Homework 3: MapReduce, Parallelism, and Stream Processing with Punctuation

Due November 22, 2021, at 10:00pm ET

1. Introduction

In this assignment, you will extend StormLite to a distributed framework, which essentially emulates Apache Storm – but in the process, also emulates MapReduce. As we hinted at in the lectures, MapReduce is really doing parallel (sharded) processing of streams of tuples, and Storm does the same thing with slightly different abstractions (e.g., streaming vs batched). Your solution will make use of REST services built in the Spark Java Framework (or your framework from Homework 1). You will test your framework by implementing a simple WordCount MapReduce program, which would normally run on Hadoop.

As usual, you should start by forking, then cloning the framework / skeleton code (project at https://bitbucket.org/upenn-cis555/555-hw3) from bitbucket. This includes a slightly expanded version of StormLite from HW2, with some MapReduce components.

2. Overview

At the end of this assignment, you will have a system consisting of two types of nodes: some number of workers and a single master. These nodes can be distributed, which means that the workers can run on different computers than the master and other workers. Coordination and data will be handled through the REST interfaces mentioned above.

As for emulating MapReduce on top of StormLite, note that StormLite gives you a basic abstraction for thinking about computation over units of data in streams. 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 they store the data that your MapReduce framework is working on; the master coordinates the workers and provides a user interface. The shuffle, which is typically done along with a sort in MapReduce after all data has been mapped, is instead done incrementally by the StormLite framework using hashing. (This will allow us to dispense key-value pairs in a streaming fashion and is useful for the vast majority of MapReduce algorithms, but of course it doesn’t allow us to directly emulate MapReduce algorithms that rely on sorting.)
A key issue is that, in a MapReduce job, the inputs end and the reduce can only start once the reducer has read all inputs. This requires that we implement a very simple distributed consensus algorithm (see 4.5.1 for details).

For our StormLite MapReduce executor, there are some key classes you will need to note. Much as we modeled StormLite off Apache Storm, these classes are heavily based on Apache Hadoop MapReduce. Recall that we spent some time in lecture discussing the different aspects of Hadoop MapReduce. Other than sort-based algorithms, you should be able to drop in Hadoop MapReduce code with minimal changes.

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

Additionally, we give you some “starter” spouts and bolts that will form the core of your implementation:

- **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 will need to modify and expand some of these, but do not change the existing interfaces as we will be dropping in our own map/reduce classes to test your system.

Both the worker and master nodes will communicate with one another via route handlers, registered with the Spark Framework or your own HW1 framework. That means that each worker and master will have a single **WorkerServer** or **MasterServer** running on separate ports, optionally on multiple machines.

We dive into their operation in more detail in later sections, but at a high level, each worker node will have a separate storage directory in which it keeps its local share of the data – as temp files and also, as relevant, as BerkeleyDB environments/tables. For instance, a worker might have a storage directory called storage/node1. 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 when architecting your storage strategy.

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. We will also assume that only one job is being run at a time.

**Changes to the StormLite API.** We’ve made some important, relatively minor, changes to StormLite to make it easier (1) for you to debug your distributed implementation, and (2) for you to look at extending StormLite to a full multithreaded execution model. Specifically, (1) the Tuple now includes an (optional) field for you to store the ID of the executor that produced it, so you can trace where it came from (all IStreamSources have a getExecutorId); (2) all methods for sending or writing take an (optional) source ID; (3) the spout nextTuple and bolt execute methods return a boolean indicating they still may produce more stream elements (if true) or they have reached end-of-stream (if false).

### 3. The master

A MasterServer will control and monitor job progress from the Web. The master presents a status page (/status) 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.

#### 3.1 Status updates from the workers

The MasterServer 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 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.
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 from a given IP:port combination.

3.2 The user-facing status (and job-launch) page

When a user requests the URL /status from the master server, the route should return a dynamic HTML document that contains (1) a section with status information about the workers, and (2) a web form for submitting jobs.

The status section should include one line for each active work (a worker is considered active if it has posted a /workerstatus within the last 30 seconds) and should be in the following comma-separated form. Make sure that your output matches exactly to be compatible with the autograder. *Again, for simplicity, you can assume we are only going to execute a single job at a time.*

```
0: port=8001, status=IDLE, job=None, keysRead=0, keysWritten=0, results=[]
1: port=8002, status=IDLE, job=Foo, keysRead=2, keysWritten=2, results=[(a, 1), (b, 1)]
```

The web form for submitting jobs should use the following form code:

```
<form method="POST" action="/submitjob">
    Job Name: <input type="text" name="jobname"/><br/>
    Class Name: <input type="text" name="classname"/><br/>
    Input Directory: <input type="text" name="input"/><br/>
    Output Directory: <input type="text" name="output"/><br/>
    Map Threads: <input type="text" name="map"/><br/>
    Reduce Threads: <input type="text" name="reduce"/><br/>
</form>
```

Parameters here are:

- 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 worker’s storage directory is set to ~/foo, the input should be read from ~/foo/bar)
- The output directory, also 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 via TestMapReduce)

When a user submits the web form on the /status page, the master node should produce the following two components, which will be used to instantiate a WorkerJob object.

