Guide to Performance and Customization

Abstract

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

V6.2

Guide to Performance and Customization

V6.2

This document was generated on April 11, 2016.

Table of Contents

List of Figures

List of Examples

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 eight 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, *Understanding VoltDB Memory Usage*
- Chapter 8, *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 six 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.
- Chapters 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 in detail how memory is used by the VoltDB server process.

- Chapter 8 provides guidelines for configuring hardware and operating systems for 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 in every partition. 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 in 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.

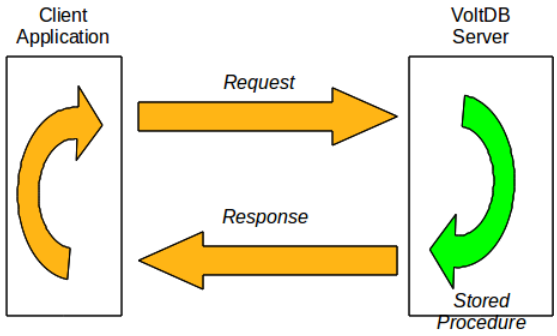
You can call VoltDB stored procedures either synchronously or asynchronously. When you call a stored procedure synchronously, your client application waits for the call to be processed before continuing. If you call a procedure asynchronously, your application continues processing once the call has been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

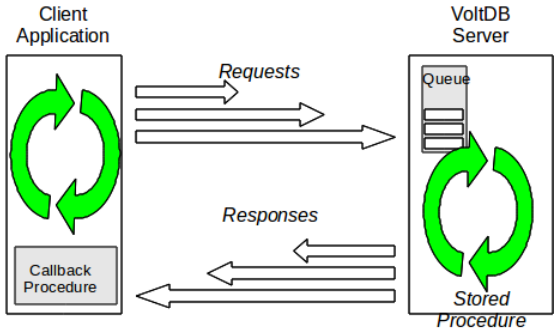
Figure 2.1. Synchronous Procedure Calls



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 Procedure-Callback 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

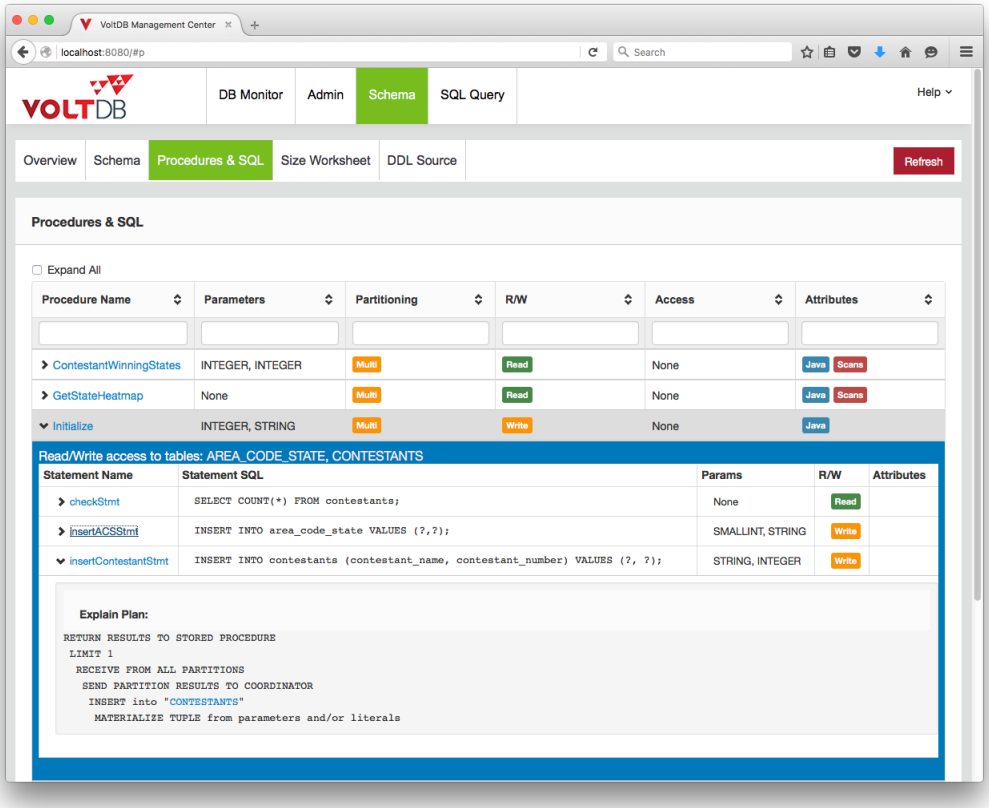
The system procedures and sqlcmd directives produce identical output. For example, if you enter the **explainproc** directive in sqlcmd with the name of a stored procedure, it displays the execution plan for every SQL statement defined in the stored procedure. You get the same results calling the @ExplainProc 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. Because the condition is a range (greater than or less than) rather than checking for an exact value match, it needs a tree rather than a hash index.

