Welcome to VoltDB

A Tutorial

Welcome to VoltDB: A Tutorial

Table of Contents

Preface

VoltDB is designed to help you achieve world-class performance in terms of throughput and scalability. At the same time, at its core, VoltDB is a relational database and can do all the things a traditional relational database can do. So before we go into the fine points of tuning a database application for maximum performance, let's just review the basics.

The following tutorial familiarizes you with the features and capabilities of VoltDB step by step, starting with its roots in SQL and walking through the individual features that help VoltDB excel in both flexibility and performance, including:

- Schema Definition
- SQL Queries
- Partitioning
- Schema Updates
- Stored Procedures

For the purposes of this tutorial, let's assume we want to learn more about the places where we live. How many cities and towns are there? Where are they? How many people live there? What other interesting facts can we find out?

Our study will take advantage of data that is freely available from government web sites. But for now, let's start with the basic structure.

How to Use This Tutorial

Of course, you can just read the tutorial to get a feel for how VoltDB works. But we encourage you to follow along using your own copy of VoltDB if you wish.

The data files used for the tutorial are freely available from public web sites; links are provided in the text. However, the initial data is quite large. So we have created a subset of source files and pre-processed data files that is available from the VoltDB web site at the following address for those who wish to try it themselves:

http://downloads.voltactivedata.com/technologies/other/tutorial_files_51.zip

For each section of the tutorial, there is a subfolder containing the necessary source files, plus one subfolder, data, containing the data files. To follow along with the tutorial, do the following:

1. Create a folder to work in.
2. Unpack the tutorial files into the folder.
3. At the beginning of each section of the tutorial:
   a. Set default to your tutorial folder.
   b. Copy the sources for that current tutorial into the your working directory, like so:

```
$ cp -r tutorial1/* ./
```

The tutorial also uses the VoltDB command line commands. Be sure to set up your environment so the commands are available to you, as described in the installation chapter of *Using VoltDB*.

Part 1: Creating the Database

In VoltDB you define your database schema using SQL data definition language (DDL) statements just like other SQL databases. So, if we want to create a database table for the places where we live, the DDL schema might look like the following:

```
CREATE TABLE towns (
   town VARCHAR(64),
   county VARCHAR(64),
   state VARCHAR(2)
);
```

The preceding schema defines a single table with three columns: town, county, and state. We could also set options, such as default values and primary keys. But for now we will keep it as simple as possible.

Starting the Database and Loading the Schema

Once you have the schema defined, you can initialize and start the database, then load your schema. There are several options available when initializing a VoltDB database, which we will discuss later. But for now, we can use the simplest **init** and **start** commands to initialize and start the database using the default options on the current machine:

```
$ voltdb init
$ voltdb start
```

The **voltdb init** command initializes a root directory that VoltDB uses to store its configuration, logs, and other disk-based information. You only need to initialize the root directory once for a production database. When doing development, where you often want to start over with new settings or a completely different schema, you can reuse the same root by using the **voltdb init --force** command between runs. We will use both methods — starting a fresh database and restarting an existing database — in this tutorial. To start with, we will re-initialize the database root each time.

The **voltdb start** command tells VoltDB to start the database process. Once startup completes, the server reports the following message:

```
Server completed initialization.
```

Now you are ready to load your schema. To do that you use the VoltDB interactive command line utility, sqlcmd. Create a new terminal window and issue the sqlcmd command from the shell prompt:

```
$ sqlcmd
SQL Command :: localhost:21212
1>
```

The VoltDB interactive SQL command line first reports what database it has connected to and then puts up a numbered prompt. At the prompt, you can enter DDL statements and SQL queries, execute stored procedures, or type "exit" to end the program and return to the shell prompt.

To load the schema, you can either type the DDL schema statements by hand or, if you have them in a file, you can use the FILE directive to process all of the DDL statements with a single command. Since we only have one table definition, we can type or cut & paste the DDL directly into sqlcmd:

```
1> CREATE TABLE towns (
2>   town VARCHAR(64),
3>   county VARCHAR(64),
4>   state VARCHAR(2)
5> );
```

Using SQL Queries

Congratulations! You have created your first VoltDB database. Of course, an empty database is not terribly useful. So the first thing you will want to do is create and retrieve a few records to prove to yourself that the database is running as you expect.

VoltDB supports all of the standard SQL query statements, such as INSERT, UPDATE, DELETE, and SELECT. You can invoke queries programmatically, through standard interfaces such as JDBC and JSON, or you can include them in stored procedures that are compiled and loaded into the database.

But for now, we'll just try some ad hoc queries using sqlcmd. Let's start by creating records using the INSERT statement. The following example creates three records, for the towns of Billerica, Buffalo, and Bay View. Be sure to include the semi-colon after each statement.

```
1> insert into towns values ('Billerica','Middlesex','MA');
2> insert into towns values ('Buffalo','Erie','NY');
3> insert into towns values ('Bay View','Erie','OH');
```

We can also use ad hoc queries to verify that our inserts worked as expected. The following queries use the SELECT statement to retrieve information about the database records.

```
4> select count(*) as total from towns;
TOTAL
------
     3

(1 row(s) affected)
5> select town, state from towns ORDER BY town;
TOWN         STATE
------------ ------
Bay View     OH
Billerica    MA
Buffalo      NY

(3 row(s) affected)
```

When you are done working with the database, you can type "exit" to end the sqlcmd session and return to the shell command prompt. Then switch back to the terminal session where you started the database and press CTRL-C to end the database process.

This ends Part One of the tutorial.

Part 2: Loading and Managing Data

As useful as ad hoc queries are, typing in data by hand is not very efficient. Fortunately, VoltDB provides several features to help automate this process.

When you define tables using the CREATE TABLE statement, VoltDB automatically creates stored procedures to insert records for each table. There is also a command line tool that uses these default stored procedures so you can load data files into your database with a single command. The **csvloader** command reads a data file, such as a comma-separated value (CSV) file, and writes each entry as a record in the specified database table using the default insert procedure.

It just so happens that there is data readily available for towns and other landmarks in the United States. The Geographic Names Information Service (GNIS), part of the U.S. Geological Survey, provides data files of officially named locations throughout the United States. In particular, we are interested in the data file for populated places. This data is available as a text file from their web site, http://geonames.usgs.gov/domestic/download_data.htm

The information provided by GNIS not only includes the name, county and state, it includes each location's position (latitude and longitude) and elevation. Since we don't need all of the information provided, we can reduce the number of columns to only the information we want.

For our purposes, let's use information about the name, county, state, and elevation of any populated places. This means we can go back and edit our schema file, `towns.sql`, to add the new columns:

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

Note that the GNIS data includes both names and numbers for identifying counties and states. We rearranged the columns to match the order in which they appear in the data file. This makes loading the data easier since, by default, the **csvloader** command assumes the data columns are in the same order as specified in the schema.

