S-Store Documentation

Release 0.1

S-Store Developers

Jun 23, 2017

Contents

CHAPTER 1

Introduction to S-Store

What is S-Store?

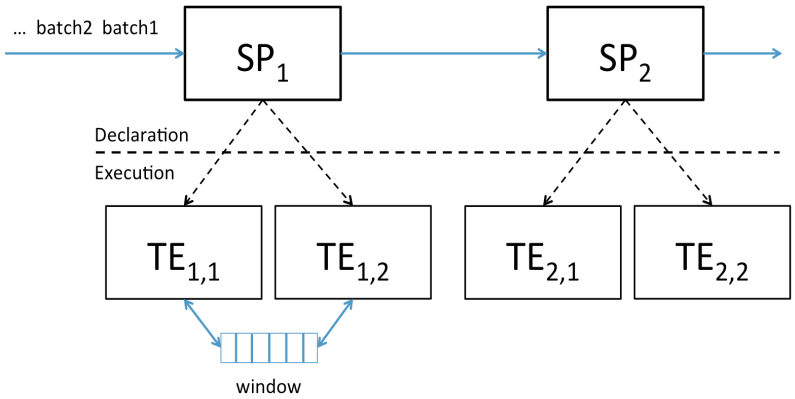
S-Store is the world's first streaming OLTP engine, which seeks to seemlessly combine online transactional processing with push-based stream processing for real-time applications. We accomplish this by designing our workloads as dataflow graphs of transactions, pushing the output of one transaction to the input of the next.

S-Store provides three fundamental guarantees, which together are found in no other system:

1. **ACID** - All updates to state are accomplished within ACID transactions
2. **Ordering** - S-Store executes on batches of data items, and ensures that batches are processed in an order consistent with their arrival.
3. **Exactly-once** - All operations are performed on data items once and only once, even in the event of failure

S-Store is designed for a variety of streaming use cases that involve shared mutable state, including real-time data ingestion, heartrate waveform analysis, and bicycle sharing applications, to name a few.

Transaction Model

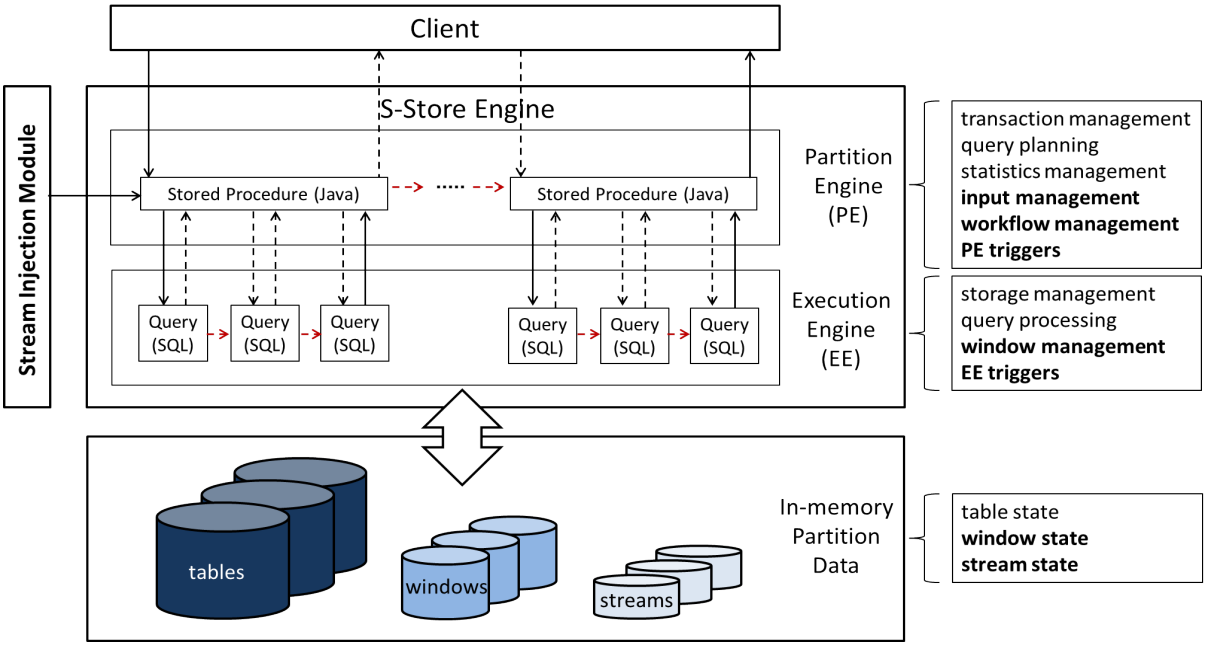


Like most streaming systems, S-Store models its workflows as a dataflow graph, a directed acyclic graph of operations. Much like OLTP systems such as H-Store, operations in S-Store are modeled as stored procedures (SP). Incoming tuples are grouped into batches, each of which must be executed as one atomic unit and ordered according to their arrival time. When a stored procedure executes one of these batches, it creates a transaction execution (TE). Transaction executions are fully ACID in their semantics, meaning that any state they touch are protected during execution and all changes are either committed or rolled back together.

In addition to these ACID semantics, S-Store ensures that each batch is executed in order according to their batch-ids. For each stored procedure, a transaction execution of batch-id B is guaranteed to commit before the TE of batch-id B+1. Similarly, for each batch-id B, stored procedure N is guaranteed to execute before stored procedure N+1, where N+1 is the next stored procedure downstream. Each of these TEs is executed once and only once, even in the event of a failure.

To learn more about applications, the transaction model, and design of S-Store, please read our publications at sstore.cs.brown.edu.

Architecture



S-Store is built on top of **H-Store**, a distributed main-memory OLTP database. You can read more about H-Store here. H-Store, in turn, is partially built on the same codebase as **VoltDB**. You can read more about VoltDB on their website. S-Store adds a number of streaming constructs to H-Store, including:

Streams - Append/delete data structures (queues) that push data from one piece of processing to another. Streams allow data to be passed from one SP to another in a dataflow graph.

Windows - Shifting materialized views that display a fixed quantity of data and shift as new data arrives. Windows can be either **tuple-based** (hold a fixed number of tuples) or **batch-based** (hold a fixed number of batches). Each window contains a **slide** value, which indicates how frequently it updates (again, in terms of tuples or batches).

Partition-Engine Triggers - Also known as PE triggers, these are attached to streams such that they trigger transactions of downstream stored procedures, allowing for push-based dataflow graphs.

