementing SQL WHERE clauses that filter query results.

There are several methods for constructing indexes in SQL:

- PRIMARY KEY column attribute
- UNIQUE or ASSUME UNIQUE column attribute
- PRIMARY KEY table constraint
- UNIQUE or ASSUME UNIQUE table constraint
- CREATE INDEX statement

Any of these methods can be used to define a "UNIQUE" index on a single column. The table constraints and CREATE INDEX statements can also define a "UNIQUE" index on multiple columns or on expressions that use one or more columns. The CREATE INDEX statement can be used to construct a non-UNIQUE index on one or more columns or expressions.

All examples in this chapter describe indexes as if they were created by the CREATE INDEX statement, but the discussion applies generally to indexes defined using any of these methods.

4.3. The Goals for Effective Indexing

The goals of effective indexing are to:

- Eliminate unused indexes
- Minimize redundancy (memory use and overhead for writes) among overlapping indexes on a table
- Minimize sequential scans of large numbers of rows

Sequential filtering always occurs when rows are accessed without the benefit of an index. This is known as a *sequential scan*. Sequential filtering can also occur on an indexed scan when there are more filters in the query than are covered by the index. The cost of sequential filtering is based on several factors. One factor is the number of filters being applied to each row. A major factor is the number of rows to which the filters must be applied.

The intent of an index is to use a more streamlined "lookup" algorithm to produce a small set of filtered rows, eliminating the need to sequentially apply (as many) filters to (as many) rows.

Since there are trade-offs and limitations involved in defining indexes, indexes may not provide complete coverage for all of the filters in a query. If any filters are not covered by an index, they must be sequentially applied. The cost of this action is typically reduced compared to a sequential scan of the data because the index reduces the two major contributing factors: the number of remaining, uncovered filters to apply, and the number of rows to which they must be applied.

The lookup algorithm used by an index is typically much more efficient than sequential application for the same set of filters and rows, but it does not have zero cost. It also slightly complicates the process

of sequentially applying any remaining filters to the remaining rows. In fact, the worst-case scenario for query filter performance is when an index's lookup algorithm is employed but fails to cover most of a query's filters and fails to eliminate most of the table's rows. This case can perform significantly worse than a sequential scan query that uses no index at all and applies all of its filters sequentially. This possibility calls for the elimination of ineffective indexes from a database.

An ideal set of index definitions minimizes the number of times that any filter is sequentially applied to any row in any query over the life of the database system. It accomplishes this with a minimum number of indexes, each of minimum complexity, to reduce persistent memory and data write computation costs.

4.4. How Indexes Work

A key insight into defining indexes is determining which of the filters in a query can be “covered” by a given index. Filters and combinations of filters qualify for coverage based on different criteria.

Each "scan" in a query, that is, each argument to a FROM clause that is not a subquery, can use up to one index defined on its table. When a table defines multiple indexes on the same table, these indexes compete in the query planner for the mission of controlling each scan in each query that uses the table. The query planner uses several criteria to evaluate which one of the table's indexes that cover one or more filters in the query is the most likely to be the most efficient.

When indexing a single column, as in "CREATE INDEX INDEX_OF_X_A ON X(A);", a covered filter can be any of the following:

- "A <op> <constant>", where <op> can be any of "=, <, >, <=, or >="
- "A BETWEEN <constant1> AND <constant2>"
- "A IN <constant-list>"
- A special case of "A LIKE <string-pattern>" where <string-pattern> contains a fixed prefix followed by a wild-card character

Here, <*constant*>, <*constant1*>, and <*constant2*> can be actual literal constants like 1.0 or 'ABC' or they can be placeholders (?) that resolve to constants at runtime. <*constant-list*> can be a list of literals or literals and parameters like ('ABC', 'BAC', 'BCA', 'ACB', 'CBA', 'BAC') or (1, 2, 3, ?) or (?, ?, ?, ?, ?) or a single vector-valued placeholder. Each of these "constants" can also be an expression of constants, such as ((1024*1024)-1).

Depending on the order in which tables are scanned in a query, called the *join order*, a covered filter can also be "A <op> <column>" where <*column*> is a column from another table in the query or any expression of a column or columns from another table and possibly constants, like B or (B || C) or SUBSTR(B||C , 1 , 4).

The join order dependency works like this: if you had two tables indexed on column A and your query is as follows, only one table could be indexed:

```
SELECT * FROM X, Y WHERE X.A = Y.A and X.B = ?;
```

The first one to be scanned would have to use a sequential table scan. If you also had an index on X.B, X could be index-scanned on B and Y could then be index-scanned on A, so a table scan would be avoided.

The availability of indexes that cover the scans of a query have a direct effect on the planners selection of the join order for a query. In this case, the planner would reject the option of scanning Y first, since that would mean one more sequential scan and one fewer index scan, and the planner prefers more index scans whenever possible on the assumption that index scans are more efficient.

When creating an index containing multiple columns, as in "CREATE INDEX INDEX_OF_X_A_B ON X(A, B);", a covered filter can be any of the forms listed above for coverage by a simpler index “ON X(A)”, regardless of the presence of a filter on B — this is used to advantage when columns are added to an index to lower its cardinality, as discussed below.

A multi-column index “ON X(A, B) can be used more effectively in queries with a combination of filters that includes a filter on A and a filter on B. To enable the more effective filtering, the first filter or *prefix filter* on A must specifically have the form of "A = ..." or "A IN ..." — possibly involving column(s) of other tables, depending on join order — while the filter on B can be any form from the longer list of covered filters, above.

A specific exception to this rule is that a filter of the form "B IN ..." does not improve the effectiveness of a filter of the form "A IN ...", but that same filter "B IN ..." can be used with a filter of the specific form "A = ...". In short, each index is restricted to applying to only one “IN” filter per query. So, when the index is covering “A IN …”, it will refuse to cover the “B IN …” filter.

This extends to indexes on greater numbers of columns, so an index "ON X(A, B, C)" can generally be used for all of the filters and filter combinations described above using A or using A and B. It can be used still more effectively on a combination of prefix filters like "A = ... " (or "A IN ...") AND "B = ..." (or "B IN ...") with an additional filter on C — but again, only the first "IN" filter improves the index effectiveness, and other “IN” filters are not covered.

When determining whether a filter can be covered as the first or prefix filter of an index (first or second filter of an index on three or more columns, etc.), the ordering of the filters always follows the ordering of the columns in the index definition. So, “CREATE INDEX INDEX_ON_X_A_B ON X(A, B)” is significantly different from “CREATE INDEX INDEX_ON_X_B_A ON X(B, A)”. In contrast, the orientation of the filters as expressed in each query does not matter at all, so "A = 1 and B > 10" has the same effect on indexing as "10 < B and A = 1" etc. The filter “A = 1” is considered the “first” filter in both cases when the index is “ON (A, B)” because A is first.

Also, other arbitrary filters can be combined in a query with “AND” without disqualifying the covered filters; these additional filters simply add (reduced) sequential filtering cost to the index scan.

But a top-level OR condition like "A = 0 OR A > 100" will disqualify all filters and will not use any index.

A general pre-condition of a query's filters eligible for coverage by a multi-column index is that the first key in the index must be filtered. So, if a query had no filter at all on A, it could not use any of the above indexes, regardless of the filters on B and/or on C. This is the condition that can cause table scans if there are not enough indexes, or if the indexes or queries are not carefully matched.

This implies that carelessly adding columns to the start of an already useful index's list can make it less useful and applicable to fewer queries. Conversely, adding columns to the end of an already useful index (rather than to the beginning) is more likely to make the index just as applicable but more effective in eliminating sequential filtering. Adding to the middle of the list can cause an index to become either more or less effective for the queries to which it applies. Any such change should be tested by reviewing the schema report and/or by benchmarking the affected queries. Optimal index use and query performance may be achieved either with the original definition of the index, with the changed definition, or by defining two indexes.

4.5. Summary

To recap, here are the best practices for defining indexes in VoltDB:

- Avoid indexes that have a column list that is simply a prefix of another index's column list. The index with the longer column list will usually serve the same queries as the shorter one. If the primary key

of table X is (A, B, C), then an index on (A, B) is of little use. An index on (B, C) may be of use in this scenario or it may be more effective to define the primary key as (B, C, A) — if B is likely to be filtered in queries where A is not equality-filtered.

- Avoid "low-cardinality" indexes — An index defined solely on a column that only has a few distinct values is usually not very effective. Because of its large number of duplicate values, it does little to narrow the set of rows to be sequentially filtered by other filters in the query. Such an index can sometimes cause the planner to fail to select a more effective index for a query or even a more efficient sequential scan. One way to increase index effectiveness of low cardinality indexes is to add other filtered columns to the index, keeping in mind that the effectiveness of an index for a query "tops out" at the first column that has an inequality filter — or before the second column that has an IN filter.

- When deciding how to order columns within an index (or primary key or unique constraint) definition, columns that are more likely to be filtered with an exact equality (such as A = ?), should be listed before columns that tend to be range-filtered (B <= ?). Queries that are run the most often or that benefit the most from indexing (perhaps because they lack filters that can be covered by other indexes) should weigh more heavily in this decision.

- In some cases, with a roughly equal mix between queries using forms like "WHERE A = ? AND B <= ?" and other queries using forms like "WHERE A > ? AND B = ?", it may be worthwhile to define indexes on both permutations — on X(A, B ...) and on X(B, A ...). Otherwise, when two or more columns in an index tend to both get equality filtered in combination, it is generally better to list a column first if it also tends to be filtered (without the other) in other queries. A possible exception to this rule is when the column has low cardinality (to avoid the ineffective use of the index).

- Placing the low-cardinality column later in the index's list prevents the index from being applied as a low-cardinality indexed filter and favors the selection of a more effective index or sequential scan.

- Any non-unique filter that is listed in the schema report as having no procedures using it is a candidate for elimination. But first, it may make sense to look for queries that would be expected to use the index and determine what they are using instead for scans on the table. It may be that the index chosen by the planner is not actually as effective and that index may be the better candidate for elimination. Also, note that indexes that are only used to support recalculation of min and max values in materialized views may be erroneously reported as unused.

- Index optimization is best accomplished iteratively, eliminating or tuning an index on a table and seeing its effect on statements before making other changes to other competing indexes.

Chapter 5. Creating Flexible Schemas With JSON

A major part of any relational database is the schema: the structure of the data as defined by the tables and columns. It is possible to change the schema when necessary. However, at any given time, each table has a set number of columns, each with a specific name and datatype.

It is possible to store unstructured data in a relational database as a "blob" using a VARBINARY or VARCHAR column. However, the database has no way to operate on your data effectively beyond simply storing and retrieving it.

Sometimes data is not as strictly organized as a relational database schema requires, but does have structure within it. For example, a table may have a set of properties, each with a different name and matching value. But not all records use the same set of properties.

JSON (JavaScript Object Notation) is a light-weight data interchange format that lets you describe data structures on the fly. JSON-encoded strings are composed of a hierarchy of key-value pairs that can be as simple or as complex as needed. More importantly, the actual structure of the object is determined at run-time and does not need to be predefined.

VoltDB gives you the ability to mix the efficiency of the relational schema with the flexibility of JSON. By using JSON-encoded columns with VoltDB SQL functions and index capabilities, you can work more naturally with JSON data while maintaining the efficiency and transactional consistency of a relational database.

5.1. Using JSON Data Structures as VoltDB Content

JSON processing is easiest to understand by example. For the purposes of this chapter, we will use the example of a single sign-on (SSO) application using VoltDB. The application needs to store login sessions for multiple online sites under a common username. Each login session must manage different properties describing the state of the session, where properties can vary from simple data values to more complex data structures. Additionally, future enhancements to the application may change what properties are required. Because of the variability of the data, a good strategy is to store this session information as JSON data structures.

To store JSON data in VoltDB, you declare the storage as a standard VARCHAR column, large enough to contain the maximum expected length of the JSON data. For example, the following table declaration includes a VARCHAR column named session_info with a maximum length of 2048 characters.

```
CREATE TABLE user_session_table (
    username           VARCHAR(200)   UNIQUE NOT NULL,
    password           VARCHAR(100)   NOT NULL,
    global_session_id  VARCHAR(200)   ASSUMEUNIQUE NOT NULL,
    last_accessed      TIMESTAMP,
    session_info       VARCHAR(2048)
);
PARTITION TABLE user_session_table ON COLUMN username;
```

The JSON data is stored as text and it is up to your application to perform the conversion from an in-memory structure to the textual representation. For example, the VoltDB software kit includes an example,

json-sessions, that provides a version of the application described here, which uses an open source package from Google called GSON to convert from *plain old Java objects* (POJOs) to text and vice versa.

Once you have a text representation of the JSON object, you insert it into the database column using standard SQL syntax. For example:

```
INSERT INTO user_session_table (username, password,
                                global_session_id,
                                last_accessed, session_info)
         VALUES (?, ?, ?, ?, ?);
```

Note that VoltDB does not validate that the text being inserted into the VARCHAR column is properly encoded JSON. Validation of the encoding occurs when accessing such columns using the JSON-specific functions, described next.

5.2. Querying JSON Data

VoltDB provides four functions to help you interact effectively with JSON data stored in VARCHAR columns. Each function takes a JSON-encoded text string as the first argument, followed by other arguments required for the operation.

Function	Description
ARRAY_ELEMENT(*json*, *array-index*)	Returns the value of a specific element in a JSON array.
ARRAY_LENGTH(*json*)	Returns the number of elements in a JSON array
FIELD(*json*, *path-name*)	Returns the value of a specific field in a JSON structure, where the field is identified by path name.
SET_FIELD(*json*, *path-name*, *new-value*)	Returns a JSON structure where the specified field's value is updated.

In the simple case where the JSON is a flat list of named fields, the FIELD() function can extract the value of one of the fields:

```
SELECT username, FIELD(session_info,'website') AS url
   FROM user_session_table WHERE username=?;
```

The FIELD() can also take a path name for more complex JSON structures. For example, if the *properties* field is itself a list of named fields, you can drill down into the structure using a path, like so:

```
SELECT username, FIELD(session_info,'website') AS url,
   FIELD(session_info,'properties.last_login') AS last_login
   FROM user_session_table WHERE username=?;
```

For structures containing arrays, you can use ARRAY_ELEMENT() to extract a specific element from the array by position. For example, the following SELECT statement extracts the first element from a JSON column named recent_history. (The array index is zero-based.)

```
SELECT username,
   ARRAY_ELEMENT(recent_history,0) AS most_recent
   FROM user_history WHERE username=?;
```

Finally, you can nest the functions to perform more complex queries. Say rather than being a separate column, the named array *recent_history* is a subelement of the *properties* field in the *session_info* column, the following query retrieves the last element from the *recent_history* array:

```
SELECT username,
   ARRAY_ELEMENT(FIELD(session_info,'properties.recent_history'),
      ARRAY_LENGTH(FIELD(session_info,'properties.recent_history'))-1)
         AS oldest
   FROM user_session_table WHERE username=?;
```

Note that, as mentioned earlier, VoltDB does not validate the correctness of the JSON-encoded string on input into a VARCHAR column. Instead, VoltDB validates the data when one of the JSON functions attempts to parse the encoded string. If the data is *not* a valid JSON-encoded string, the function will throw an exception at run-time and rollback the transaction.