Finally, we can use some shell magic to trim out the unwanted columns and rows from the data file. The following script selects the desired columns and removes any records with empty fields:

```
$ cut --delimiter="|" --fields=2,4-7,16 POP_PLACES_20120801.txt \
  | grep -v "|$" \
  | grep -v "||" > data/towns.txt
```

To save time and space, the resulting file containing only the data we need is included with the tutorial files in a subfolder as `data/towns.txt`.

Restarting the Database

Because we changed the schema and reordered the columns of the Towns table, we want to start over with an empty database and reload the schema. Alternately, if the database is still running you could do a DROP TABLE and CREATE TABLE to delete any existing data and replace the table definition.

Later we'll learn how to restart an existing database and recover its schema and content. But for now, we'll use the same commands you learned earlier to re-initialize the root directory and start an empty database in one terminal session and load the schema using sqlcmd in another. You'll want to add the **--force** argument to the **voltdb init** command to indicate you don't need any of the old command logs or snapshots from your previous session. And this time we will load the schema from our DDL file using the FILE directive:

```
  [terminal 1]
$ voltdb init --force
$ voltdb start

  [terminal 2]
$ sqlcmd
1> FILE towns.sql;
Command succeeded.
2> exit
```

Loading the Data

Once the database is running and the schema loaded, we are ready to insert our new data. To do this, set default to the `/data` subfolder in your tutorial directory, and use the **csvloader** command to load the data file:

```
$ cd data
$ csvloader --separator "|"❶  --skip 1❷   \
           --file towns.txt❸  towns❹
```

In the preceding commands:

❶ The `--separator` flag lets you specify the character separating the individual data entries. Since the GNIS data is not a standard CSV, we use `--separator` to identify the correct delimiter.
❷ The data file includes a line with column headings. The `--skip 1` flag tells csvloader to skip the first line.
❸ The `--file` flag tells csvloader what file to use as input. If you do not specify a file, csvloader uses standard input as the source for the data.
❹ The argument, `towns`, tells csvloader which database table to load the data into.

The csvloader loads all of the records into the database and it generates three log files: one listing any errors that occurred, one listing any records it could not load from the data file, and a summary report including statistics on how long the loading process took and how many records were loaded.

Querying the Database

Now that we have real data, we can perform more interesting queries. For example, which towns are at the highest elevation, or how many locations in the United States have the same name?

```
$ sqlcmd
1> SELECT town,state,elevation from towns order by elevation desc limit 5;
TOWN                       STATE  ELEVATION
-------------------------- ------ ----------
Corona (historical)        CO           3573
Quartzville (historical)   CO           3529
Logtown (historical)       CO           3524
Tomboy (historical)        CO           3508
Rexford (historical)       CO           3484


(5 row(s) affected)
2> select town, count(town) as duplicates from towns
3> group by town order by duplicates desc limit 5;
TOWN            DUPLICATES
--------------- -----------
Midway                  215
Fairview                213
Oak Grove               167
Five Points             150
Riverside               130


(5 row(s) affected)
```

As we can see, the five highest towns are all what appear to be abandoned mining towns in the Rocky Mountains. And Springfield, as common as it is, doesn't make it into the top five named places in the United States.

We can make even more interesting discoveries when we combine data. We already have information about locations and elevation. The US Census Bureau can also provide us with information about population density. Population data for individual towns and counties in the United States can be downloaded from their web site, http://www.census.gov/popest/data/index.html.

To add the new data, we must add a new table to the database. So let's edit our schema to add a table for population that we will call *people*. While we are at it, we can create indexes for both tables, using the columns that will be used most frequently for searching and sorting, state_num and county_num.

```
CREATE TABLE towns (
   town VARCHAR(64),
   state VARCHAR(2),
   state_num TINYINT NOT NULL,
   county VARCHAR(64),
   county_num SMALLINT NOT NULL,
   elevation INTEGER
);
CREATE TABLE people (
  state_num TINYINT NOT NULL,
  county_num SMALLINT NOT NULL,
  state VARCHAR(20),
  county VARCHAR(64),
  population INTEGER
);
CREATE INDEX town_idx ON towns (state_num, county_num);
CREATE INDEX people_idx ON people (state_num, county_num);
```

Once again, we put the columns in the same order as they appear in the data file. We also need to trim the data file to remove extraneous columns. The census bureau data includes both measured and estimated values. For the tutorial, we will focus on one population number.

The shell command to trim the data file is the following. (Again, the resulting data file is available as part of the downloadable tutorial package.)

```
$ grep -v "^040," CO-EST2011-Alldata.csv \
     | cut --delimiter="," --fields=4-8  > people.txt
```

Once we have the data and the new DDL statements, we can update the database schema. We could stop and restart the database, load the new schema from our text file and reload the data. But we don't have to. Since we are not changing the Towns table or adding a unique index, we can make our changes to the running database by simply cutting and pasting the new DDL statements into sqlcmd:

```
$ sqlcmd
1> CREATE TABLE people (
2>  state_num TINYINT NOT NULL,
3>  county_num SMALLINT NOT NULL,
4>  state VARCHAR(20),
5>  county VARCHAR(64),
6>  population INTEGER
7> );
8> CREATE INDEX town_idx ON towns (state_num, county_num);
9> CREATE INDEX people_idx ON people (state_num, county_num);
```

Once we create the new table and indexes, we can load the accompanying data file:

```
$ cd data
$ csvloader --file people.txt --skip 1 people
```

At this point we now have two tables loaded with data. Now we can join the tables to look for correlations between elevation and population, like so:

```
$ sqlcmd
1> select top 5 min(t.elevation) as height,
2> t.state,t.county, max(p.population)
3> from towns as t, people as p
4> where t.state_num=p.state_num and t.county_num=p.county_num
5> group by t.state, t.county order by height desc;
HEIGHT  STATE  COUNTY    C4
------- ------ --------- ------
   2754 CO     Lake        7310
   2640 CO     Hinsdale     843
   2609 CO     Mineral      712
   2523 CO     San Juan     699
   2454 CO     Summit     27994

(5 row(s) affected)
```

It turns out that, even discounting ghost towns that have no population, the five inhabited counties with highest elevation are all in Colorado. In fact, if we reverse the select expression to find the lowest inhabited counties (by changing the sort order from descending to ascending), the lowest is Inyo county in California — the home of Death Valley!

This ends Part Two of the tutorial.

Part 3: Partitioning

Now you have the hang of the basic features of VoltDB as a relational database, it's time to start looking at what makes VoltDB unique. One of the most important features of VoltDB is partitioning.

Partitioning organizes the contents of a database table into separate autonomous units. Similar to sharding, VoltDB partitioning is unique because:

- VoltDB partitions the database tables automatically, based on a partitioning column you specify. You do not have to manually manage the partitions.
- You can have multiple partitions, or sites, on a single server. In other words, partitioning is not just for scaling the data volume, it helps performance as well.
- VoltDB partitions both the data *and* the processing that accesses that data, which is how VoltDB leverages the throughput improvements parallelism provides.

Partitioned Tables

You partition a table by specifying the partitioning column as part of your schema. If a table is partitioned, each time you insert a row into that table, VoltDB decides which partition the row goes into based on the value of the partitioning column. So, for example, if you partition the Towns table on the column Name, the records for all towns with the same name end up in the same partition.

