CIS 5550: Internet & Web Systems

Spring 2023

Assignment 7: Enhanced Analytics Engine

Due March 20, 2023, at 10:00pm EDT

1 Overview

For HW6, you built a simple distributed analytics engine called *Flame* that was loosely based on Apache Spark. At the end of HW6, the engine could do some basic operations (all the necessary ingredients for WordCount, the "Hello World" of analytics), but there were still lots of operations missing, including several that we will need for crawling, for indexing, and for computing PageRank. The goal of HW7 is to add enough operations to support these tasks. We still won't be anywhere near the full functionality of Apache Spark, so, in the later assignments and/or during the project, you may sometimes find that your initial approach won't work because it would require an operation that Flame does not support. In this situation, you will have to either change your approach slightly (we've verified that all of the components can be built with the operations we are adding here), or, alternatively, you can simply add more operations as needed.

Most of the new operations should be fairly straightforward, with two exceptions: `join` takes *two* input tables, and `fold` needs to do some master-side aggregation in addition to the steps that the workers are taking.

2 Requirements

Please start by downloading the HW7 package from http://cis5550.net/hw7.zip. This should basically contain the same things as the HW6 package, except that the test suite is different and that the `FlameRDD`, `FlamePairRDD`, and `FlameContext` interfaces have additional methods. Your solution must meet all of the requirements from HW6, with the following additions:

Functionality: Your solution should correctly implement all of the functions in `FlameContext`, `FlameRDD`, and `FlamePairRDD`, including the ones that have been added in the HW7 framework. The expected behavior is documented in the interface files.

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 `HW7`, which should contain 1) the `README` file from the HW7 package, with all the fields filled in; 2) the files `webserver.jar` and `kvs.jar` from the HW7 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-hw7.zip` and, from the `HW7` 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 (and the features from HW6) are required, not just the ones that are covered by the tests we provided!

Step #1: Copy over your HW6 code. You can simply cut and paste your code from HW6 into the HW7 project. Be sure to include the web server and KVS libraries (from the `lib` directory) as well!

Step #2: Create dummy implementations of the new methods. As with HW6, a good way to start is to create dummy implementations of all the new methods, so your code will at least compile (again). Methods without a return value can be left empty; the others can return `null` for now.

Step #3: Implement the master-local methods. Next, implement the `count()`, `saveAsTable()`, and `take()` methods. These should be easy because none of them involves the workers; in fact, the first two should be one-liners because `count()` can simply return the size of the table in the KVS, and `saveAsTable()` can simply rename the table in the KVS (and update the RDD to keep track of the name change). `KVSClient` has methods for both. For `take()`, you can simply scan the underlying table using `KVSClient` and return the values in the first n rows (or fewer if the RDD isn't large enough). At this point, the `count`, `save`, and `take` test cases should work.

Step #4: Implement `fromTable`. For `fromTable`, you can follow our usual pattern from HW6: in `FlameContextImpl`, call `invokeOperation` with a new route name, and then, in `Worker`, add a POST route with that name. The route should scan the range of the input table that the worker has been assigned, invoke the lambda for each row it finds, and store the resulting string (if any) in the output table. Now the `fromtable` test case should work.

Step #5: Implement `flatMap` and `flatMapToPair`. Next, implement the `flatMapToPair()` in `FlameRDDImpl` and the two additional map operations in `FlamePairRDDImpl`. These should be very similar to the `flatMap()` operation you implemented in `FlameRDDImpl` for HW6; the only difference is that the input and/or the outputs of the lambda are pairs. Remember, when you read a table that represents a `PairRDD`, each `Row` can represent multiple key-value pairs, all with the same key (`Row.key()`) but with different values, each in a separate column. When you write to a table that represents a `PairRDD`, you'll need to make sure that each new value gets a unique column name, otherwise you'll risk that some values are overwritten and dropped from the `PairRDD`. After this, the `map1`, `map2`, and `map3` test cases should work.

Step #6: Implement `distinct`. Now, implement `RDD.distinct()`. This can be very easy, with the following simple trick: put each value v from the input table into a row with key v (and column name `value`, as usual)! If the RDD contains duplicates, they will overwrite each other, and leave only a single instance at the end. With this, the `distinct` test case should work.

Step #7: Implement `join`. The `join()` operation is a bit special because, unlike all the other operations we will implement, it has *two* input tables. So you'll need to add an extra argument to the POST URL. The actual implementation is not difficult, however: all you need to do is scan one of the input tables and, for each row key k, look up the row with key k in the other input table. (Remember, the operation only joins pairs whose keys exist in *both* input tables!) The two rows might have more than one column (that is, they

might each represent more than one tuple, all with the same key), so you'll need a nested loop to produce all combinations of values from the first table and values from the second table. The only remaining complication is that, for each combination, you'll need to generate a unique column name for the output table – for instance, by concatenating hashes of the two column names from the input tables (which should already be unique within their respective rows), separated by some character that doesn't normally occur in column names. Now the `join` test case should work.

Step #8: Implement `fold`. This leaves the `fold()` operation. This is mostly analogous to the `foldByKey()` operation you had implemented for HW6, except that you now need to aggregate over an entire range of KVS keys on each worker, not just over the values in a given `Row`. The big difference is that, with `fold()`, there is no output table: instead, each worker should return the final value of its accumulator to the master, which should do one final round of aggregation to combine the accumulators of the various workers. With this, the final test case – `fold` – should work.

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 `HW7` 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.

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 from HW6 continues to apply; we will not give credit for solutions that scan or collect entire RDDs.

Implement `filter` (+5 points): Add a new method called `filter(b)` to `FlameRDDImpl`, where `b` is a boolean predicate. The method should return another RDD with only those elements from the original RDD on which the predicate evaluates to `true`.

Points	Feature(s)	Test case(s)
5 points	`RDD.count()`	`count`
5 points	`RDD.saveAsTable()`	`save`
5 points	`RDD.take()`	`take`
5 points	`RDD.fromTable()`	`fromtable`
5 points	`RDD.mapToPair()`	`map1`
5 points	`PairRDD.flatMap()`	`map2`
5 points	`PairRDD.flatMapToPair()`	`map3`
5 points	`RDD.distinct()`	`distinct`
15 points	`PairRDD.join()`	`join`
10 points	`RDD.fold()`	`fold`
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

Implement `mapPartitions` (+5 points): Add a new method called `mapPartitions(L)` to `FlameRDDImpl` that takes a lambda `L` that is given an `Iterator<String>` and returns another `Iterator<String>`. The lambda should be invoked once on each worker, with an iterator that contains the RDD elements that worker is working on (see `KVSClient.scan()`); the elements in the iterator that `L` returns should be stored in another RDD, which `mapPartitions` should return.

Implement `cogroup` (+5 points): Add a new method called `cogroup(R)` to `FlamePairRDDImpl`. This method should return a new `PairRDD` that contains, for each key `k` that exists in either the original RDD or in `R`, a pair `(k,"[X],[Y]")`, where `X` and `Y` are comma-separated lists of the values from the original RDD and from `R`, respectively. For instance, if the original RDD contains `(fruit,apple)` and `(fruit,banana)` and `R` contains `(fruit,cherry)`, `(fruit,date)` and `(fruit,fig)`, the result should contain a pair with key `fruit` and value `[apple,banana],[cherry,date,fig]`.