Performance Guide

Abstract

This book explains how to optimize the performance of applications using VoltDB.

V2.8.1

Performance Guide

V2.8.1

This document was generated on August 06, 2012.

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. Other books — specifically *Getting Started with VoltDB* and *Using VoltDB* — describe the basic features of the 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 six chapters:

- Chapter 1, *Introduction*
- Chapter 2, *Hello, World! Revisited*
- Chapter 3, *Understanding VoltDB Execution Plans*
- Chapter 4, *Understanding VoltDB Memory Usage*
- Chapter 5, *Managing Time*

2. Other Resources

This book provides recommendations for optimizing VoltDB applications. 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:

- *Getting Started with VoltDB* 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 Management Guide* provides information on managing VoltDB clusters, with a focus on the VoltDB Enterprise Manager and REST API.

These books and more resources are available on the web from `http://www.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 *Getting Started with VoltDB* 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 3) explains how to design your database schema, stored procedures, and client applications for maximum performance.
- Chapter 4 explains in detail how memory is used by the VoltDB server process.
- Chapter 5 provides guidelines for configuring hardware and operating systems for running a VoltDB cluster.

Chapter 2. Hello, World! Revisited

The *Getting Started with VoltDB* manual includes a Hello World tutorial that teaches you how to create a VoltDB database, including the stored procedures and 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 have learned 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 once you install the VoltDB software in the `doc/tutorials/helloworldrevisited` subfolder.

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 either the database DDL or in the project definition file. If a table is not listed as being partitioned, it becomes a replicated table by default.

The advantage of specifying the partitioning in the DDL is that it keeps all of the information about the schema in one location. 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 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, to change the procedure's "signature" to multipartitioned rather than single-partitioned. We do that by changing the @ProcInfo annotation at the beginning of the procedure. The new assertion is as follows:

```
@ProcInfo(
    singlePartition = false
)
```

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 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 @ProcInfo annotation must identify the parameter for the EMAIL value. The resulting RegisterUser procedure looks like this:

```
import org.voltdb.*;

@ProcInfo(
    partitionInfo = "USERACCOUNT.EMAIL: 0",
    singlePartition = true
)

public class RegisterUser extends VoltProcedure {

  public final SQLStmt insertuser = new SQLStmt(
      "INSERT INTO USERACCOUNT VALUES (?,?,?,?,?);"
  );

  public VoltTable[] run( String email, String firstname,
                  String lastname, String language)
      throws VoltAbortException {

              // Insert a new record
          voltQueueSQL( insertuser, email, firstname,
                        lastname, null, language);
          return voltExecuteSQL();
      }
}
```

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.

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.*;

@ProcInfo(
    partitionInfo = "USERACCOUNT.EMAIL: 0",
    singlePartition = true
)

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();
      }
}
```

2.4. Using Asynchronous Stored Procedure Calls

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

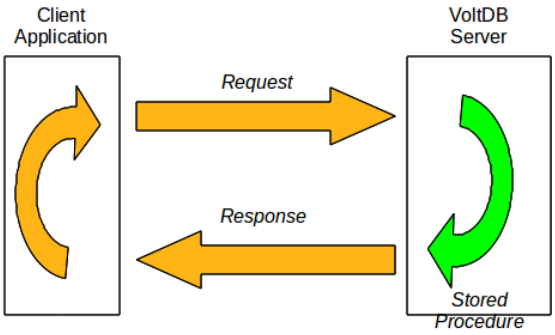
You can call VoltDB stored procedures either synchronously or asynchronously. When you call a stored procedure synchronously, your client application waits for the call to be processed before continuing. If you call a procedure asynchronously, your application continues processing once the call has been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

2.4.1. Understanding Asynchronous Programming

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

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

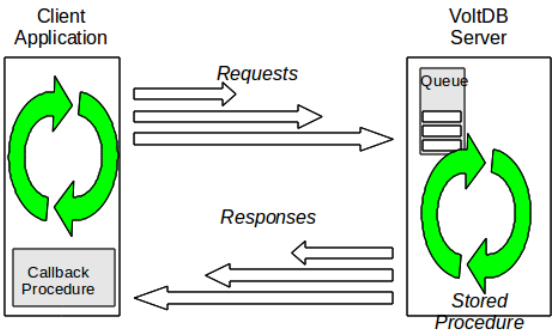
Figure 2.1. Synchronous Procedure Calls



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

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

Figure 2.2. Asynchronous Procedure Calls



2.4.2. The Callback Procedure

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

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

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

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

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

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

       }
    }
```