However, although partitioning by name may be reasonable in terms of evenly distributing the records, the goal of partitioning is to distribute both the data *and* the processing. We don't often compare information about towns with the same name. Whereas, comparing towns within a given geographic region is very common. So let's partition the records by state so we can quickly do things like finding the largest or highest town within a given state.

Both the Towns and the People tables have columns for the state name. However, they are slightly different; one uses the state abbreviation and one uses the full name. To be consistent, we can use the State_num column instead, which is common to both tables.

To partition the tables, we simply add a PARTITION TABLE statement to the database schema. Here are the statements we can add to our schema to partition both tables by the State_num column:

```
PARTITION TABLE towns ON COLUMN state_num;
PARTITION TABLE people ON COLUMN state_num;
```

Having added partitioning information, we can stop the database, re-initialize, restart and reload the schema and data. This time, rather than using CTRL-C to kill the database process, we can use the **voltadmin shutdown** command. The **voltadmin** commands perform administrative functions for a database cluster and **shutdown** performs an orderly shutdown of the database whether a single node or a 15 node cluster. So go to the second terminal session and use **voltadmin shutdown** to stop the database:

```
$ voltadmin shutdown
```

Then you can re-initialize and start the database and load the new schema and data files:

```
  [terminal 1]
$ voltdb init --force
$ voltdb start

  [terminal 2]
$ sqlcmd
1> FILE towns.sql;
Command succeeded.
2> exit
$ cd data
$ csvloader --separator "|"  --skip 1   \
            --file towns.txt  towns
$ csvloader --file people.txt --skip 1 people
```

The first thing you might notice, without doing any other queries, is that loading the data files is faster. In fact, when csvloader runs, it creates three log files summarizing the results of the loading process. One of these files, `csvloader_TABLE-NAME_insert_report.log`, describes how long the process took and the average transactions per second (TPS). Comparing the load times before and after adding partitioning shows that adding partitioning increases the ingestion rate for the Towns table from approximately 5,000 to 16,000 TPS — more than three times as fast! This performance improvement is a result of parallelizing the stored procedure calls across eight sites per host. Increasing the number of sites per host can provide additional improvements, assuming the server has the core processors necessary to manage the additional threads.

Replicated Tables

As mentioned earlier, the two tables Towns and People both have a VARCHAR column for the state name, but its use is not consistent. Instead we use the State_num column to do partitioning and joining of the two tables.

The State_num column contains the FIPS number. That is, a federal standardized identifier assigned to each state. The FIPS number ensures unique and consistent identification of the state. However, as useful as the FIPS number is for computation, most people think of their location by name, not number. So it would be useful to have a consistent name to go along with the number.

Instead of attempting to modify the fields in the individual tables, we can normalize our schema and create a separate table that provides an authoritative state name for each state number. Again, the federal government makes this information freely available from the U.S. Environmental Protection Agency web site, http://www.epa.gov/enviro/html/codes/state.html. Although it is not directly downloadable as a data file, a copy of the FIPS numbers and names for all of the states is included in the tutorial files in the data subfolder as `data/state.txt`.

So let's go and add a table definition for this data to our schema:

```
CREATE TABLE states (
   abbreviation VARCHAR(20),
   state_num TINYINT,
   name VARCHAR(20),
   PRIMARY KEY (state_num)
);
```

This sort of lookup table is very common in relational databases. They reduce redundancy and ensure data consistency. Two of the most common attributes of lookup tables are that they are relatively small in size and they are static. That is, they are primarily read-only.

It would be possible to partition the States table on the State_num column, like we do the Towns and People tables. However, when a table is relatively small and not updated frequently, it is better to replicate it to all partitions. This way, even if another table is partitioned (such as a customer table partitioned on last name), stored procedures can join the two tables, no matter what partition the procedure executes in.

Tables where all the records are available to all the partitions are called *replicated tables*. Note that tables are replicated by default. So to make the States table a replicated table, we simply include the CREATE TABLE statement without an accompanying PARTITION TABLE statement.

One last caveat concerning replicated tables: the benefits of having the data replicated is that it can be read from any individual partition. However, the deficit is that any updates or inserts to a replicated table must be executed for all partitions at once. This sort of multi-partition procedure reduces the benefits of parallel processing and impacts throughput. Which is why you should not replicate tables that are frequently updated.

This ends Part Three of the tutorial.

Part 4: Schema Updates and Durability

Thus far in the tutorial we have re-initialized and restarted the database from scratch and reloaded the schema and data manually each time we changed the schema. This is sometimes the easiest way to make changes when you are first developing your application and making frequent changes. However, as your application — and the data it uses — becomes more complex, you want to maintain your database state across sessions.

You may have noticed that in the previous section of the tutorial we defined the States table but did not add it to the running database yet. That is because we want to demonstrate ways of modifying the database without having to start from scratch each time.

Preserving the Database

First let's talk about durability. VoltDB is an in-memory database. Each time you re-initialize and start the database with the **init** and **start** commands, it starts a new, empty database. Obviously, in real business situations you want the data to persist. VoltDB has several features that preserve the database contents across sessions.

The easiest way to preserve the database is to use *command logging*, which is enabled by default for the VoltDB Enterprise Edition. Command logging logs all of the database activity, including schema and data changes, to disk. If the database ever stops, you can recover the command log simply by restarting the database with the **voltdb start** command.

If you are using the Enterprise Edition, try it now. Stop the database process with **voltadmin shutdown**, then use **voltdb start** (without **voltdb init**) to restore the database to its previous state:

```
$ voltadmin shutdown
$ voltdb start
```

Command logging makes saving and restoring your database easy and automatic. Alternately, you can save and restore your database using *snapshots*. Snapshots are a complete disk-based representation of a VoltDB database, including everything needed to reproduce the database after a shutdown. You can create a snapshot of a running VoltDB database at anytime using the **voltadmin save** command. For example:

```
 voltadmin save
```

By default, the snapshot is saved to a subfolder of the database root directory. Alternately, you can specify the location and name of the snapshot files as arguments to the **voltadmin save** command. But there is an advantage to saving the snapshot to the default location. Because if there are any snapshots in the root directory, the **voltdb start** command automatically restores the most recent snapshot when the database restarts.

To make it even easier, VoltDB let's you create a final snapshot when you shutdown the database simply by adding the **--save** argument to the **shutdown** command. This is the recommended way to shutdown the database when not using command logging. Let's try it:

```
$ voltadmin shutdown --save
$ voltdb start
```

We can verify that the database was restored by doing some simple SQL queries in our other terminal session:

```
$ sqlcmd
SQL Command :: localhost:21212
1> select count(*) from towns;
C1
-------
 193297


(1 row(s) affected)
2> select count(*) from people;
C1
------
 81691


(1 row(s) affected)
```

Adding and Removing Tables

Now that we know how to save and restore the database, we can add the States table we defined in Part Three. Adding and dropping tables can be done "on the fly", while the database is running, using the sqlcmd utility. To add tables, you simply use the CREATE TABLE statement, like we did before. When modifying existing tables you can use the ALTER TABLE statement. Alternately, if you are not concerned with preserving existing data in the table, you can do a DROP TABLE followed by CREATE TABLE to replace the table definition.