VoltDB creates tree indexes by default. However, you can explicitly specify the type of index by adding the string "tree" or "hash" (upper or lower case) to the constraint name. For example, we can explicitly specify that we want a tree index by using "tree" in the index name:

```
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 flightTimeTreeIdx ON FLIGHT ( departtime);
```

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

```
RETURN RESULTS TO STORED PROCEDURE
 INDEX SCAN of "FLIGHT" using "FLIGHTTIMETREEIDX"
 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 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.

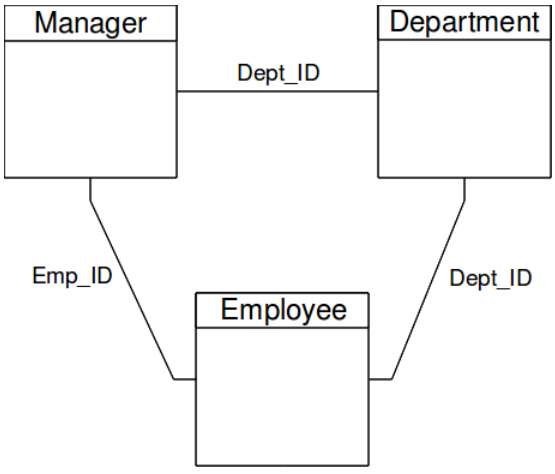
See the following chapter, Chapter 4, *Using Indexes Effectively*, for more information on tuning your database through effective use of indexes.

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 catalog, 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 catalog 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 catalog 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. Tree Indexes vs. Hash Indexes

Note the above discussion applies to the VoltDB default *tree* indexes. You can also define a more specialized hash index, which only covers queries that use equality or "IN" (limited to one "IN" filter per query, as above) to filter *all* of the keys in the index.

So, a hash index on column A requires a filter that looks like "A = ..." or "A IN ...". A hash index on A, B requires a filter that looks like "A = ... AND B = ..." or "A IN ... and B = ..." or "A = ... AND B IN ..." but will have no effect on only filters like "A = ..." or "A = ... AND B >= ..." etc.

The more flexible default tree indexes are recommended over hash indexes, except in special cases where equality lookup performance on all keys is super-critical and filters on just the prefix key or range filter performance on the complete set of key(s) is not critical. Even under these conditions, we recommend benchmarking hash indexes against tree indexes to verify their benefit for the scale of your data.

4.6. 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 catalog 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 new 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

Let’s assume that you want to implement a single sign-on (SSO) application using VoltDB. You wish to store the login session for a set of different online sites under a common username. Each login session could hold different user state, simple data values or possibly more complex structures. Additionally, future sessions could hold just about anything. Because of the variability of the data, a good strategy would be to JSON-encode it. The VoltDB table schema for this application might look like the following:

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

Common across all sessions would be the username, password, perhaps a global session ID, and a last accessed timestamp. Because you wish to support millions of simultaneous logins, it is best to partition the table based on the username column.

This schema is from the json-sessions sample application that comes with the VoltDB server software.

Ultimately, the sample inserts the JSON-encoded session into the database using a simple standard SQL statement:

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

The json-sessions sample models each type of session being tracked as a plain old Java object (POJO). To simplify encoding these session types into JSON, the sample uses an open source package from Google called GSON. GSON can convert POJOs to/from JSON, greatly simplifying the JSON processing in the example.

Note that VoltDB does not, at present time, validate that data inserted into a VARCHAR column is properly encoded JSON. Validation of encoding occurs during query time, as described in the next section.

5.2. Querying JSON Data in VoltDB

The VoltDB FIELD() column function helps you interact with JSON encoded data. Using the user_session_table schema above, let's assume the table is populated with rows similar to the following:

```
SELECT username, json_data FROM user_session_table
    ORDER BY username LIMIT 10

USERNAME    JSON_DATA
----------- ------------------------------------------------------------
user-1      {"read_only_user":true,"site":"VoltDB Management","props":
            {"last-login":"1356537244991"}}
user-10     {"role":"reader","site":"VoltDB Blog","props":{"last-login":
            "1356537252380"}}
user-1000   {"role":"reader","site":"VoltDB Blog","props":{"last-login":
            "1356537251017"}}
user-10000  {"read_only_user":false,"site":"VoltDB Management","props":
            {"last-login":"1356537246249"}}
user-10002  {"role":"reader","site":"VoltDB Blog","props":{"last-login":
            "1356537250566"}}
user-10003  {"read_only_user":false,"site":"VoltDB Management","props":
            {"last-login":"1356537252187"}}
user-10004  {"moderator":false,"download_count":0,"site":"VoltDB Forum",
            "props":{"last-login":"1356537244170"}}
user-10005  {"moderator":false,"download_count":0,"site":"VoltDB Forum",
            "props":{"last-login":"1356537250381"}}
user-10006  {"moderator":false,"download_count":0,"site":"VoltDB Forum",
            "props":{"last-login":"1356537245804"}}
user-10009  {"moderator":false,"download_count":0,"site":"VoltDB Forum",
            "props":{"last-login":"1356537249792"}}
```

By using the FIELD() function, a query can return only those rows where the login session is from the “VoltDB Forum”. The query and results would look as follows:

```
SELECT username, json_data FROM user_session_table
    WHERE field(json_data, 'site')='VoltDB Forum'
    ORDER BY username LIMIT 10

USERNAME     JSON_DATA
-----------  ------------------------------------------------------------------------
user-10004   {"moderator":false,"download_count":0,"site":"VoltDB Forum","props":
```

```
             {"last-login":"1356537244170"}}
user-10005   {"moderator":false,"download_count":0,"site":"VoltDB Forum","props":
             {"last-login":"1356537250381"}}
user-10006   {"moderator":false,"download_count":0,"site":"VoltDB Forum","props":
             {"last-login":"1356537245804"}}
user-10009   {"moderator":false,"download_count":0,"site":"VoltDB Forum","props":
             {"last-login":"1356537249792"}}
user-10013   {"moderator":false,"download_count":0,"site":"VoltDB  Forum","props":
             {"last-login":"1356537250681","client_language":"Java"}}
user-10014   {"moderator":false,"download_count":1,"site":"VoltDB Forum","props":
             {"last-login":"1356537251345","download_version":"v2.7"}}
user-10015   {"moderator":false,"download_count":0,"site":"VoltDB Forum","props":
             {"last-login":"1356537250817"}}
user-10016   {"moderator":false,"download_count":0,"site":"VoltDB Forum","props":
             {"last-login":"1356537244761"}}
user-10017   {"moderator":false,"download_count":1,"site":"VoltDB Forum","props":
             {"last-login":"1356537253137","download_version":"v3.0"}}
user-10027   {"moderator":false,"download_count":0,"site":"VoltDB Forum","props":
             {"last-login":"1356537248096","client_language":"Java"}}
```

Note that the FIELD() function assumes that the VARChAR column value is valid JSON. If the value is not valid JSON, the query fails with an appropriate error message.

Say you want to refine the result even further and find those Forum sessions that had downloaded any 2.x version content. You can use nested FIELD() function invocations to drill deeper into the JSON structure. For example, the following query fetches the properties for the VoltDB Forum session and then further extracts the download_version field, ultimately pattern matching on the value using the SQL LIKE clause:

```
SELECT username, json_data FROM user_session_table
    WHERE field(field(json_data, 'props'), 'download_version')
           LIKE 'v2%' ORDER BY username LIMIT 10

USERNAME    JSON_DATA
-----------           ------------------------------------------------------
user-10014  {"moderator":false,"download_count":1,"site":"VoltDB Forum",
            "props":{"last-login":"1356537251345","download_version":"v2.7"}}
user-10030  {"moderator":false,"download_count":1,"site":"VoltDB Forum",
            "props":{"last-login":"1356537250413","download_version":"v2.7"}}
user-10052  {"moderator":false,"download_count":1,"site":"VoltDB Forum",
            "props":{"last-login":"1356537250274","download_version":"v2.7"}}
user-10087  {"moderator":false,"download_count":1,"site":"VoltDB Forum",
            "props":{"last-login":"1356537247453","download_version":"v2.7"}}
user-10103  {"moderator":false,"download_count":1,"site":"VoltDB Forum",
            "props":{"last-login":"1356537247822","download_version":"v2.7",
            "client_language":"Java"}}
user-10170  {"moderator":false,"download_count":1,"site":"VoltDB Forum",
            "props":{"last-login":"1356537250308","download_version":"v2.7"}}
user-1018   {"moderator":false,"download_count":1,"site":"VoltDB Forum",
            "props":{"last-login":"1356537252219","download_version":"v2.7"}}
user-10223  {"moderator":false,"download_count":1,"site":"VoltDB Forum",
            "props":{"last-login":"1356537248629","download_version":"v2.7"}}
user-10226  {"moderator":false,"download_count":1,"site":"VoltDB Forum",
            "props":{"last-login":"1356537249328","download_version":"v2.7",
            "client_language":"Java"}}
```

```
user-10227  {"moderator":false,"download_count":1,"site":"VoltDB Forum",
            "props":{"last-login":"1356537252425","download_version":"v2.7"}}
```

5.3. Indexing JSON Fields

The queries executed in the previous section all require a full table scan to compute the results. With large data sets these queries could be costly in terms of compute cycles and time. To speed up query execution for these types of queries, should they be executed frequently, you can define an index on the commonly accessed fields. Again the FIELD() function comes into play. VoltDB supports defining function-based indexes.

To significantly improve the query execution time of the queries in the prior section, the following two indexes should be created:

```
CREATE INDEX session_site_moderator
    ON user_session_table (field(json_data, 'site'),
                field(json_data, 'moderator'), username);