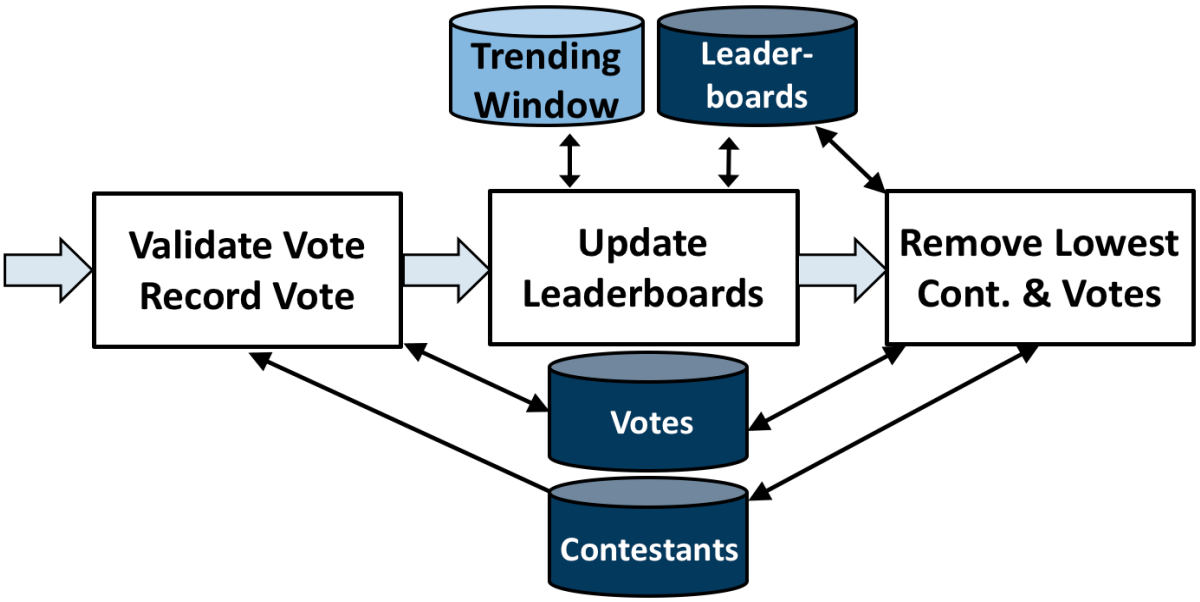
Execution-Engine Triggers - Also known as EE triggers, these are SQL statements attached to either windows or streams that execute when specific conditions are met (when new tuples are inserted for streams, and when the window slides in the case of windows).

At the moment, S-Store operates in single-node mode only, which means that there is no opportunity for parallelism. The S-Store scheduler currently executes batches completely serially, meaning that an entire batch is processed to completion within the context of a dataflow graph before the next batch is started.

As a main-memory database, S-Store features disk-based command log-based recovery. In case of failure, the user can replay S-Store's command logs in order to completely recreate the state of the system at the time of failure.

Running Example (Voter w/ Leaderboards)

S-Store comes with a number of benchmarks, including a simple streaming example meant to showcase the functionalities of S-Store. This benchmark, called votersstoreexample in the code, mimics an online voting competition in which the audience votes for their favorite contestant, a sliding window is generated of the current leaderboard, and periodically, based on who has the least votes in that moment, a contestant is removed from the running.



This workload can be broken down into three stored procedures:

Vote - This procedure validates and records a new vote for a contestant, then passes the vote downstream.

GenerateLeaderboard - This procedure creates a sliding window that indicates the current leaderboard of who has the most/least votes.

DeleteContestant - When a specific number of votes has been collected, the contestant with the fewest votes will be removed.

As shown in the diagram above, each procedure shares state with other procedures, making it necessary to use transactions for correct state management. By default, the benchmark takes a single tuple per batch, but can be configured to instead operate on larger batches of tuples.

Using S-Store

S-Store is licensed under the terms of the GNU Affero General Public License Version 3 as published by the Free Software Foundation. See the GNU Affero General Public License for more details. All software is provided as-is.

S-Store can be downloaded on GitHub

CHAPTER 2

Deploying and Executing S-Store

Quick Start (Dockerized)

The easiest way to build an S-Store instance is using a Docker image. Docker is a software container platform designed to allow code to run in a virtual container, with all requirements for installation already included. The biggest advantage to this method is that you do not need to worry about the requirements for S-Store, but instead can run an instance on any system that can run Docker. To learn more about how to install Docker on your specific system, visit https://www.docker.com/

1. Clone S-Store from github using the following command:

```
git clone http://github.com/s-store/s-store.git
```

2. Once you have Docker installed, you can then build the S-Store image. In a terminal, change to the directory that you just cloned S-Store into, and run the following command. This will download the fully-compiled S-Store docker image and start a new Docker container.

```
./scripts/setup_docker.sh
```

3. Once the script has finished, you will be connected to the S-Store container (named sstore-console) in the terminal where you ran the script. In the same terminal, navigate to the S-Store directory and restart the ssh service using the following commands:

```
cd root/s-store
service ssh restart
```

4. From there, you are free to run any benchmark available in the S-Store library. By default, the benchmark votersstoreexample is precompiled and ready to run. You can try running this using the following command:

```
ant sstore-benchmark -Dproject=votersstoreexample -Dclient.txnrate=1000
```

This will run the votersstoreexample benchmark for 60 seconds, submitting 1000 new tuples per second. You will also be able to run any other benchmark available in the S-Store library using the following commands:

ant sstore-prepare -Dproject={benchmark} ant sstore-benchmark -Dproject={benchmark}

Note: From the Docker container, you will also be able to pull any changes from git and have access to all git commands.

5. If you wish to run commands on S-Store in an interactive way, you will need to run two terminals. Once you are running the sstore-container in the first terminal, run the following command in the first terminal:

```
ant sstore-prepare -Dproject={BENCHMARK}
ant sstore-benchmark-console -Dproject={BENCHMARK}
```

For example, if the chosen benchmark is "votersstoreexample," the terminal will look roughly like this:

```
============================
===== Running containers=====
============================
root@799e8144766d:/# cd root/s-store
root@799e8144766d:~/s-store# service ssh restart
 * Restarting OpenBSD Secure Shell server sshd
↪                                        start-stop-daemon: warning: failed to kill␣
↪11521: No such process                                                            ␣
↪                                                    [ OK ]
root@799e8144766d:~/s-store# ant sstore-benchmark-console -Dproject=votersstoreexample
Buildfile: /root/s-store/build.xml

sstore-benchmark-console: ...
```

Then, in a second terminal, you will need to connect to the running container using the following commands:

```
docker exec -it sstore-console /bin/bash
cd /root/s-store
./sstore {BENCHMARK}
```

```
docker exec -it sstore-console /bin/bash
root@799e8144766d:/# cd /root/s-store
root@799e8144766d:~/s-store# ./sstore votersstoreexample
00:05:36,016 [main] (ParametersUtil.java:165) WARN  - Failed to find␣
↪ParameterMappingSet file 'votersstoreexample.mappings' in 'votersstoreexample.jar'
 ___      ___ _____ ___  ___ ___
/ __|___/ __|_   _/ _ \| _ \ __|
\__ \___\__ \ | || (_) |   / _|
|___/   |___/ |_| \___/|_|_\___|

Connected to localhost:21212 / Server Version: 1.0.01
```

Once connected to this second terminal, you can run SQL statements in order to query the database. For instance, if running votersstoreexample, you could run the following SQL statement to retrieve the number of tuples in the Votes table:

```
sstore> SELECT COUNT(*) FROM VOTES;
Result #1 / 1
  ??????
  ? C1 ?
  ? -- ?
  ? 69 ?
  ??????
1 row in set (0.55 sec)
```

Note: There are also a variety of statistics tools available as well. Check the Statistics section for more details.

6. To clean up any existing docker containers that are no longer needed, simply exit the running docker container by closing any running S-Store instances using ctrl+C. Then, simply use the following commands to exit the running docker container and clean up container:

```
exit
./scripts/cleanup_containers.sh
```

7. Some other general docker commands that you might want to use:

List all images and detailed information:

```
docker images
```

Check active and inactive containers and obtain any container's id:

```
docker ps -a
```

Manual Start (Environment Setup on Native Linux)

S-Store is easy to set up on any Linux machine, and is recommended as the easiest method of developing new benchmarks. You will need a **64-bit version of Linux** with at least 2 cores and a recommended 6 GB of RAM available. Native S-Store has the same requirements as its parent system, H-Store. These are:

- gcc/g++ +4.3
- JDK 1.6/1.7
- Python +2.7
- Ant +1.7
- Valgrind +3.5

Note: S-Store does **not** support JDK 1.8 at this time. You will need to use JDK 1.6 or 1.7. If you are running a machine with JDK 1.8 installed, you can either install 1.7 alongside it, or install S-Store within a virtual machine.