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

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

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

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

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

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

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

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

         }
    }
```

2.4.3. Making an Asynchronous Procedure Call

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

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

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

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

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

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

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
- 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 perform these tasks. However, for maximum performance it is best if all nodes do their share. If only one node is interacting with the client application, managing the queue could 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 all nodes in the cluster. When there are multiple connections, the client interface uses 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 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", "Hello",   "World",  language[0]);
    myApp.callProcedure("Insert", "Bonjour", "Monde",  language[1]);
    myApp.callProcedure("Insert", "Hola",    "Mundo",  language[2]);
    myApp.callProcedure("Insert", "Hej",     "Verden", language[3]);
    myApp.callProcedure("Insert", "Ciao",    "Mondo",  language[4]);
} 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 compiles a project definition file, it reviews possible execution plans for the SQL statements in the stored procedures. 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 textual 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

When you compile your project definition file, VoltDB creates the folder `./debugoutput` (in the current default path) and writes files summarizing the planning process into subfolders. Many of these files contain more information than necessary to help the end user. But one folder contains a summary in human readable form.

The folder `./debugoutput/statement-winner-plans/` contains the final plans chosen for execution. In other words, a description of the plans that VoltDB chose for executing each and every SQL statement in your project. These execution plans describe the order in which tables are scanned and which indexes, if any, are used to access those tables. The execution plans in `./debugoutput/statement-winner-plans` are also included in the runtime catalog in the folder `/plans`. So the execution plans are available to anyone with a copy of the resulting catalog.

Let's look at the Hello World tutorial as an example. The Hello World example is very simple; it has two stored procedures with one SQL statement each. When VoltDB compiles the project, it creates one file in the `statement-winner-plans` folder for each statement. The files are named according to the stored procedure name and SQL statement name. So, for Hello World, there are two files:

- Insert-sql.txt
- Select-sql.txt

Note that in both stored procedures, the SQLStmt instance is named *sql*. If the SQLStmt uses a different name, the plan file reflects that difference. (For example, in the Voter application, the SQL statement in the Results stored procedure is named *resultStmt*. Therefore, the plan file is created as `Results-resultStmt.txt`.)

3.3. Reading the Execution Plan and Optimizing Your SQL Statements

Within the execution plan is an ordered representation of the winning plan. The plan can be read bottom up to understand the order in which the plan is executed. So, for example, looking at the SELECT statement in the Hello World example, we see that the Select-sql.txt plan file contains the following:

```
❶ SQL: SELECT HELLO, WORLD FROM HELLOWORLD  WHERE DIALECT = ?;
❷ COST: 6.0
❸ PLAN:
❹
❺ RETURN RESULTS TO STORED PROCEDURE
❻  INDEX SCAN of "HELLOWORLD" using "SYS_IDX_SYS_PK_10018_10019" (unique-scan covering)
```

The first two lines list the SQL statement itself and a "cost" for the selected plan. The cost value is an abstract internal calculation used to determine which of the many possible plans the compiler chooses as the winner. In general, the more complex the SQL statement, the higher the cost for any given plan. However, this cost value does not have any bearing on the actual potential performance of the statement itself. It is simply a relative ordering of the possible plans.

The real content of the execution plan appears in the "Plan" section, starting with line 3.

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

- Performing an indexed scan of the HELLOWORLD table (line 6)
- Returning the selected row(s) to the stored procedure (line 5)

You will notice that line 6 is indented to indicate it is a child of the preceding statement. In other words, it must be completed before the results can be returned.

You will also notice that the scan of the HELLOWORLD table specifies the index as SYS_IDX_SYS_PK_10018_10019. The use of system-generated values is not particularly meaningful, but is a consequence of the index not being explicitly named in the database schema. If you want to use the execution plans to evaluate your schema and stored procedures, it is a good idea to explicitly name the indexes in your DDL. For example, if we modify the Hello World example to explicitly name the partitioning column:

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

The execution plan incorporates the stated name, becoming more readable:

```
SQL: SELECT HELLO, WORLD FROM HELLOWORLD  WHERE DIALECT = ?;
COST: 6.0
PLAN:

RETURN RESULTS TO STORED PROCEDURE
```

```
 INDEX SCAN of "HELLOWORLD" using "SYS_IDX_PK_DIALECT_10018" (unique-scan covering)
```

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, if you look at the execution plans for the Voter example that comes with the VoltDB software, you can see a more complex execution plan for the GetStateHeatmap stored procedure:

```
SQL: SELECT contestant_number, state, SUM(num_votes) AS num_votes
     FROM v_votes_by_contestant_number_state GROUP BY contestant_number,
     state ORDER BY 2 ASC, 3 DESC;
COST: 8000000.0
PLAN:

RETURN RESULTS TO STORED PROCEDURE
 ORDER BY (SORT)
  AGGREGATION ops: sum
   RECEIVE FROM ALL PARTITIONS
    SEND PARTITION RESULTS TO COORDINATOR
     AGGREGATION ops: sum
      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.
- Use an aggregate function to sum the votes for each contestant.
- Return the results from each partition to the initiator that is coordinating the multi-partition transaction.
- The results from all partitions are then summed together.
- The combined results are then sorted.
- And finally the results are returned 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:

```
SQL: SELECT HELLO, WORLD FROM HELLOWORLD  WHERE DIALECT = ?;
COST: 2000000.0
PLAN:

RETURN RESULTS TO STORED PROCEDURE
 SEQUENTIAL SCAN of "HELLOWORLD"
```

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 throughout Chapter 3 of 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:

```
SQL: SELECT * FROM FLIGHT WHERE origin=? AND destination=?
     AND departtime>? AND departtime < ?;
COST: 2000000.0
PLAN:

RETURN RESULTS TO STORED PROCEDURE
 SEQUENTIAL SCAN of "FLIGHT"
```

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:

```
SQL: SELECT * FROM FLIGHT WHERE origin=? AND destination=?
     AND DEPARTTIME>? AND DEPARTTIME < ?;
COST: 144.0
PLAN:

RETURN RESULTS TO STORED PROCEDURE
 INDEX SCAN of "FLIGHT" using "FLIGHTTIMETREEIDX" (range-scan covering)
```

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 execution plans to make sure your application is not performing any unexpected sequential (non-indexed) scans. You can use the Linux `grep` command to do this, like so:

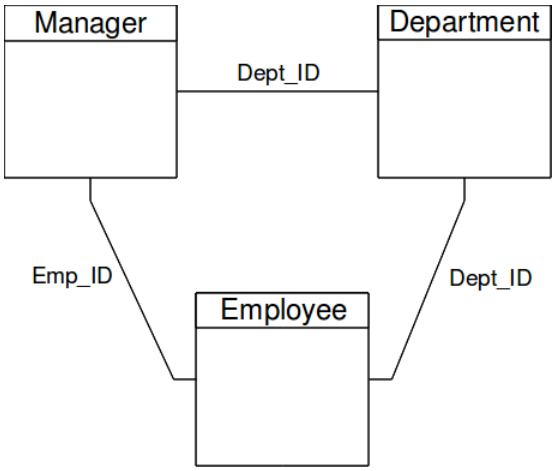
```
$ grep -i "sequential scan" debugoutput/statement-winner-plans/*
```

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:

```
SQL: SELECT d.dept_name, d.dept_id,e.emp_id, e.first_name, e.last_name,
     m.emp_id FROM MANAGER AS M, DEPARTMENT AS D, EMPLOYEE AS E
     WHERE m.emp_id=? AND m.dept_id=d.dept_id AND m.dept_id=e.dept_id
     ORDER BY e.last_name, e.first_name
COST: 8000000.0
PLAN:

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. For example, the declaration of the preceding SQL query, including join order, would look like this:

```
public final SQLStmt FindEmpByMgr = new SQLStmt(
    "SELECT d.dept_name, d.dept_id, e.emp_id, " +
    "e.first_name, e.last_name, m.emp_id " +
    "FROM MANAGER AS M, DEPARTMENT AS D, EMPLOYEE AS E " +
    "WHERE m.emp_id=? AND m.dept_id=d.dept_id AND m.dept_id=e.dept_id " +
    "ORDER BY e.last_name, e.first_name",
    "manager,department,employee");
```

For simple stored procedures where the SQL query is defined in the project definition file, you specify the joinorder as an attribute to the <sql> element. For example, if the preceding query is defined within the project definition file, you specify the join order by enclosing the query in the tags `<sql joinorder="manager,department,employee"> ... </sql>`. If a query joins seven or more tables (either in a stored procedure or in the project definition file), 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:

```
SQL: SELECT d.dept_name, d.dept_id,e.emp_id, e.first_name, e.last_name,
     m.emp_id FROM MANAGER AS M, DEPARTMENT AS D, EMPLOYEE AS E
     WHERE m.emp_id=? AND m.dept_id=d.dept_id AND m.dept_id=e.dept_id
     ORDER BY e.last_name, e.first_name
COST: 8000000.0
PLAN:

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. 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.

