Assignment 3: MapReduce- and Storm-Style Computation

Due November 14, 2016, at 10:00pm EDT

1 Introduction
In this assignment, you will extend StormLite to a distributed framework, which essentially emulates Apache Storm – and in the process, also emulates MapReduce. You’ll do this by hooking in a servlet and your servlet container from Homework 1 (or Jetty). You will test your framework by implementing a simple WordCount.

As usual, you should start by checking out the framework / skeleton code (project upenn-cis455/HW3) from bitbucket. This includes an expanded version of StormLite with some MapReduce components.

2 Overview
StormLite gives you a basic abstraction for thinking about computation over units of data in streams. You’ll extend it to have a MapReduce-style architecture with two types of compute nodes: a number of workers and a single master. We will run map and reduce functions within StormLite bolts (with a spout that can read key-value data). Workers run spouts and bolts, and store the data that your MapReduce framework is working on; the master coordinates the workers and provides a user interface. A key issue is that in a MapReduce job, the inputs end and the reduce can only start once all inputs have been read.

For our StormLite MapReduce executor, there are some key classes.

- **Config** (the key-value pairs) will be used to specify various parameters on how your MapReduce engine’s topology is created and how it executes.
- **Job** is the basic class specifying what the map and reduce functions are. Think of it as a simplified Hadoop.

Additionally, we give you some spouts and bolts to help out:

- **FileSpout** is a file reader that reads a given file. If you have workers 0 and 1, FileSpout will read file.0 and file.1 on those machines, respectively.
- **MapBolt** runs map() from an instance of Job (as specified in the Config parameters)
- **ReduceBolt** runs reduce() from an instance of Job (as specified in the Config parameters).

  **ReduceBolt** buffers tuples until it knows that there is no more input (i.e., all potential senders have marked “end of stream”).
- **PrintBolt** simply prints the output to the console.

You’ll need to modify and expand some of these.
Each worker node will have a storage directory in which it keeps its local share of the data—temp files and also, as relevant, as BerkeleyDB environments/tables. For instance, a worker might have a storage directory called ~/store. Each worker may have multiple operators and threads, each with a bolt executor ID (such as 1234); you can use this to generate filenames or BerkeleyDB table names unique to the executor. Keep in mind that there will usually be many workers, and that the data will be distributed among them; for instance, you might have a system with 10 workers and 100,000 web pages total, but each worker might only store 10,000 of them. You should also keep in mind that there might be multiple executor threads running on a worker.

The master will have a status page on which it displays the list of workers that are currently online, as well as some information about each (e.g., the worker's IP address, and what the worker is doing). To keep this list up to date, the workers will periodically send some information about themselves to the master. The status page will also have an input form that allows the administrator to specify a MapReduce job to run, as well as some parameters (such as a subdirectory of the storage directory to read data from, a subdirectory to write the output to, etc.). When the administrator submits this form, the master forwards this information to each of the workers, which then begin processing the data.

The master will be implemented as a servlet—in other words, the master node will run your servlet container from HW1 with a MasterServlet.

The workers have their own WorkerServer for communication with one another and with the master. It’s basically a servlet but using a simplified API. You’ll need to run multiple WorkerServers, on multiple ports optionally on multiple machines. The actual MapReduce Jobs will simply be classes that implement a special interface (which contains a map and a reduce function). In a MapReduce framework like Hadoop, these classes would be sent from the master to the workers; to simplify the assignment, we will assume that these classes are already in the classpath on each worker.

3 The master

A first step will be to implement the MasterServlet, which will control and monitor progress from the Web. You can run it in Jetty or your servlet engine.

3.1 Status updates from the workers

The MasterServlet should provide a way for the workers to report their status. Specifically, it should accept GET requests for the URL /workerstatus, with the following parameters (in the query string):

- port: the port number on which the worker is listening for HTTP requests (e.g., port=4711)
- status: mapping, waiting, reducing or idle, depending on what the worker is doing (e.g., status=idle)
- job: the name of the class that is currently being run (for instance, job=edu.upenn.cis455.mapreduce.job.MyJob)
- keysRead: the number of keys that have been read so far (if the status is mapping or reducing), the number of keys that were read by the last map (if the status is waiting) or zero if the status is idle
- keysWritten: the number of keys that have been written so far (if the status is mapping or reducing), the number of keys that were written by the last map (if the status is waiting) or zero if the status is idle

---

1 Or Jetty, as optionally used in HW2.
the number of keys that were written by the last reduce (if the status is idle). If the node has never run any jobs, return 0.