In the case of the States table we are adding a new table so we can simply type (or copy & paste) the CREATE TABLE statement into the sqlcmd prompt. We can also use the **show tables** directive to verify that our new table has been added.

```
$ sqlcmd
SQL Command :: localhost:21212
1> CREATE TABLE states (
2>    abbreviation VARCHAR(20),
3>    state_num TINYINT,
4>    name VARCHAR(20),
5>    PRIMARY KEY (state_num)
6> );
Command successful
7> show tables;

--- User Tables ------------------------------------------
PEOPLE
STATES
TOWNS

--- User Views -------------------------------------------

--- User Export Streams ----------------------------------------

8> exit
```

Next we can load the state information from the data file. Finally, we can use the **voltadmin save** command to save a complete copy of the database.

```
$ csvloader --skip 1 -f data/states.csv states
$ voltadmin save
```

Updating Existing Tables

Now that we have a definitive lookup table for information about the states, we no longer need the redundant columns in the Towns and People tables. We want to keep the FIPS column, State_num, but can remove the State column from each table. Our updated schema for the two tables looks like this:

```
CREATE TABLE towns (
   town VARCHAR(64),
--   state VARCHAR(2),
   state_num TINYINT NOT NULL,
   county VARCHAR(64),
   county_num SMALLINT NOT NULL,
   elevation INTEGER
);
CREATE TABLE people (
  state_num TINYINT NOT NULL,
  county_num SMALLINT NOT NULL,
--  state VARCHAR(20),
  town VARCHAR(64),
  population INTEGER
);
```

It is good to have the complete schema on file in case we want to restart from scratch. (Or if we want to recreate the database on another server.) However, to modify an existing schema under development it is often easier just to use ALTER TABLE statements. So to modify the running database to match our new schema, we can use ALTER TABLE with the DROP COLUMN clause from the sqlcmd prompt:

```
$ sqlcmd
SQL Command :: localhost:21212
1> ALTER TABLE towns DROP COLUMN state;
Command successful
2> ALTER TABLE people DROP COLUMN state;
Command successful
```

Many schema changes, including adding, removing, and modifying tables, columns, and indexes can be done on the fly. However, there are a few limitations. For example, you cannot add new unique constraints to a table that already has data in it. In this case, you can DROP and CREATE the table with the new constraints if you do not need to save the data. Or, if you need to preserve the data and you know that it will not violate the new constraint, you can save the data to a snapshot using an explicit directory, re-initialize the database root directory, restart, and reload the new schema, then restore the data from the snapshot into the updated schema using the **voltadmin restore** command.

This ends Part Four of the tutorial.

Part 5: Stored Procedures

We now have a complete database that we can interact with using SQL queries. For example, we can find the least populous county for any given state (California, for example) with the following SQL query:

```
$ sqlcmd
1> SELECT TOP 1 county, abbreviation, population
2>     FROM people, states WHERE people.state_num=6
3>     AND people.state_num=states.state_num
4>     ORDER BY population ASC;
```

However, typing in the same query with a different state number over and over again gets tiring very quickly. The situation gets worse as the queries get more complex.

Simple Stored Procedures

For queries you run frequently, only changing the input, you can create a simple stored procedure. *Stored procedures* let you define the query once and change the input values when you execute the procedure. Stored procedures have an additional benefit; because they are pre-compiled, the queries do not need to be planned at runtime, reducing the time it takes for each query to execute.

To create simple stored procedures — that is, procedures consisting of a single SQL query — you can define the entire procedure in your database schema using the CREATE PROCEDURE AS statement. So, for example to turn our previous query into a stored procedure, we might add the following statement to our schema:

```
CREATE PROCEDURE leastpopulated❶ AS
  SELECT TOP 1 county, abbreviation, population
    FROM people, states WHERE people.state_num=?❷
    AND people.state_num=states.state_num
    ORDER BY population ASC;
```

In the CREATE PROCEDURE AS statement:

❶ The label, in this case *leastpopulated*, is the name given to the stored procedure.
❷ Question marks are used as placeholders for values that will be input at runtime.

In addition to creating the stored procedure, we can also specify if it is single-partitioned or not. When you partition a stored procedure, you associate it with a specific partition based on the table that it accesses. For example, the preceding query accesses the People table and, more importantly, narrows the focus to a specific value of the partitioning column, State_num.

Note that you can access more than one table in a single-partition procedure, as we do in the preceding example. However, all of the data you access must be in that partition. In other words, all the tables you access must be partitioned on the same key value or, for read-only SELECT statements, you can also include replicated tables.

So we can partition our new procedure on the People table by adding a PARTITION ON clause and specifying the table and partitioning column. The complete statement is as follows:

```
CREATE PROCEDURE leastpopulated
   PARTITION ON TABLE people COLUMN state_num
AS
   SELECT TOP 1 county, abbreviation, population
     FROM people, states WHERE people.state_num=?
     AND people.state_num=states.state_num
     ORDER BY population ASC;
```

Now when we invoke the stored procedure, it is executed only in the partition where the State_num column matches the first argument to the procedure, leaving the other partitions free to process other requests.

Of course, before we can use the procedure we need to add it to the database. Modifying stored procedures can be done on the fly, like adding and removing tables. So we do not need to restart the database, just type the CREATE PROCEDURE statement at the sqlcmd prompt:

```
sqlcmd>
1> CREATE PROCEDURE leastpopulated
2>   PARTITION ON TABLE people COLUMN state_num
3> AS
4>    SELECT TOP 1 county, abbreviation, population
5>      FROM people, states WHERE people.state_num=?
6>      AND people.state_num=states.state_num
7>      ORDER BY population ASC;
```

Once we update the schema, the new procedure becomes available. So we can now execute the query multiple times for different states simply by changing the argument to the procedure:

```
1> exec leastpopulated 6;
COUNTY         ABBREVIATION  POPULATION
-------------- ------------- -----------
Alpine County  CA                   1175


(1 row(s) affected)
2> exec leastpopulated 48;
COUNTY         ABBREVIATION  POPULATION
-------------- ------------- -----------
Loving County  TX                     82


(1 row(s) affected)
```

Writing More Powerful Stored Procedures

Simple stored procedures written purely in SQL are very handy as short cuts. However, some procedures are more complex, requiring multiple queries and additional computation based on query results. For more involved procedures, VoltDB supports writing stored procedures in Java.