4.1. 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.

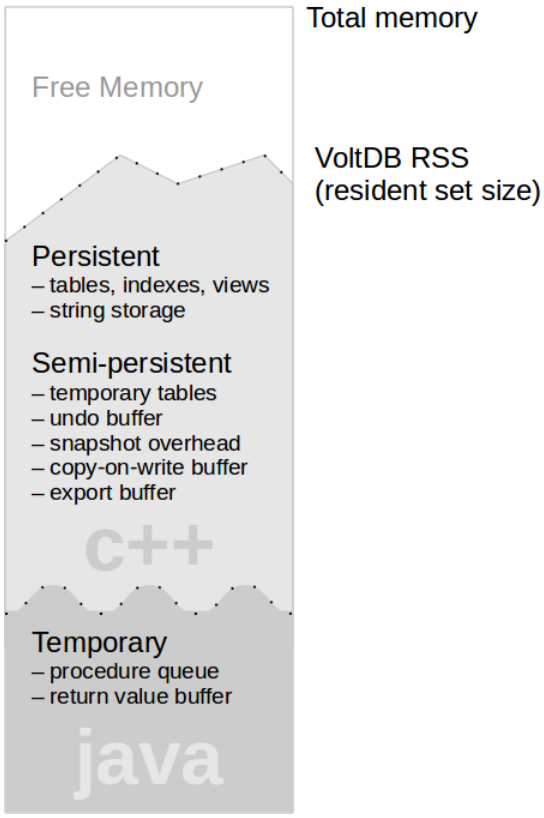
Semi-persistent memory is also used for buffers related to system activities such as snapshots and export. While a snapshot is occurring, a certain amount of memory is required for overhead, as well as copy-on-write buffers. Normally, snapshots are written directly from the tables in memory, thus requiring 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 an export receiver retrieves them.

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

Figure 4.1. 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.

4.2. 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 4.2. Details of Memory Usage During and After an SQL Statement

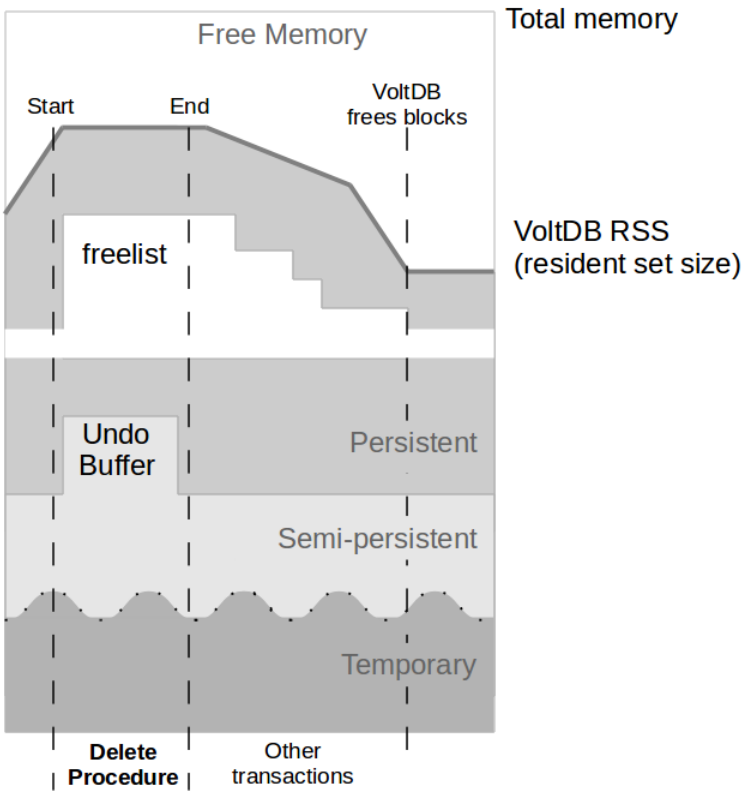


Figure 4.2, “Details of Memory Usage During and After an SQL Statement” illustrates how a delete operation can have a delayed effect on overall memory allocation.

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 4.3, “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 there is no client receiver running 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 a client is written out as export overflow to disk. So memory used for export queues does not grow indefinitely.