5.3. Updating JSON Data

The JSON functions can not only query the JSON-encoded data, they let you modify a JSON-encoded string in place. The SET_FIELD() function takes JSON data as input and returns the same JSON structure but with the specified field updated. For example, using the *user_session_table* described in the previous section, it is possible to update the *last_login* property in the JSON column *session_info* with a single SQL statement:

```
UPDATE user_session_table
   SET session_info = SET_FIELD(session_info,'properties.last_login',?)
   WHERE username=? AND global_session_id=?;
```

Again, the JSON functions can be nested to perform more complex operations. In the following example, the UPDATE statement takes a full JSON encoded-string as input for the *session_info* column, but merges it with the *properties* value from the existing record.

```
CREATE PROCEDURE merge_properties AS
   PARTITION ON TABLE user_session_table COLUMN username
   AS
      UPDATE user_session_table
         SET session_inf = SET_FIELD(?,'properties',
             FIELD(session_info,'properties'))
         WHERE username=?;
```

5.4. Indexing JSON Fields

The JSON functions perform significant string processing to encode and decode the JSON structures. If the table does not have appropriate indexes, the extra processing required for JSON columns added to the need to scan all of the records in the table can quickly result in undesirable latency. The solution is to index the pertinent JSON operations.

To speed up query execution for JSON queries, you can define an index on the commonly accessed fields, including the commonly used function instances. For example, the following index definitions can significantly improve the execution time for queries described in the previous sections:

```
CREATE INDEX session_index_last_login
    ON user_session_table (
      field(session_info, 'properties.last_login'),
      username
    );

CREATE INDEX session_index_history
    ON user_session_table (
```

```
        ARRAY_ELEMENT(FIELD(session_info,'properties.recent_history'),
           ARRAY_LENGTH(FIELD(session_info,'properties.recent_history'))-1),
        username
    );
```

These are fully functional SQL indexes. Whenever you create or update a record in the *user_session_table* table, VoltDB executes the JSON functions and stores the result inside the index. When you query by that same function in the future, VoltDB uses the index avoids both the table scan and repeating the JSON string processing.

5.5. Summary: Using JSON in VoltDB

One of the major benefits of encoding data as JSON is that you do not have to predefine what structure or shape the data has. Further, the shape of the data can vary from one row to the next. The schema for JSON columns are defined by the data itself, rather than having to define the structure using SQL ahead of time. If a new field is needed, you simply create the appropriate data object in whatever programming language you are using, serialize it to JSON, and store it in VoltDB. This avoids the need to change the database schema anytime these temporary data structures change.

On the other hand, when using JSON columns you should be aware of the following drawbacks:

- Your application must be intelligent enough to interpret the various structures that appear in the JSON columns.
- Evaluating JSON data using the VoltDB JSON functions requires additional processing overhead. JSON is excellent for managing fluid data structures, but it is always significantly faster to used a predefined SQL schema when possible.
- You must be sensitive to when indexes are needed, should query patterns or the data structures change. This may, for example, require you to add (or modify) indexes based on the existence of new JSON fields.

Another point of note is that there is a size limit for JSON values. In VoltDB, the VARCHAR columns used to store JSON values are limited to one megabyte (1048576 bytes). JSON support lets you augment the existing relational model within VoltDB. However, it is not intended or appropriate as a replacement for pure blob-oriented document stores.

Chapter 6. Creating Geospatial Applications

VoltDB provides standard datatypes for storing common numeric and textual content. It also provides support for JSON within VARCHAR columns to handle semi-structured content, as described in the preceding chapter. But not all application data can be efficiently managed using just the standard datatypes.

One example of an application area requiring special handling is geospatial data — that is, information about locations and regions on the earth. It is possible to store geospatial data using standard datatypes; for example, storing longitude and latitude as two separate FLOAT columns. However, by storing the data in generic datatype columns the information loses its context. For example, using separate columns it is impossible to tell how far apart two points are or whether those points are part of a larger geometric shape.

To simplify the use of geospatial information, VoltDB includes two geospatial datatypes and a number of functions that help you evaluate and operate on that data. This chapter describes the new datatypes and provides basic information on how to input and use that data in stored procedures and SQL queries.

6.1. The Geospatial Datatypes

VoltDB supports two geospatial datatypes:

- GEOGRAPHY
- GEOGRAPHY_POINT

The GEOGRAPHY datatype supports geographical regions defined as polygons. The GEOGRAPHY_POINT datatype defines a single point using a pair of longitude and latitude values. Both datatypes can be represented as text in an industry format known as Well Known Text (WKT) defined by the Open Geospatial Consortium (OGC). VoltDB provides functions for converting WKT representations to both GEOGRAPHY and GEOGRPAHY_POINT values. WKT is also how values of these types are displayed by sqlcmd and the VoltDB Management Center. Since GEOGRAPHY_POINT is the simpler of the two points, we will discuss it first.

6.1.1. The GEOGRAPHY_POINT Datatype

A GEOGRAPHY_POINT represents a single point on earth as defined by a longitude and latitude value. The WKT representation of a GEOGRAPHY_POINT value is the following:

POINT (*longitude-value latitude-value*)

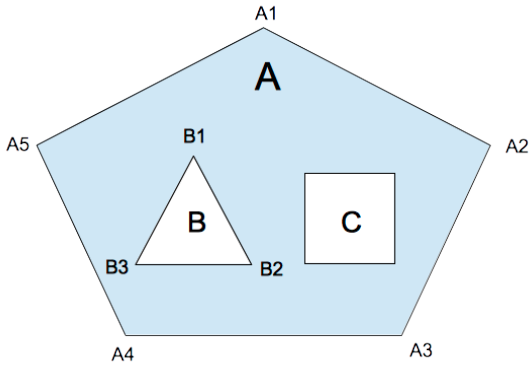
The longitude is a floating point value between 180 and -180 inclusive. The latitude is a floating point value between 90 and -90 inclusive.

6.1.2. The GEOGRAPHY Datatype

The GEOGRAPHY datatype defines a bounded region of the earth represented by one or more polygons. The first polygon, or *ring*, describes the outer boundary of the region. Subsequent rings within the WKT representation describe "holes" within the outer region. So, for example, the following shaded region is described by three rings:

- The outer ring, A

- Two inner rings, B and C



In the WKT representation, the outer ring must be listed first, with the vertices listed in counter-clockwise order (e.g. A5, A4, A3, A2, A1). The inner rings, if any, are listed next with the vertices in clockwise order (e.g. B1, B2, B3). The lines of the rings must not cross or overlap and the description of each ring must list the starting vertex twice: as both the first and last vertex.

Note that, although the individual rings must not cross and vertices must be in the correct order for the geospatial functions to generate valid results, the correctness of the polygon is *not* checked by VoltDB when the GEOGRAPHY data is inserted. If you are unsure of the correctness of the originating data, you can use the ISVALID() and ISINVALIDREASON() functions to validate GEOGRAPHY values within a SQL query.

The WKT representation of a GEOGRAPHY value is the following, where each vertex-list is a comma-separated list of longitude and latitude values describing a single ring:

POLYGON ((*vertex-list*) [, (*vertex-list*)]...)

For example, the simplest polygon, which consists of a single outer ring of three vertices, could be represented like this:

```
POLYGON ( ( 1.5 3.0, 0.0 0.0, 3.0 0.0, 1.5 3.0 ) )
```

For a polygon with two inner rings, the WKT might look like the following:

```
POLYGON ( ( 1.5 3.0, 0.0 0.0, 3.0 0.0, 1.5 3.0 ),
          ( 1.0 1.0, 1.5 0.5, 0.5 0.5, 1.0 1.0 ),
          ( 2.0 1.0, 2.5 0.5, 1.5 0.5, 2.0 1.0 )  )
```

6.1.3. Sizing GEOGRAPHY Columns

GEOGRAPHY polygons, unlike GEOGRAPHY_POINT values, do not have a fixed size. The memory required to store a GEOGRAPHY column varies depending on the number of rings and the number of vertices in each ring. In this way, GEOGRAPHY columns are treated much like VARCHAR and VARBINARY columns when VoltDB manages their allocation and storage.

For convenience, VoltDB provides a default maximum size for GEOGRAPHY columns. So if you declare the column without a specific size, it is assigned a maximum size of 32 kilobytes (32768 bytes):

```
CREATE TABLE Country (
  Name VARCHAR(32),
  Border GEOGRAPHY
```

```
);
```

However, for very large polygons, this default size may be too small. Or, if you have GEOGRAPHY columns mixed with large VARCHAR columns in a single table, the default may be too large because there is a two megabyte limit for the sum of the columns in a single table.

You can specify your own maximum size for a GEOGRAPHY column, in bytes, by including the maximum size in parentheses after the datatype keyword, the same way you do for VARCHAR columns. For example, the following CREATE TABLE statement defines the maximum size of the *border* column as 1024 bytes:

```
CREATE TABLE Country (
  Name VARCHAR(32),
  Border GEOGRAPHY(1024)
);
```

To determine how much space is required to store any polygon, use the following calculation:

- 40 bytes for the polygon
- 43 bytes for every ring
- 24 bytes for every vertex

Note that when counting the vertices, although the starting vertex must be listed twice in the WKT representation, it is only stored once and therefore only counted once in the memory allocation. For example, the memory calculation for a polygon with an outer ring with 10 vertices and 3 inner rings with 8 vertices each would be the following:

```
    40 bytes
   172 bytes  ( 43 X 4 rings )
   816 bytes  ( 24 X 34 total vertices )
------------
  1028 bytes total
```

The largest maximum size you can specify for a GEOGRAPHY column, or any column in VoltDB, is one megabyte.

6.1.4. How Geospatial Values are Interpreted

The earth itself is not uniformly round. However, measurements accurate enough for most applications can be achieved by assuming a perfect sphere and mapping the longitude and latitude coordinates onto that sphere. For calculating distances between locations and areas of regions VoltDB assumes a sphere with a radius matching the mean radius of the earth, or 6,371,008.8 meters. Although an approximation, this model provides distance calculations accurate to within three tenths of a percent (0.3%) of other, more elaborate geospatial models. What this means is, when calculating the distance between two points that are a kilometer apart, the answer computed by the DISTANCE() function may vary up to 3 meters from calculations using other techniques.

6.2. Entering Geospatial Data

As mentioned earlier, Well Known Text (WKT) is the standard presentation VoltDB uses for ingesting and reporting on geospatial data. However, you cannot insert WKT text strings directly as geospatial values. Instead, VoltDB provides SQL functions and Java classes and methods for translating from WKT to the internal geospatial values.

In SQL statements you can use the POINTFROMTEXT() and POLYGONFROMTEXT() functions to generate the appropriate geospatial datatypes from WKT. For example, the following SQL statement inserts the geographic location of New York City into the GEOGRAPHY_POINT column *location*:

```
INSERT INTO CITIES (name, location) VALUES
    ('New York City', POINTFROMTEXT('POINT(-74.0059 40.7127)');
```

In a Java stored procedure you can generate and store a GEOGRAPHY or GEOGRAPHY_POINT value from WKT using the classes `GeographyValue` and `GeographyPointValue` and the method `.fromWKT()`. For example, the following stored procedure takes two Java String objects, converts them to GEOGRAPHY and GEOGRAPHY_POINT values, then inserts them into a record via placeholders in the SQL statement:

```
import org.voltdb.*;
import org.voltdb.types.GeographyValue;
import org.voltdb.types.GeographyPointValue;

public class InsertGeo extends VoltProcedure {

  public final SQLStmt insertrec = new SQLStmt(
     "INSERT INTO region VALUES (?,?,?);" );

  public VoltTable[] run(
     String name, String poly, String point)
     throws VoltAbortException {

       GeographyValue g = GeographyValue.fromWKT(poly);
       GeographyPointValue p = GeographyPointValue.fromWKT(point);

       voltQueueSQL( insertrec, name, p, g);
       return voltExecuteSQL();
   }
}
```

A third option is to use the .fromWKT() method to create instances of GeographyValue and GeographyPointValue in the client application and pass the data to the stored procedure as native geospatial types.

When retrieving geospatial data from the database, the ASTEXT() SQL function converts from a GEOGRAPHY or GEOGRAPHY_POINT value to its textual representation. (You can also use the CAST(*value* AS VARCHAR) function). In a stored procedure or Java client application, you can use the `.toString()` method of the `GeographyValue` or `GeographyPointValue` class.

6.3. Working With Geospatial Data

In addition to the classes, methods, and functions to insert and extract geospatial data from the database, VoltDB provides other SQL functions to help you manipulate the data. The functions fall into three main categories:

- Converting to and from WKT representations:

  ASTEXT()
  POLYGONFROMTEXT()
  POINTFROMTEXT()

VALIDPOLYGONFROMTEXT()

- Performing geospatial calculations:

  AREA()
  CENTROID()
  CONTAINS()
  DISTANCE()
  DWITHIN()
  LATITUDE()
  LONGITUDE()

- Analyzing the structure of a region:

  MAKEVALIDPOLYGON()
  ISVALID()
  ISINVALIDREASON()
  NUMINTERIORRINGS()
  NUMPOINTS()

The following sections provide examples of using these functions on both locations and regions.

6.3.1. Working With Locations

For geospatial locations, the data is often available as numeric values — longitude and latitude — rather than as WKT. In this case, you need to convert the numeric data to WKT before converting and inserting it as a GEOGRAPHY_POINT value.

For example, The *VoltDB Tutorial* uses data from the US Geographic Names Information Service (GNIS) to create a database of geographic locations. The original source data also includes the longitude and latitude of those locations. So it is easy to modify the database schema to add a location for each town:

```
CREATE TABLE towns (
   town VARCHAR(64),
   state VARCHAR(2),
   state_num TINYINT NOT NULL,
   county VARCHAR(64),
   county_num SMALLINT NOT NULL,
   location GEOGRAPHY_POINT,
   elevation INTEGER
);
```

However, the incoming data includes two floating point values rather than a GEOGRAPHY_POINT value or WKT. One solution is to create a simple stored procedure to perform the conversion to WKT and insert the record using the POINTFROMTEXT() function:

```
public class InsertTown extends VoltProcedure {

  public final SQLStmt insertrec = new SQLStmt(
      "INSERT INTO TOWNS VALUES (?,?,?,?,?,POINTFROMTEXT(?),?);"
  );

  public VoltTable[] run(   String t, String s, byte sn,
                            String c, short cn,
                            double latitude, double longitude,
```

```
                            long e)
        throws VoltAbortException {
            String wkt = "POINT( "  +
                 String.valueOf(longitude) + " " +
                 String.valueOf(latitude) + ")";
            voltQueueSQL( insertrec, t,s,sn, c, cn, wkt, e);
            return voltExecuteSQL();
        }
}
```

Once the data is imported into the database, it is possible to use the geospatial functions to perform meaningful queries on the locations, such as determining which town is closest to a specific location (such as a cell phone user):

```
SELECT town, state FROM TOWNS
   ORDER BY DISTANCE(location,CAST(? AS GEOGRAPHY_POINT))
   ASC LIMIT 1;
```

Or which town is furthest north:

```
SELECT town, state FROM TOWNS ORDER BY LATITUDE(location) DESC LIMIT 1;
```

6.3.2. Working With Regions

The textual representation for regions, or polygons, are not as easily constructed as geographic points. Therefore if you do not have region data already in WKT, your client application will need to generate WKT from whatever source data you are using.