It isn't necessary to be a Java programming wizard to write VoltDB stored procedures. All VoltDB stored procedures have the same basic structure. For example, the following code reproduces the simple stored procedure *leastpopulated* we wrote in the previous section using Java:

```
import org.voltdb.*;❶
```

```
public class LeastPopulated extends VoltProcedure {❷

  public final SQLStmt getLeast = new SQLStmt(
      " SELECT TOP 1 county, abbreviation, population "
    + " FROM people, states WHERE people.state_num=?"
    + " AND people.state_num=states.state_num"
    + " ORDER BY population ASC;" );❸

  public VoltTable[] run(int state_num)❹
      throws VoltAbortException {

          voltQueueSQL( getLeast, state_num );❺
          return voltExecuteSQL();❻

      }
}
```

In this example:

❶ We start by importing the necessary VoltDB classes and methods.
❷ The procedure itself is defined as a Java class. The Java class name is the name we use at runtime to invoke the procedure. In this case, the procedure name is *LeastPopulated*.
❸ At the beginning of the class, you declare the SQL queries that the stored procedure will use. Here we use the same SQL query from the simple stored procedure, including the use of a question mark as a placeholder.
❹ The body of the procedure is a single *run* method. The arguments to the run method are the arguments that must be provided when invoking the procedure at runtime.
❺ Within the run method, the procedure queues one or more queries, specifying the SQL query name, declared in step 3, and the arguments to be used for the placeholders. (Here we only have the one query with one argument, the state number.)
❻ Finally, a call executes all of the queued queries and the results of those queries are returned to the calling application.

Now, writing a Java stored procedure to execute a single SQL query is overkill. But it does illustrate the basic structure of the procedure.

Java stored procedures become important when designing more complex interactions with the database. One of the most important aspects of VoltDB stored procedures is that each stored procedure is executed as a complete unit, a transaction, that either succeeds or fails as a whole. If any errors occur during the transaction, earlier queries in the transaction are rolled back before a response is returned to the calling application, or any further work is done by the partition.

One such transaction might be updating the database. It just so happens that the population data from the U.S. Census Bureau contains both actual census results and estimated population numbers for following years. If we want to update the database to replace the 2010 results with the 2011 estimated statistics (or some future estimates), we would need a procedure to:

1. Check to see if a record already exists for the specified state and county.

2. If so, use the SQL UPDATE statement to update the record.

3. If not, use an INSERT statement to create a new record.

We can do that by extending our original sample Java stored procedure. We can start be giving the Java class a descriptive name, *UpdatePeople*. Next we include the three SQL statements we will use (SELECT,

UPDATE, and INSERT). We also need to add more arguments to the procedure to provide data for all of the columns in the People table. Finally, we add the query invocations and conditional logic needed. Note that we queue and execute the SELECT statement first, then evaluate its results (that is, whether there is at least one record or not) before queuing either the UPDATE or INSERT statement.

The following is the completed stored procedure source code.

```
import org.voltdb.*;

public class UpdatePeople extends VoltProcedure {

  public final SQLStmt findCurrent = new SQLStmt(
      " SELECT population FROM people WHERE state_num=? AND county_num=?"
    + " ORDER BY population;");
  public final SQLStmt updateExisting = new SQLStmt(
      " UPDATE people SET population=?"
    + " WHERE state_num=? AND county_num=?;");
  public final SQLStmt addNew = new SQLStmt(
      " INSERT INTO people VALUES (?,?,?,?);");

  public VoltTable[] run(byte state_num,
                         short county_num,
                         String county,
                         long population)
      throws VoltAbortException {

          voltQueueSQL( findCurrent, state_num, county_num );
          VoltTable[] results = voltExecuteSQL();

          if (results[0].getRowCount() > 0) { // found a record
             voltQueueSQL( updateExisting, population,
                                           state_num,
                                           county_num );

          } else {  // no existing record
             voltQueueSQL( addNew, state_num,
                                   county_num,
                                   county,
                                   population);

          }
          return voltExecuteSQL();
      }
}
```

Compiling Java Stored Procedures

Once we write the Java stored procedure, we need to load it into the database and then declare it in DDL the same way we do with simple stored procedures. But first, the Java class itself needs compiling. We use the Java compiler, **javac**, to compile the procedure the same way we would any other Java program.

When compiling stored procedures, the Java compiler must be able to find the VoltDB classes and methods imported at the beginning of the procedure. To do that, we must include the VoltDB libraries in the Java *classpath*. The libraries are in the subfolder `/voltdb` where you installed VoltDB. For example, if you

installed VoltDB in the directory `/opt/voltdb`, the command to compile the UpdatePeople procedure is the following:

```
$ javac -cp "$CLASSPATH:/opt/voltdb/voltdb/*"  UpdatePeople.java
```

Once we compile the source code into a Java class, we need to package it (and any other Java stored procedures and classes the database uses) into a Jar file and load it into the database. Jar files are a standard format for compressing and packaging Java files. You use the **jar** command to create a Jar file, specifying the name of the Jar file to create and the files to include. For example, you can package the `UpdatePeople.class` file you created in the previous step into a Jar file named `storedprocs.jar` with the following command:

```
$ jar cvf storedprocs.jar *.class
```

Once you package the stored procedures into a Jar file, you can then load them into the database using the sqlcmd **load classes** directive. For example:

```
$ sqlcmd
1> load classes storedprocs.jar;
```

Finally, we can declare the stored procedure in our schema, in much the same way simple stored procedures are declared. But this time we use the CREATE PROCEDURE FROM CLASS statement, specifying the class name rather than the SQL query. We can also partition the procedure on the People table, since all of the queries are constrained to a specific value of State_num, the partitioning column. Here is the statement we add to the schema.

```
CREATE PROCEDURE
   PARTITION ON TABLE people COLUMN state_num
   FROM CLASS UpdatePeople;
```

Notice that you do not need to specify the name of the procedure after "CREATE PROCEDURE" because, unlike simple stored procedures, the CREATE PROCEDURE FROM CLASS statement takes the procedure name from the name of the class; in this case, *UpdatePeople*.

Go ahead and enter the CREATE PROCEDURE FROM CLASS statement at the sqlcmd prompt to bring your database up to date:

```
$ sqlcmd
1> CREATE PROCEDURE
2>   PARTITION ON TABLE people COLUMN state_num
3>   FROM CLASS UpdatePeople;
```

Putting it All Together

OK. Now we have a Java stored procedure and an updated schema. We are ready to try them out.

Obviously, we don't want to invoke our new procedure manually for each record in the People table. We could write a program to do it for us. Fortunately, there is a program already available that we can use.

The **csvloader** command normally uses the default INSERT procedures to load data into a table. However, you can specify an different procedure if you wish. So we can use **csvloader** to invoke our new procedure to update the database with every record in the data file.

First we must filter the data to the columns we need. We use the same shell commands we used to create the initial input file, except we switch to selecting the column with data for the 2011 estimate rather than

the actual census results. We can save this file as `data/people2011.txt` (which is included with the source files):

```
$ grep -v "^040," data/CO-EST2011-Alldata.csv \
| cut --delimiter="," --fields=4,5,7,11 > data/people2011.txt
```

Before we update the database, let's just check to see which are the two counties with the smallest population:

```
$ sqlcmd
SQL Command :: localhost:21212
1> SELECT TOP 2 county, abbreviation, population
2> FROM people,states WHERE people.state_num=states.state_num
3> ORDER BY population ASC;
COUNTY          ABBREVIATION  POPULATION
--------------- ------------- -----------
Loving County   TX                     82
Kalawao County  HI                     90


(2 row(s) affected)
```

Now we can run **csvloader** to update the database, using the `-p` flag indicating that we are specifying a stored procedure name rather than a table name:

```
$ csvloader --skip 1 --file data/people2011.txt \
       -p UpdatePeople
```

And finally, we can check to see the results of the update by repeating our earlier query:

```
$ sqlcmd
SQL Command :: localhost:21212
1> SELECT TOP 2 county, abbreviation, population
2> FROM people,states WHERE people.state_num=states.state_num
3> ORDER BY population ASC;
COUNTY          ABBREVIATION  POPULATION
--------------- ------------- -----------
Kalawao County  HI                     90
Loving County   TX                     94


(2 row(s) affected)
```

Aha! In fact, the estimates show that Loving County, Texas is growing and is no longer the smallest!