- **results:** the set of results (up to the first 100) output by each worker. This will of course be empty until the ReduceBolt has actually produced output and fed it to the next stream.

We are going to assume that only one job is being run at any point in time. Note that the worker is *not* reporting its IP address because the master can get it from the request. (The port number is needed because the worker will usually use different ports for listening and for sending requests.) The master should keep this information, along with the time it has last received a /workerstatus request with a given IP:port combination.

### 3.2 The user-facing status / launch page

When a client requests the URL /status from the master servlet, the servlet should return a web page that contains (1) a table with status information about the workers, and (2) a web form for submitting jobs. The table should contain one row for each active worker (a worker is considered active if it has posted a /workerstatus within the last 30 seconds) and columns for (1) IP:port, (2) the status; (3) the job; (4) the keys read, and (5) the keys written.

*For simplicity, you can assume we are only going to execute a single job at a time.*

The web form for submitting jobs should contain fields for:

- The class name of the job (e.g., edu.upenn.cis.cis455.mapreduce.job.MyJob)
- The input directory, relative to the storage directory (e.g., if this is set to bar and the storage directory is set to ~/foo, the input should be read from ~/foo/bar)
- The output directory, relative to the storage directory
- The number of map threads (MapBolt executors) to run on each worker
- The number of reduce threads (ReduceBolt executors) to run on each worker

### 3.3 Launching jobs via the Master (or TestMapReduce)

When the administrator submits the web form on the /status page, the master node should first create an appropriate Topology using the FileSpout, MapBolt, ReduceBolt, PrintBolt or equivalent, and so on. Given the Topology and a Config specifying the MapReduce parameters, the Master should instantiate a WorkerJob object. The WorkerJob consists of two main parts:

1. The **Topology** can be built with the TopologyBuilder, and as with HW2M2, it describes the bolts and spouts and their connection. The same topology (as defined by the master) will get constructed and executed on each worker.

   Please see the TestMapReduce class as a model of how to assemble a MapReduce workflow with a FileSpout, a MapSpout and a ReduceSpout. As mentioned several times previously, the actual map and reduce classes are named in the Config (described next).

2. The **Config** is a series of key-value pairs specifying configuration information. Such information includes the list of workers, the specific integer index of the worker receiving the WorkJob (“here’s the topology, the list of 5 workers, and you are worker #3”), the number of executors etc. Again, please see the sample code.
Your first use of RESTful Web services (and Jackson). Now the Master will POST a message to the URL /definejob on each active worker. The POST should send JSON representing the WorkerJob object to the worker’s /definejob Handler (this can be nicely done with Jackson’s ObjectMapper). The Master should send this to each worker.

Finally, the Master should send a blank POST to /runjob on each WorkerServer, to actually start the computation. Now the worker will create a topology, link it to the other workers in the config, and periodically “check back in” with the server.

Test driver / worker startup. We’ve included a simple TestMapReduce class to show the basics of how this is done. You’ll likely want to use that class to run workers (execute it with an integer to indicate which worker in the workerList specified in the Config). You can also run it with an integer index and a non-blank parameter to designate one node as the initiator of the MapReduce computation.

4 The Worker

4.1 Basic computation in a distributed setting

In a distributed, multithreaded setting, there can be multiple executors running “copies” of the same spout/bolt in different threads. Moreover, different workers will have their own local copies. At each stage, as tuples are output by a spout/bolt, they can be routed by the OutputCollector and StreamRouter to the next bolt – either directly to that bolt or to another worker via a “SenderBolt”. Let’s look at the picture below:

The two rows represent two workers. On the top worker, we have one executor of a FileSpout, which feeds its output stream to two local Map Bolt executors, as well as two remote Map Bolt executors running on the bottom worker, representing 4 possible destinations. Next the Map Bolts send data to two Reduce Bolt executors (one on each worker). Finally, the Reduce Bolts send to a single Printer Bolt that produces the final output.