4.3. 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.

4.4. 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.

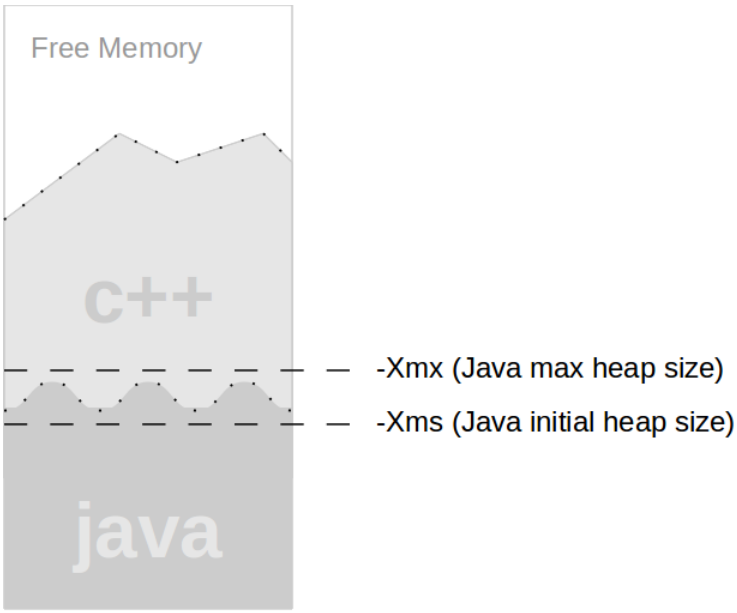
4.5. Controlling How Memory is Allocated

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 the `-Xmx` argument for the maximum heap size, you can control the maximum amount of memory that will be used for temporary storage. In general, VoltDB does not recommend setting the maximum heap size greater than 2 gigabytes for the VoltDB server — any additional space over 2 gigabytes is likely to go unused. If your servers are constrained, you can set a lower maximum heap size to allow more memory for persistent and semi-persistent storage.

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

Figure 4.3. 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.

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.

4.6. 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 5. Managing Time

Because time is used to coordinate transactions, it is important to ensure a stable and consistent view of time within a VoltDB cluster. NTP is the recommended service for managing time for VoltDB. However, there are many different options to the NTP service and countless ways to configure it. This chapter explains the challenges related to time management for a VoltDB cluster and suggests one possible solution.

If you are familiar with NTP or another service and have a preferred method for using it, you may want to read only Section 5.1, “The Importance of Time” and Section 5.2.2, “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

5.1. The Importance of Time

Time is very important to VoltDB. Because transactions are globally ordered based on time, differences in the clocks between individual nodes in the database cluster can have a dramatic effect on performance.

When the cluster starts up, VoltDB determines the maximum amount of skew (that is, the difference in clock time) between the nodes. If the skew is greater than 100 milliseconds (1/10th of a second), the VoltDB cluster won't start. However, even if the skew is less than 100 milliseconds, a large time skew can impact performance.

The problem is that when a VoltDB cluster node looks at its queue to find the lowest ordered transaction, it waits to see if another node is submitting a transaction with a lower ID. Normally, this pause is too small to notice. However, if there is significant clock skew, that waiting period is increased to accommodate the cluster-wide skew between nodes. This extended pause introduces additional latency to the transactions and can impact overall throughput. Which is why keeping time skew to a minimum is important.

5.2. 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.

5.2.1. 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 reference server

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. However, for VoltDB it is best to use a single reference server. 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.

5.2.2. Troubleshooting Issues with Time

In many cases, the preceding basic configuration is sufficient. However, there are a number of time-related issues that can arise within a VoltDB 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)
```

There is no fixed rule as to what is a good or bad value for skew. However, if the skew is greater than 10 milliseconds, it could impact performance and you should consider reconfiguring NTP to reduce the skew.

The next step is to determine what is causing the skew. The most common issues when using NTP to manage time are:

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

5.2.3. 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].

[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.

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

By combining the burst, iburst, minpoll, and maxpoll arguments, you can increase the frequency that the NTP daemon synchronizes and thereby reduce the potential drift of the local server's clock. However, you should not use these arguments with public servers, such as the ones included in the NTP configuration 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 5.2.4, “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.

5.2.4. 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 5.1, “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 5.1. 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
```

5.3. 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.

5.3.1. 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 5.2.4, “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.

5.3.2. 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 is not suitable for VoltDB 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 5.2.4, “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.