Part 6: Client Applications

We now have a working sample database with data. We even wrote a stored procedure demonstrating how to update the data. To run the stored procedure we used the pre-existing csvloader utility. However, most applications require more logic than a single stored procedure. Understanding how to integrate calls to the database into your client applications is key to producing a complete business solution, In this lesson, we explain how to interact with VoltDB from client applications.

VoltDB provides client libraries in a number of different programming languages, each with their own unique syntax, supported datatypes, and capabilities. However, the general process for calling VoltDB from client applications is the same no matter what programming language you use:

1. Create a client connection to the database.

2. Make one of more calls to stored procedures and interpret their results.

3. Close the connection when you are done.

This lesson will show you how to perform these steps in several different languages.

But before we begin, this lesson makes use of the Volt Management Console (VMC), which is a browser-based management tool for VoltDB, and in particular the JSON application programming interface (API) VMC supports. VMC is a separately installable and runnable component of Volt, which you can download and install the same way you do the Volt server software. Before beginning this lesson you should install and run VMC. For example, if you unpack the VMC tar file into your top level directory and rename the directory vmc, you can start VMC with the following command, using the ampersand to run it as a detached process:

```
$ ~/vmc/bin/vmc &
```

Making the Sample Application Interactive

As interesting as the population and location information is, it isn't terribly dynamic. Population does not change that quickly and locations even less so. Creating an interactive application around this data alone is difficult. However, if we add just one more layer of data things get interesting.

The United States National Weather Service (part of the Department of Commerce) issues notices describing dangerous weather conditions. These alerts are available online in XML format and include the state and county FIPS numbers of the areas affected by each weather advisory. This means it is possible to load weather advisories correlated to the same locations for which we have population and elevation data. Not only is it possible to list the weather alerts for a given state and county, we could also determine which events have the highest impact, in terms of population affected.

Designing the Solution

To make use of this new data, we can build a solution composed of two separate applications:

- One to load the weather advisory data
- Another to fetch the alerts for a specific location

This matches the natural break down of activities, since loading the data can be repeated periodically — every five or ten minutes say — to ensure the database has the latest information. Whereas fetching the alerts would normally be triggered by a user request.

At any given time, there are only a few hundred weather alerts and the alerts are updated only every 5-10 minutes on the NWS web site. Because it is a small data set updated infrequently, the alerts would normally be a good candidate for a replicated table. However, in this case, there can be — and usually are — multiple state/county pairs associated with each alert. Also, performance of user requests to look up alerts for a specific state and county could be critically important depending on the volume and use of that function within the business solution.

So we can normalize the data into two separate tables: nws_alert for storing general information about the alerts and local_event which correlates each alert (identified by a unique ID) to the state and county it applies to. This second table can be partitioned on the same column, state_num, as the towns and people tables. The new tables and associated indexes look like this:

```
CREATE TABLE nws_event (
   id VARCHAR(256) NOT NULL,
   type VARCHAR(128),
   severity VARCHAR(128),
   SUMMARY VARCHAR(1024),
   starttime TIMESTAMP,
   endtime TIMESTAMP,
   updated TIMESTAMP,
   PRIMARY KEY (id)
);

CREATE TABLE local_event (
    state_num TINYINT NOT NULL,
    county_num SMALLINT NOT NULL,
    id VARCHAR(256) NOT NULL
);

CREATE INDEX local_event_idx ON local_event (state_num, county_num);
CREATE INDEX nws_event_idx ON nws_event (id);

PARTITION TABLE local_event ON COLUMN state_num;
```

It is possible to add the new table declarations to the existing schema file. However, it is possible to load multiple separate schema files into the database using sqlcmd, as long as the DDL statements don't overlap or conflict. So to help organize our source files, we can create the new table declarations in a separate schema file, `weather.sql`. We will also need some new stored procedures, so we won't load the new DDL statements right now. But you can view the weather.sql file in your tutorial directory to see the new table declarations.

Designing the Stored Procedures for Data Access

Having defined the schema, we can now define the stored procedures that the client applications need. The first application, which loads the weather alerts, needs two stored procedures:

- FindAlert — to determine if a given alert already exists in the database
- LoadAlert — to insert the information into both the nws_alert and local_alert table

The first stored procedure is a simple SQL query based on the id column and can be defined in the schema. The second procedure needs to create a record in the replicated table nws_alert and then as many records in local_alert as needed. Additionally, the input file lists the state and county FIPS numbers as a string of six digit values separated by spaces rather than as separate fields. As a result, the second procedure

must be written in Java so it can queue multiple queries and decipher the input values before using them as query arguments. You can find the code for this stored procedure in the file `LoadAlert.java` in the tutorial directory.

These procedures are not partitioned because they access the replicated table nws_alert and — in the case of the second procedure — must insert records into the partitioned table local_alert using multiple different partitioning column values.

Finally, we also need a stored procedure to retrieve the alerts associated with a specific state and county. In this case, we can partition the procedure based on the state_num field. This last procedure is called GetAlertsByLocation.

The following procedure declarations complete the `weather.sql` schema file:

```
CREATE PROCEDURE FindAlert AS
   SELECT id, updated FROM nws_event
   WHERE id = ?;

CREATE PROCEDURE FROM CLASS LoadAlert;

CREATE PROCEDURE GetAlertsByLocation
   PARTITION ON TABLE local_event COLUMN state_num
   AS SELECT w.id, w.summary, w.type, w.severity,
             w.starttime, w.endtime
             FROM nws_event as w, local_event as l
             WHERE l.id=w.id and
                   l.state_num=? and l.county_num = ? and
                   w.endtime > TO_TIMESTAMP(MILLISECOND,?)
             ORDER BY w.endtime;
```

Now the stored procedures are written and the additional schema file created, we can compile the Java stored procedure, package both it and the UpdatePeople procedure from Part Five into a Jar file, and load both the procedures and schema into the database. Note that you must load the stored procedures first, so the database can find the class file when it processes the CREATE PROCEDURE FROM CLASS statement in the schema:

```
$ javac -cp "$CLASSPATH:/opt/voltdb/voltdb/*"  LoadAlert.java
$ jar cvf storedprocs.jar *.class
$ sqlcmd
1> load classes storedprocs.jar;
2> file weather.sql;
```

Creating the LoadWeather Client Application

The goal of the first client application, LoadWeather, is to read the weathers alerts from the National Weather Service and load them into the database. The basic program logic is:

1. Read and parse the NWS alerts feed.

2. For each alert, first check if it already exists in the database using the FindAlert procedure.

   - If yes, move on.

