Visitor Cookie Storage Engine Test Harness

Setting up and running the test harness is relatively easy. The following instructions should guide you through the process of installing and running the test harness.

Storage Engine Driver abstraction

The storage engine driver should implement the calls specified in the v-cookie abstraction layer. This is done by creating a class derived from VCookieStore. Only the three main methods (SaveVCookie, LoadVCookie, and DeleteOldVCookies) are required to be implemented. The remaining three (DeleteVCookie, GetVCookieCount, and GetVCookie) are optional but may help storage writers debug their code. Note however that the optional methods must still be defined (in order to prevent compiler errors related to instantiating a class with pure virtual functions) even though they do not need to be fully implemented.

```
class VCookieStore {
public:
    virtual ~VCookieStore () {}

    virtual bool SaveVCookie (VCookie const &vcookie) = 0;

    // Only VCookie::VCookie should ever call LoadVCookie
    virtual bool LoadVCookie (VCookie &vcookie) = 0;

    // returns the number of cookies that were deleted (if determining the number is expensive
    // we can probably change the return type to void)
    // Solutions have a lot of ways to implement this. Deletes really only need to happen once
    // per month, so a solution could create a new DB/table each month and any new or updated
    // vcookies are written to this new table. After 12 months, simply delete the oldest table.
    // However, this does require more space, because active users will have versions of their
    // vcookies in multiple tables. The size increase depends on how many visitors return
    // regularly and how long cookies survive in their browser environment.
    virtual unsigned long long DeleteOldVCookies (time_t purgeOlderThanThis) = 0;

    // These functions are not required. We don't currently have a use case for them
    // but they may facilitate testing. When we deploy a final implemantion, these will likely
    // not be part of it, and therefore efficiency is not a major concern for these functions.
    virtual bool DeleteVCookie (VCookie &vcookie) = 0;
    virtual unsigned long long GetVCookieCount () const = 0;
    virtual bool GetVCookie (VCookie &vcookie, unsigned long long index) const = 0;

    // If a VCookie implementation uses Key/Value pairs, it can use these serialization functions
    // for the value portion
    // The key would be the userid/visid. We will likely optimize these in the future, possibly
    // including compression of the data blob.
    // Since "last hit time" is used to expire a cookie, most implemenations will probably want
    // to store it separately from the blob so that they can search/delete old vcookies without
    // having to deserialize the whole vcookie
    static void Serialize (VCookie const &vcookie, std::vector<char> &buffer,
       bool saveLastHitTime=true);
    static bool Deserialize (VCookie &vcookie, const std::vector<char> &buffer);
};
```

A sample, in memory, v-cookie store implementation is provided in the abstraction directory. See the VCStoreInMemory.h file.

Unit testing storage engine drivers

Also included with the abstraction code is a simple unit test framework and source file (test.cpp). No makefile is included but the unit test is straight forward to compile. Edit the test.cpp source file to

include the appropriate header for your storage engine driver. Change the instantiation of the driver on line 88 to use your new class. Compile test.cpp and any implementation sources for your engine and then run the resulting unit test executable.

Building the test harness

The test harness is written in C++ and must be compiled for your environment and storage engine. The make file includes a mechanism to compile the test harness for different abstraction implementations.

To add your implementation you must follow two simple rules:

1. Your class name and the name of your header file must be the same. By convention, the name should be VCStore{vendor}.h. For example, the MySQL driver would be VCStoreMySQL.h and the class name would be VCStoreMySQL.
2. The include file should be in the same directory with the test harness source files and make file.

Modify the make file to include a new entry for your target. Start by duplicating one of the existing entries and modify the compile command as necessary (i.e. adding sources and libraries). By convention, we use two or three characters and an underscore followed by "testharness". Again, using the example of MySQL, we might name the target msq_testharness.

You also need to add the name of your new target to the end of rm command under the "clean" target so that it gets deleted when you invoke `make clean`.

Configuring tests

The test harness reads configuration information from a config file or from standard input. The config file supports the following parameters.

- *threads* the number of client request threads to create; default 10
- *requests* number of requests to submit (per thread); default 10,000
- *request-rate* number of requests per second (per thread), 0=unlimited; default 1000
- *replay-rate* rate multiplier for playback (1=100%)
- *replay-file* data warehouse file to read requests from (multiple allowed)

The rate of requests can be controlled one of two mutually exclusive ways; either by specifying a target request rate (requests per second) or by playing back request at some multiple of their original rate based on the timestamps captured in the data warehouse file. If the rate multiplier is specified, it will override the request rate parameter.

Request rate mode

When the test harness is running in "request rate" mode it will throttle each thread to the specified rate of requests per second. Note that this is the maximum rate and that the test harness can only limit requests to this rate, not accelerate a slow driver to perform faster than it is able. Request rate mode is useful for load testing and for exception testing (simulating failed server nodes, resharding / balancing the server cluster, etc.).