Once you have the WKT representation, you can insert the data using a simple stored procedure similar to the example given above for locations. Since the data is already in WKT, you can even define the stored procedure using a CREATE PROCEDURE AS statement. The following example defines a table for storing information about the names and regions of national parks. It also defines the insert procedure for ingesting records from existing WKT data:

```
CREATE TABLE parks (
   park VARCHAR(64),
   park_code VARCHAR(2),
   border GEOGRAPHY
);
CREATE PROCEDURE InsertPark AS
   INSERT INTO parks VALUES (?,?, POLYGONFROMTEXT(?) );
```

As mentioned before, VoltDB does not validate the structure of the GEOGRAPHY polygon on input. So, if you are not positive the WKT representation meets the rules for a valid polygon, you should use the ISVALID() function on the GEOGRAPHY value before or after insertion to verify that your data is correct. For example, the following SQL statement uses the ISVALID() and ISINVALIDREASON() functions to report on all invalid park regions and the reason for the exception:

```
SELECT park, park_code, ISINVALIDREASON(border)
  FROM Parks WHERE NOT ISVALID(border) ORDER BY park;
```

Alternately, you can use the VALIDPOLGYONFROMTEXT() function which combines the POLYGONFROMTEXT() and ISVALID() functions into a single function, ensuring that only valid polygons are generated. The preceding InsertPark can be rewritten to validate the incoming data like so:

```
CREATE PROCEDURE InsertPark AS
```

```
INSERT INTO parks VALUES (?,?, VALIDPOLYGONFROMTEXT(?) );
```

Of course, the rewritten procedure will take incrementally longer because it performs both the conversion and validation. However, it performs these functions in a single step. The VALIDPOLYGONFROMTEXT() function will also correct simple errors in the WKT input. Specifically, it will correct any rings where the vertices are listed in the wrong direction.

Once you know your GEOGRAPHY data is valid, you can use the geospatial SQL functions to perform meaningful queries on the data. (If the polygons are not valid, the geospatial functions will not generate an error but will also *not* produce meaningful results.) The functions that perform calculations on GEOGRAPHY values are:

- AREA() — the area of a region
- CENTROID() — the geographic center point of a region
- CONTAINS() — Whether a region contains a given point
- DISTANCE() — distance between a point and a region (or between two points)

For example, the following SQL queries determine the three largest parks, what parks are closest to a given town, and what towns are contained with the region of a given park:

```
SELECT park, AREA(border) FROM Parks
   ORDER BY AREA(border) DESC LIMIT 3;

SELECT p.park, DISTANCE(p.border,t.location)
   FROM parks AS P, towns AS T WHERE t.town=?
   ORDER BY DISTANCE(p.border,t.location) ASC LIMIT 5;

SELECT t.town FROM parks AS P, towns AS T
   WHERE p.park=? AND CONTAINS(p.border,t.location);
```

Chapter 7. Creating Custom Importers, Exporters, and Formatters

VoltDB includes built-in export and import connectors for a number of standard formats, such as CSV files, JDBC, Kafka topics, and so on. If you have a data source or destination not currently covered by connectors provided by VoltDB, you could write a custom application to perform the translation. However, you would then need to manually coordinate the starting and stopping of your application with the starting and stopping of the database.

A better approach is to create a custom import or export connector. Custom connectors run within the VoltDB process and use the standard mechanisms in VoltDB for synchronizing the running of the connector with the database itself. You write custom connectors as Java classes, packaged in a JAR file, which VoltDB can access at runtime. This chapter provides instructions and sample code for writing, installing, and configuring custom export and import connectors. It also describes how to write custom formatters that can be used to interpret the input coming from an import connector.

7.1. Writing a Custom Exporter

Warning

Note that the VoltDB export subsystem has been extensively enhanced and improved and the original custom export interface is now deprecated, since it no longer supports all the necessary features. The following sections describe the latest custom interface (introduced in VoltDB V8), which uses the same method names, but uses different signatures.

To use the new, supported interface, the onBlockStart(), onBlockCompletion(), and processRow() methods must accept a single exportRow object, as described in the following sections. The older, deprecated interface where the methods accept no arguments, or in the case of processRow(), two arguments, will no longer be supported after the next major release.

An export connector, known internally as an ExportClient, is a Java class that receives blocks of row data when data is inserted into a stream within the database. The export connector is responsible for formatting and passing those rows to the downstream export target. The following sections describe:

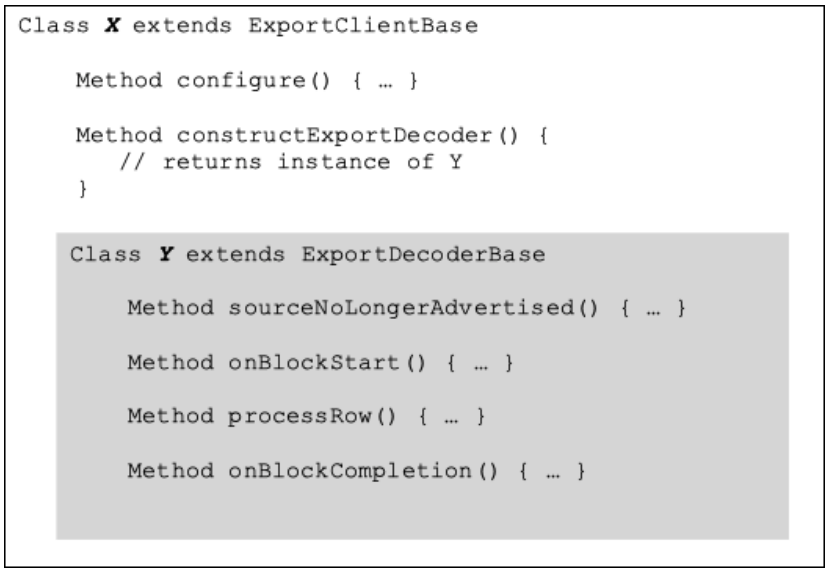
- The Structure and Workflow of the Export Client
- How to Use Custom Properties to Configure the Client
- How to Compile and Install the Client
- How to Configure the Export Client

7.1.1. The Structure and Workflow of the Export Client

The custom export client is declared as a Java subclass extending ExportClientBase. Within the subclass you can override the configure() method, which receives any properties defined in the export connection configuration. The subclass must also declare the constructExportDecoder() method, which in turn generates a subclass of ExportDecoderBase. This extension of ExportDecoderBase is the class that does the actual work at run time and must override the primary methods sourceNoLongerAdvertised(),

onBlockStart(), processRow(), and onBlockCompletion. Figure 7.1, "Structure of the Custom Export Class" illustrates the structure of the custom client.

Figure 7.1. Structure of the Custom Export Class



At run time, VoltDB passes data to the export client in blocks that are roughly 2MB in size but do *not* align with transactions. A block is guaranteed to contain complete rows — that is, no single SQL INSERT to an export stream is split across blocks. The handoff from the internal VoltDB producer to the custom export client follows a simple pattern:

```
producer -> client.onBlockStart
foreach row in block:
    producer -> client.processRow
producer -> client.onBlockCompletion
```

Each time the pattern executes, it runs within a single thread. Therefore, it is not necessary to synchronize accesses to the data structures used in client.onBlockStart, client.processRow, and client.onBlockCompletion unless they are used in other threads as well.

For each row of data, the processRow() method is called. The object passed in as an argument contains a full description of the row, including the column names, types, and values, which your client code can iterate over. For example:

```
public boolean processRow(ExportRow row)
              throws RestartBlockException {

   for (int i =0; i < row.values.length; i++) {
      String column_name = row.names.get(i);
      Object column_value = row.values[i];
          // do work . . .
   }
```

Note that each row starts with six columns of metadata, including the transaction ID and timestamp. If you do not need this information, you can skip the first six columns. Also, within each block of export data, the schema is constant. However, it is possible for the schema to change between blocks (if a schema change is applied to the database while export is active). The custom client can evaluate the ExportRow

object passed into the onBlockStart() or processRow() method to recognize any changes to the schema and configure the downstream system as necessary to accept the new data.

As a rule, the custom client must balance the requirements to execute and return control quickly, so as not to block other export threads (since connectors share a limited thread pool), against the need to ensure that the individual rows are accepted and acknowledged by the downstream system before onBlockCompletion() returns. If the client fails at onBlockStart, processRow or onBlockCompletion, the export client must throw a RestartBlockException to prevent VoltDB from acknowledging (ACKing) and dropping the export data from its durability control. This point deserves repeating: if the custom ExportClient runs onBlockStart, processRow and onBlockCompletion without throwing the correct exception, VoltDB assumes the data is remotely durable and that the VoltDB database can discard that export block.

The ExportClient must not return from onBlockCompletion until it ensures the downstream target acknowledges receipt of the data.

7.1.2. How to Use Custom Properties to Configure the Client

Properties, set in the deployment file as part of the export configuration, let you pass information to the export connector. For example, if the user needs to pass the export connector a file location or IP address for the export target. What properties are necessary or supported is up to you as the author of the export client to decide.

The properties specified in the deployment file are passed to the export client as a Properties object argument to the configure() method every time the connector starts. That is, whenever the database starts with the connector enabled or whenever the schema or deployment is modified (for example, by a **voltadmin update** command).

The configure() method can either iterate over the Properties object or it can look for specific entries as needed. For example:

```
public void configure(Properties config) throws Exception {

        // Check for specific property value
    if config.containsKey("filename") {
       filename = config.getProperty("filename");
     }
}
```

7.1.3. How to Compile and Install the Client

Once your export client code is complete, you need to compile, package, and install the connector on the appropriate VoltDB servers. You compile the export client like other Java methods. Be sure to include the VoltDB server jar file in the classpath. For example, if VoltDB is installed in a directory called voltdb in your home directory, the command could be:

```
$ javac -cp "$HOME/voltdb/voltdb/*:./" -d obj \
   org.voltdb.exportclient/MyExportClient.java
```

After compiling the source code, you must package the resulting class into a JAR file, like so:

```
$ jar cvf myexportclient.jar -C obj .
```

Finally you must install the JAR file in the lib/extension folder where VoltDB is installed on *all* servers in the cluster that will be running the export client. For, example, if you are running a single node cluster

on the current node, where VoltDB has been installed as $HOME/voltdb, you can copy the JAR file with the following command:

```
$ cp myexportclient.jar $HOME/voltdb/lib/extension/
```

7.1.4. How to Configure the Export Client

Once your custom export client is installed you can configure and start it. Custom export clients are configured like any other export connector, by adding a `<configure>` section to `<export>` in the deployment file (or configuring it interactively in the VoltDB Management Center). For custom clients, you declare the connector `type` as "custom" and add the `exportconnectorclass` attribute specifying the connector's Java classpath. For example:

```
<export>
   <configuration enabled="true" target="myspecial" type="custom"
    exportconnectorclass="org.voltdb.exportclient.MyExportClient" >
     <property name="filename">myexportfile.txt</property>
  </configuration>
</export>
```

Any properties listed in the `<configuration>` ("filename" in this example) are passed to the custom export client as arguments to the configure() method, as described in Section 7.1.2, “How to Use Custom Properties to Configure the Client”. See the chapter on "Importing and Exporting Live Data" in the *Using VoltDB* manual for more information on configuring export connectors.

7.2. Writing a Custom Importer

An import connector is a set of Java classes that configure the connector and then iteratively retrieve data from a remote source and pass it into VoltDB by invoking stored procedures. Unlike the export connector, which is responsible for formatting the data between source and target, the VoltDB import architecture allows for the use a separate formatter to translate the inbound data into a set of Java objects that can be passed as parameters to a stored procedure.

Import connectors are packaged as OSGi (Open Service Gateway Initiative) bundles, so they can be started and stopped easily from within the server process. However, for the purposes of writing a custom importer, VoltDB handles much of the OSGi infrastructure in the abstract classes associated with the import client. As a result, your import connector only needs to provide some of the classes normally required for an OSGi bundle. Specifically, a custom importer must provide the classes and methods described in Table 7.1, “Structure of the Custom Importer”.

Table 7.1. Structure of the Custom Importer

Class	Method	Description
implementation of **ImporterConfig**	*class constructor*	
	getFormatterBuilder()	Returns the FormatterBuilder method of the specified format.
	getResourceID()	Returns a unique resource ID for this importer instance.
	URI()	Returns the URI for the current importer instance.
extension of **AbstractImporterFactory**	create()	Returns an instance of the AbstractImporter implementation.

Class	Method	Description
	createImporterConfigurations()	Returns a map of configuration information.
	getTypeName()	Returns the name of the AbstractImporter class as a string.
	isImporterRunEverywhere()	Returns true or false.
extension of **AbstractImporter**	getName()	Returns the name of the AbstractImporter class as a string.
	accept()	Performs the actual data import. Should check to see if stop() has been called.
	stop()	Completes the import process.
	URI()	Returns the URI for the current importer instance.

Having all the right parts in place is extremely important, since if the bundle is incomplete or incorrect, the server process will crash when the importer starts. So the best way to create a new custom importer is to take an existing example — including the associated ant build script — and modify it as needed. You can find an example custom importer in the VoltDB public github repository at the following URL:

https://github.com/VoltDB/voltdb/tree/master/src/frontend/org/voltdb/importclient/socket

The following sections describe:

- Writing the custom importer using Java
- Compiling, packaging, and installing the importer
- Configuring and running the importer

7.2.1. Designing and Coding a Custom Importer

One of the most important decisions you must make when planning your custom importer is whether to run a single importer process for the cluster or to design a run-everywhere importer. A single importer process ensures only one instance of the importer is running at any given time. That means on a cluster, only one node will run the import connector process.

The following sections discuss run-everywhere vs. single process and managing the starting and stopping of the import connector.

7.2.1.1. Run-Everywhere vs. Single Process

A *run-everywhere* import connector starts a separate import process on each node in the cluster. A run-everywhere connector can improve performance since it distributes the work across the cluster. However, it means that the connector must negotiate the distribution of the work to avoid importing duplicate copies of the data.

Run-everywhere connectors are especially useful where the import process uses a "push" model rather than a "pull'. That is, if the connector opens a port and accepts data sent to the port, then the data source(s) can proactively connect and "push" data to that port, making the data source responsible for the distribution to the multiple servers of the VoltDB cluster.

You specify whether you are creating a single importer process or run-everywhere connector in the isImporterRunEverywhere() method of the Importer class. If the method returns true, importer processes

are created on every server. If the method returns false, only one importer process is created at any given time.

7.2.1.2. Managing the Starting and Stopping of the Import Process

When the custom importer is enabled, the ImporterFactory create() method is invoked, which in turn creates instances of the ImporterConfig and Importer classes. The VoltDB import infrastructure then calls the Importer accept() method for each importer process that is created.

The accept() method does the bulk of the work of iteratively fetching data from the appropriate sources, calling the formatter to structure the content of each row, and invoking the appropriate stored procedure to insert the data into the database. Two important points to keep in mind:

- If the accept() method fails for any reason of returns to the caller, the importer will stop until the next time it is initialized. (That is, when the database restarts or is paused and resumed.)
- On each iteration, the accept() method should check to see if the close() method has been called, so it can clean up any pending imports and then return to the caller.