   - If no, insert the alert using the LoadAlert procedure.

Since this application will be run periodically, we should write it in a programming language that allows for easy parsing of XML and can be run from the command line. Python meets these requirements so we will use it for the example application.

The first task for the client application is to include all the libraries we need. In this case we need the VoltDB client library and standard Python libraries for input/output and parsing XML. The start of our Python program looks like this:

```
import sys
from xml.dom.minidom import parseString
from voltdbclient import *
```

The beginning of the program also contains code to read and parse the XML from standard input and define some useful functions. You can find this in the program `LoadWeather.py` in the tutorial directory.

More importantly, we must, as mentioned before, create a client connection. In Python this is done by creating an instance of the FastSerializer:

```
client = FastSerializer("localhost", 21212)
```

In Python, we must also declare any stored procedures we intend to use. In this case, we must declare FindAlert and LoadAlert:

```
finder = VoltProcedure( client, "FindAlert", [
FastSerializer.VOLTTYPE_STRING,
] )

loader = VoltProcedure( client, "LoadAlert", [
FastSerializer.VOLTTYPE_STRING,
FastSerializer.VOLTTYPE_STRING,
FastSerializer.VOLTTYPE_STRING,
FastSerializer.VOLTTYPE_STRING,
FastSerializer.VOLTTYPE_STRING,
FastSerializer.VOLTTYPE_STRING,
FastSerializer.VOLTTYPE_STRING,
FastSerializer.VOLTTYPE_STRING
] )
```

The bulk of the work of the application is a set of loops that walk through each alert in the XML structure checking if it already exists in the database and, if not, adding it. Again, the code for parsing the XML can be found in the tutorial directory if you are interested. But the code for calling the VoltDB stored procedures is the following:

```
# Check to see if the alert is already in the database.
response = finder.call([ id ])
if (response.tables):
    if (response.tables[0].tuples):
          # Existing alert
          cOld += 1
    else:
          # New alert
          response = loader.call([ id, wtype, severity, summary,
                                   starttime, endtime, updated, fips])
          if response.status == 1:
                cLoaded += 1
```

Note how the application uses the response from the procedures in two different ways:

- The response from FindAlert (finder) is used to check if any records were returned. If so, the alert already exists in the database.
- The response from LoadAlert (loader) is used to verify the status of the call. If the return status is one, or success, then we know the alert was successfully added to the database.

There is additional information in the procedure response besides just a status code and the data returned by the queries. But LoadWeather shows two of the most commonly used components.

The last step, once all the alerts are processed, is to close the connection to the database:

```
client.close()
```

Running the LoadWeather Application

Because Python is a scripting language, you do not need to compile your code before running it. However, you do need to tell Python where to find any custom libraries, such as the VoltDB client. Simply add the location of the VoltDB client library to the environment variable PYTHONPATH. For example, if VoltDB is installed in your home directory as the folder `~/voltdb`, the command to use is:

```
$ export PYTHONPATH="$HOME/voltdb/lib/python/"
```

Once you define PYTHONPATH, you are ready to run LoadWeather. Of course, you will also need weather alerts data to load. A sample file of weather data is included in the tutorial files data directory:

```
$ python LoadWeather.py < data/alerts.xml
```

Or you can pipe the most recent alerts directly from the NWS web site:

```
$ curl https://alerts.weather.gov/cap/us.php?x=0 | python LoadWeather.py
```

Creating the GetWeather Application

Now the database contains weather data, we can write the second half of the solution — an application to retrieve all of the alerts associated with a specific location. In a real world example, the GetWeather application is relatively simple, consisting of a single call to the GetAlertsByLocation stored procedure for the user's current location. Run manually, one query at a time, this is not much of a test of VoltDB — or any database. But in practice, where there can be hundreds or thousands of users running the application simultaneously, VoltDB's performance profile excels.

To demonstrate both aspects of our hypothetical solution, we can write two versions of the GetWeather application:

- A user interface showing what it looks like to the user and how easy it is to integrate VoltDB into such applications.
- A high-performance application to emulate real world loads on the database.

VoltDB in User Applications

The first example adds VoltDB access to a user application. In this case, a web interface implemented in HTML and Javascript. You can find the complete application in the `/MyWeather` folder in the tutorial directory. Run the application by opening the file `GetWeather.html` in a web browser. If you use the

sample alert data included in the tutorial directory, you can look up alerts for Androscoggin County in Maine to see what the warnings look like to the user.

Most of the application revolves around the user interface, including HTML. CSS, and Javascript code to display the initial form and format the results. Only a very small part of the code is related to VoltDB access.

In fact, for applications like this VoltDB has simplified programming interfaces that do not require the explicit setup and tear down of a normal database application. In this case, we can use the JSON interface, which does not require you to open and close an explicit connection. Instead, you simply call the database with your query and it returns the results in standard JavaScript Object Notation (JSON). VoltDB takes care of managing the connection, pooling queries, and so on.

So the actual database call only takes up two statements in the file GetAlerts.js;

- One to construct the URL that will be invoked, identifying the database server, the stored procedure (GetAlertsByLocation), and the parameters.
- Another to do the actual invocation and specify a callback routine.

```
var url = "http://localhost:8080/api/2.0/" +
          "?Procedure=GetAlertsByLocation&Parameters=" +
          "[" + statenum + "," + countynum + "," + currenttime + "]";
callJSON(url,"loadAlertsCallback");
```

Once the stored procedure completes, the callback routine is invoked. The callback routine uses the procedure response, this time in JSON format, much the same way the LoadWeather application does. First it checks the status to make sure the procedure succeeded and then it parses the results and formats them for display to the user.

```
function loadAlertsCallback(data) {

    if (data.status == 1) {
      var output = "";
      var results = data.results[0];
      if (results.length > 0) {
          var datarow = null;
          for (var i = 0; i < results.length; i++) {
              datarow = results[i];
              var link = datarow['ID'];
              var descr = datarow['SUMMARY'];
              var type = datarow['TYPE'];
              var severity = datarow['SEVERITY'];
              var starttime = datarow['STARTTIME']/1000;
              var endtime = datarow['ENDTIME']/1000;
              output += '<p><a href="' + link + '">' + type + '</a> '
                        + severity + '<br/>' + descr + '</p>';
          }
       }  else {
          output = "<p>No current weather alerts for this location.</p>";
       }
       var panel = document.getElementById('alerts');
       panel.innerHTML = "<h3>Current Alerts</h3>" + output;


    } else {
```

```
        alert("Failure: " + data.statusstring);
    }
}
```

VoltDB in High Performance Applications

The second example of GetWeather emulates many users accessing the database at the same time. It is very similar to the voter sample application that comes with the VoltDB software.

