CIS 5550: Internet & Web Systems

Spring 2023

Assignment 6: Analytics Engine

Due March 13, 2023, at 10:00pm EDT

1 Overview

For this assignment, you will build a simple distributed analytics engine called *Flame* that is loosely based on Apache Spark. Like Spark, Flame will be able to work with large data sets that are spread across several nodes (we'll call them RDDs, just like in Spark), and it will support some basic operations on these data sets, such as `flatMap`, `fold`, or `join`. However, to keep the assignment manageable Flame will obviously have only a tiny fraction of Spark's functionality; among other things, Flame will just have a small number of operations, and it will not support any data types other than `String`.

Flame won't maintain the data sets directly; instead, it will use the key-value store from HW4+5. Both the normal RDDs and the PairRDDs will be represented as tables in the key-value store. The rows in the normal RDDs will each have a unique key and a column called `value` that contains the value. The rows in the PairRDDs will each have a key k and potentially several columns with unique names; each such column will contain a value v_i, to represent a key-value pair (k, v_i). Thus, a single row can represent multiple pairs; this is important for operations like `join`, where we will need to access all of the values for a given key.

Like the key-value store from HW4+5, Flame will have a single master node and several worker nodes. The master will keep track of which workers are in the system, and it is also responsible for executing jobs, which are submitted to it as JAR files. When a new job is submitted, the master does roughly the following:

1. It adds the JAR file to its classpath, and then sends a copy to each worker node, which adds it to the classpath as well.
2. It loads an initial class, whose name the user specifies, and invokes the `run` function on this class. This is given an initial `FlameContext` object, as well as any command-line arguments the user has provided.
3. The `run` method can use the context to create some initial RDDs/PairRDDs (e.g., by invoking `parallelize`). When a new RDD is created, Flame loads the corresponding data into a new KVS table and then saves the name of this table in the RDD object it returns to the job.
4. When the job invokes a method (say, `flatMap`) on an RDD or PairRDD, Flame will send a message to each worker to tell the worker what to do. For instance, the message could tell the worker what operation is being invoked, which KVS table the data is coming from, which table the output should go to, and which range of keys the worker should process. If the method takes a lambda as an argument, the message will contain a serialized version of this as well. The worker will then run the necessary steps and report back to the master. Once all workers have reported back, the master will return a new RDD/PairRDD object to the caller.
5. When the job terminates, the master will send any output back to the client.

For communication, Flame will use your HTTP server from homeworks 1–3: both the master and the workers will run a local HTTP server, and messages will be sent as HTTP requests. In case you weren't

able to complete some of the earlier assignments or do not fully trust your implementations, we will provide reference implementations of both the HTTP server and the key-value store.

As in the earlier assignments, please do use Google, answers from the discussion group, the Java API reference and, if necessary, a good Java book to solve simple problems on your own. If none of these steps solve your problem, please post on the discussion group, and we will be happy to help! Feel free to use the Spark API reference for inspiration, but please keep in mind that Flame works a bit differently here and there, so this handout should be your primary source of information.

2 Requirements

Please start by downloading the HW6 package from http://cis5550.net/hw6.zip. This contains 1) a `README` file with a short questionnaire, 2) an Eclipse project definition (which you can ignore if you are not using Eclipse), an interface (`KVS`), 3) five classes in the `cis5550.tools` package that provide some useful building blocks, 4) three interfaces (`FlameContext`, `FlameRDD`, and `FlamePairRDD`, 5) three classes (`FlamePair`, `Master`, and `Worker`), 6) a tool for submitting jobs (`FlameSubmit`), and 7) a test job (`FlameTest`). It also contains simple implementation of HW3 (`lib/webserver.jar`) and HW5 (`lib/kvs.jar`). Your solution must meet the following requirements:

Command-line arguments: Both your `Master` and your `Worker` should accept two command-line parameters. The first is the port number on which each should run the HTTP server. In the case of `Worker`, the second parameter is the IP and port of the Flame master (Example: `1.2.3.4:8000`). In the case of `Master`, it is the IP and port of the *KVS* master.