7.2.2. Packaging and Installing a Custom Importer

Once the custom importer code is ready, you need to compile and package it as an OSGi-compliant JAR file. There are a number of OSGi properties that need to be set in the JAR file manifest. So it is easiest to use an ant build file to compile and package the files. The following is an excerpt from an example build.xml file for a custom importer project:

```
<!-- Simple build file to build socket stream importer -->
<project name="customimport"      basedir="." default="customimporter">
<property name='base.dir'         location='.' />
<property name='bundles.dir'      location='./bundles' />
<property name='build.dir'        location='./obj' />

   <target name="buildbundle" depends="customimporter"/>

   <resources id="default.imports.resource">
      <string>org.osgi.framework;version=&amp;quot;[1.6,2)&amp;quot;</string>
      <string>org.voltcore.network</string>
      <string>org.voltdb.importer</string>
      <string>org.voltdb.client</string>
      <string>org.voltdb.importer.formatter</string>
      <string>org.apache.log4j</string>
      <string>org.slf4j</string>
      <string>jsr166y</string>
      <string>org.voltcore.utils</string>
      <string>org.voltcore.logging</string>
      <string>com.google_voltpatches.common.base</string>
      <string>com.google_voltpatches.common.collect</string>
      <string>com.google_voltpatches.common.net</string>
      <string>com.google_voltpatches.common.io</string>
      <string>com.google_voltpatches.common.util.concurrent</string>
   </resources>

   <pathconvert property="default.imports"
                refid="default.imports.resource" pathsep=","/>
```

```
   <target name="customimporter">

     <!-- Compile source files -->
     [ . . . ]

     <!-- Build OSGi bundle -->
     <antcall target="osgibundle">
         <param name="bundle.name" value="mycustomimporter"/>
         <param name="activator" value="MyCustomImporterFactory"/>
         <param name="bundle.displayname" value="MyCustomImporter"/>
         <param name="include.classpattern" value="mycustomimporter/*.class"/>
      </antcall>
   </target>

   <target name="osgibundle">
      <mkdir dir="${bundles.dir}" />
      <jar destfile="${bundles.dir}//${bundle.name}.jar" basedir="${build.dir}">
         <include name="${include.classpattern}"/>
         <manifest>
            <attribute name="Bundle-Activator" value="${activator}" />
            <attribute name="Bundle-ManifestVersion" value="2" />
            <attribute name="Bundle-Name"
                       value="${bundle.displayname} OSGi Bundle" />
            <attribute name="Bundle-SymbolicName"
                       value="${bundle.displayname}" />
            <attribute name="Bundle-Version" value="1.0.0" />
            <attribute name="DynamicImport-Package" value="*" />
            <attribute name="Import-Package" value="${default.imports}" />
         </manifest>
      </jar>
   </target>
</project>
```

Once you create the OSGi bundle, you install the custom importer by copying it to the `bundles` folder inside the root directory of the VoltDB installation on every server in the cluster. For example, if VoltDB is installed in `/opt/voltdb`, copy your custom importer JAR file to `/opt/voltdb/bundles/`.

7.2.3. Configuring and Running a Custom Importer

Once the custom importer is installed on the VoltDB servers, you can configure and start the importer using the database configuration file. You can configure import either before the database starts or after the database is running using the **voltadmin update** command.

In the configuration use the `<import>` and `<configuration>` elements to declare your custom importer. Specify the `type` as "custom" and identify the custom importer bundle in the `module` attribute specifying the name of the JAR file. For example:

```
<import>
   <configuration type="custom" module="mycustomimporter.jar">
     [ . . . ]
```

If the custom importer requires additional information, you can provide it in properties passed to the ImporterConfig class. For example:

```
<import>
   <configuration type="custom" module="mycustomimporter.jar">
     <property name="datasource">my.data.source</property>
     <property name="timeout">5m</property>
  </configuration>
</import>
```

As soon as the configuration is enabled, the import processes will be initialized and the custom importer accept() method invoked by the VoltDB import infrastructure.

7.3. Writing a Custom Formatter

A formatter is a module that takes a row of data received by an import connector, interprets the contents, and translates it into individual column values. The default formatter that is provided with VoltDB parses comma-separated values (CSV) data. However, if the data you are importing is in a different format, you can write a custom formatter to perform this translation step.

You provide a custom formatter as an OSGi (Open Service Gateway Initiative) bundle. However, much of the standard work of an OSGi bundle is handled by the VoltDB import framework. So you only need to provide selected components as described in the following sections.

Note

Custom formatters can be used with both custom and built-in import connectors and with the standalone kafkaloader utility.

The following sections describe:

- The structure of the custom formatter
- Compiling and packaging custom formatter bundles
- Installing and invoking custom formatters
- Using custom formatters with the kafkaloader utility

7.3.1. The Structure of the Custom Formatter

The custom formatter must contain at least two Java classes: one that implements the `org.voltdb.importer.formatter.Formatter` interface and one that extends the `org.voltdb.importer.formatter.AbstractFormatterFactory` interface.

For the sake of example, let's assume the custom formatter classes are called MyFormatter and MyFormatterFactory. When the associated import connector is initialized, the VoltDB importer infrastructure calls the classes' methods in the following order:

- `MyFormatterFactory.create`() is called once, to initialize the formatter. The `create` method must return an instance of the MyFormatter class.
- `MyFormatter.MyFormatter()` is invoked once when an instance of the MyFormatter class is initialized in the preceding step.
- MyFormatter.transform`()` is called from the import connector every time it retrieves a record from the data source.

In many cases, the easiest way to create custom class is to modify an existing example. And VoltDB provides an example formatter that you can use as a base for your customizations in the VoltDB github at the following URL:

https://github.com/VoltDB/voltdb/tree/master/tests/test_apps/kafkaimporter/custom_formatter/formatter

The next sections describe how to modify this example — or how to create a custom formatter from scratch, if you wish.

7.3.1.1. The AbstractFormatterFactory Interface and Class

You must create a class that extends the AbstractFormatterFactory class. However, within that class all you need to change is overriding the create() method to return an instance of your implementation of the Formatter interface. So, assuming the new class names use the prefix "MyFormatter" and using the example formatter provided in github, all you need to modify are the items highlighted in the following example:

```
package myformatter;

import org.voltdb.importer.formatter.AbstractFormatterFactory;

public class MyFormatterFactory extends AbstractFormatterFactory {
    /**
     * Creates and returns the formatter object.
     */
    @Override
    public MyFormatter create() {
        MyFormatter formatter = new MyFormatter(m_formatName, m_formatProps);
        return formatter;
    }
}
```

7.3.1.2. The Formatter Interface and Class

The bulk of the work of a custom formatter occurs in the class that implements the Formatter interface. Within that class, you must have at least one method that overrides the default `transform()` method. You can, optionally, include a method that initializes the class and handles any properties that need to be passed into the formatter from the import configuration.

7.3.1.2.1. Initializing the Formatter Class

The method that initializes the class has the same name as the class (in our example, MyFormatter). The method accepts two parameters: a string and a list of properties. The string contains the name of the formatter as specified in the database configuration file (see Section 7.3.3.2, “Configuring and Invoking Custom Formatters”). This string will, by definition, match the name of the class itself. The second parameter is a collection of Java Property objects representing properties set in the configuration file using the `<format-property>` element and certain VoltDB built-in properties, whose names all start with two underscores.

If the custom formatter doesn't require any information from the configuration, you do not need to include this method. However, if your formatter does require additional information, this class can retrieve and store information provided in the import configuration. For example, the `MyFormatter()` method in the following implementation looks for a "column_width" property and stores it for later use by the `transform()` method:

```
package myformatter;

import java.util.Properties;
import org.voltdb.importer.formatter.FormatException;
import org.voltdb.importer.formatter.Formatter;

public class MyFormatter implements Formatter {

    String column_width = "";

    MyFormatter (String formatName, Properties prop) {
        column_width = prop.getProperty("column_width");
    }
```

7.3.1.2.2. Transforming the Data

The method that does the actual work of formatting the incoming data is the transform() method. This method receives the incoming data as a Java byte buffer and is expected to return an array of Java objects representing the input parameters, which will be passed to the specified stored procedure to insert the data into the database.

For example, If the custom formatter expects data in fixed-width columns, the method might look like this:

```
@Override
public Object[] transform(ByteBuffer payload) throws FormatException {

   String buffer = new String(payload.array());
   ArrayList<Object> list = new ArrayList<Object>();

   int position = 0;
   while (position < buffer.length()) {
      int endpoint = Math.min(position+column_width, buffer.length());
      list.add(buffer.substring(position,endpoint));
      position += column_width;
   }
   return list.toArray();
}
```

7.3.2. Compiling and Packaging Custom Formatter Bundles

Once the custom formatter source code is complete, you are ready to compile and package the formatter as an OSGi bundle.

When compiling the source code, be sure to include the VoltDB JAR files in the Java classpath. For example, if VoltDB is installed in the folder `/opt/voltdb`, you will need to include `/opt/voltdb/voltdb/*` and `/opt/voltdb/lib/*` in the classpath.

You will also need to include a number of OSGi-specific attributes in the final JAR file manifest. For example, you must include the *Bundle-Activator* attribute pointing to the FormatterFactory class. To ensure all the necessary properties are set, it is easiest to use the ant utility and an ant build file. The following is an example `build.xml` file, with the items that you must modify highlighted in bold text:

```
<project default="build">
   <path id='project.classpath'>
      <!-- Replace this with the path to the VoltDB jars -->
```

```
      <fileset dir='/opt/voltdb'>
        <include name='voltdb/*.jar' />
        <include name='lib/*.jar' />
      </fileset>
  </path>

  <target name="build" depends="clean, dist, formatter"/>

  <target name="clean">
    <delete dir="obj"/>
    <delete file="myformatter.jar"/>
  </target>

  <target name="dist">
    <mkdir dir="obj"/>
    <javac srcdir="src" destdir="obj">
      <classpath refid="project.classpath"/>
    </javac>
  </target>

  <target name="formatter">
    <jar destfile="myformatter.jar" basedir="obj">
      <include name="myformatter/MyFormatter.class"/>
      <include name="myformatter/MyFormatterFactory.class"/>
      <manifest>
        <attribute name="Bundle-Activator"
                   value="myformatter.MyFormatterFactory" />
        <attribute name="Bundle-ManifestVersion" value="2" />
        <attribute name="Bundle-Name" value="My Formatter OSGi Bundle" />
        <attribute name="Bundle-SymbolicName" value="MyFormatter" />
        <attribute name="Bundle-Version" value="1.0.0" />
        <attribute name="DynamicImport-Package" value="*" />
      </manifest>
    </jar>
  </target>
</project>
```

7.3.3. Installing and Invoking Custom Formatters

Once you have built and packaged the custom formatter, you are ready to install and use it in your VoltDB infrastructure.

7.3.3.1. Installing Custom Formatters

To install the custom formatter, you simply copy the formatter JAR file (in the preceding examples, `myformatter.jar`) to the `bundles` folder in the VoltDB installation on every server in the cluster. For example, if VoltDB is installed in `/opt/voltdb`:

```
$ cp obj/myformatter.jar /opt/voltdb/bundles/
```

7.3.3.2. Configuring and Invoking Custom Formatters

Once the JAR file is available to all VoltDB instances, you can configure and invoke the custom formatter as part of the import configuration. Note that the import configuration can be changed either before the

database cluster is started or while the database is running using either the **voltadmin update** command of the web-based VoltDB Management Center.

You choose the formatter as part of the import configuration using the `format` attribute of the `<configuration>` element in the database configuration file. Normally, you use the built-in "csv" format. However, to select a custom formatter, set the `format` attribute to the name of the formatter JAR file and its class name. For example:

```
<import>
   <configuration type="kafka" format="myformatter.jar/MyFormatter" >
     [ . . . ]
```

Storing your custom JAR in the bundles directory is recommended. However, if you choose to keep your custom code elsewhere, you can still reference it in the configuration by including the absolute path to the file location as part of the `format` attribute. For example, if your JAR file is in the /etc/myapp folder, the `format` attribute value would be "file:/etc/myapp/myformatter.jar/MyFormatter". The formatter JAR must be in the same location on *all* nodes of the cluster.

Within the import configuration, you can also include any properties that the formatter needs using the `<format-property>` element. For example, in the preceding example, the custom formatter expects a property called "column_width", so the configuration might look like this:

```
<import>
   <configuration type="kafka" format="myformatter.jar/MyFormatter" >
     <property name="brokers">kafka.myorg.org:9092</property>
     <property name="topics">customer</property>
     <property name="procedure">CUSTOMER.insert</property>
     <format-property name="column_width">15</format-property>
  </configuration>
<import>
```

7.3.4. Using Custom Formatters With the kafkaloader Utility

You can also use custom formatters with the standalone kafkaloader utility. To use a custom formatter with kafkaloader you must:

- Declare environment variables for FORMATTER_LIB and ZK_LIB
- Create a formatter properties file specifying the formatter class and any formatter-specific properties the formatter requires.

The environment variables define the paths to the formatter JAR file and the Apache ZooKeeper libraries, respectively. (Note that ZooKeeper does not need to be running, but you must have a copy of the standard ZooKeeper libraries installed and accessible via the ZK_LIB environment variable.)

The formatter properties file must contain, at a minimum, a "formatter" property that is assigned to the formatter class of the custom formatter. It can contain other properties required by the formatter. The following is the properties file for kafkaloader that matches the example given in the previous section to configure the custom formatter using the built-in importer infrastructure:

```
formatter=MyFormatter
column_width=15
```

If both your formatter and the ZooKeeper libraries are in a folder `myformatter` under your home directory, along with the preceding properties file, you could start the kafkaloader utility with the following commands to use the custom formatter:

```
$ export FORMATTER_LIB="$HOME/myformatter/"
$ export ZKLIB="$HOME/myformatter/"
$ kafkaloader --formatter=$HOME/myformatter/formatter.config \
  --topic=customer --zookeeper=kafkahost:2181
```

Chapter 8. Creating Custom SQL Functions

VoltDB provides many built-in functions for use in SQL statements that are described in an appendix to the Using VoltDB manual. These built-in functions perform a variety of tasks, such as data conversion, string matching, geometric calculations, and so on.

There are two types of SQL functions: *scalar* functions and *aggregate* functions. A scalar function takes a set of arguments and produces a single result value. DATEADD() is an example of a scalar function that takes a timestamp, a value, and a unit type such as MINUTE or HOUR, and produces a value that represents the addition of the specified units to the input timestamp. An aggregate functions takes multiple values — a table column or expression — as part of a query and produces a value that aggregates the results from all of the input. MAX() is an example of a built-in aggregate function that calculates the maximum value of all the inputs associated with the query's constraints.

However, not all possible functions are built in and there may be cases where you have an application-specific function that needs to be performed repeatedly. Rather than writing code to perform the operation as part of the application or within a stored procedure, VoltDB lets you create and declare your own functions that can be invoked directly from within SQL queries and data manipulation statements just like built-in functions.

Just as there are two types of built-in functions, you can create two types of user-defined functions. For user-defined scalar functions, you write the function as a Java method that takes the specified input arguments and returns a single value. User-defined aggregate functions are slightly more complex. You create aggregate functions as a Java class with specific methods to initiate the function, collect and process the input values, and return the aggregated result.

In both cases, there are the same three steps to creating a *user-defined function*:

1. Write the code to perform the function as a Java method (for scalar functions) or a Java class (for aggregate functions).
2. Load the Java class that includes the user-defined function into the database.
3. Declare the function using the CREATE FUNCTION or CREATE AGGREGATE FUNCTION statement, associating it to the Java class and/or method.

The following sections describe how to perform each of these tasks, as well as how to invoke the function in SQL statements once it has been declared.

8.1. Writing a User-Defined Scalar Function

You write user-defined scalar functions as Java methods. If you are creating multiple scalar functions, you can put them all in a single Java class or in separate Java classes. Whichever is most convenient for you and the management of your code base.

The number and datatypes of the method's parameters define the number and types of the function's arguments. For example, if you declare the method as having two parameters, a Java `int` and `String`, the function will have two arguments, a VoltDB INTEGER and a VARCHAR. Similarly, the return datatype of the method itself determines the datatype that the function returns.

Because user-defined functions are executable within SQL statements and stored procedures, the methods must obey the same rules concerning determinism as stored procedures. That is, avoid any actions

that introduce values that may vary from one system to another, such as system time, random number generation, or I/O with indeterminate results. See the section on determinism in the *Using VoltDB* manual for details.