1. Install the required packages with the following commands:

```
sudo apt-get update
sudo apt-get --yes install subversion gcc g++ openjdk-7-jdk valgrind ant
```

2. In order to run S-Store, your machine needs to have OpenSSH enabled and you must be allowed to login to localhost without a password:

```
sudo apt-get --yes install openssh-server
ssh-keygen -t rsa # Do not enter a password
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

Execute this simple test to make sure everything is set up properly:

```
ssh -o StrictHostKeyChecking=no localhost "date"
```

You should see the date printed without having to put in a password. If this fails, then check your permissions in the ~/.ssh/ directory.

The S-Store source code can be downloaded from the Github repository using the following command:

```
git clone http://github.com/s-store/s-store.git
```

Once the code is downloaded and the desired branch selected, run the following command on the root directory of S-Store:

```
ant build
```

Note: This will build all of the portions of the S-Store codebase. Depending on the development environment, this can take a good bit of time. If your development is limited to benchmarks only, it is much quicker to simply rebuild the Java portion of the codebase using "ant build-java".

Note: S-Store must be run on a 64 bit Linux machine, preferably with at least 6 GB of RAM. If you have a Mac or Windows machine, I recommend installing a virtual machine using a free service such as VirtualBox. VirtualBox can be downloaded at www.virtualbox.org.

Compiling and Executing a Benchmark

Executing S-Store is very similar to executing H-Store, documented here. All commands, including **hstore-prepare**, **hstore-benchmark**, **catalog-info**, and **hstore-invoke** work as expected, in addition to the **hstore terminal tool**, which can be extremely helpful to view what actually exists in each table.

When running S-Store on a single node, these are the commands you will want to run. Note that you will need to recompile each time you make changes to your code.

```
ant clean-java build-java
ant sstore-prepare -Dproject=$benchmarkname
ant sstore-benchmark -Dproject=$benchmarkname $parameters
```

Or simply use the included shell script, which will run each command for you:

```
./runsstorev1.sh $benchmarkname $txnspersecond "other parameters here"
```

The runsstorev1.sh shell script uses a number of parameters that are desired by most S-Store runs, including the use of a single non-blocking client and disabling logging. If you want to run the script without those parameters, you can easily override them by re-adding the parameters with your desired values.

Interacting with a Live Database

Like most databases, it is possible to interact directly with a live S-Store database. Because S-Store is a main-memory database, it will need to reload data into its table objects every time it restarts. To interact with an S-Store database, you can run an existing benchmark in a way that does not shut down the system once the data has been loaded. The easiest way to do this is to use the following command:

```
ant sstore-benchmark-console -Dproject=$benchmarkname $parameters
```

This will automatically set the "noshutdown" parameter to true. Once S-Store is running, open another terminal window in the same root directory as S-Store. From there, you can open an interactive S-Store terminal by running (in a new terminal!):

```
./sstore $benchmarkname
```

From this interactive terminal, you can run adhoc SQL statements, as well as statistics_ transactions. This terminal window can remain open even once S-Store is stopped, and will automatically reconnect to a new S-Store instance run from the same root directory. However, clearly you will be unable to query the database when it is not running.

Environmental Parameters

S-Store adds a number of enviroment parameters to H-Store's base parameters. To use these parameters at runtime, use "-D" and then the parameter name (for instance, "-Dclient.txnrate=[txnrate]"). A full list of H-Store's parameters can be found here:

- Global Parameters
- Site Parameters
- Client Parameters

Some of the most helpful S-Store parameters are listed below:

client.txnrate:

- Default: 1000
- Permitted Type: integer
- Indicates the number of transactions per second that are being submitted to the engine (per client). If using the streamgenerator, it is recommended that you set this parameter to "-1", as this will cause the client to send as many transaction requests per second as are provided by the streamgenerator.

client.threads_per_host:

- Default: 1
- Permitted Type: integer
- Indicates the number of client threads that will be submitting transaction requests to the engine.

client.duration:

- Default: 60000
- Permitted Type: integer
- Indicates the period of time the benchmark will run, in milliseconds.

client.benchmark_param_0:

- Default: 0
- Permitted Type: integer
- Generic input parameter that can be used within a benchmark.

client.benchmark_param_str:

- Default: NULL

- Permitted Type: String
- Generic input parameter that can be used within a benchmark.

site.commandlog_enable:

- Default: false
- Permitted Type: boolean
- Indicates whether commands are being logged to disk.

noshutdown:

- Default: false
- Permitted Type: boolean
- Keeps S-Store running, even after the benchmark has completed.

noexecute:

- Default: false
- Permitted Type: boolean
- Causes the benchmark to run, but no requests to be sent from the client.

There are several S-Store-specific parameters as well. They are:

global.sstore:

- Default: true
- Permitted Type: boolean
- Enables S-Store and its related functionality. When set to false, the system should operate as pure H-Store.

global.sstore_scheduler:

- Default: true
- Permitted Type: boolean
- Enables the serial scheduler, which ensures that when a procedure triggers another procedure, that transaction is scheduled before any other.

global.weak_recovery:

- Default: true
- Permitted Type: boolean
- Enables the weak recovery mechanism, which only logs the "border" stored transactions that exist at the beginning of a workflow. If not enabled, then strong recovery is used instead.

global.sstore_frontend_trigger:

- Default: true
- Permitted Type: boolean
- Enables frontend (PE) triggers.

client.input_port:

- Default: 21001
- Permitted Type: integer
- Specifies which port the streamgenerator should connect to

client.input_host:

- Default: “localhost”
- Permitted Type: String
- Specifies which hostname the streamgenerator should connect to

client.bigdawg_port:

- Default: 21002
- Permitted Type: integer
- Specifies the port to be used to connect to BigDAWG

CHAPTER 3

Writing Applications/Benchmarks in S-Store

The most common use of S-Store is the creation of applications, or benchmarks. S-Store supports benchmarks with both streaming workloads and/or OLTP workloads.

Creating a New Benchmark

To begin creating a benchmark, please follow the instructions in the H-Store documentation, linked here. The high-level overview of these steps are below:

1. Create a **new directory** in the source tree under the src/benchmarks directory that will contain your benchmark. The high-level directory for your benchmark should contain a separate directory for the stored procedures. In the example shown below, the "votersstoreexample" benchmark is being created under the "edu.brown.benchmark" package:

```
mkdir -p src/benchmarks/edu/brown/benchmark/votersstoreexample
mkdir src/benchmarks/edu/brown/benchmark/votersstoreexample/procedures
```

Note: More details listed here: H-Store Setup.

2. Add your benchmark to the **ProjectType** class, located in src/frontend/edu/brown/utils/ProjectType.java

```
public enum ProjectType {
        // "Benchmark Identifier" ("Benchmark Name", "Benchmark Package")
        VOTERSSTOREEXAMPLE ("VoterSStoreExample", "edu.brown.benchmark.
↪votersstoreexample");
        ...
}
```

Note: More details listed here: H-Store ProjectType.

3. Create a **schema file** (DDL file) in your benchmark directory. In votersstoreexample, the schema file is named votersstoreexample-ddl.sql

Note: More details listed here: H-Store Schema.

4. Create **stored procedures** for your benchmark in the "procedures" folder created earlier. These stored procedures will extend the VoltProcedure class. They should contain parameterized SQL statements that are queued within the run(...) method.

Note: More details listed here: H-Store StoredProcedures.

5. Create the **project builder** class, which will extend the AbstractProjectBuilder. It will define 1) the benchmark's Data Loader class, 2) the benchmark's Client Driver class, 3) the default stored procedures that are included, and 4) the default partitioning scheme for the tables/windows/streams in the database. The Data Loader and Client Driver class files must be defined in two static parameters m_clientClass and m_loaderClass, respectively.

Note: In single-sited S-Store, the partitioning scheme used is not important, as all state will live on the same partition. It is still recommended to include a partitioning scheme nonetheless. More details listed here: H-Store ProjectBuilder.

6. Create the **data loader**, which is used to populate the database with initial data before the benchmark begins.

Note: More details listed here: H-Store DataLoader.

7. Create a **client driver** class, which will submit transaction requests at a rate specified at runtime. This will be covered in further detail in the next section.
8. Create a **benchmark specification** file in properties/benchmarks/ that will be named [classname].properties. By default, S-Store will look for the specification file in this directory based on the value of the project parameter. This file needs to contain a single line, the location of the project builder class. Here is the example of votersstoreexample.properties:

```
builder = edu.brown.benchmark.votersstoreexample.VoterSStoreExampleProjectBuilder
```

Note: More details listed here: H-Store Specification.

While the core benchmark pieces, such as the schema, stored procedures, project builder, and client are fundamentally the same, there are some important differences between an S-Store workload that features dataflow graphs and an H-Store OLTP workload. These differences are listed below.

Creating Batches and a Client

S-Store executes and orders its dataflow processing in terms of batches. A batch can be considered a group of one or more tuples that should be executed as a single, atomic unit. Each batch includes a batch_id, which is associated with the time at which the batch arrived. These batch_ids are attached to transactions incoming tuples are processed in order. Batches and batch_ids are currently created on the client side.

Programming a client is very similar to the process described in the H-Store documentation (H-Store Client.), but with some key differences. There are two primary methods of ingesting data from the client to the engine. The first method,

similar to H-Store, is to generate new tuples directly within the client. This method is best when data is fabricated, as the input rate can easily be controlled at runtime. The second method is to use the StreamGenerator tool (documented here) to simulate an incoming stream.

The client consists of two major methods for repeatedly inserting new tuples into the engine: the runLoop() method and the runOnce() method. Within the runOnce() method, the user can define a segment of code that runs x times per second, where x is defined by the -Dclient.txnrate parameter at runtime for each client. An example can be found below:

```
static AtomicLong batchid = new AtomicLong(0); //monotonically-increasing batchid
                        //increased when a new batch is created
TupleGenerator tuplegenerator = new TupleGenerator(); //custom-designed tuple␣
↪generator class for fabricating new tuples

protected boolean runOnce() throws IOException {

        Client client = this.getClientHandle();

        //create a new tuple, with a pre-defined schema
        Object tuple = tupleGenerator.createNewTuple();

        //asynchronous call for stream procedures
        boolean response = client.callStreamProcedure(callback,
                                                                                ␣
↪           "SP1",
                                                                                ␣
↪           batchid.getAndIncrement(),
                                                                                ␣
↪           tuple.t_id,
                                                                                ␣
↪           tuple.t_value);

        return response;
}
```

The runLoop() method runs as one would expect a loop to: as many times as possible, with no hesitation. runLoop() is best used with the streamgenerator, as it automatically ingests tuples at whatever rate the streamgenerator is producing them. The easiest way to code in such a way that both the runOnce() and runLoop() method can be used is to place all of the inner loop code within runOnce(), and then call runOnce() repeatedly from within runLoop(), like so:

```
//to use "runLoop" instead of "runOnce," set the client.txnrate param to -1 at runtime
public void runLoop() {
        try {
                while (true) {
                        try {
                        runOnce();
                        } catch (Exception e) {
                                failedTuples.incrementAndGet();
                        }
                } // WHILE
        } catch (Exception e) {
                e.printStackTrace();
        }
}
```

As new tuples arrive, it is up to the client to group them into batches. This can be done in several ways, but the easiest way is to create a String array with a size equal to the maximum number of tuples that you intend to send per batch. In each iteration of the loop, the runOnce method takes in a new tuple and adds it to a batch. When an entire batch is

ready, that batch is submitted to the system by calling the client.callStreamProcedure(Callback, ProcName, Batch_Id, Tuples) method. An example of this can be found below.

```
static AtomicLong batchid = new AtomicLong(0);

protected boolean runOnce() throws IOException {
        String tuple = tuple_id + "," + tuple_value; //create tuple, DO NOT include a␣
↪batch_id
        curTuples[i++] = tuple;
        if (BenchmarkConstants.NUM_PER_BATCH == i) { // We have a complete batch now.
                Client client = this.getClientHandle();
                boolean response = client.callStreamProcedure(callback, "SP1",␣
↪batchid.getAndIncrement(), (Object[]) curTuples);
                i = 0;
                curTuples = new String[BenchmarkConstants.NUM_PER_BATCH];
        }
}
```

runOnce()/runLoop() can easily be connected to the StreamGenerator using a clientSocket and BufferedInputStream, as shown below:

```
static AtomicLong batchid = new AtomicLong(0);

public void runLoop() {
        Socket clientSocket = null;

        try {

                clientSocket = new Socket(BenchmarkConstants.STREAMINGESTOR_HOST,␣
↪BenchmarkConstants.STREAMINGESTOR_PORT);
                clientSocket.setSoTimeout(5000);

                BufferedInputStream in = new BufferedInputStream(clientSocket.
↪getInputStream());

                int i = 0;
                while (true) {
                        int length = in.read();
                        if (length == -1 || length == 0) { //end of input stream
                                if (i > 0) {
                                        Client client = this.getClientHandle();
                                        boolean response = client.
↪callStreamProcedure(callback, "SP1", batchid.getAndIncrement(), (Object[])␣
↪curTuples);
                                        i = 0;
                                }
                                break;
                        }
                        byte[] messageByte = new byte[length];
                        in.read(messageByte);
                        String tuple = new String(messageByte);
                        curTuples[i++] = tuple;
                        if (BenchmarkConstants.NUM_PER_BATCH == i) {
                                // We have a complete batch now.
                                Client client = this.getClientHandle();
                                boolean response = client.
↪callStreamProcedure(callback, "SP1", batchid.getAndIncrement(), (Object[])␣
↪curTuples);
```

```
                                i = 0;
                                curTuples = new String[BenchmarkConstants.NUM_PER_
↪BATCH];
                        }
                }
        } catch (Exception e) {
                e.printStackTrace();
        }
}
```

Creating Tables, Windows, and Streams

As is the case in H-Store, application schemas are defined in a DDL file (H-Store DDL). The DDL file must be named the same as your benchmark, followed by "-ddl.sql".

There are three primary types of state in S-Store applications: Tables, Streams, and Windows. All three types of state are defined as tables, and all three are fully recoverable.

Tables constitute the primary "shared mutable state" of S-Store. Any publicly writeable data (accessible to all OLTP or ad-hoc queries) should be defined in a table. Creating tables is identical to both VoltDB and H-Store. The table schema and any indexes are defined as in the example below:

```
CREATE TABLE T1
(
tuple_id        bigint    NOT NULL,
tuple_val       integer   NOT NULL,
CONSTRAINT PK_T1 PRIMARY KEY (tuple_id)
);
```

Note: Partition keys for tables are defined in the ProjectBuilder class.

Streams are the primary method of moving information from one stored procedure to another within a dataflow graph. While the data is primarily passed through stored procedure arguments, it is important to also store the data in persistent streams as well for recovery purposes. Streams are logically append and remove only. For now, it is left to the application developer to prevent any updates to data items in a stream. Stream creation is very similar to table creation. An example of a stream is shown below.