To make this type of a flow work, we need to address two different issues: (1) how to partition the data as each spout/bolt sends to the next spout/bolt, and (2) how to determine when to switch from the map step, which accumulates tuples, to the reduce step, which processes them.

In the figure above, the File Spouts produce outputs that need to be split across 4 map bolt executors (2 local, 2 remote). The circle with a 4 in it represents the StreamRouter that “round robin” across the 4 different potential destinations so each gets the same load. For 2 of the potential destinations, we want to send the data to the other worker; it needs to receive the data and “round robin” among its two map bolt executors.
In the next step, the Map Bolts’ StreamRouters need to partition data by key – “shard” the data using a FieldBased grouping. For keys hashing to an even number, the data should go to the Reduce Bolt running on the top node; for odd numbers, the data should go to the Reduce Bolt running on the bottom node.

For certain other settings, you may want to use the RoundRobin grouping (it just rotates among all destinations). If you want a single destination, such as the PrintBolt in our sample code, you can use a First grouping.

4.2 Distributed communication

We have laid most of the ground work for worker-to-worker communication, but you need to take the final steps. When a tuple is produced, it is sent to the StreamRouter class and its subclasses (FieldBased for sharding, RoundRobin for round-robining, First for the first child). The StreamRouter has a list of potential destination bolts.

1. In the local case, these destinations are literally instances of the IRichBolt object (added during topology construction by the DistributedCluster) representing different executors of your bolt class.
2. In the distributed case, there should be a SenderBolt (added by the DistributedCluster) for each potential remote destination. We need to be careful that there is a SenderBolt for each remote worker’s bolt executors, such that hash-based partitioning will predictably send data to the same executor.

Please take a look at the SenderBolt which should make an HTTP POST call to send a JSON Tuple to the destination, and the WorkerServer’s “/pushdata” route. (For details on how routes work, please see the Spark Framework documentation at http://sparkjava.com/documentation.html; basically you are matching on requests starting with /pushdata/:stream, where the :stream represents an additional parameter, namely the stream name, that is included in the request path.)

4.3 Handling /runjob, and ensuring larger-than-memory support

When the worker receives a /runjob POST from the master, this is handled by the WorkerServer. This will parse the WorkerJob and its Config and Topology objects.

Incorporating the mapper and reducer objects. You should modify the MapBolt and ReduceBolt to instantiate objects of the mapClass and reduceClass, respectively, and call them as necessary.

Detecting end-of-stream with multiple senders. Observe that the MapBolt can call its job object’s map function at any point when it receives a tuple – unless the tuple is an end-of-stream (EOS), marking that there is no more data from the source. If it’s an EOS, the bolt should check (and record) how many such messages it has received. Only when it has received an EOS from each potential source executor (each executor on each node that can send a message) has the full end-of-stream been reached. This is a type of consensus among all of the executors. Given that the number of spout, map, and reduce executors per node is specified, you should be able to compute just how many EOS messages to wait for. The MapBolt should then propagate a single end-of-stream message to its next operator.

Similarly, ReduceBolt simply buffers state until it has received EOS messages from all of the upstream MapBolt executors. At this point, it can iterate over all keys, reduce their values, and output results. Once it is done with this, it can trigger end-of-stream.

Queuing up big data for “reduce”ing. The current ReduceBolt uses an in-memory hash table to store keys and sets of values. You should replace this with a BerkeleyDB “table”. Please remember to set up
your BerkeleyDB environment within your ~/store directory – not in /tmp or elsewhere! You should clear the database each time you run a job. Note that this is equivalent to the “sort” stage of MapReduce, except that we are using a B+ Tree instead of a sort algorithm to do the grouping.

4.4 Recording MapReduce Output
MapReduce typically produces large outputs. Currently, the same code routes the output of the ReduceBolt to a PrintBolt – which just writes output to the console. Instead, it should write to the file output.txt (as comma-delimited (key, value) lines) in the storage directory.

4.5 Status, updates, shutdown
It’s important to keep in mind that each worker on a node has a series of bolt executors that each have a reference to a shared Configuration (a set of key-value parameters) and a shared TopologyContext describing work state. Your worker will always use the MapBolt and ReduceBolt, with parameters specifying the map/reduce classes, the number of executors and worker nodes, and so on.