1. Create a Topology using TopologyBuilder that, like in HW2M2, 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, MapBolt, ReduceBolt, and PrintBolt. As mentioned several times previously, the actual map and reduce classes are named in the Config (described next).

2. Populate a Config with 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 WorkerJob (“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). The Master should then POST a JSON representation of WorkerJob to a /definejob handler on each active worker. You may find Jackson’s ObjectMapper useful for marshalling/unmarshalling JSON.

Finally, the Master should send a blank POST to /runjob on each WorkerServer, to start the computation. Upon invoking this route, 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 have included a simple TestMapReduce class to show the basics of how this is done. You will 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.

Final (non-test) version. You should make sure your pom.xml has corresponding entries to launch the Master (execution goal master) and two workers (worker1 and worker2; you will have to add worker2 to the pom.xml). In our version:

- MasterServer takes the port it should listen on for REST requests.
- WorkerServer takes an argument for the port number, the address/port of the master (e.g., localhost:45555), and the storage directory (e.g., storage/node1).

You may also change this interface if you like, as long as all of the following commands work.

```bash
mvn exec:java # starts master
mvn exec:java@{master,worker1,worker2}
```

4 The Worker, in Detail

4.1 Execution of map/reduce in bolts

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 worker nodes. On the top worker, we have one executor of a FileSpout, which feeds its output stream to two local MapBolt executors, as well as two remote MapBolt executors running on the bottom worker, representing 4 possible destinations. Next, the MapBolts send data to two ReduceBolt executors (one on each worker). Finally, the ReduceBolts send to a single PrintBolt 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.

4.2 Sharded execution on StormLite

Let us consider (first, non-distributed) sharded execution across multiple StormLite executors. In the figure above, the FileSpouts produce outputs that need to be split across 4 map bolt executors (2 local, 2 remote, which we will discuss in Section 4.4). The circle with a 4 in it represents the StreamRouter that shards 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. To implement this, you will want to understand the StreamRouter logic well.

In the next step, the MapBolts’ 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 ReduceBolt running on the top node; for odd numbers, the data should go to the ReduceBolt running on the bottom node.

For certain other settings, you may want to use the shuffleGrouping (it just rotates among all destinations). If you want a single destination, such as the PrintBolt in our sample code, you can use a firstGrouping. In case it is useful, there is also an allGrouping which will broadcast to all nodes. (You can see how these work in BoltDeclarer and in the routers subpackage.)
4.3 Processing end-of-stream and changing execution phases

The big difference in abstraction between a stream engine and MapReduce is that there is an end to all MapReduce jobs. After the FileSpout has read all data, it should notify the MapBolt, which in turn should notify the ReduceBolt, which should then trigger the reduce computation! For this case, we create a special end-of-stream Tuple (see Tuple.isEndOfStream()). Note that the StreamRouter also has a special executeEndOfStream() function. End of streams are special because all destination executors need to see each end-of-stream message, i.e., it needs to be broadcast, regardless of the grouping policy.

4.4 Distributed communication

We have laid most of the groundwork 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-robinning, 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.
3. You should probably avoid the Round Robin policy for the distributed cases, as other policies will be better for keeping data local.

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. The use of Spark parameters should be familiar.

The WorkerServer’s /pushdata route should take the input tuple, serialize it back into an object, and send via the StreamRouter to the appropriate destinations. Think carefully about when to use StreamRouter’s execute method vs the alternative called executeLocally.

4.5 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.

4.5.1 Some key steps

Actually launching the topology. Right now, the /runjob handler does not launch the topology even though it is created – you will need to remedy this. We have separated out the handler into a RunJobRoute class to make it easier to update this.
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. You should use Java Reflection to do this. See http://www.rizzoweb.com/java/dynamicInstantiation.html for an example of how this can be done.

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. This represents a “vote” from one source executor that end-of-stream has been reached. If it is an EOS, the bolt should check (and record) how many such votes 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 simple example of consensus – much simpler than the algorithms we will talk about later in the course. 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. We have included a helper class called ConsensusTracker for recording the votes.