Job submission: The master should accept POSTs to `/submit`, which must contain 1) a JAR file in the HTTP body, and 2) a query parameter called `class`, which must specify the name of a class in the JAR file. If the optional query parameters `arg1`, `arg2`, ..., are present, they contain URL-encoded arguments that should be passed to the job. Both the master and the workers should add the JAR file to their classpaths; then the master should invoke a method called `run` on the specified class, and pass it 1) an object that implements `FlameContext`, and 2) an array of strings with the (URL-decoded) arguments, if any. If the method returns normally, the POST request should return a 200 status and, in the body, any outputs that the method passed to `FlameContext.output()`. (If output() was invoked more than once, its argument should be concatenated.) If the method could not be invoked or was not found, the request should return a 400 status with an explanatory message; if the method threw an exception, the request should return a 500 status and a stack trace. (We have already implemented *some* of this functionality for you in `Master` and `Worker`.)

Functionality: Your solution should correctly implement the functions in `FlameContext`, `FlameRDD`, and `FlamePairRDD`; the expected behavior is documented in the interface files.

Parallelism: When your solution runs with multiple workers, each worker should be given roughly the same amount of work, and if a Flame worker is running on the same machine as a KVS worker, the former should work on the latter's keys, if possible. (We have already implemented this functionality for you in the `Partitioner` class.) *The workers should work on each operation in parallel!*

Encoding: The RDDs should be encoded as discussed in Section 1: the table that represents an RDD $R := \{v_1, v_2, \ldots, v_n\}$ should have n rows; row #i should have an arbitrary but unique key and a column called `value`, which should contain v_i. The keys should be at least somewhat random, so the table will be spread roughly evenly across the workers. The table that represents a PairRDD $P := \{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$ should have as many rows as there are unique keys in the RDD; the row for an RDD key k should have k as its KVS key, and a column with a unique name for each value v_j, where $(k, v_j) \in P$; the value in that column should be v_j.

User interface: The master should support a GET route for `/`, which should return an HTML page (content type `text/html` that has a table with a row for each Flame worker; each row should contain the IP address and the port number of the worker.

Compatibility: Your solution *must* work with the unmodified reference implementations in `lib/webserver.jar` and `lib/kvs.jar`, which we will use for grading.

Packaging: Your solution should be in a directory called `HW6`, which should contain 1) the `README` file from the HW6 package, with all the fields filled in; 2) the files `webserver.jar` and `kvs.jar` from the HW6 package in a subdirectory called `lib`; and 3) a subdirectory called `src` with *all* of your source code, including the files we provided, and in the directory structure Java expects (with subdirectories for packages). Your solution *must* compile without errors if you unpack `submit-hw6.zip` and, from the `HW6` folder within it, run `javac -cp lib/webserver.jar:lib/kvs.jar --source-path src src/cis5550/flame/Master.java` and `javac -cp lib/webserver.jar:lib/kvs.jar --source-path src src/cis5550/flame/Worker.java`. Please do try this before you submit! Submissions that fail this basic check will receive a zero.

3 Suggested approach

We suggest that you use the steps below to solve this assignment; however, feel free to use a different approach, or to ignore this section entirely. We will give credit for your solution, as long as it meets the requirements above.

As with most of the earlier assignments, the HW6 package comes with a small test suite. However, please keep in mind that *all* the features from Section 2 are required, not just the ones that are covered by the tests we provided!

Step #1: Create dummy implementations of the interfaces. A good way to start is to create dummy implementations of all the interfaces – perhaps a `FlameContextImpl`, a `FlameRDDImpl`, and a `FlamePairRDDImpl`. Methods without a return value can be left empty; the others can return `null` for now. With that, your code should at least compile (but it won't do anything useful yet!).

Step #2: Add the output mechanism. Next, implement the context's `output()` function. Jobs can call this at any time, but the outputs are only returned (in the body of the HTTP response to the `/submit` request) when the job terminates, so you'll need to store them somewhere in the meantime. At this point, the `output` test should work. If the test fails, feel free to have a look at the code in `cis5550.test` to see what it does; however, the test suite will use a compiled JAR file from the `tests` directory, so changes to the source code of a test case will not have any effect unless you also recompile the corresponding JAR file.