Periodic background thread. Once a job is launched, the worker should look for a Config parameter called master, which should be in the form IP:port (Example: 158.138.53.72:3000). As part of the creation of the LocalCluster, it should then create a thread that issues a GET /workerstatus to this IP and port once every 10 seconds; see 3.1 for the required parameters. The thread should be set up in such a way that it does not abort or crash if the connection fails for some reason (e.g., when the master is down, or has not started yet). Note that the background thread will also need to interface with the output.txt files above, in order to report on progress.

Shutdown. You should implement a /shutdown servlet for the Master and it should also trigger a shutdown of the known workers. (On the worker side, you will need to extend WorkerServer.)

5 WordCount
For testing, you should write a simple MapReduce job that implements WordCount. The name of the class should be edu.upenn.cis455.mapreduce.job.WordCount. Please submit the source code for this class along with your servlet and remaining code.

6 Testing and debugging
Recall that, for HW1, you assumed that the host name was localhost. This won't be a good assumption for this assignment anymore, particularly if you choose to deploy your workers on Amazon EC2, so you should change your servlet container to accept any other hostname.

For testing, you can run a master and a few workers locally in your VM (but on different port numbers, and with different storage directories).

A good way to start is to implement the master's status page first (you can test this by issuing a few /workerstatus GETs manually). Next, you could write a simple worker servlet that issues /workerstatus to the master, and test whether this works (by starting a few workers and checking that they show up on the master's status page). After this, you could implement the server's web form, and test it with dummy implementations on the workers, etc.
Caution: If you are not completely confident that your servlet container from HW1 works correctly, you may want to initially develop and test your servlets with Jetty. Once they work with Jetty, you can then run them in your own servlet container and fix any additional bugs that may be triggered there.

7 Requirements
Your solution must meet the following requirements (please read carefully!):

1. Your master servlet must be called `edu.upenn.cis455.mapreduce.MasterServlet`.
2. The format of the various GET and POST messages must be exactly as specified in the handout.
3. MapReduce jobs must implement the `Job` interface that came with your framework code in Bitbucket.
4. IP and port number of the master, and the location of the storage directory, must be read from Config parameters as described above, and may not be hard-coded.
5. Your submission must contain a) the entire source code for the servlets and updated classes, b) the source code for WordCount, c) an ant build script called `build.xml`, and d) a README file. The README file must contain 1) your full name and SEAS login name, 2) a description of features implemented, 3) any extra credit claimed, 4) a list of source files included, and 5) brief instructions on how to run the application on your application server. You must also complete all the yes/no questions.
6. When your submission is unpacked in the original VM image and the ant build script is run, your solution must compile correctly. Please test this before submitting! We will of course include a validator on the submission server. Please note this is totally advisory and is NOT part of the grading script.
7. Your ant script should instantiate two nodes on the local machine: a Master/worker node with the WorkerServer on port 8000, and a separate worker node on port 8001. The Master servlet should be assigned a port as in past homeworks. The two nodes will be launched in separate Java virtual machines.
8. Your servlet must display your full name and SEAS login name on the master's status page. We use this as a sanity check during grading.
9. Your solution must be submitted via OpenSubmit before 10:00pm EDT on the deadline. The project name should be `hw3`.
10. Your code must contain a reasonable amount of useful documentation.

Reminder: All the code you submit (other than any code we have provided) must have been written by you personally, and you may not collaborate with anyone else on this assignment. Copying code from the web is considered plagiarism.

8 Extra Credit (Due as part of the submission)

8.1 Dynamic deployment (+5%)
Extend your master and worker servlets such that the administrator can upload the `.class` file for a new job via the web interface (i.e., it doesn't have to previously exist in the workers' class paths).

8.2 Batched communication (+5%)
Extend `/pushdata` and your worker-to-worker communication to support propagation of multiple tuples at once, instead of one tuple at a time (still accounting for end-of-stream messages etc). Measure the running times before and after this change, and show that it produces speedups.
8.3 Multithreaded execution (+10%)
In the current implementation of DistributedCluster, each node has a single execution thread because events from the same stream need to be handled sequentially – which isn’t guaranteed for the multithreaded executor. To solve this, you’ll need to “split” the event queue in the distributed cluster, such that independent bolts can be run in different threads but each bolt’s tasks are handled in order.