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 the worker’s provided storage directory – not in 555-hw3/database/ or elsewhere! You should clear the database each time you run a job. Note that, in terms of group values, this is equivalent to the “sort” stage of MapReduce, except that we are using BerkeleyDB’s B+ Tree instead of a sort algorithm.

4.6 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 a local file output.txt (as (key, value) pairs separated by newlines) in the appropriate output directory within the storage directory.

4.7 Status, updates, shutdown

It is 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 Config (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. One of the command-line parameters to WorkerServer should be the hostname/IP and port of the MasterServer, e.g., 158.138.53.72:80 or localhost:45555. As part of initialization, workers should create a thread that issues a GET /workerstatus to this address 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 route for the Master and it should also trigger a shutdown of the known workers. (On the worker side, you will similarly need to extend WorkerServer.)

### 5. Simple MapReduce Program: WordCount

For testing, you should write a simple MapReduce job that implements **WordCount** over your StormLite engine. The name of the class should be `edu.upenn.cis455.mapreduce.job.WordCount` and it should provide both map and reduce functions (we have supplied essentially a blank class). Note that we will be testing other MapReduce programs, so you should too.

Please submit the source code for the WordCount class along with your MapReduce implementation.

### 6. Testing and debugging

Recall that, for HW1, you assumed that the host name was localhost. Now you should think about having “real” hostnames in a distributed setting. For testing, you can run a master and a few workers locally in your Vagrant 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 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.

### 7. Tips and Tricks

Here are a few tips that might make your development a bit easier.

1. You should expect to run multiple instances of your WorkerServer (from the Terminal or via Maven). Each instance needs (1) its own IP port, and (2) its own directory for BerkeleyDB. The latter is necessary because there is a lock file that cannot be shared across processes.
2. One of your nodes should also have the **MasterServer**. You can decide whether your Master also does all of the tasks of a Worker or not. In your default `mvn exec:java` configuration, the Master should be on Port 45555, and the other nodes can be on 8001, 8002, ... as necessary.
3. As mentioned in the spec, workers should periodically call /workerstatus on the Master, and they should do this even if they aren't doing any work. The Master should use this to track the set of eligible workers.
4. If you set the MasterServer to handle both /workerstatus and /status, as in the instructions, this will allow you to have a data structure in the MasterServer that tracks all workers.

8. Requirements

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

1. Your master server must be called edu.upenn.cis455.mapreduce.master.MasterServer. This should include a main method that launches the Master server appropriately, monitoring /status and /workerstatus. You should make sure its command line arguments match those in the pom.xml; see #8 below.

2. Your workers must be called with edu.upenn.cis455.mapreduce.worker.WorkerServer. Here the main method should register handlers for the worker routes. You should make sure its command line arguments match those in your pom.xml; see #8 below.

3. The format of the various GET and POST messages must be exactly as specified in the handout.

4. MapReduce jobs must implement the Job interface that came with your framework code in Bitbucket. This is what your programmer is coding to!

5. For the master, it should take a local port to listen on. For workers, the local port, IP and port number of the master, and the location of the storage directory. All must be read from command-line parameters, and may not be hard-coded.

6. Your submission must contain a) the entire source code for the updated classes, b) the source code for WordCount, c) a pom.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, and 4) a list of source files you modified.

7. When your submission is unpacked and the mvn build script is run, your solution must compile correctly. Please test this before submitting! We will include a validator on the submission server.

8. Your pom.xml, via mvn exec:java, should have three different options to run three nodes on the local machine: “exec:java@master”, a Master node with the MasterServer on port 45555; “exec:java@worker1” and “exec:java@worker2”, which run two separate WorkerServer / worker nodes on ports 8001 and 8002. Each node will be launched in a separate Java virtual machine.

9. Your server implementation must display your full name and SEAS login name on the master's status page. We use this as a sanity check during grading.

10. Your solution must be submitted by 10:00pm ET on the deadline.

11. Your code must contain a reasonable amount of useful documentation and test cases.

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.
9. Extra Credit (Due as part of the submission)

9.1 Dynamic deployment (+5%)
Extend your master and worker servers 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).

9.2 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.

9.3 Multithreaded execution (+15%)
In the provided implementation of DistributedCluster, there is a single execution pool with one thread, because events from the same stream need to be handled sequentially and need to preserve order on the output stream -- if not, your end-of-stream markers won’t make sense. This is not guaranteed if we simply add more threads to the executor. To solve this, you will want to build a separate thread and queue for each spout and bolt executor.