Step #3: Add simple RDDs. Next, implement the context's `parallelize()` method. This gets a list of strings, which it is supposed to load into an RDD. First, pick a fresh table name – for instance, you could have some kind of unique jobID (such as the time the job was started), followed by a sequence number. Then, use the `KVSClient` to upload the strings in the list to this table; the column name should be `value`, the value should be the string from the list, and the row key should be a random, unique string – for instance, you can use `Hasher` to hash the strings `1`, `2`, `3`, ..., and so on. Then create a new instance of `FlameRDDImpl`, store the table name somewhere in that instance, and then return it. Now implement the `collect()` method in `FlameRDDImpl`; this should simply scan the table (using `KVSClient.scan()`) and return a list with all the elements in the `value` column. Now the `collect` test should work.

Step #4: Build the framework for RDD operations. The actual RDD operations will be a somewhat bigger lift, but if you are careful, you will be able to reuse much of the code for the other operations. Typically, each operation takes some "input" RDD or PairRDD (the one on which it is invoked) and produces another "output" RDD or PairRDD with the results. So the workers will need to read some data from the corresponding "input" tables and write the results into a fresh "output" table. The rough workflow is as follows: 1) A method on `RDD` or `PairRDD` is invoked, usually with a lambda argument; 2) the master serializes the lambda and sends it to each worker (in a HTTP POST request), along with the names of the input and output tables, the hostname and port of the KVS master, and a range of keys this worker should work on; and 3) the master waits until all the workers have completed the operation, and then returns a new RDD or PairRDD that contains the name of the output table.

Because the process is very similar for each operation, it makes sense to implement a generic function (say, `invokeOperation`), perhaps in `FlameContextImpl`, that the various operations in `FlameRDDImpl` and `FlamePairRDDImpl` can invoke. The only bits that differ between operations are 1) the name of the operation (this can be a `String` argument) and 2) the lambda (this can be a `byte[]` argument). The function should first generate a fresh name for the output table (just like `parallelize()` did earlier). Then it should use the `Partitioner` class to find a good assignment of key ranges to Flame workers: invoke `addKVSWorker` for each KVS worker and give it the worker's address, the worker's ID as the start of the KVS key range for which this worker is responsible, and the next worker's ID as the end of that range. The only exception is the last worker, which is responsible for all keys higher than its ID *and* for all keys lower than the lowest ID of any worker, so `addKVSWorker` needs to be called twice; use `null` for the "highest" and "lowest" IDs, respectively. Then call `addFlameWorker` for each Flame worker, and finally call the partitioner's `assignPartitions()` method; this should return a vector of `Partition` objects that each contain a key range and an assigned Flame worker.

Next, send a HTTP request to each worker to tell it what operation to perform. You can use `HTTP.doRequest` to send messages, but be sure to send all the requests in parallel (from multiple threads); otherwise only one worker will be active at any given time. Then wait for all the responses to arrive (you can use `Thread.join()`) and check whether any requests failed or returned status codes other than 200. If so, your function should report this to the caller.

Step #5: Implement `flatMap()`. Now you can implement your first "real" RDD operation! It may be easiest to start with the `flatMap()` in `FlameRDDImpl`. Call the function you wrote in the previous step (`invokeOperation`) with the name of the operation (maybe `/rdd/flatMap`) and the serialized lambda as arguments; you can serialize the lambda using the `Serializer` class we have given you. This should cause the master to send POSTs to `/rdd/flatMap` on each worker. In `Worker`, define a POST route for `/rdd/flatMap`. In this route, decode the query parameters; this should give you the names of the input and output tables, the host and port of the KVS master, and the key range the worker should work on. Also, deserialize the lambda using `Serializer`; be sure to provide the name of the job's JAR file, so `Serializer` can load any extra `.class` files that may be needed. Then use `KVSClient` to scan the keys in this range, and (using the `op` method) invoke the lambda on the `value` column in each row; it will return either `null` or an `Iterable`. If it is the latter, iterate over the elements and put them into the output table in the KVS. As before, be sure to make the row keys unique, so, if the lambda returns the same value more than once, the values will all show up in separate rows instead of overwriting each other. Now the `flatmap` test case should work.

Step #6: Implement PairRDD's `collect()`. Next, let's start adding support for PairRDDs. Start by adding the `collect()` method; this should be roughly analogous to the method in `FlameRDDImpl`, except that here, a single `Row` of data can contain several key-value pairs – so you will need to iterate over the `columns()` of each `Row` and, for each value v you find in a column, output a pair (k, v), where k is the key of the `Row`.