CREATE INDEX session_props
    ON user_session_table
        (field(field(json_data, 'props'), 'download_version'),
         field(field(json_data, 'props'), 'client_language'),
         username);
```

These are fully functional SQL indexes. Whenever you create or update a record in the user_session_table table, VoltDB runs the FIELD() function to extract the specified field from the JSON value and stores the result inside the index. When you query by that same field in the future, VoltDB will use the index and avoid the table scan. Additionally, using the index usually avoids JSON string processing.

5.4. Summary: Considerations When Using JSON in VoltDB

One of the major benefits of encoding data as a JSON field is that you don not have to predefine what structure or shape that data will have. Further, the shape of the data can vary from one row to the next.

In the json-sessions example, the schema for the JSON column is defined by the Java objects themselves, on the fly, rather than having to define the structure using SQL ahead of time. If a new Java object, such as a new session type, is needed, you simply create the Java object, serialize it to JSON, and store it in VoltDB. This avoids the need to propagate a new catalog with DDL changes to the database.

On the flip-side, using a variable schema in the JSON column means that your application must be intelligent enough to understand the various structures that appear in that JSON column. Further, you must be sensitive to when indexes are needed, should query patterns change in your application along with the shape of the data. This may, for example, require you to add (or modify) indexes based on the existence of new fields that are now frequently queried.

Another point of note is the size limit for JSON values. In VoltDB, VARCHAR columns, in which JSON values are stored, are limited to 1MB, exactly 1024^2 bytes (1048576 bytes). In this way, JSON support lets you augment the existing relational model within VoltDB. 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*)

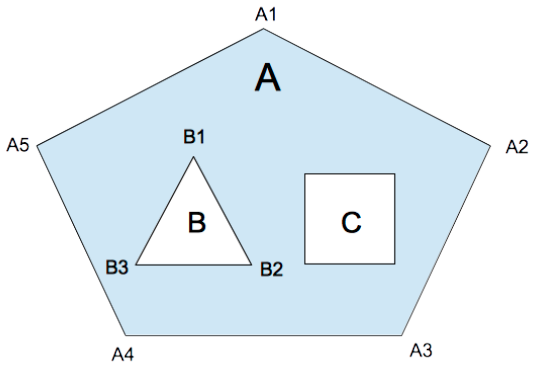
The longitude is a floating point value between 180 and -180 inclusive. The latitude is a floating point value between 90 and -90 inclusive.

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() function 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 COLUMNS (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:

  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,?) 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.

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. 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.

7.1. 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.

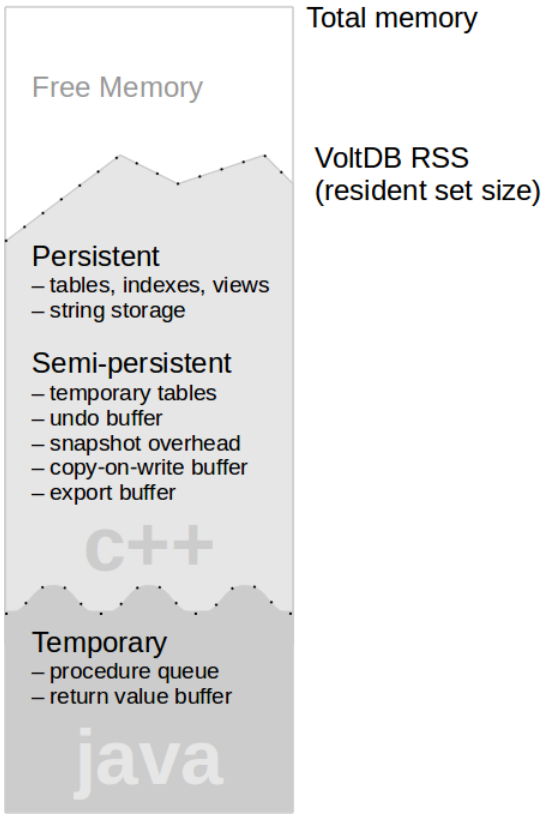
Semi-persistent memory is also used for buffers related to system activities such as snapshots and export. While a snapshot is occurring, a certain amount of memory is required for overhead, as well as copy-on-write buffers. Normally, snapshots are written directly from the tables in memory, thus requiring 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-only tables 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 7.1, “The Three Types of Memory in VoltDB” illustrates how the three types of memory are allocated in VoltDB.

Figure 7.1. 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.