For example, say you need to convert distances from imperial or US units to metric measurements. You might define your function with two arguments: a floating-point value representing the measurement and a string unit identifying the units (such as "feet", "yards", or "miles"). So your Java source code would need to declare the method as accepting two parameters: a `double` and a `String`. It should also be declared as returning a `double` value.

```
package myapp.sql.functions;
import org.voltdb.*;
public class Conversion {

  public double us2metric( double value, String units )
              throws VoltAbortException {
```

Note the method is declared as throwing a VoltAbortException. This is useful for error handling. By throwing a VoltAbortException, it is possible for the function to handle errors gracefully and notify the VoltDB runtime to rollback the current transaction. For example, the first step in the method might be to validate the input and make sure the units argument has a known value:

```
units = units.toUpperCase().trim();
if (!units.equals("FEET") &&
    !units.equals("YARDS") &&
    !units.equals("MILES") )
    throw new VoltAbortException("Unrecognized selector.");
```

The bulk of the method will focus on performing the actual task the function is designed for. The key point is to make sure it returns the appropriate datatype object that correlates to the VoltDB datatype you want the function to return in the SQL statement. In the previous example, the method is declared as returning a `double`, which matches the VoltDB FLOAT type. See the appendix on Datatype compatibility in the *Using VoltDB* manual for details on the mapping of Java and VoltDB datatypes. But, in brief, the mapping of Java to VoltDB datatypes is as follows:

byte or Byte → TINYINT
short or Short → SMALLINT
int or Integer → INTEGER
long or Long → BIGINT
double or Double → FLOAT
BigDecimal → DECIMAL
String → VARCHAR
byte[] or Byte[] → VARBINARY

You can define parameters for VoltDB-specific datatypes by using the object types included in the VoltDB Java packages. For example:

org.voltdb.types.GeographyValue → GEOGRAPHY
org.voltdb.types.GeographyPointValue → GEOGRAPHY_POINT
org.voltdb.types.TimestampType → TIMESTAMP

8.2. Writing a User-Defined Aggregate Function

Aggregate functions are more complex than scalar functions because the program code does not receive a predefined number of arguments, it must handle a variable number of rows. It must also be able to

aggregate results from multiple partitions in the case of partitioned tables. To make this possible, user-defined aggregate functions are written as a Java class with four required methods:

- `start()` — This method initializes the function and is called once when the function is first invoked.
- `assemble( arg )` — This method processes the argument to the function and is called once for each record matching the constraints defined by the query.
- `combine( class-instance )` — For partitioned tables in a multi-partition procedure, this method combines the results of one partition into the results of another and is called iteratively until the results from all of the partitions are aggregated.
- `end()` — This method finalizes the function, returning the function result. It is called once after all of the input has been processed and, if appropriate, the partitions combined.

Let's take a look at a simple example of an aggregate function to understand how this works. The MAX() built-in function returns the maximum value from a set of inputs and CHAR_LENGTH() returns the length of a text string. So you can combine them to find the length of the longest string. But what if you want to know what is the longest word in the set of strings? For this you need to write your own aggregate function.

First, the function must import and implement the `VoltUDAggregate` interface. Within the class, the datatype of the `assemble` method's parameter defines the type of the function's argument. For example, if you declare the method as having one String parameter, the function will accept one argument, a VoltDB VARCHAR expression. Similarly, the datatype of the `end` method determines the datatype that the function returns. See the appendix on Datatype compatibility in the *Using VoltDB* manual for details on the mapping of Java and VoltDB datatypes.

So the Java source code of a longest word function might look like the following:

```
package fadvisor.functions;

import java.io.Serializable;
import org.voltdb.VoltUDAggregate;

public class LongestWord
   implements Serializable, VoltUDAggregate<String, LongestWord> {

      private String longest;

      public void start() {
        /* Initialize value. */
        longest = "";
      }

      public void assemble (String txt) {
         /* Break into uppercase words, then check each word. */
         String[] words = txt.toUpperCase().split("\\W+");
         for (int i=0;i<words.length;i++) {
            if (words[i].length() > longest.length()) {
                longest = words[i];
            } else {
                /* If the same length, alphabetize. */
                if ( words[i].length() == longest.length() &
                     longest.compareTo(words[i]) > 0 ) {
                     longest = words[i];
```

```
                }
            }

        }
    }

    public void combine(LongestWord other) {
       /* Merge the results from each partition. */
       if (other.longest.length() > longest.length()) {
          longest = other.longest;
       } else {
          if ( other.longest.length() == longest.length() &
               longest.compareTo(other.longest) > 0 ) {
               longest = other.longest;
          }
       }
    }

    public String end() {
       /* Return the result. */
      return longest;

    }
}
```

The bulk of the `assemble` and `combine` methods perform the actual task the function is designed for. The key point is to make sure the end method returns the appropriate datatype object that correlates to the VoltDB datatype you want the function to return in the SQL statement. In this example, the method is declared as returning a `String`, matching the VoltDB VARCHAR type.

8.3. Loading User-Defined Functions into the Database

Once you have written, compiled, and tested the Java code for your SQL functions, you can load them into the database. You load user-defined functions the same way you load stored procedures, by packaging the Java class or classes into a JAR file and then loading the JAR file using the LOAD CLASSES statement. For example:

```
$ export CLASSPATH="$CLASSPATH:/opt/voltdb/voltdb/*"
$ javac -d ./obj src/myapp/functions/*.java
$ jar cvf myfunctions.jar -C obj .
$ sqlcmd
1> load classes myfunctions.jar;
```

You can package multiple function methods and classes or a combination of functions and stored procedure classes into a single JAR file. The key is that you must load the classes containing the aggregate functions and scalar function methods into the database before you can declare them as SQL functions.

8.4. Declaring a User-Defined Function

Once the Java class containing the method is loaded, you can declare the function itself. You declare scalar functions using the CREATE FUNCTION statement, specifying the name of the function and the

associated Java class path and method name. For example, if you want to call the function associated with the scalar example US2METRIC, the CREATE FUNCTION statement looks like this:

```
CREATE FUNCTION US2METRIC FROM METHOD myapp.functions.Conversion.us2metric;
```

You declare aggregate functions using the CREATE AGGREGATE FUNCTION statement, specifying the name of the function and the associated Java class. For example, if you want to call the function associated with the aggregate example LONGEST_WORD, the CREATE AGGREGATE FUNCTION statement looks like this:

```
CREATE AGGREGATE FUNCTION LONGEST_WORD FROM CLASS myapp.functions.LongestWord;
```

Note that although the function names are not case sensitive, the class path and method names *are* and must be specified in the correct mix of upper and lower-case.

8.5. Invoking User-Defined Functions in SQL Statements

Once the class is loaded and the function declared, you can include the user-defined function in SQL queries and data manipulation (DML) statements just as you normally use built-in functions in SQL. For example, if a stored procedure needs to convert an entry in miles to the equivalent measurement in kilometers, the stored procedure definition might look like the following:

```
CREATE PROCEDURE Expense_in_miles AS
    INSERT INTO Expense_item (item_name, distance, destination)
    VALUES (?, US2METRIC(?,'MILES'), ?);
```

Similarly, if you want a procedure to find the longest word in each product description for a specific category, the stored procedure definition might look like this:

```
CREATE PROCEDURE big_words AS
    SELECT product_name, longest_word(description) FROM products
      WHERE category=? GROUP BY product_name;
```

Note that user-defined functions can be used in queries and DML but *cannot* be used in data definition statements, such as CREATE statements that define indexes, tables, or views.

Chapter 9. Creating Custom Tasks

You can schedule common repetitive tasks using the CREATE TASK SQL statement. CREATE TASK lets you schedule any system or user-defined stored procedure to be run on a specified schedule. However, there are times when you need the schedule, or the procedure itself, to be more responsive: adjusting to the current state of the database or the results of the previous task runs. This is what custom tasks help you accomplish.

Custom tasks let you modify the characteristics of the task, including:

- What procedure is executed by the task
- The arguments to the task procedure
- The time interval between executions of the task

The following sections describe how to design, develop and implement custom tasks.

9.1. Overview of How Custom Tasks Work

You write custom tasks as Java classes implementing one of three interfaces, depending on what you want to customize:

- **ActionGenerator** — Customizes the stored procedure to call and the parameters to use
- **IntervalGenerator** — Customizes the interval between procedure calls
- **ActionScheduler** — Customizes all three aspects of the task: the stored procedure to call, its parameters, and the interval between calls

Once you write the class for a custom task, you implement it in VoltDB in two steps:

1. You load the class into the database the same way you would a stored procedure, by compiling and packaging it as a JAR file and then loading the JAR file into the database with the LOAD CLASSES directive.
2. You then declare the custom task using the CREATE TASK statement, replacing the argument to the appropriate clause with FROM CLASS. For example, a custom procedure call replaces `PROCEDURE {procedure-name}` with `PROCEDURE FROM CLASS {class-name}`, a custom interval replaces `ON SCHEDULE {type-and-value}` with `ON SCHEDULE FROM CLASS {class-name}`, and a custom scheduler replaces both the `PROCEDURE` and `ON SCHEDULE` clauses with `FROM CLASS {class-name}`. You can optionally specify arguments to the custom class using the `WITH` clause.

For example, the following SQL statements declare one custom task of each type, assuming their classes are packaged in the JAR `mytasks.jar`:

```
LOAD CLASSES mytasks.jar;
CREATE TASK variableproc
   ON SCHEDULE EVERY 5 SECONDS
   PROCEDURE FROM CLASS MyVariableProc WITH (1,2,3);
CREATE TASK variableinterval
   ON SCHEDULE FROM CLASS MyVariableInterval WITH (1,2,3)
   PROCEDURE cleanupchoir;
CREATE TASK variabletask
   FROM CLASS MyVariableTask WITH (1,2,3);
```

In all three cases, the sequence of events at run time is the same. Once you declare and enable a custom task, the database:

1. Invokes the custom task's `initialize()` method once, passing the parameters specified in the WITH clause of the CREATE TASK statement.

2. Invokes the custom task's `getFirst<item>()` method once. The actual method name varies depending on the interface; `getFirstAction()` for actions, `getFirstInterval` for intervals, and `getFirstScheduledAction()` for both action and interval. The method must return an object with the custom values and a callback method. VoltDB uses this information to schedule the first invocation of the task.

3. Iteratively, invokes the specified callback method, passing the results of the stored procedure as an argument. The callback then repeats the process of returning an object with the next custom values and callback.

This sequence of events is repeated any time the database is restarted, the schema is changed, or the database is paused and then resumed.

The following sections provide examples of using each type of customization. See the javadoc for additional details about the classes and methods that can assist your task design.

9.2. Modifying the Procedure Call and Arguments

To vary the stored procedure called by a task or the arguments to that procedure, you can create a custom task that implements the ActionGenerator interface. The custom task consists of the Java class you create, and the SQL statements to load and declare the task. At run time, for each invocation of the task, VoltDB uses the custom class to determine what procedure to call and its arguments and the task declaration to determine when to run the task.

Let's look at an example. Say we want a task that periodically deletes unused sessions. The stored procedure to do this might look like the following, where the batch size — the maximum number of records to delete — is passed in as an argument to the procedure:

```
CREATE PROCEDURE PurgeOldSessions DIRECTED AS
    DELETE FROM session
       WHERE last_access < DATEADD(MINUTE,-5,NOW())
       ORDER BY sessionID  ASC LIMIT ?;
```

We could schedule this procedure using a static task and a fixed batch size. But then there is no guarantee that the task can keep up with the volume of expired sessions. Instead, we would like to be able to adjust the batch size to accommodate changing workloads.

To do this we can create a custom task that checks how many records were actually deleted in each run. If the count of deleted records is less than 80% of the batch size, reduce the batch size for the next run. On the other hand, if the number deleted equals the batch size, increase the batch size incrementally until it matches the workload. Finally, we can use parameters to the custom task to specify the minimum, maximum, and starting batch size.

9.2.1. Designing a Java Class That Implements ActionGenerator

The first step is to write a Java class that implements the ActionGenerator interface. This must be a static class whose constructor takes no arguments. In our example, we will also declare class variables for a

helper object that is used in subsequent methods plus minimum, maximum, and current values for the batch size.

The class must, at a minimum, declare or override three methods:

❶ **initialize()**

The initialize() method is called first and receives a helper object (inserted automatically by the task subsystem), plus any parameters defined by the task definition in SQL. In our example there will be three parameters from the task definition: a minimum, maximum, and starting value for the batch size, which the method stores in the variables declared earlier. So the initialize() method ends up having a total of four arguments.

❷ **GetFirstAction()**

The getFirstAction() method is called when the task starts; that is, when the task is first defined or when the database starts or resumes. The method must return an Action object, which includes the procedure to call, arguments to that procedure, and a callback method to be invoked once the first invocation is completed. In our example, the method creates an action using the PurgeOldSessions stored procedure, the initial batch size, and the callback procedure declared in the next step.

❸ **A callback method**

Finally, your custom task must have a callback method (the method you specify when creating an Action object), which is invoked once the specified procedure instance completes. The callback method must return another Action, similar to getFirstAction.

In our example, the callback compares the current batch size to the number of records deleted by the last run and makes appropriate adjustments. It decreases the batch size if less than 80% were deleted, it increases the batch size if the full batch was used, and it keeps the size within the specified minimum and maximum. In this case, we are changing the arguments only, not the stored procedure invoked. Although that is possible if your application requires it.

Note that the callback method can be a name of your choosing. It does not have to be callback(). You are also not constrained to using just one callback; you might select different callback methods based on which stored procedure you are invoking or the current application context.

Example 9.1, “Custom Task Implementing ActionGenerator” shows the completed example task class, with the key elements highlighted.

Example 9.1. Custom Task Implementing ActionGenerator

```
package mytasks;

import java.util.concurrent.TimeUnit;
import org.voltdb.VoltTable;
import org.voltdb.client.ClientResponse;
import org.voltdb.task.*;

public class PurgeBatches implements ActionGenerator {

    private TaskHelper helper;
    private long min, max, batchsize;

    public void initialize(TaskHelper helper,          ❶
                long min, long max, long batch) {
        this.min = min;
        this.max = max;
        this.batchsize = batch;
    }

    public Action getFirstAction() {                   ❷
        return Action.procedureCall(this::callback,
               "PurgeOldSessions", this.batchsize);
    }

        /*
        * Callback to handle the result of the task
        * and return next Action.
        */
    private Action callback(ActionResult result) {     ❸

        /* Find out how many records were deleted */
        ClientResponse response = result.getResponse();
        VoltTable[] results = response.getResults();
        long count = results[0].fetchRow(0).getLong(0);

         /* If less than 80%, decrease by 10% */
        if (count < this.batchsize * .8)
            this.batchsize -= this.max * .1;

        /* If equal to batch size, increase by 10% */
        if (count == this.batchsize)
            this.batchsize += this.max * .1;

            /* Stay within min & max */
        if (this.batchsize > this.max) this.batchsize = this.max;
        if (this.batchsize < this.min) this.batchsize = this.min;

        return Action.procedureCall(this::callback,
               "PurgeOldSessions", this.batchsize);
    }
}
```

9.2.2. Compiling and Loading the Class into VoltDB

Once you complete your Java source code, you need to compile, debug, and package it into a JAR file so it can be loaded into VoltDB. You compile and package task classes the same way you compile and package stored procedures. In fact, you can package tasks and procedures into the same or separate JARs if you choose. The following example compiles the Java classes in the `src` folder and packages them into the JAR file `sessiontasks.jar`:

```
$ javac -classpath "/opt/voltdb/voltdb/*" \
        -d ./obj  src/*.java
$ jar  cvf  sessiontasks.jar -C obj .
```

You then load the classes from the JAR file into VoltDB using the sqlcmd LOAD CLASSES directive:

```
LOAD CLASSES sessiontasks.jar;
```

9.2.3. Declaring the Task

Finally, once the custom class is loaded into the database, you can declare the task and start it running. You declare the task using the CREATE TASK statement, replacing a procedure name with the FROM CLASS clause specifying the classpath to your new class. In our example, the custom task also requires three arguments: a minimum, maximum and starting batch size.