Step #7: Implement `mapToPair()`. Now move on to `mapToPair()`. If you did the `invokeOperation` method correctly, the master side of this should be very simple – you simply call `invokeOperation` again, but this time with a different name (maybe `/rdd/mapToPair`). As before, you then need to add a POST route to the workers, where you decode the query parameters, deserialize the lambda, etc. (In fact, it may make sense to move this code to a function of its own, to avoid duplication!) Only the code for the actual operation will be different: here, you hand the string from the input table to the lambda and get back a pair (k, v); v will need to go into a row k in the output table, with a unique column name. Since the input table already has a unique key for each element, you can simply use that key (which is otherwise meaningless) as the column name. Now the `maptopair` test case should work.

Step #8: Implement `foldByKey()`. Finally, implement `foldByKey()`. On the master side, the main complication is that `foldByKey()` has an additional argument, which you need to include in the requests you send to the workers, perhaps as an additional query parameter. (Don't forget to urlencode it!) On the worker side, you can add a POST route as usual, which will scan the input table as before. For each `Row` the woker finds, it should initialize an "accumulator" with the zero element, then iterate over all the columns and call the lambda for each value (with the value and the current accumulator as arguments) to compute a new accumulator. Finally, it should insert a key-value pair into the output table, whose key is identical to the key in the input table, and whose value is the final value of the accumulator. In this case, the column name can be anything because there will be only one value for each key in the output table. Now the `foldbykey` test case should work.

Step #9: Try WordCount. As a final test, you can run the `FlameWordCount` job that is included with the framework code. You can provide several lines of text as arguments, and the output should be a list of the words in the text you provided, along with the number of times each appears.

Final step: Double-check the requirements! During initial development, it makes sense to focus on passing the tests; however, please keep in mind that the test cases are not meant to be exhaustive! Once you've finished the above steps, we strongly suggest that you read over the requirements from Section 2 again and make sure that you haven't overlooked anything.

3.1 Submitting your solution

When you are ready to submit your solution, please do the following:

1. Double-check that your solution is in a `HW6` directory within your GitHub repository, and that it has the required layout (see the end of Section 2). Verify that your code compiles! Section 2 specifies the command that the autograder will run.
2. Be sure to `git add` any new files you created, then commit your changes and push them to GitHub. Please *do not* add any generated files, such as `.class` files!
3. Submit your solution on Gradescope. Choose GitHub as the submission method, pick your CIS5550 repository, and select the `main` branch.
4. Wait for the autograder to run, inspect the results, and have a closer look at any unexpected failures.

We strongly recommend that you submit your solution at least a few hours before the deadline, so there will be enough time to resolve any problems.

4 Grading

Like each of the nine homework assignments, this assignment is worth 100 points, which will be allocated as shown in Table 1.

Points	Feature(s)	Test case(s)
5 points	Output mechanism	`output`
15 points	Parallelization; collecting RDDs	`collect`
25 points	`flatMap()` operation	`flatmap`
10 points	`mapToPair()`; collecting PairRDDs	`maptopair`
10 points	`foldByKey()` operation	`foldbykey`
30 points	Other tests that we have not published	
5 points	README file completed correctly	
100 points	Total without extra credit	

Table 1: Point distribution

5 Extra credit

If you like, you can implement the following additional features for some extra credit. If you do, please indicate this in the README file you submit! Notice that the parallelism requirement continues to apply; we will not give credit for solutions that scan or collect entire RDDs.

Implement `intersection` (+5 points): Add a new method called `intersection(R)` to `FlameRDDImpl` that takes another RDD `R` as its argument, and returns an RDD that contains *only* the elements that are present in *both* RDDs. The result should contain only one instance of each element x, even if both input RDDs contain multiple instances of x.

Implement sampling (+5 points): Add a new method called `sample(f)` to `FlameRDDImpl`, where `f` is a `double` that specifies the probability with which each element of the original RDD should be sampled.

Implement `groupBy` (+5 points): Add a method called `groupBy(L)` to `FlameRDDImpl` that takes a lambda `L` as its argument, which should accept a string and return a string. The method should apply this lambda to each element in the RDD and then return a PairRDD with elements $(k, v \ldots)$, where k is a string that `L` returned for at least one element in the original RDD, and $v \ldots$ is a comma-separated list of elements in the original RDD for which `L` returned k.