7.2. 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 7.2. Details of Memory Usage During and After an SQL Statement

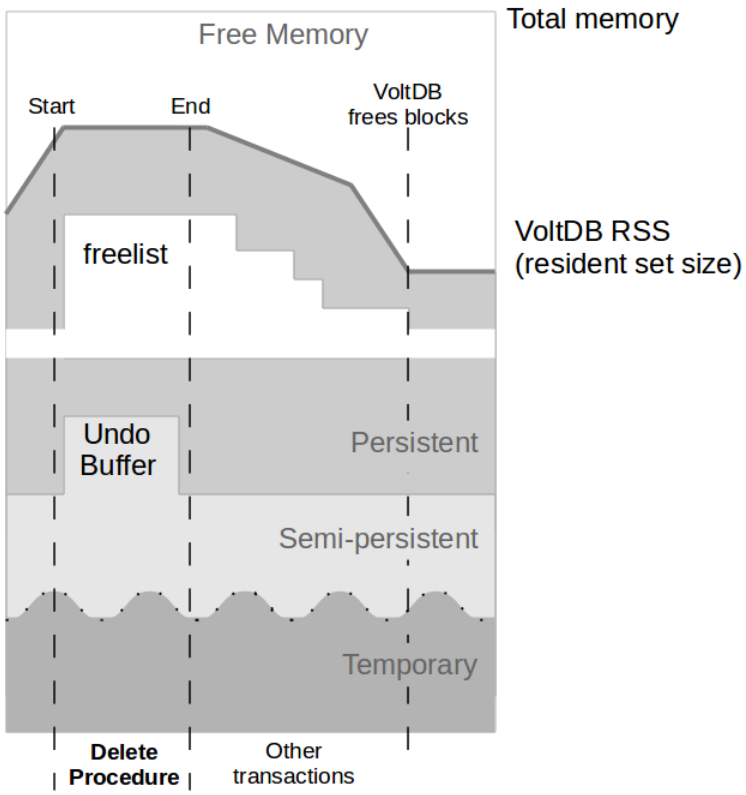


Figure 7.2, “Details of Memory Usage During and After an SQL Statement” illustrates how a delete operation can have a delayed effect on overall memory allocation.

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 7.3, “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.

7.3. 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.

7.4. 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.

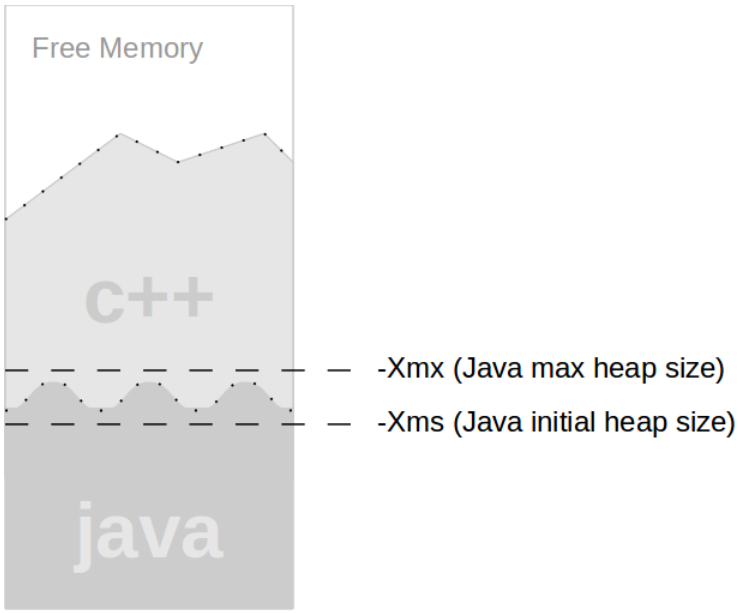
For temporary storage (which is managed in Java), VoltDB cannot explicitly control memory allocation and deallocation and relies on the Java virtual machine (JVM) to manage memory appropriately. The JVM decides when and how to collect free space from unused objects. This means that the VoltDB server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

7.5. 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 7.3, “Controlling the Java Heap Size” illustrates how the `-Xms` and `-Xmx` arguments can be used to control the overall size of temporary storage.

Figure 7.3. 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.

7.6. 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.
TUPLEDATA	Persistent	The amount of memory in use to store inline table data.

Column	Type of Storage	Description
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 8. Managing Time

In previous versions of VoltDB, all transactions were globally coordinated, using system time to order and schedule each transaction. As a result, even small differences in clock time between nodes could impact latency in the system.

Starting with version 3.0, transactions are no longer globally coordinated and differences in system clocks no longer directly impact database latency. However, there are still some 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. 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 8.1, “The Importance of Time” and Section 8.2.2, “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

8.1. 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.

8.2. 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.

8.2.1. Basic Configuration

To manage time effectively 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 most systems, starting the NTP daemon happens automatically on startup. You do not need to perform this action manually. However, 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.

8.2.2. 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

8.2.3. 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.

[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".
[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.

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 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 8.2.4, “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.

8.2.4. 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 8.1, “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 8.1. 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
```

8.3. 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.

8.3.1. 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 8.2.4, “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.

8.3.2. 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 8.2.4, “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.