Because the stored procedure that the custom class specifies is a directed procedure (that is, it runs separately on every partition on the cluster), the task must be declared to RUN ON PARTITIONS. If the procedure was not directed, the task could be run on PARTITIONS, DATABASE, or HOSTS. However, for partitioned tables it is often necessary to have partitioned or directed procedures if the procedure's statements need to both ORDER BY and LIMIT the rows.

The following statement creates the custom task with a minimum batch size of 100 records, a maximum of 2,000, and a starting size of 1,000.

```
CREATE TASK batchcleanup
  ON SCHEDULE EVERY 5 SECONDS
  PROCEDURE FROM CLASS mytasks.PurgeBatches
     WITH (100,2000,1000)
     RUN ON PARTITIONS;
```

Note that the task starts as soon as it is declared, unless you include the DISABLE clause. Alternately, you can use the ALTER TASK statement to change the state of the task. For example, the following statement disables our newly created task:

```
ALTER TASK batchcleanup DISABLE;
```

9.3. Modifying the Interval Between Invocations

To customize the interval between invocations of the task's stored procedure, you create a custom task that implements the IntervalGenerator interface. The custom task consists of the Java class you create, and the SQL statements to load and declare the task. At run time, for each invocation of the task, VoltDB uses the custom class to determine how long to wait before invoking the stored procedure specified in the task definition.

Let's look at an example. Say we want a task, similar to the previous example, that periodically deletes unused sessions. The task can use the same stored procedure as before, this time with a fixed batch size passed in as an argument when declaring the task:

```
CREATE PROCEDURE PurgeOldSessions DIRECTED AS
    DELETE FROM session
       WHERE last_access < DATEADD(MINUTE,-5,NOW())
       ORDER BY sessionID  ASC LIMIT ?;
```

Now we want to customize how frequently the procedure runs — increasing the frequency if it always deletes the full batch, or decreasing the frequency if fewer records are deleted each time. To do this we create a custom task that checks how many records were actually deleted in each run. If the count of deleted records is less than 80% of the batch size, increase the interval between runs. On the other hand, if the number deleted records equals the batch size, increase the frequency by reducing the interval. Finally, we can use parameters to the custom task to specify the minimum, maximum, and starting frequency, as well as the batch size.

9.3.1. Designing a Java Class That Implements IntervalGenerator

The first step is to write a Java class that implements the IntervalGenerator interface. This must be a static class whose constructor takes no arguments. In our example, we will also declare class variables for a helper object that is used in subsequent methods and minimum, maximum, and initial values for interval, plus the batch size.

The class must, at a minimum, declare or override three methods:

❶ **initialize()**

The `initialize()` method is called first and receives a helper object inserted automatically by the subsystem that manages tasks, plus any parameters defined by the task definition in SQL. In our example there are four parameters from the task definition, which the method stores in the variables declared earlier. So the `initialize()` method ends up having a total of five arguments.

❷ **GetFirstInterval()**

The `getFirstInterval()` method is called when the task starts; that is, when the task is first defined or when the database starts or resumes. The method must return an Interval object, which specifies the length of time to wait until the first execution of the task as well as a callback to invoke after the task runs. In our example, we initialize the Interval object with the current interval value, the MILLISECONDS time unit, and the callback defined in the next step.

❸ **A callback method**

Finally, your custom task must have a callback method (the method you specify when creating the Interval object), which is invoked when the task completes. The callback method must return another Interval, similar to `getFirstInterval()`.

In our example, the callback compares the batch size to the number of records deleted by the last run and makes appropriate adjustments to the next interval. It increases the frequency by reducing the interval if the full batch was used, it decreases the frequency by extending the interval if less than 80% was used, and it keeps the interval within the specified minimum and maximum.

Note that the callback method can be any name you choose. It does not have to be `callback()` and you can specify different callback methods each time if your application logic requires it.

Example 9.2, “Custom Task Implementing IntervalGenerator” shows the completed example task class, with the key elements highlighted.

Example 9.2. Custom Task Implementing IntervalGenerator

```
package mytasks;

import java.util.concurrent.TimeUnit;
import org.voltdb.VoltTable;
import org.voltdb.client.ClientResponse;
import org.voltdb.task.*;

public class PurgeIntervals implements IntervalGenerator {

    private TaskHelper helper;
    private long min, max, delta, batchsize;

    public void initialize(TaskHelper helper,                ❶
                long min, long max,
                long delta, long batch) {
        this.min = min;
        this.max = max;
        this.batchsize = batch;
    }

    public Interval getFirstInterval() {                     ❷
        return new Interval(this.delta, TimeUnit.MILLISECONDS,
                            this::callback);
    }

        /*
        * Callback to handle the result of the task
        * and return next Action.
        */
    private Interval callback(ActionResult result) {         ❸

        /* Find out how many records were deleted */
        ClientResponse response = result.getResponse();
        VoltTable[] results = response.getResults();
        long count = results[0].fetchRow(0).getLong(0);

         /* If less than 80%, increase by 10% */
        if (count < this.batchsize * .8)
            this.delta += this.max * .1;

        /* If equal to batch size, decrease by 10% */
        if (count == this.batchsize)
            this.delta -= this.max * .1;

            /* Stay within min & max */
        if (this.delta > this.max) this.delta = this.max;
        if (this.delta < this.min) this.delta = this.min;

        return new Interval(this.delta, TimeUnit.MILLISECONDS,
                            this::callback);
    }
}
```

9.3.2. Compiling and Loading the Class into VoltDB

Once you complete your Java source code, you compile, debug, and package it into a JAR file the same way you compile and package stored procedures. In fact, you can package tasks, procedures, and other classes (such as user-defined functions) into a single or separate JARs depending on your application and operational needs. The following example compiles the Java classes in the `src/` folder and packages them into the JAR file `sessiontasks.jar`:

```
$ javac -classpath "/opt/voltdb/voltdb/*" \
        -d ./obj  src/*.java
$ jar  cvf  sessiontasks.jar -C obj .
```

You then load the classes from the JAR file into VoltDB using the sqlcmd LOAD CLASSES directive:

```
LOAD CLASSES sessiontasks.jar;
```

9.3.3. Declaring the Task

Finally, once the custom class is loaded into the database, you can declare the task and start it running. You declare the task using the CREATE TASK statement, replacing the ON SCHEDULE static interval with FROM CLASS specifying the classpath of your new class. In our example, the custom task also requires four arguments: a minimum, maximum and starting interval, plus the batch size. (The same batch size passed to the stored procedure PurgeOldSessions.) The following statement creates the custom task with a minimum interval of 100ms (a tenth of a second), a maximum of 10 seconds, an initial interval of 1 second, and a batch size of 500 records.

```
CREATE TASK timecleanup
  ON SCHEDULE FROM CLASS mytasks.PurgeIntervals
     WITH (100,10000,1000,500)
  PROCEDURE PurgeOldSessions WITH (500)
     RUN ON PARTITIONS;
```

Because the stored procedure PurgeOldSessions is a directed procedure (that is, it runs separately on every partition on the cluster), the task must be declared to RUN ON PARTITIONS.

The task starts as soon as it is declared, unless you include the DISABLE clause. Alternately, you can use the ALTER TASK statement to change the state of the task. For example, the following statement disables our newly created task:

```
ALTER TASK timecleanup DISABLE;
```

9.4. Modifying Both the Procedure and Interval

To allow full customization of the task — including the procedure, its arguments, and the interval between invocations — you create a custom task that implements the ActionScheduler interface. The custom task consists of the Java class you create, and the SQL statements to load and declare the task. At run time, for each invocation of the task, VoltDB uses the custom class to determine what procedure to invoke, what arguments to pass to the procedure, and how long to wait before invoking it.

Let's look at an example. The previous examples create tasks for purging old sessions from a database. But how can you tell that these tasks work without a real application? One way is to create a test database with tasks to emulate the expected workload, generating new sessions on an ongoing basis.

For our example, we want a task that generates sessions, but not so many the number of sessions created exceeds the number deleted. So our custom task needs to perform two actions:

1. Check to see how many session records are in the database.

2. If the task hasn't reached its target goal, generate up to 1,000 new sessions. Then repeat step 1.

In other words, we need two separate procedures: one to count the number of session records and another to insert a new session record. To count all session records in the database, the first procedure, *CountSessions*, must be multi-partitioned. (It also checks for the maximum value of the session ID so it can generate an incrementally unique ID.) But the second procedure, *AddSession*, can be partitioned since it is inserting into a partitioned table. This way the task reproduces the actions and performance of a running application.

```
CREATE PROCEDURE CountSessions AS
    SELECT COUNT(*), MAX(sessionID) FROM session;

CREATE PROCEDURE AddSession PARTITION ON TABLE SESSION COLUMN sessionID AS
    INSERT INTO session (sessionID,userID) VALUES(?,?);
```

The custom task will decide which procedure to invoke, and how long to wait between invocations, based on the results of the previous execution. There are many ways to do this, but for the sake of example, our custom task uses two separate callback methods: one to evaluate the results of the *CountSessions* procedure and one to evaluate the results of the *AddSession* procedure, as described in the next section.

9.4.1. Designing a Java Class That Implements ActionScheduler

The majority of the work of a custom class is performed by a Java class that implements a task interface; in this case, the ActionScheduler interface. It must be a static class whose constructor takes no arguments. The class must, at a minimum, declare or override three methods:

❶ **initialize()**

The `initialize()` method is called first and receives a helper object inserted automatically by the subsystem that manages tasks, plus any parameters defined by the task definition in SQL. In our example there is one parameter in the task definition: the target number of records to create. So the `initialize()` method has a total of two arguments.

❷ **GetFirstInterval()**

The `getFirstInterval()` method is called when the task starts; that is, when the task is first defined or when the database starts or resumes. The method must return a ScheduledAction object, which specifies the length of time to wait until the first execution of the task, a callback to invoke after the task runs, plus the name of the procedure and any arguments the procedure requires. In our example, we initialize the ScheduledAction object with no wait time (that is, an interval of zero), the callback for the CheckSessions procedure, and the procedure itself.

❸ **callback methods**

After each iteration of the task, VoltDB invokes the specified callback procedure. In our example, there are two callback methods:

- *checkcallback* — The task specifies this as the callback method to invoke after each iteration of the *CheckSessions* stored procedure. The callback checks the results for the number of session records. (it also saves the highest value of the session ID so it can generate an incrementally unique ID for each new record.) If the number of records is less than the goal, it schedules the *AddSession*

procedure as the next task, specifying `loadcallback()` as the callback method. If the goal has been met, no more records need to be added so the callback backs off on the frequency (increasing the interval) and schedules the CheckSessions procedure again.

- *loadcallback* — After each execution of the *AddSession* procedure, the callback reduces the batch size by one. If the batch is not complete, the callback reschedules the AddSession procedure, waiting 2 milliseconds. If the batch of inserts has been completed (that is, the batch size is down to zero), the callback schedules the CheckSessions procedure, specifying its callback method `checkcallback()` to see if there is now a full complement of session records.

Two things to note about this process are that if the session table is filled to the specified goal, the checkcallback uses the ability to customize the interval to reduce the frequency of checking — to minimize the impact on other transactions. Also, besides scheduling different stored procedures at different times, it passes different arguments as well, inserting a unique session ID and randomized user ID for each *AddSession* invocation.

Example 9.3, "Custom Task Implementing ActionScheduler" shows the completed example task class, with the key elements highlighted.

Example 9.3. Custom Task Implementing ActionScheduler

```
package mytasks;

import java.util.Random;

import java.util.concurrent.TimeUnit;
import org.voltdb.VoltTable;
import org.voltdb.client.ClientResponse;
import org.voltdb.task.*;

public  class LoadSessions implements ActionScheduler {

  private TaskHelper helper;
  private long batch, wait, nextid, goal;

  public void initialize(TaskHelper helper, long goal) {              ❶
    this.goal = goal;
    this.wait = 0;
  }

  public ScheduledAction getFirstScheduledAction() {                  ❷
    return ScheduledAction.procedureCall(0, TimeUnit.MILLISECONDS,
       this::checkcallback, "CountSessions");
  }

  /*
   * Callbacks to handle the results of the check task
   * and the load task
   */

  private ScheduledAction checkcallback(ActionResult result) {        ❸

    ClientResponse response = result.getResponse();
    VoltTable[] results = response.getResults();
```

```
    long recordcount = results[0].fetchRow(0).getLong(0);
    this.nextid = results[0].fetchRow(0).getLong(1);

    if (recordcount == 0) this.nextid = 0; /* start fresh*/
    this.batch = this.goal - recordcount;

     if (this.batch > 0) {
       /* Start loading data. Max batch size is 1,000 records */
       this.batch = (this.batch < 1000) ? this.batch : 1000;
       this.nextid++;
       return ScheduledAction.procedureCall(2, TimeUnit.MILLISECONDS,
               this::loadcallback, "AddSession",
               this.nextid,randomuser());
    } else {
       /* schedule the next check. */
       this.wait += 500;
       if (this.wait > 60000) this.wait = 60000;
       return ScheduledAction.procedureCall(this.wait, TimeUnit.MILLISECONDS,
              this::checkcallback, "CountSessions");
    }
  }

  private ScheduledAction loadcallback(ActionResult result) {        ❸
    this.batch--;
    if (this.batch > 0 ) {
       /* Load next session */
       this.nextid++;
       return ScheduledAction.procedureCall(2, TimeUnit.MILLISECONDS,
               this::loadcallback, "AddSession",
               this.nextid,randomuser());
    } else {
       /* schedule the next check. */
       System.out.println(" Done.");
       this.wait = 0;
       return ScheduledAction.procedureCall(this.wait, TimeUnit.MILLISECONDS,
            this::checkcallback, "CountSessions");
    }
  }

  private long randomuser() { return new Random().nextInt(1001); }

}
```

9.4.2. Compiling and Loading the Class into VoltDB

Once you complete your Java source code, you compile, debug, and package it into a JAR file the same way you compile and package stored procedures. You can package tasks, procedures, and other classes (such as user-defined functions) into a single or separate JARs depending on your application and operational needs. The following example compiles the Java classes in the `src` folder and packages it into the JAR file `sessiontasks.jar`:

```
$ javac -classpath "/opt/voltdb/voltdb/*" \
        -d ./obj  src/*.java
$ jar  cvf  sessiontasks.jar -C obj .
```

You then load the classes from the JAR file into VoltDB using the sqlcmd LOAD CLASSES directive:

```
LOAD CLASSES sessiontasks.jar;
```

9.4.3. Declaring the Task

Once the custom class is loaded into the database, you can declare the task and start it running. You declare the task using the CREATE TASK statement, replacing both the ON SCHEDULE and PROCEDURE clauses with a single FROM CLASS clause specifying the classpath of your new class. In our example, the custom task also requires one argument: the target value for the maximum number of session records to create. The following statement creates the custom task with a goal of 10,000 records.

```
CREATE TASK loadsessions
  FROM CLASS mytasks.LoadSessions WITH (10000)
  RUN ON DATABASE;
```

Note that the task is defined to RUN ON DATABASE. This means only one instance of the task is running at one time. However, the stored procedures will run as defined; that is, the *CheckSessions* procedure will run as a multi-partitioned procedure and each instance of *AddSession* will be executed on a specific partition based on the unique partition ID passed in as an argument at run-time.