In this case we can write the application in Java. As we did before with LoadWeather, we need to import the VoltDB libraries and open a connection to the database. The code to do that in Java looks like this:

```
import org.voltdb.*;
import org.voltdb.client.*;

   [ . . . ]

        /*
         * Instantiate a client and connect to the database.
         */
        Client2Config config = new Client2Config();
        Client2 client = ClientFactory.createClient(config);
        client.connectSync("localhost");
```

The program then does one ad hoc query. You can add ad hoc queries to your applications by calling the @AdHoc system procedure with the SQL statement you want to execute as the only argument to the call. Normally, it is best for performance to always use stored procedures since they are precompiled and can be partitioned. However, in this case, where the query is run only once at the beginning of the program to get a list of valid county numbers per state, there is little or no negative impact.

You use system procedures such as @AdHoc just as you would your own stored procedures, identifying the procedure and any arguments in the callProcedure method. Again, we use the status in the procedure response to verify that the procedure completed successfully.

```
ClientResponse response = client.callProcedureSync("@AdHoc",
    "Select state_num, max(county_num) from people " +
    "group by state_num order by state_num;");
if (response.getStatus() != ClientResponse.SUCCESS){
    System.err.println(response.getStatusString());
    System.exit(-1);
}
```

The bulk of the application is a program loop that randomly assigns a state and county number and looks up weather alerts for that location using the GetAlertsByLocation stored procedure. The major difference here is that rather than calling the procedure synchronously and waiting for the results, we call the procedure asynchronously and move immediately on to the next call.

```
while ( currenttime - starttime < timelimit) {

      // Pick a state and county
    int s = 1 + (int)(Math.random() * maxstate);
    int c = 1 + (int)(Math.random() * states[s]);

      // Get the alerts
    client.callProcedureAsync("GetAlertsByLocation",
```

```
            s, c, new Date().getTime())
            .whenComplete(GetWeather::alertCallback);

        currenttime = System.currentTimeMillis();
        if (currenttime > lastreport + reportdelta) {
            DisplayInfo(currenttime-lastreport);
            lastreport = currenttime;
         }
}
```

Asynchronous procedure calls are very useful for high velocity applications because they ensure that the database always has queries to process. If you call stored procedures synchronously, one at a time, the database can only process queries as quickly as your application can send them. Between each stored procedure call, while your application is processing the results and setting up the next procedure call, the database is idle. Essentially, all of the parallelism and partitioning of the database are wasted while your application does other work.

By calling stored procedures asynchronously, the database can queue the queries and process multiple single-partitioned queries in parallel, while your application sets up the next procedure invocation. In other words, both your application and the database can run at top speed. This is also a good way to emulate multiple synchronous clients accessing the database simultaneously.

The asynchronous calls in the Volt Client2 return a `CompletableFuture`, which is a standard Java object that provides a useful variety of ways to manage asynchronous execution. Once an asynchronous procedure call completes, the CompletableFuture becomes "completed", and can execute futher actions.

In the example, we use `whenComplete()` to cause our `alertCallback` routine to be called. It is called with two arguments, a Volt `ClientResponse` and a Java `Throwable`, only one of which will be non-null. The presence of the Throwable indicates that an exception was thrown in the Client2 API during processing.

Assuming that the ClientResponse is provided rather than the Throwable, the callback procedure then processes the procedure response which it receives as an argument, just as your application would after a synchronous call.

```
static void alertCallback(ClientResponse response, Throwable error) {
    if (error != null) {
        System.err.printf("\n** " + error + "\n");
    } else {
        if (response.getStatus() == ClientResponse.SUCCESS) {
            VoltTable tuples = response.getResults()[0];
            tuples.resetRowPosition();
            while (tuples.advanceRow()) {
                String id = tuples.getString(0);
                String summary = tuples.getString(1);
                String type = tuples.getString(2);
                String severity = tuples.getString(3);
                long starttime = tuples.getTimestampAsLong(4);
                long endtime = tuples.getTimestampAsLong(5);
            }
        }
        txns++;
        if (txns % 50000 == 0) System.out.print(".");
    }
}
```

There is one important consideration for artificial load-generating programs such as this one. The callProcedureAsync method queues the procedure call in the Client2 API and immediately returns. The queue has a finite length, and we can add new entries faster than they can be sent to the cluster, so we are almost certain to overflow the queue. For the tutorial, we use the idea of a gate (class `Client2RequestGate`) which must be open before a new call can be queued, and which will close on receipt of a warning notification that the queue is approaching its limit. The gate will be reopened when the pending queue length falls sufficiently low.

The parameters that control this are set in the `Client2Config` that is used in client creation. In this tutorial, the maximum size of the queue is set to 3000, with a warning being given at 2000, which allows sufficient time to arrange for the main application thread to slow down. Sending can resume when the queue drops to 500 procedure calls. Finally, the warning and resume events are connected to our gate instance by defining a request-backpressure handler.

```
Client2Config config = new Client2Config()
            .clientRequestLimit(3000)
            .clientRequestBackpressureLevel(2000, 500)
            .requestBackpressureHandler(gate::operate);
```

The main thread ensures the gate is open before each procedure call is queued. This call returns without waiting, as long as there are no more than 2000 requests outstanding.

```
gate.waitOpen();
client.callProcedureAsync(...);
```

Finally, once the application has run for the predefined time period (by default, five minutes) it prints out one final report and closes the connection.

```
    // one final report
if (txns > 0 && currenttime > lastreport)
   DisplayInfo(currenttime - lastreport);

client.close();
```

Running the GetWeather Application

How you compile and run your client applications depends on the programming language they are written in. For Java programs, like the sample GetWeather application, you need to include the VoltDB JAR file in the class path. If you installed VoltDB as `~/voltdb`, a subdirectory of your home directory, you can add the VoltDB and associated JAR files and your current working directory to the Java classpath like so:

```
$ export CLASSPATH="$CLASSPATH:$HOME/voltdb/voltdb/*:$HOME/voltdb/lib/*:./"
```

You can then compile and run the application using standard Java commands:

```
$ javac GetWeather.java
$ java GetWeather
Emulating read queries of weather alerts by location...
.........................
1403551 Transactions in 30 seconds (46783 TPS)
.........................
1550652 Transactions in 30 seconds (51674 TPS)
```

As the application runs, it periodically shows metrics on the number of transactions processed. These values will vary based on the type of server, the configuration of the VoltDB cluster (sites per host, notes in the cluster, etc) and other environmental factors. But the numbers give you a rough feel for the performance of your database under load.

In Conclusion

How your database schema is structured, how tables and procedures are partitioned, and how your client application is designed, will all impact performance. When writing client applications, although the specifics vary for each programming language, the basics are:

- Create a connection to the database.
- Call stored procedures and interpret the results. Use asynchronous calls where possible to maximize throughput.
- Close the connection when done.

Much more information about how to design your client application and how to tune it and the database for maximum performance can be found online in the *Using VoltDB* manual and *VoltDB Performance Guide*.

Part 7: Next Steps

This tutorial has introduced you to the basic features of VoltDB. There are many more capabilities within VoltDB to optimize and enhance your database applications. To learn more about individual features and capabilities, see the *Using VoltDB* manual for more information.