Replay rate mode

When running in "replay rate" mode, the test harness creates a virtual clock that steps along at the specified multiple of real time. This virtual clock is used to gate the requests from the data warehouse file based on the original timestamps recorded in the file.

Note that the *replay-file* parameter may be specified multiple times and files will be processed in the order that they are specified in the configuration file. Files should be listed in chronological order in the config file. **This has implications for the "replay rate" mode** because the internal virtual clock is based on the first hit that is read from the first file. If later files are not in chronological sync with the previous file, playback may cease (because the next timestamp is far into the virtual future) or may run unconstrained (because the next timestamp is in the virtual past).

Running the test harness

The test harness is intended to be run in a terminal environment. It does not provide a GUI interface. The test harness supports the following command line options.

- *--version* display the version string
- *--help* display basic help

In single client operation, the name of the configuration file should be specified on the command line. When running in a cluster of request nodes, the config file may be specified on the command line or configuration data may be read from standard input.

Running the test harness on a cluster

Since each instance of the test harness is simply a command line application, there are a variety of ways to launch them across a cluster of machines. The simplest is to start the app on multiple machines manually. While this works, it can be cumbersome to do, especially repeatedly.

Another simple solution is to use gearman (see www.gearman.org). Gearman is a generic application framework for managing a service on a cluster of machines. In this case, the 'service' is running the testharness. The test harness has been tested with gearman 0.34.

Install gearman using the usual package manager (yum, apt-get, etc.) for your system or build from source.

Start the gearman daemon on one box using something like:

```
gearmand -d
```

Once the gearman daemon is running, it is a simple matter to start a 'worker' to run the test harness in the background via:

```
gearman –h server_name –w –f vcookieTestHarness ./testharness &
```

Once all the workers are started, simply submit as many jobs to gearman as desired to start that many clients. Since the test harness will accept the config file via standard input, you can submit a job as follows:

```
gearman –h server_name –f vcookieTestHarness < test.conf
```

This will start one worker with the test config file. If you want to start more than one worker, simply repeat the above command multiple times. Note that gearman, by default, waits for the worker to finish and writes the output of the worker to standard out. Therefore, to run multiple copies at the same time, you probably want something like:

```
gearman –h server_name –f vcookieTestHarness < test.conf > test.out1 &
gearman –h server_name –f vcookieTestHarness < test.conf > test.out2 &
gearman –h server_name –f vcookieTestHarness < test.conf > test.out3 &
```

In all of the above commands, replace *server_name* with the machine name or IP address of the box running the gearman daemon (wherever you ran *gearmand -d*).

Test harness output

The output of the test harness includes some basic information about the configuration and operation of the tool. The most important part are the "rate = " lines for each child thread. These lines list the number of vcookie requests that were serviced each second by that thread. These numbers should be relatively consistent across all the child threads. If the rates are inconsistent, this may indicate that there are too many threads and some are being starved for resources. An example of the output using the in memory testing database driver is shown below.

```
> mem_testharness fast.conf
Adobe Visitor Profile Storage Test Harness 1.2 using VCStoreInMemory on n1.example.com

Reading config from fast.conf
Config: threads = 4; requests = 25000; request rate = 0

26747: Creating thread 1
26747: Creating thread 2
26747: Child 1 is alive at 1353100626
26747: Creating thread 3
26747: Child 2 is alive at 1353100626
26747: Creating thread 4
26747: Child 3 is alive at 1353100626
26747: Child 4 is alive at 1353100626
26747: Child 3 is done at 1353100630
26747-3: rate = 6368 6501 6580 6454
26747-3: readAvgNS = 1214 1295 1356 1440
26747-3: writeAvgNS = 5989 6097 6214 6229
26747: Child 2 is done at 1353100630
26747-2: rate = 5881 6490 6597 6439
26747-2: readAvgNS = 1350 1313 1356 1396
26747-2: writeAvgNS = 6573 6104 6253 6277
26747: Child 1 is done at 1353100630
26747-1: rate = 6265 6485 6301 6337
26747-1: readAvgNS = 1262 1298 1386 1371
26747-1: writeAvgNS = 6079 6099 6508 6337
26747: Child 4 is done at 1353100630
26747-4: rate = 5824 6506 6323 6416
26747-4: readAvgNS = 1391 1305 1394 1384
26747-4: writeAvgNS = 6639 6081 6464 6250

26747: aggregate rate = 24338 25982 25801 25646
26747: aggregate readAvgNS = 1301 1302 1372 1397
26747: aggregate writeAvgNS = 6308 6095 6357 6272
```

This shows that four child threads were created and each one serviced approximately 6,500 requests per second for an aggregate total of nearly 26,000 requests per second. This data can be captured and evaluated in Excel or other tools to plot graphs of performance for a given architecture.

Framework performance testing

You can test the speed of the entire framework with no storage engine by using the nop_testharness executable built as part of the make. This implements the store procedures by simply returning appropriate result codes rather than doing any actual work. This allows testing the maximum speed of the test harness, including reading hits, etc., in a given hardware environment.