The task starts as soon as it is declared, unless you include the DISABLE clause. Alternately, you can use the ALTER TASK statement to change the state of the task. For example, the following statement disables our newly created task:

```
3> ALTER TASK loadsessions DISABLE;
```

Chapter 10. Understanding VoltDB Memory Usage

VoltDB is an in-memory database. Storing data in memory has the advantage of eliminating the performance penalty of disk accesses (among other things). However, with the complex interaction of VoltDB memory usage and how operating systems allocate and deallocate memory, it can be tricky understanding exactly how much memory is being used at any given time. For example, deleting rows of data can result in a temporary increase in memory usage, which seems counterintuitive at first.

This chapter explains how VoltDB uses memory, the impact of system memory allocation and deallocation functions on your database's memory utilization, and variables available to you to help control memory usage.

10.1. How VoltDB Uses Memory

The memory that VoltDB uses can be grouped, loosely, into three buckets:

- Persistent
- Semi-persistent
- Temporary

Persistent memory is, as you might expect, the memory used for storing actual database records, including tables, indexes, and views. The larger the volume of data in the database, the more memory required to store it. String and varbinary columns longer than 63 bytes are not stored in line. Instead they are stored as pointers to the content in a separate string storage area, which is also part of persistent memory.

Semi-persistent memory is used for temporary storage while processing SQL statements and certain system procedures. In particular, semi-persistent memory includes temporary tables and the undo buffer.

- Temporary tables are where data is processed as part of an SQL statement. For example, if you execute an SQL statement like `SELECT * FROM flight WHERE DESTINATION='LAX'`, all of the tuples meeting the selection criteria are copied into temporary tables before being returned to the initiator. If the stored procedure is multi-partitioned, each partition creates a copy of its tuples and the initiator merges the multiple copies.
- The undo buffer is also associated with the execution of SQL statements. Any tuples that are modified or deleted as part of an SQL statement are recorded in the undo buffer until the transaction is committed or rolled back.

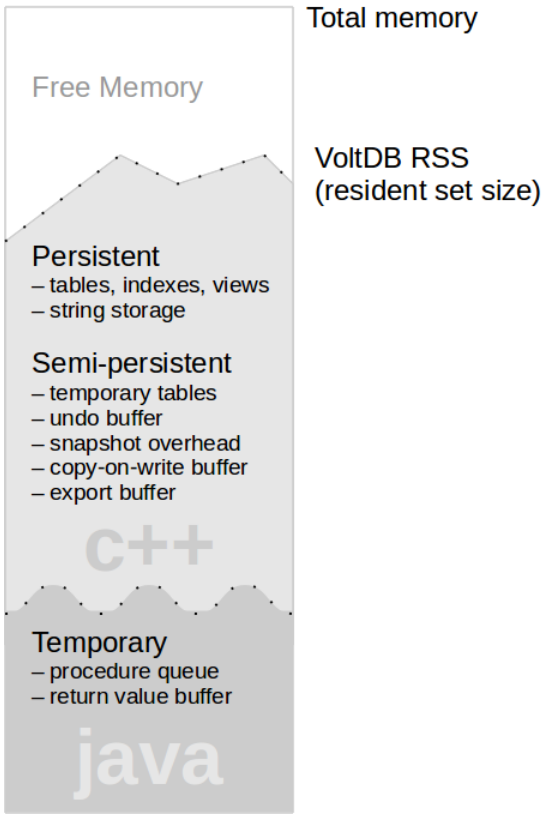
Semi-persistent memory is also used for buffers related to system activities such as snapshots and export. While a snapshot is occurring, a certain amount of memory is required for overhead, as well as copy-on-write buffers. Normally, snapshots are written directly from the tables in memory, thus requiring no additional overhead. However, if snapshots are non-blocking (performed asynchronously while other transactions are executing), any tuples that need to be modified before they are written to the snapshot get copied into semi-persistent memory. This technique is known as "copy-on-write". The consequence is that mixing asynchronous snapshots with frequent deletes and updates will increase the memory usage.

Similarly, when export is enabled, any insertions into export streams are written to an export buffer in semi-persistent memory until the export connector sends the data to the export target.

Temporary memory is used by VoltDB to manage the queueing and distribution of procedures to the individual partitions. Temporary memory includes the queue of pending procedure invocations as well as buffers for the return values for the completed procedures (until the client application retrieves them).

Figure 10.1, “The Three Types of Memory in VoltDB” illustrates how the three types of memory are allocated in VoltDB.

Figure 10.1. The Three Types of Memory in VoltDB



The sum of the persistent, semi-persistent, and temporary memory is what makes up the total memory (what is referred to as resident set size, or RSS) used by VoltDB on the server.

10.2. Actions that Impact Memory Usage

There are a number of actions that impact the amount of memory VoltDB uses during operation. Obviously, the more data that is stored within the partition (including all tables, indexes, and views), the more memory is required for persistent storage. Similarly for snapshotting and export, when these functions are enabled, they require some amount of semi-persistent storage. However, under normal conditions, the memory requirements for snapshotting and export should be relatively consistent over time.

Temporary storage, on the other hand, fluctuates depending on the workload and type of transactions being executed. If the client applications are "firehosing" (sending stored procedure requests faster than the servers can process them), the temporary storage required for pending procedure invocations will grow. Similarly, if the parameters being submitted to the procedures or the data being returned is large in size (up to 50 megabytes per procedure), the buffer for return values can grow significantly.

The nature of the workload also has an impact on the amount of semi-persistent storage. Read-only queries do not require space in the undo buffer. However, complex queries and queries that return large data sets

require space for temporary tables. On the other hand, update and delete queries can take up significant space in the undo buffer, especially when a single transaction (or stored procedure) performs multiple queries, each requiring undo support.

The use of the temporary and semi-persistent storage explains fluctuations that can be seen in overall memory utilization of servers running VoltDB. Although delete operations do eventually release memory used by the persistent storage, they initially require more memory in the undo buffer and for any temporary table operations. Once the entire transaction is complete and committed, the space in persistent storage and undo buffer is freed up. Note, however, that the unused space may not immediately be visible in the system RSS reports. The amount of memory *in use* and the amount of memory *allocated* can vary as a result of the interaction of several different memory management schemes that all come into play.

When VoltDB frees up space in persistent storage, it does not immediately return that memory to the operating system. Instead, it keeps track of unused space, which is then reused the next time a tuple is stored. Over time, memory can become fragmented. If the fragmentation reaches a preset level, the memory is compacted and unused space is deallocated and returned to the operating system.

Figure 10.2. Details of Memory Usage During and After an SQL Statement

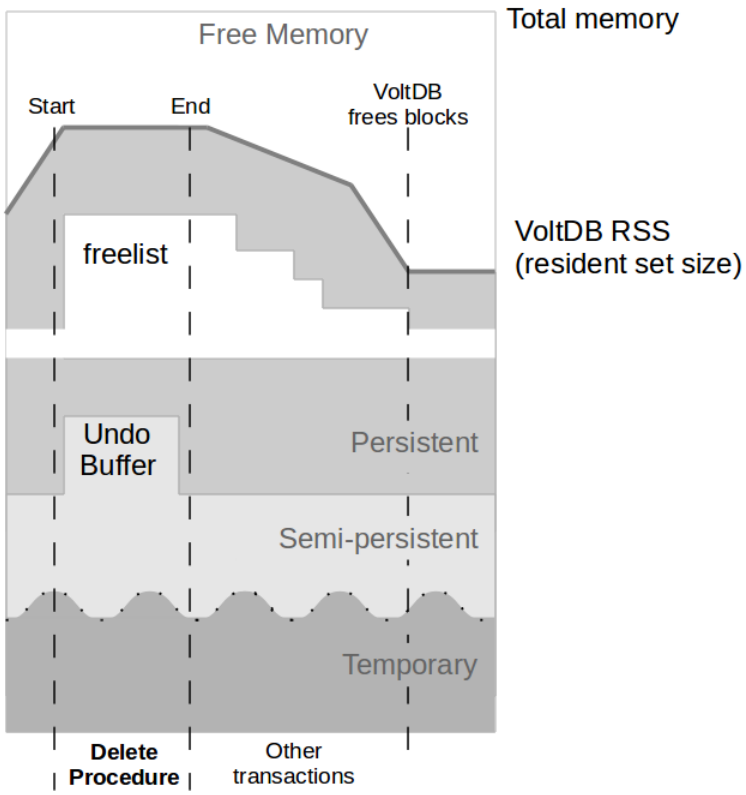


Figure 10.2, “Details of Memory Usage During and After an SQL Statement” illustrates how a delete operation can have a delayed effect on overall memory allocation.

1. At the beginning of the transaction, the deleted tuples are recorded in the semi-persistent undo buffer, increasing memory usage. Any freed persistent storage is returned to the VoltDB list of free space.

2. At the end of the transaction, the undo buffer is freed. However, the storage for the deleted tuples in persistent storage is managed and may not be released immediately.

3. Over time, free memory is used for new tuples, until...

4. At some point, VoltDB compacts any fragmented memory and releases unused blocks to the system.

How and when memory is actually deallocated depends on what that memory is being used for and how it is managed. The following section Section 10.3, “How VoltDB Manages Memory” describes how VoltDB manages memory in more detail.

Finally, there are some combinations of factors that can aggravate the fluctuations in memory usage. The memory required for snapshotting is usually not significant. However, if non-blocking snapshots are intermixed with update-heavy transactions, the snapshot copy-on-write buffer can grow rapidly.

Similarly, the memory used for export can grow if export is enabled but the connector cannot reach the target destination to clear the export buffer. However, the export buffer size is constrained; after a certain point any additional export data that is not acknowledged by the connector is written out as export overflow to disk. So memory used for export queues does not grow indefinitely.

10.3. How VoltDB Manages Memory

To manage memory effectively, VoltDB does not immediately release all unused memory. Allocating and deallocating small chunks of memory frequently can be expensive. Instead, VoltDB manages unused memory until larger chunks are available. Similarly, the Java runtime and the operating system perform their own memory pooling techniques.

As a result, RSS is not an exact measurement of actual memory usage. However, VoltDB offers statistics that provide a detailed breakdown of how it is using the memory that it has currently allocated. These statistics provide a more meaningful representation of VoltDB's memory usage than the lump sum allocation reported by the operating system RSS.

VoltDB manages memory for persistent and semi-persistent storage aggressively to ensure unused space is compacted and released when available. In some cases, memory is returned to the operating system, making the RSS more responsive to changes in the database contents. In other cases, where memory is managed as a pool of resources, VoltDB provides detailed statistics on what memory is allocated and what is actually in use.

Persistent storage for database tables (tuples) and indexes is compacted when fragmentation reaches a set percentage of total memory. Compaction eliminates fragmentation and allows memory to be returned to the operating system as the database volume changes. At the same time, storage for variable data such as strings and varbinary data greater than 63 bytes in length is being managed as a pool of resources. Free memory in the pool is not immediately returned to the operating system. VoltDB holds and reuses memory that is allocated but unused for these objects.

The consequence of these changes is that when you delete rows, the allocated memory for VoltDB (as shown by RSS) may go up during the delete operation (to allow for the undo buffer), but then it will go down — by differing amounts — based on the type of content that is deleted. Memory for tuples not containing large strings or binary data is returned to the operating system quickly. Memory for large string and binary data is not returned but is held for later reuse.

In other words, the pool size for non-inline string and binary data tends to reach a maximum size (based on the maximum required for your application workload) and then stabilize. Whereas memory for indexes as well as numeric and short string data oscillates as your application needs vary.

To help you understand these changes, the @Statistics system procedure tells you how much memory VoltDB is using and how much unused memory is being held for each type of content. These statistics provide a more accurate view of actual memory usage than the lump sum value of system RSS.

10.4. How Memory is Allocated and Deallocated

Technically, persistent and semi-persistent memory within VoltDB is managed using code written in C++. Temporary memory is managed using code written in Java. What language the source code is written in is not usually relevant, except in the case of memory, because different languages manage memory differently. C++ uses the traditional explicit allocation and deallocation of memory, where the application code controls exactly how and when memory is assigned and deassigned. In Java, memory is not explicitly allocated and deallocated. Instead, Java uses what is called "garbage collection" to free up memory that is not in use.

To complicate matters, the language libraries themselves do some performance optimizations to avoid allocating and deallocating memory from the operating system too frequently. So even if VoltDB explicitly frees memory in persistent or semi-persistent storage, that memory may not be immediately returned to the operating system or alter the process's perceived RSS value.

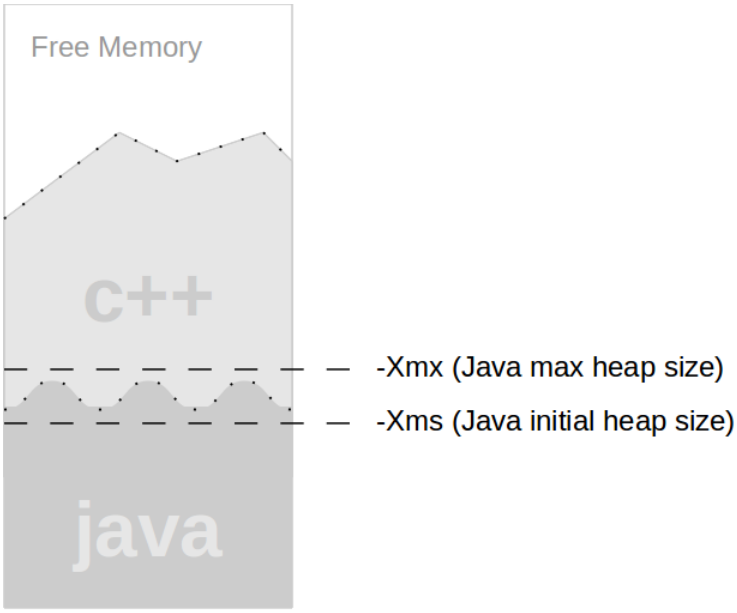
For temporary storage (which is managed in Java), VoltDB cannot explicitly control memory allocation and deallocation and relies on the Java virtual machine (JVM) to manage memory appropriately. The JVM decides when and how to collect free space from unused objects. This means that the VoltDB server cannot directly control if and when the memory associated with temporary storage is returned to the operating system.

10.5. Controlling How Memory is Allocated

Despite the fact that you as a developer or database administrator cannot control *when* temporary storage is allocated and freed, you can control *how much* memory is used. Java provides a way to specify the size of the heap, the portion of memory the JVM uses to store runtime data such as class instances, arrays, etc. The `-Xms` and `-Xmx` arguments to the `java` command specify the initial and maximum heap size, respectively.

By setting both the `-Xmx` and `-Xms` arguments, you can control not only the maximum amount of memory used, but also the amount of fluctuation that can occur. Figure 10.3, “Controlling the Java Heap Size” illustrates how the `-Xms` and `-Xmx` arguments can be used to control the overall size of temporary storage.

Figure 10.3. Controlling the Java Heap Size



However, you must be careful when setting the values for the Java heap size, since the JVM will not exceed the value you set as a maximum. It is possible, under some conditions, to force a Java out-of-memory error if the maximum heap size is not large enough for the temporary storage VoltDB requires. See the *VoltDB Planning Guide* for recommendations on calculating the appropriate heap size for your specific application.