```
CREATE STREAM S1
(
tuple_id        bigint          NOT NULL,
tuple_val       integer         NOT NULL,
batch_id        bigint          NOT NULL
);
```

Note: Automatic garbage collection on Streams is left to future functionality. The application developer should ensure that expired data items within Streams are garbage collected once the tuples are no longer needed (i.e. once the downstream SP has committed).

Windows hold a fixed quantity of data that updates as new data arrives. Windows can be either **tuple-based**, meaning that they always hold a fixed number of tuples, or **batch-based**, meaning that they hold a fixed number of batches at

any given time. Windows update periodically as a specific quantity of tuples or batches arrive. This is known as the window's **slide** value.

In order to create a window, the user must first create a stream that features the same schema as the window. This stream must feature two columns to be used by the system, but not by the user: *WSTART* and *WEND*. Both columns are to be left nullable, and should be of the INTEGER data type. Aside from defining these columns, the user does not need to be concerned with them. In the case of batch-based windows, the user must define a third column, *ts*, of the bigint data type. This column corresponds with the batch-id, and determines when the window slides. Unlike *WSTART* and *WEND*, the *ts* column must be managed by the user, and should be used as though it were a *batch_id* column. An example of this base stream is defined below:

```
CREATE STREAM stream_for_win
(
tuple_id        bigint          NOT NULL,
tuple_val       integer         NOT NULL,
ts                      bigint          NOT NULL,
WSTART          integer, --an integer column that is only used behind the scenes for␣
↪window management
WEND            integer --an integer column that is only used behind the scenes for␣
↪window management
);
```

Once the template stream has been defined, the window can be defined based on that. An example of a tuple-based window is below:

```
CREATE WINDOW tuple_win ON stream_for_win ROWS [*number of rows*] SLIDE [*slide␣
↪size*];
```

An example of a batch-based window is below:

```
CREATE WINDOW batch_win ON stream_for_win RANGE [*number of batches*] SLIDE [*slide␣
↪size*];
```

It is important to keep in mind that the window is its own separate data structure. When inserting tuples into a window, they should be directly inserted into the window rather than the base stream. Additionally, both the *WSTART* and *WEND* columns should be ignored during insert. An example insert statement is shown below:

```
//insert into window
public final SQLStmt insertProcTwoWinStmt = new SQLStmt(
        "INSERT INTO tuple_win (tuple_id, tuple_val, ts) VALUES (?,?,?);"
);
```

Windows slides are handled automatically by the system, as the user would expect. As new tuples/batches arrive, they are staged behind the scenes until enough tuples/batches arrive to slide the window by the appropriate amount. Garbage collection is handled automatically, meaning that the user does ever need to manually delete tuples from a window.

Note: In tuple-based window, no ordering is maintained within tuples in a batch. This means that if a stored procedure is replayed upon recovery, the result may differ from the original value. The results will remain consistent with our guarantees, however.

It is possible to attach an Execution Engine trigger to a window, as described below. EE triggers execute on each window slide, not necessarily on each tuple insertion.

Creating OLTP Stored Procedures

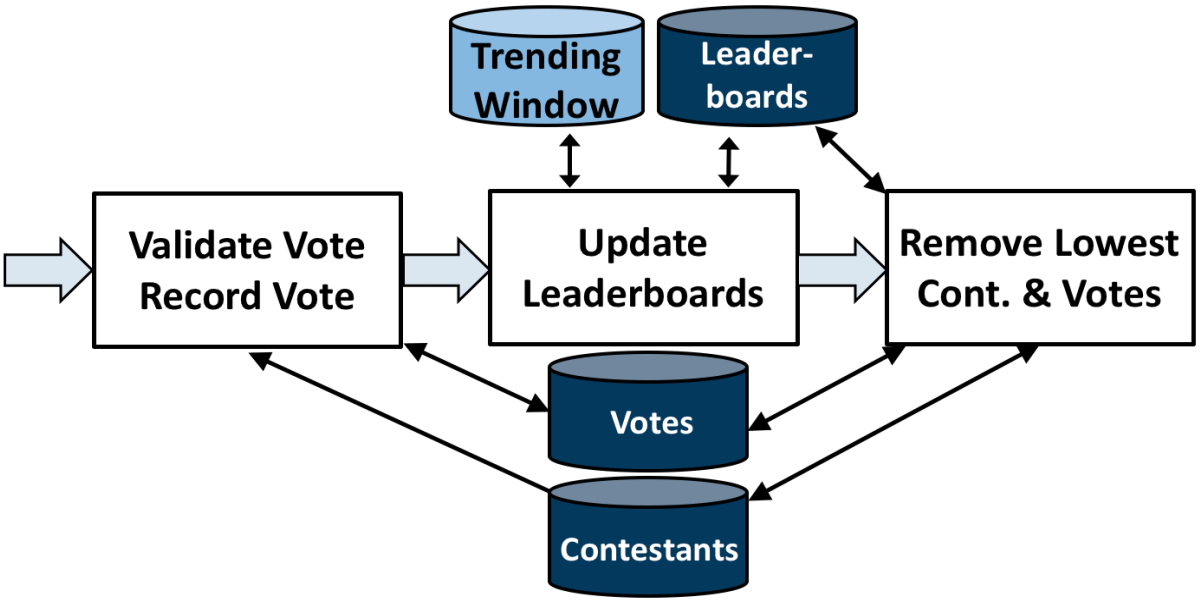
The primary unit of execution in S-Store are **stored procedures**. Each execution of an S-Store stored procedure on an input batch results in a **transaction** with full ACID properties. The definition of a stored procedure is very similar to that of H-Store Procedures (H-Store Procedures). Constant SQL statements are defined and then submitted to the engine with parameters to be executed in batches. An example of an OLTP stored procedure can be seen below.

```
@ProcInfo(
        partitionInfo = "t.t_id:0", //indicates that the partition that should be␣
↪accessed in this SP
                                    //corresponds to the "0th" parameter of the␣
↪run(params) method
                                    //hashed to match the t_id column of table t
        singlePartition = true;
)
public class OLTP extends VoltProcedure {

        public final SQLStmt insertOutputStream = "INSERT INTO t (t_id, t_val) VALUES␣
↪(?,?);"; //parameterized insert

        //the part of the stored procedure that actually runs on execution
        public long run(int t_id, int t_val) {

                voltQueueSQL(insertTValue, t_id, t_val);
                VoltTable response = voltExecuteSQL();

                return BenchmarkConstants.SUCCESS; //return a long that indicates␣
↪success
        }
}
```

In OLTP stored procedures, it is possible to pass any number of parameters into the "run()" method. These parameters should then be used with parameterized SQL statements, as shown above. The parameterized SQL statements are queued using the voltQueueSQL() method, and then submitted together to the Execution Engine using the voltExecuteSQL() method.

Creating Dataflow Graph Stored Procedures (Partition Engine Triggers)

Like most streaming systems, the main method of programming a workload in S-Store is via **dataflow graphs**. A dataflow graph in S-Store is a series of stored procedures which are connected via streams in a directed acyclic graph.



By default, each stored procedure in a dataflow graph executes on each batch that arrives from the input. When a stored procedure commits on an input batch, the S-Store scheduler automatically triggers a transaction execution of the downstream stored procedure. For each stored procedure, batch *b* should commit before batch *b+1*, and for each batch, stored procedure *t* is guaranteed to commit before transaction *t+1*. See the S-Store Engine section for more details on how this occurs and in what order the transactions will execute.

Below is an example of a dataflow graph SP, otherwise known as a Streaming SP:

```
@ProcInfo(
        partitionNum = 0; //states which partition this SP runs on
        singlePartition = true;
)
public class SP2 extends VoltProcedure {
        protected void toSetTriggerTableName()
        {
                addTriggerTable("proc_one_out");
        }

        public final SQLStmt getBatchId = "SELECT batch_id FROM proc_one_out ORDER BY␣
↪batch_id LIMIT 1";

        public final SQLStmt getInputStream = "SELECT t_id, t_val FROM proc_one_out␣
↪WHERE batch_id = ?;"; //define SQL statements here

        public final SQLStmt deleteInputStream = "DELETE * FROM proc_one_out WHERE␣
↪batch_id = ?;";

        public final SQLStmt insertOutputStream = "INSERT INTO proc_two_out (t_id, t_
↪val, batch_id) VALUES (?,?,?);"; //parameterized insert

        //the part of the stored procedure that actually runs on execution
        public long run(int part_id) {

                voltQueueSQL(getBatchId);
                VoltTable response = voltExecuteSQL();
                long batch_id = response[0].fetchRow(0).getLong("batch_id");
```

```
                //procedure work happens here
                voltQueueSQL(getInputStream, batch_id); //get tuples from the stream␣
↪for the given batch_id
                voltQueueSQL(deleteInputStream, batch_id); //manually remove tuples␣
↪from the stream
                response = voltExecuteSQL(); //returns results as an array of␣
↪VoltTables

                //iterates through all rows of the response to the first query
                for(int i = 0; i < response[0].getRowCount()) {
                        VoltTableRow row = response[0].fetchRow(i); //get the next row
                        long t_id = row.getLong("t_id");
                        int t_val = (int)row.getLong("t_val"); //integer is not an␣
↪option, use "long" and cast

                        //insert tuple into downstream
                        voltQueueSQL(insertOutputStream, t_id, t_val+10, batch_id);
                        voltExecuteSQL();
                }

                return BenchmarkConstants.SUCCESS;
        }
}
```

Dataflow graphs are defined as a series of triggering procedures, which are defined in each individual SP of the graph. At the beginning of each dataflow SP, the user should define what input stream triggers this particular SP within the *toSetTriggerTableName()* function. An example of this for *SP2* as listed below:

```
protected void toSetTriggerTableName() {
        addTriggerTable("proc_one_out"); //defines which stream will trigger this␣
↪procedure, as a tuple is inserted into it
}
```

Dataflow stored procedures are required to take in a single parameter:

int part_id - This parameter will automatically be filled in with the partitionNum ProcInfo parameter set at the beginning of the SP. It is irrelevant for single-partition S-Store, but will be used in the distributed version.

Again, currently stream maintenance is handled by the developer. It is very important that the developer at the minimum 1) pull the most recent information from the input stream, 2) delete the same info from the input stream, and 3) insert new stream information into the output stream, if necessary. Because single-node S-Store

Passing Data Along Streams using VoltStreams

Stream data is passed from procedure to procedure using VoltStreams as arguments. VoltStreams are attached to Stream tables that are defined in the DDL. The stream tables used should include a *batch_id* column of long data type, in addition to whatever other schema is required.

As mentioned in the previous section, downstream stored procedures are activated with every transaction invocation. This ensures that every SP executes for every batch_id, regardless of whether that batch contains new data that must be processed.

When data is being passed downstream, it must be inserted into a stream database object. The downstream transaction should then find the earliest batch_id in the stream, and use that to read the batch from the same stream database object. It should then manually perform garbage collection on the batch. The SQL statements required for this are shown below.

```
public final SQLStmt getBatchId = "SELECT batch_id FROM proc_one_out ORDER BY batch_
↪id LIMIT 1";
public final SQLStmt getInputStream = "SELECT t_id, t_val FROM proc_one_out WHERE␣
↪batch_id = ?;";
public final SQLStmt deleteInputStream = "DELETE * FROM proc_one_out WHERE batch_id =␣
↪?;";
```

Then, those SQL statements can be executed in batches, using the following commands:

```
voltQueueSQL(getBatchId);
VoltTable response = voltExecuteSQL();
long batch_id = response[0].fetchRow(0).getLong("batch_id"); //finds the batch_id␣
↪value

voltQueueSQL(getInputStream, batch_id);
voltQueueSQL(deleteInputStream, batch_id);
response = voltExecuteSQL();
```

Note: Garbage collection is not currently implemented for stream tables. Tuples can be removed from the stream in the same transaction that is reading from it, as the transactions are guaranteed to either fully commit or rollback.

Execution Engine Triggers

Execution Engine triggers (also known as **EE triggers** or **backend triggers**) are SQL statements that are attached to tables, windows, or streams. These triggers execute the attached SQL code immediately upon the insertion of a tuple. Note that if a batch of many tuples is inserted with one command, the trigger will fire once for each insertion.

EE triggers are defined in a way that is similar to stored procedures. They are placed in the "procedures" package of the benchmark, and similarly declared within the ProjectBuilder class. Any EE trigger object extends the VoltTrigger class. The stream/window/table to which the trigger is attached must be defined by overriding the "toSetStreamName()" method, which will return the target object name.

```
protected String toSetStreamName() {
        return "s1";
}
```

Each SQL statement that should be run upon tuple insert is then defined. These statements will run sequentially. Usually an "INSERT INTO...SELECT" statement will be used in order to somehow manipulate the data and push it downstream. Here is an example:

```
public final SQLStmt thisStmtName = new SQLStmt(
        "INSERT INTO sometable SELECT * FROM thisstream;"
);
```

EE triggers have different semantics depending on what type of object they are attached to. For streams and tables, the triggers execute the attached SQL code immediately upon the insertion of a tuple. Note that if a batch of many tuples is inserted with one command, the trigger will fire once for each insertion. Tuples are automatically garbage collected once the attached SQL has finished running.

EE triggers attached to windows, however, operate differently. Rather than firing on the insertion of new tuples, the triggers instead fire on the sliding of the window. This is particularly useful for aggregating the contents of a window upon slide and pushing it into a downstream table or stream.

There are some limitations. EE triggers are unable to accept parameterized SQL statements, but both joins and aggregates can be used. Additionally, EE triggers are unable to activate a PE trigger. This means that if a tuple is inserted into a PE trigger stream directly from an EE trigger, the downstream stored procedure will not be activated.

CHAPTER 4

S-Store Engine

Batching

S-Store dataflow graphs process all tuples in atomic batches, which have been defined by the client. These batches can have zero, one, or several tuples. Batch-ids are assigned by the client, attached to the input tuples, and sent to the engine with the transaction request.

Scheduler

The scheduler is responsible for ensuring that streaming transactions execute in the proper order. For each stored procedure, earlier batches are guaranteed to execute before later ones. For each batch, earlier stored procedures (in the dataflow graph) are guaranteed to execute before the later ones. The scheduler is also responsible for ensuring that each transaction executes once and only once.

In single-node S-Store, there is no opportunity for parallelism in transactions, and thus all dataflow graphs are executed serially. This means that when one SP executes on batch B, the next SP in the dataflow graph will then immediately execute on the same batch B before any other streaming transaction occurs. This has the added benefit of serving a similar purpose to nested transactions, as it prevents other transactions from accessing the state until a batch has completely executed within the dataflow graph.

Note: In the distributed case, stored procedures are assigned to a specific "home" node for distributed scheduling. This is future functionality.

Windows and EE Triggers

Under the hood, streams and windows are largely defined as extensions on SQL tables. This ensures that all state in an application is transactional and fully recoverable. Tuple management in windows in particular are handled in the

execution engine. New tuples are staged in the window upon insert, marked as invisible, only to be marked as visible when enough tuples/batches arrive for a window slide.

Execution engine triggers that are defined on a stream or table activate every time a new tuple is inserted. On a window, on the other hand, the EE triggers only execute when the window slides.

Logging

S-Store provides two types of logging, which can be toggled using "-Dglobal.weak_recovery" at runtime.

Strong recovery logging is very similar to that of H-Store/OLTP logging. Every transaction within the dataflow graph is logged, and every OLTP transaction is logged. This ensures that, on recovery, the transactions are replayed in the exact order that they were originally executed. When recovering using strong recovery, the scheduler's triggering of downstream transactions must be disabled in order to ensure that transactions are not replayed twice. Once the log has been fully replayed, then the scheduler should be re-enabled and allowed to generate transactions from any unprocessed tuples remaining in the streams.

Weak recovery logging, on the other hand, only records OLTP transactions and **border transactions**, or transactions at the beginning on the dataflow graph (on this particular node). When recovering using weak recovery, the scheduler remains on, so that every transaction then triggers a downstream transaction in the dataflow graph, as would happen in normal runtime. This recovery method guarantees that all transactions will execute exactly-once and in compliance with S-Store's ordering guarantees, but it does not necessarily ensure that the transactions will be replayed in the *exact* order as the original runtime.

By default, S-Store uses group commit for transaction logging in order to batch write transactions to disk. It is possible to disable group commit at runtime using the "-Dsite.commandlog_timeout=-1" option, though this will have a very large impact on performance. If this option is chosen, weak recovery is highly recommended.

CHAPTER 5

Stream Generator Tool

The Stream Generator is a simple tool able to take a flat CSV file and convert it to a stream of tuples, represented by strings. It is primarily used to simulate streams of tuples when no actual stream is available.

You will need to install Maven in order to compile the Stream Generator. You can find instructions for this at maven.apache.org, or by running:

```
sudo apt-get update
sudo apt-get install maven
```

Once Maven is installed, install the Stream Generator by navigating to the tools/streamgenerator directory. Then, run the command:

```
mvn install
```

To use the Stream Generator, it is recommended that you use the "stream-generator-v1-jar-with-dependencies.jar." A typical use looks like this:

```
java -jar target/streamgenerator.jar -f <file> -t <throughput> -d <duration> -p <port>
↪ -m <maxtupels>
```

The parameters:

- **-f <file>:** the CSV file that contains the data to stream
- **-t <throughput>:** the rate at which tuples are being sent to the system (per second)
- **-d <duration>:** the amount of time that the streamgenerator will run
- **-p <port>:** the port that the stream data will be sent to
- **-m <maxtuples>:** the maximum number of tuples that the streamgenerator will send

---

Note: The application/benchmark will need to be configured to receive tuples from the streamgenerator, and the benchmark configuration run with the setting "-Dclient.txnrate=-1" in order to receive as many tuples as possible from the stream.

Note: Batch-ids are assigned in the client and should not be pre-assigned by the streamgenerator.

CHAPTER 6

Connecting S-Store to BigDAWG

What is BigDAWG?

BigDAWG is a research polystore system developed by Intel. It supports heterogeneous database engines, multiple programming languages and complex analysis on a variety of workloads. BigDAWG provides a single user interface for querying several systems, allowing a user to potentially request data from multiple systems within a single query. It also contains the ability to easily and safely migrate data from one system to another. More information on BigDAWG is available on the BigDAWG website.

As a transactional streaming system, S-Store is able to serve several roles within BigDAWG. It can be used as a main-memory relational engine, much like its parent system, H-Store. It can be used as a pure streaming system. Or, if used as a hybrid of the two, S-Store is able to serve as a streaming data ingestion engine, able to transform incoming data items as they arrive and then migrate them to the appropriate engine.

Benchmark

BigDAWG features a sample workload operating on the MIMIC II dataset. We demonstrate the connection of S-Store and BigDAWG using the same dataset, with the S-Store benchmark mimic2bigdawg. In this configuration, S-Store is responsible for data ingestion into the polystore, specifically into Postgres. Benchmark mimic2bigdawg injects data into table medevents in S-Store, and S-Store periodically pushes data in medevents to table mimic2v26.medevents in Postgres. Analytical queries can be posted to BigDAWG. If mimic2v26.medevents is included in a query, BigDAWG will pull the data from S-Store first before executing the query in Postgres. This guarantees that the user always obtains the most fresh data injected into BigDAWG. We demonstrate this functionality by the dockerized BigDAWG and S-Store.

Setting up BigDAWG via Docker

Connecting S-Store to BigDAWG is easiest using Docker containers. Starting a BigDAWG cluster is easy, and only requires access to the BigDAWG repository.

Start a terminal. In the terminal, check out BigDAWG, switch to the sstore-injection branch, compile and execute using the following commands:

```
git clone https://github.com/bigdawg-istc/bigdawg.git
cd bigdawg
git checkout sstore-injection
./compile
cd provisions
./setup_bigdawg_docker.sh
```

S-Store ingests data in a rate of 100 tuples per second, for 10 minutes by default. After the ingestion is finished, S-Store stays online.

Note:

1. BigDAWG only compiles in JDK 8.
2. If BigDAWG is installed on Ubuntu, setup_bigdawg_docker.sh may reports errors during the setup of the BigDAWG catalog in Postgres. This is likely caused by Docker's default storage driver aufs. This can be fixed by switching the storage driver to devicemapper.
3. If BigDAWG is installed on Mac, please compile and run the setup script in Docker Quickstart Terminal as described in the BigDAWG documentation.

Querying through BigDAWG/JDBC

Once BigDAWG is started, it may still take S-Store a few minutes to be ready for data ingestion. Start a second terminal window. After one to two minutes, if BigDAWG is running on Ubuntu, execute the following query in the terminal:

```
curl -X POST -d "bdrel(select count(*) from mimic2v26.medevents;)" http://
↪localhost:8080/bigdawg/query/
```

If BigDAWG is running on Mac, execute the following query in the terminal:

```
curl -X POST -d "bdrel(select count(*) from mimic2v26.medevents;)" http://192.168.99.
↪100:8080/bigdawg/query/
```

The above query shows the amount of tuples in table mimic2v26.medevents in Postgres that have been migrated from S-Store.

Pushing data from S-Store to Postgres

When BigDAWG is started, it deletes the historical data in mimic2v26.medevents in Postgres by dropping the table. Once S-Store comes alive, BigDAWG recreates table mimic2v26.medevents in Postgres by the table definition in S-Store, and it starts to push data from S-Store to Postgres. Currently data is pushed from S-Store to Postgres on a time-based fashion only. The time between two pushes is defined in bigdawg/profiles/dev/dev-config.properties. The name of the entry is "sstore.injection.migrationGap", with the unit of millisecond, and is set to one minute (60000 milliseconds) by default, i.e., S-Store pushes data to Postgres once per minute.

Pulling data from S-Store

Data in a table is pulled from S-Store to Postgres for each SQL query that requires the table. Currently we support queries that require only one table a time from S-Store for transactional safety. The support for pulling multiple tables for one query is not yet provided, but is in progress.

Pushing/Pulling data via Binanry Format

Data are migrated from S-Store to Postgres in CSV format by default. The support for binary format is in progress.

CHAPTER 7

S-Store Statistics

Client-side Statistics

By default, when running S-Store benchmarks, transaction and latency statistics are provided in the terminal at a specified interval of time (by default, every 10 seconds). It is important to note that these statistics are recorded and managed on the client side, within the callback of the procedures. An example of how to manage these statistics is provided below:

```
private class Callback implements ProcedureCallback {
        private final int idx; //which SP id is associated with this callback

        public Callback(int idx) {
                this.idx = idx;
        }

        public void clientCallback(ClientResponse clientResponse) {
                incrementTransactionCounter(clientResponse, idx);
        }
}
```

Note: These statistics are only valid for transactions that originate at the client. Any transactions that occur later in the dataflow graph are not recorded on the client side, and must instead be viewed on the server side.

At the end of the benchmark, the S-Store client will output the transaction counter statistics to the terminal for all procedures run during the benchmark. While these statistics do not include latency or throughput, they provide an indication of how many downstream stored procedures executed during the benchmark period.

Server-side Statistics

In addition to the client-side statistics, the user can also view server-side statistics via the S-Store terminal. The easiest way to view these statistics is to open a separate S-Store terminal. To do this, use the "-Dnoshutdown=true" argument when running the benchmark. Once the benchmark has completed but the S-Store instance is still running, open another terminal and run the interactive S-Store terminal.

If running Dockerized S-Store, use the following command. You can find the container's ID by running "docker images":

If running native S-Store, you can simply run:

```
./sstore {BENCHMARK}
```

Once the S-Store terminal is running, you can query a variety of statistics using H-Store's @Statistics system transaction, which is fully documented here.

The most useful of these commands are:

```
exec @Statistics txncounter 1 #gives the transaction counter statistics
exec @Statistics table 1 #gives statistics about the tuples in each table
```

CHAPTER 8

Supported Features

Below is a list of supported features in S-Store. All features are directly implemented into the engine unless otherwise specified. Missing features are not directly supported in this release, but are in development for future releases

GENERAL MODEL:

- Write stored procedures: **SUPPORTED**
- Create dataflow graphs of stored procedures: **SUPPORTED**
- Pass data from one stored procedure to another: **SUPPORTED**
- Batch incoming stream data: **MANUALLY WRITTEN IN APPLICATIONS**
- OLTP transactions: **SUPPORTED**
- Nested transactions: **LIMITED SUPPORT - SCHEDULER SERIALIZES TXNS**
- Stream garbage collection: **MANUALLY WRITTEN IN APPLICATIONS**
- Distributed S-Store: **FUTURE RELEASE**

WINDOWS:

- Window incoming data: **SUPPORTED**
- Pass data from windows in-engine: **SUPPORTED**
- Group windowed data: **FUTURE RELEASE**
- Trigger SPs with window: **MANUALLY WRITTEN IN APPLICATIONS**
- Limiting window access from other SPs: **MANUALLY WRITTEN IN APPLICATIONS**
- Window garbage collection: **SUPPORTED**

ENGINE:

- Stored procedures execute ACID transactionally: **SUPPORTED**
- Ordered execution for dataflow graphs of SPs: **SUPPORTED**
- Proper triggering from one SP to next in dataflow graph: **SUPPORTED**

- Exactly-once - No repeat transactions: **SUPPORTED**

RECOVERY:

- Logging for weak recovery: **SUPPORTED**
- Logging for strong recovery: **SUPPORTED**
- Recovery for weak logging: **MANUALLY WRITTEN IN APPLICATIONS**
- Recovery for strong logging: **SUPPORTED**
- Snapshotting: **FUTURE RELEASE**
- Group commit: **SUPPORTED**
- Non-group commit: **SUPPORTED**

BIG DAWG:

- Query through JDBC: **SUPPORTED**
- Connect to BigDAWG: **SUPPORTED**
- Query from BigDAWG: **SUPPORTED**
- Migrate data from S-Store to Postgres: **SUPPORTED**
- Migrate data from Postgres to S-Store: **FUTURE RELEASE**

STATISTICS:

- Transaction counter (engine): **SUPPORTED**
- Transaction counter (client): **BORDER TXNS ONLY**
- Transaction Latency: **BORDER TXNS ONLY**
- Dataflow statistics: **FUTURE RELEASE**
- Table statistics: **SUPPORTED**

CHAPTER 9

S-Store Documentation

Introduction

Welcome to S-Store! S-Store is the world's first streaming OLTP engine, which seeks to seemlessly combine online transactional processing with push-based stream processing for real-time applications. We accomplish this by designing our workloads as dataflow graphs of transactions, pushing the output of one transaction to the input of the next.

S-Store provides three fundamental guarantees, which together are found in no other system:

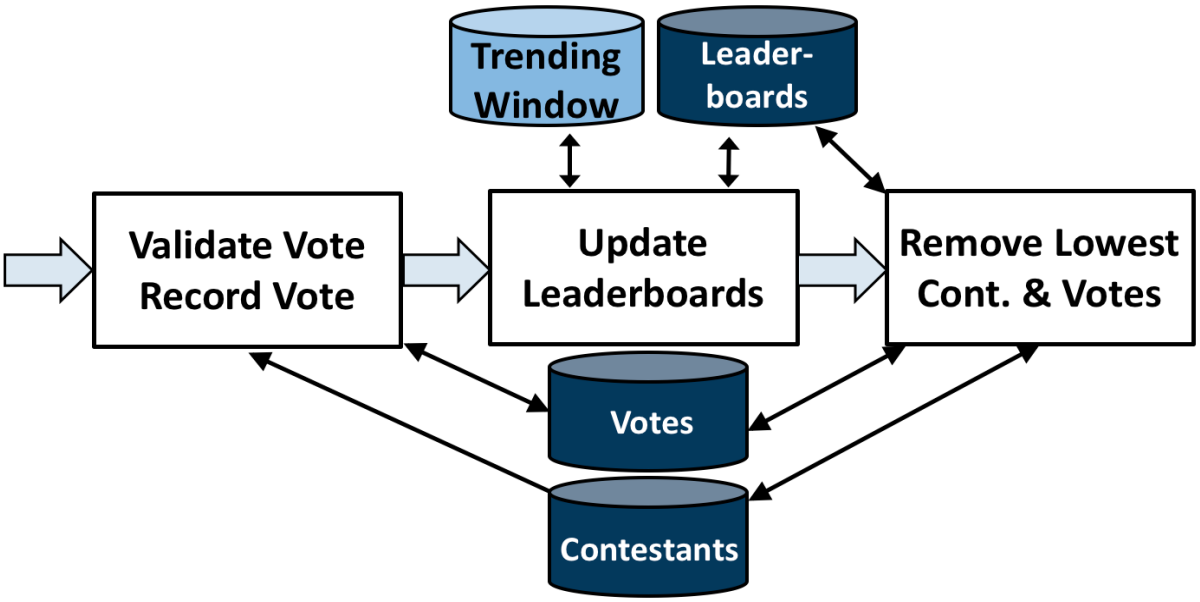
1. **ACID** - All updates to state are accomplished within ACID transactions
2. **Ordering** - S-Store executes on batches of data items, and ensures that batches are processed in an order consistent with their arrival.
3. **Exactly-once** - All operations are performed on data items once and only once, even in the event of failure

S-Store is designed for a variety of streaming use cases that involve shared mutable state, including real-time data ingestion, heartrate waveform analysis, and bicycle sharing applications, to name a few. To learn more about applications, the transaction model, and design of S-Store, please read our publications at sstore.cs.brown.edu.

S-Store is built on top of H-Store, a distributed main-memory OLTP database. You can read more about H-Store here.

A simple example

S-Store comes with a number of benchmarks, including a simple streaming example meant to showcase the functionalities of S-Store. This benchmark, votersstoreexample, mimics an online voting competition in which the audience votes for their favorite contestant, a sliding window is generated of the current leaderboard, and periodically, based on who has the least votes in that moment, a contestant is removed from the running.



This workload can be broken down into three stored procedures: Vote (collect the audience's votes), Generate Leaderboard (update the sliding window), and Delete Contestant (remove the lowest contestant every X votes).

Get the code

S-Store is licensed under the terms of the GNU Affero General Public License Version 3 as published by the Free Software Foundation. See the GNU Affero General Public License for more details. All software is provided as-is.

S-Store can be downloaded on GitHub