Remember, temporary storage is used to queue the procedure requests and responses. If you are using synchronous procedures calls (and therefore little or no queuing on the server) a small heap size is acceptable. Also, if the size of the procedure invocations (in terms of the arguments passed into the procedures) and the return values are small, a lower heap size is acceptable. But if you are invoking procedures asynchronously with large argument lists or return values, be very careful when setting a low maximum heap size.

10.6. Understanding Memory Usage for Specific Applications

To help understand the memory usage for a specific VoltDB database, the @Statistics system procedure provides memory usage information. The "MEMORY" keyword returns a separate row of data for each server in the cluster, with columns providing information about the different aspects of memory usage, as described in the following table.

Column	Type of Storage	Description
JAVAUSED	Temporary	The amount of memory currently in use for temporary storage.
JAVAUNUSED	Temporary	The maximum amount of Java heap allocated but not currently in use.
TUPLECOUNT	Persistent	The number of tuples currently being held in memory.

Column	Type of Storage	Description
TUPLEDATA	Persistent	The amount of memory in use to store inline table data.
TUPLEALLOCATED	Persistent	The amount of memory allocated for table storage. This includes the amount in use and any free space currently being held by VoltDB.
INDEXMEMORY	Persistent	The approximate amount of memory in use to store indexes.
STRINGMEMORY	Persistent	The approximate amount of memory in use for non-inline string and binary storage.
POOLEDMEMORY	Persistent	The total amount allocated to pooled memory, including the memory assigned for storing strings, indexes, free lists, and metadata associated with tuple storage.
RSS	All	The resident set size for the VoltDB server process.

You can use periodic calls to the @Statistics system procedure with the "MEMORY" keyword to track your database cluster's memory usage in detail. But if you are only looking for an overall picture, VoltDB provides simple graphs at runtime.

Connect to a VoltDB server's HTTP port (by default, http://<server-name>:8080) to see graphs of basic memory usage for that server, including total resident set size (RSS), used Java heap and unused Java heap. Memory statistics are collected and displayed over three different time frames: 2 minutes, 30 minutes, and 24 hours. Click on the web browser's refresh button to update the charts.

Chapter 11. Managing Time

In VoltDB, differences in system clocks do not directly impact database latency since each partition operates indpendently. However, there are several database activities that need to be globally managed, such as when the database starts or failed nodes rejoin the cluster. For these activities, differences in clock time can impact — or, if the skew is large enough, even interrupt — proper operation.

That is why it is important to ensure a stable and consistent view of time within a VoltDB cluster using a time synchronization service such as NTP or chrony. This chapter presents some best practices for configuring and managing time using NTP.

If you are familiar with NTP or another service and have a preferred method for using it, you may want to read only Section 11.1, “The Importance of Time” and Section 11.2.2, “Troubleshooting Issues with Time”. If you are not familiar with NTP, this chapter suggests an approach that has proven to provide useful results in most situations.

The following sections explain:

- Why time is important to a VoltDB cluster
- How to use NTP to manage time across the cluster
- Special considerations when using VoltDB in a hosted or cloud environment

11.1. The Importance of Time

Because certain operations require coordination between the server nodes, it is important that they agree on what time it is. When the database process starts, VoltDB determines the maximum amount of skew (that is, the difference in clock time) between the individual nodes in the cluster. If the skew is greater than 200 milliseconds (2/10ths of a second), the VoltDB cluster refuses start.

11.2. Using NTP to Manage Time

NTP (Network Time Protocol) is a protocol and a set of system tools that help synchronize time across servers. The actual purpose of NTP is to keep an individual node's clock "accurate". This is done by having the node periodically synchronize its clock with a reference server. You can specify multiple servers to provide redundancy in case one or more time servers are unavailable.

The important point to note here is that VoltDB doesn't care whether the cluster view of time is "correct" from a global perspective, but it does care that they all have the same view. In other words, it is important that the nodes all synchronize to the same reference time and server.

11.2.1. Basic Configuration

To manage time effectively with NTP on a VoltDB cluster you must:

- Start NTP on each node
- Point each instance of NTP to the same set of reference servers

You start NTP by starting the NTP[1] service, or daemon, on your system. On many systems, starting the NTP daemon happens automatically on startup. You do not need to perform this action manually. However,

[1]The name of the NTP service varies from system to system. For Debian-based operating systems, such as Ubuntu, the service name is "ntp". For Red Hat-based distributions, such as CentOS, the service name is "ntpd".

if you need to make adjustments to the NTP configuration, it is useful to know how to stop and start the service. For example, the following command starts the daemon[2]:

```
$ service ntp start -x
```

You specify the time server(s) in the NTP configuration file (usually `/etc/ntp.conf`). You can specify multiple servers, one server per line. For example:

```
server clock.psu.edu
```

The configuration file is read when the NTP service starts. So, if you change the configuration file after NTP is running, stop and restart the service to have the new configuration options take affect.

11.2.2. Troubleshooting Issues with Time

In many cases, the preceding basic configuration is sufficient. However, there are issues that can arise time varies within the cluster.

If you are unsure whether a difference between the clocks in your cluster is causing performance issues for your database, the first step is to determine how much clock skew is present. When the VoltDB server starts it reports the maximum clock skew as part of its startup routine. For example:

```
INFO - HOST: Maximum clock/network skew is 12 milliseconds (according to leader)
```

If the skew is greater than 200 milliseconds, the cluster refuses to start. But even if the skew is around 100 milliseconds, the difference can delay certain operations and the nodes may drift farther apart in the future. The most common issues when using NTP to manage time are:

- Time drifts between adjustments
- Different time servers reporting different times

11.2.3. Correcting Common Problems with Time

The NTP daemon checks the time servers periodically and adjusts the system clock to account for any drift between the local clock and the reference server (by default, somewhere between every 1 to 17 minutes). If the local clock drifts too much during that interval, it may never be able to fully correct itself or provide a consistent time value to VoltDB.

You can reduce the polling interval by setting the minpoll and maxpoll arguments as part of the server definition in the NTP configuration file. By setting minpoll and maxpoll to a low value (measured as exponential values of 2 seconds), you can ensure that the VoltDB server checks more frequently. For example, setting minpoll and maxpoll to 4 (that is, 16 seconds), you ensure the daemon polls the reference server approximately every minute[3].

It is also possible that the poll does not get a response. When this happens, the NTP daemon normally waits for the next interval before checking again. To increase the likelihood of receiving a new reference time — especially in environments with network fluctuations — you can use the burst and iburst arguments to increase the number of polls during each internal.

By combining the burst, iburst, minpoll, and maxpoll arguments, you can increase the frequency that the NTP daemon synchronizes and thereby reduce the potential drift of the local server's clock. However, you should not use these arguments with public servers, such as the ones included in the NTP configuration

[2]Use of the -x option is recommended. This option causes NTP to "slew" time — slowly increasing or decreasing the clock to adjust time — instead of making one-time jumps that could create sudden changes in clock skew for the entire cluster.
[3]The default values for minpoll and maxpoll are 6 and 10, respectively. The allowable value for both is any integer between 4 and 17 inclusive.

file by default. Excessive polling of public servers is considered impolite. Instead, you should only use these arguments with a private server (as described in Section 11.2.4, “Example NTP Configuration”). For example, the ntp.conf entry might look like the following:

```
server myntpsvr iburst burst minpoll 4 maxpoll 4
```

Even if your system synchronizes with an NTP server, there can be skew between the reference servers themselves. Remember, the goal of NTP is to synchronize your system with a reference time source, not necessarily to reduce the skew between multiple local systems. Even if the polling frequency is improved for each node in a VoltDB cluster, the skew between them may never reach an acceptable value if they are synchronizing against different reference servers.

This situation is made worse by the fact that the most common host names for reference servers (such as ntp.ubuntu.com) are not actual IP addresses, but rather front ends to a pool of servers. So even if the VoltDB nodes have the same NTP configuration file, they might not end up synchronizing against the same physical reference server.

You can determine what actual servers your system is using to synchronize by using the NTP query tool (ntpq) with the -p argument. The tool displays a list of the servers it has selected, with an asterisk (*) next to the server currently in use and plus signs (+) next to alternatives in case the primary server is unavailable. For example:

```
$ ntpq -p
     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
+dns3.cit.cornel 192.5.41.209     2 u   14 1024  377   32.185    2.605   0.778
-louie.udel.edu  128.4.1.20       2 u  297 1024  377   22.060    3.643   0.920
 gilbreth.ecn.pu .STEP.          16 u    - 1024    0    0.000    0.000   0.000
*otc2.psu.edu    128.118.2.33     2 u  883 1024  377   29.776    1.963   0.901
+europium.canoni 193.79.237.14    2 u 1017 1024  377   90.207    2.741   0.874
```

Note that NTP does not necessarily choose the first server on the list and that the generic host names are resolved to different physical servers.

So, although it is normal to have multiple servers listed in the NTP configuration file for redundancy, it can introduce differences in the local system clocks. If the maximum skew for a VoltDB cluster is consistently outside of acceptable values, you should take the following steps:

- Change from using generic host names to specific server IP addresses (such as otc2.psu.edu or 128.118.2.33 in the preceding example)
- List only one NTP server to ensure all VoltDB nodes synchronize against the same reference point

Of course, using only one reference server for time introduces a single point of failure to your environment. If the reference server is not available, the database nodes receive no new reference values for time. The nodes continue to synchronize as best they can, based on the last valid reference time and historical information about skew. But over time, the clock skew within the cluster will start to drift.

11.2.4. Example NTP Configuration

You can provide both redundancy and maintain a single source for time synchronization, by creating your own NTP server.

NTP assumes a hierarchy (or strata) of servers, where each level of server synchronizes against servers one level up and provides synchronization to servers one level down. You can create your own reference server by inserting a server between your cluster nodes and the normal reference servers.

For example, assume you have a node myntpsvr that uses the default NTP configuration for setting its own clock. It can list multiple reference servers and use the generic host names, since the actual time does not matter, just that all cluster nodes agree on a single source.

Then the VoltDB cluster nodes list your private NTP server as their one and only reference node. By doing this, all the nodes synchronize against a single source, which has strong availability since it is within the same physical infrastructure as the database cluster.

Of course, there is always the possibility that access to your own NTP server could fail, in which case the database nodes need a fallback to ensure they continue to synchronize against the same source. You can achieve this by:

- Adding all of the cluster nodes as *peers* of the current node in the NTP configuration file
- Adding the current node (localhost) as its own server and setting it as a low level stratum (for example, stratum 10)

By listing the nodes of the cluster as peers, you ensure that when the reference server (myntpsvr in this example) becomes unavailable, the nodes will negotiate between themselves on an alternative source. At the same time, listing localhost (127.127.0.1) as a server tells the node that it can use itself as a reference server. In other words, the cluster nodes will agree among themselves to use one of their own as the reference server for synchronizing time. Finally, by using the fudge statement to set the stratum of localhost to 10, you ensure that the cluster will only pick one of its own members as a reference server for NTP if the primary server is unavailable.

Example 11.1, “Custom NTP Configuration File” shows what the resulting NTP configuration file might look like. This configuration can be the same on all nodes of the cluster, since `peer` entries referencing the current node are ignored.

Example 11.1. Custom NTP Configuration File

```
server myntpsvr burst iburst minpoll 4 maxpoll 4

peer voltsvr1 burst iburst minpoll 4 maxpoll 4
peer voltsvr2 burst iburst minpoll 4 maxpoll 4
peer voltsvr3 burst iburst minpoll 4 maxpoll 4

server 127.127.0.1
fudge 127.127.0.1 stratum 10
```

11.3. Configuring NTP in a Hosted, Virtual, or Cloud Environment

The preceding recommendations for using NTP work equally well in a managed or a hosted environment. However, there are some additional issues that can arise when working in a hosted environment that should be considered.

In a locally managed environment, you have complete control over both the hardware and software configuration. This means you can ensure that the VoltDB cluster nodes are connected to the same switch and in close proximity to a private NTP server, guaranteeing the best network performance within the cluster and to the NTP reference server.

In a hosted environment, you may not have control over the physical arrangement of servers but you usually have control of the software configuration.

In a virtualized or cloud environment, you have no control over — or even knowledge of — the hardware configuration. You are often using a predefined system image or "instance", including the operating system and time management configuration, which may not be appropriate for VoltDB. There are configuration changes you should consider making each time you "spin up" a new virtual server.

11.3.1. Considerations for Hosted Environments

In situations where you have control over the selection and configuration of the server operating system and services, the preceding recommendations for configuring NTP should be sufficient. The key concern would be those aspects of the environment you do not have control over: network bandwidth and reliability. Again, the recommended NTP configuration in Section 11.2.4, “Example NTP Configuration”, especially the use of a local timer server and peer relationship within the cluster, should provide reliable time management despite any network fluctuations.

11.3.2. Considerations for Virtual and Cloud Environments

In virtual or cloud environments, you usually do not have control over either the hardware or the initial software configuration. New servers are instantiated from a common system image, or "instance", with default configurations for the operating system and time management. This presents two problems for establishing a reliable environment for VoltDB:

- The default configuration may not be sufficient and must be overridden
- Because of the prior issue, there can be considerable clock skew that must be corrected before running VoltDB

Virtualization allows multiple virtual servers to run on a single piece of hardware. To do this, prepackaged "instances" of an operating system are booted under a virtual machine manager. These instances are designed to support the majority of applications, most of which do not have extensive requirements for clock synchronization, As a result, the instances often use default NTP configurations or none at all.

When you spin up a new virtual server, in most cases you need to reconfigure NTP, changing the configuration file as described in Section 11.2.4, “Example NTP Configuration” and restarting the service.

In some cases, NTP is not used at all. Instead, the operating system synchronizes its (virtual) clock against the clock of the physical server on which it runs. You need to override this setting before installing, configuring, and starting NTP. For example, when running early instances of Ubuntu in EC2 under the Xen hypervisor, you must modify the file `/proc/sys/xen/independent_wallclock` to avoid the hypervisor performing the clock synchronization. For example:

```
$ echo "1" > /proc/sys/xen/independent_wallclock
$ apt-get install -y ntp
```

This particular approach is specific to the Xen hypervisor. Other virtualization engines may use a different approach for controlling the system clock. See the documentation for your specific virtualization environment for details.

Once NTP is running and managing the system clock, it can take a considerable amount of time for the clocks to synchronize if the initial skew is large. You can reduce this initial delay by forcing synchronization before you start VoltDB. You can do this performing the following steps as the user `root`:

1. Stop the NTP service.
2. Use the `ntpdate` command to synchronize against a specific reference server. Do this several times until the reported skew is consistently low. (It will never effectively be less than a millisecond — a thousandth of a second — but can be reduced to a few milliseconds.)

3. Restart the NTP Service.

For example, if your local time server's IP address is 10.10.56.1, the commands might look like this:

```
$ service ntp stop
 * Stopping NTP server ntpd                                              [ OK ]
$ ntpdate -p 8 10.10.56.1
20 Oct 09:21:04 ntpdate[2795]: adjust time server 10.10.56.1 offset 0.008294 sec
$ ntpdate -p 8 10.10.56.1
20 Oct 09:21:08 ntpdate[2797]: adjust time server 10.10.56.1 offset 0.002518 sec
$ ntpdate -p 8 10.10.56.1
20 Oct 09:21:12 ntpdate[2798]: adjust time server 10.10.56.1 offset 0.001459 sec
$ service ntp start -x
 * Starting NTP server ntpd                                              [ OK ]
```

Once NTP is configured and the skew between the individual clocks and the reference server has been minimized, you can safely start the VoltDB database.