CIS 5550: Internet & Web Systems

Spring 2023

Assignment 1: Static Web Server

Due January 30, 2023, at 10:00pm EST

1 Overview

For this assignment, you will build a simple HTTP server that serves static files from a given directory. This server will eventually be used for the frontend of your search engine, and it is also going to be reused in several other homework assignemnts, e.g., as a communication layer for our Spark clone.

To keep the assignment manageable, we will be using an older version of the HTTP protocol (HTTP/1.1), and we will omit a lot of features you would find in a production-grade web server. Still, your server will work with any web browser, and, in HW3, we will deploy it in the Cloud.

In contrast to your intro programming courses, we will *not* provide framework code for assignments; the idea is to write the solution from scratch, just like you would for a "real" project after you graduate. We did prepare a small test suite, however; please see below for details. We will not give style points, and we do not require you to implement your own test cases, but of course we encourage you to do so.

As in HW0, please do try to solve simple problems on your own – please keep the Java API reference and a good Java book handy, please try googling unexpected errors or exceptions, and please do check the discussion group for existing solutions before you post a question. If you aren't sure how to implement a particular feature, please have a look at Section 3.

2 Requirements

Please start by downloading the HW1 package from http://cis5550.net/hw1.zip. This contains a `README` file with a short questionnaire, an Eclipse project definition (which you can ignore if you are not using Eclipse), and a small test suite. Your solution must meet the following requirements:

Invoking the server: The main class of your server should be `cis5550.webserver.Server`. This class should have a `main()` method that should accept two parameters: 1) a port number, and 2) the name of a directory that contains the files the server will return. It should be possible to run your server from the command line like this: `java cis5550.webserver.Server 8000 foo/bar`. If the server is not invoked with the correct number of arguments, it should print your full name (e.g., `Written by Andreas Haeberlen`) and exit.

Basic operation: When the server receives a `GET` request, it should append the URL to the directory that was given on the command line, and return the file with that name if it exists. For instance, if the directory is `foo/bar` and the request is `GET index.html HTTP/1.1`, your server should return `foo/bar/index.html`. When the server receives a `HEAD` request, it should handle it just like a `GET` request, except that it should not send a message body. The headers (including `Content-Length`) should be the same as if the request were a `GET`.

Error handling: Your server should send the following error codes:

- `400 Bad Request` if the method, URL, protocol, or `Host:` header are missing from the request;
- `403 Forbidden` if the requested file exists but is not readable;
- `404 Not Found` if the requested file does not exist;
- `405 Not Allowed` if the method is `POST` or `PUT`.
- `501 Not Implemented` if the method is something other than `GET`, `HEAD`, `POST`, or `PUT`; and
- `505 HTTP Version Not Supported` if the protocol is anything other than `HTTP/1.1`.

Headers: Your server should be able to handle persistent connections – that is, if a request has a `Content-Length` header to indicate that it contains a body, the server should use this header to determine where the current request ends and the next one begins. In its response, the server should include at least the `Content-Length`, `Server`, and `Content-Type` headers; the content type should be set to `image/jpeg` if the requested file has the extension `.jpg` or `.jpeg`; `text/plain` if it has the extension `.txt`; `text/html` if the extension is `.html`; and `application/octet-stream` otherwise.

Concurrency: Your server should be able to handle multiple requests concurrently. You can accomplish this by launching a fresh thread whenever the main thread has accepted a new connection, and by doing all the protocol logic in this new thread, while the main thread goes back and waits for the next connection.

Security: As a basic security feature, your server should return a 403 error code if the requested URL contains the string `..` (two dots), to prevent clients from downloading files outside the specified directory.

Miscellaneous: Your server should be able to handle any number of headers, in any order; keep in mind that header names are not case sensitive. It should be able to correctly return files of any size and type, including binary files. You should not assume that requests will arrive in any particular granularity, especially not all at once; they could arrive one byte at a time. Keep in mind that the line separator in HTTP is CRLF (a carriage return, followed by a line feed); your server should correctly parse requests that use this separator, and it should use the separator correctly in the responses. When your server is idle and there are no requests, it should not consume any CPU resources; please do not use "busy waiting" anywhere in your code. And finally, the server should have reasonable perfomance: with 1kB files, it should be able to handle individual requests in one second or less, and it should be able to handle at least ten requests per second. For any details that are not specified here, please see the HTTP/1.1 specification in RFC 2616.

Packaging: Your solution should be in a directory called `HW1`, which should contain 1) the `README` file from the HW1 package, with all the fields filled in, and 2) a subdirectory called `src` with *all* of your source code, in the directory structure Java expects (with subdirectories for packages). Your solution *must* compile without errors if you run `javac --source-path src src/cis5550/webserver/Server.java` fom within the `HW1` folder. 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.

The HW1 package contains a small test suite in `cis5550.test.HW1Test`, which includes a subset of the tests we will use for grading. Several of the steps below will reference these tests. To run the entire test suite, you can simply invoke `HW1Test`; to run only a few specific tests, you can provide the names of these tests as command-line arguments. Please keep in mind that *all* the features from Section 2 are required, not just the ones that are covered by the test suite!

Step #1: Open a server socket. Start by writing a simple program that opens a `ServerSocket` on the port that was specified on the command line, accepts a connection, prints whatever it receives, and then exits. For now, ignore multithreading and everything else. Then run your program, using port 8000 (which is non-privileged and a common port for testing HTTP), and invoke the `req` test from the test suite. You should see your server print a `GET` request, including a few headers. If this isn't working, double-check that your program actually opened the correct port, e.g., using the `netstat` tool.

Step #2: Add some logging. When debugging server code, it's critical to have some kind of log, so you can tell what the server did, or tried to do, in what order. A simple, Log4j-inspired logger (`cis5550.tools.Logger`) is included with your HW1 package. To use this, you can simply include something like the following in each class you write: `private static final Logger logger = Logger.getLogger(YourClassName.class)`. Then you can write log entries with something like `logger.info("Incoming connection from "+sock.getRemoteSocketAddress())`. There are methods for different log levels (`fatal`, `error`, `warn`, `info`, `debug`). You can then configure the behavior by creating a file called `log.properties` in the directory where you will run your solution. A like like `cis5550.webserver.Server = WARN` in that file tells the logger that, for the web server class, you want only warnings, errors, and fatal errors to be logged. (You can specify different log levels for each class.) And a line `log = someLogFile` tells the logger to write the messages to a file called `someLogFile`; if you do not include this line, the output will go to the terminal. If you do not have a `log.properties` file at all, logging will be disabled. Add a log entry when a connection is accepted, create a `log.properties` file, and make sure that the entry appears.

Step #3: Send a dummy response. Next, try sending a simple HTTP response once the connection is open. You can do this by wrapping a `PrintWriter` around the connection's `OutputStream`. Try sending `HTTP/1.1 200 OK`, followed by a CRLF, then `Content-Type: text/plain` and a CRLF, `Server: XXX` and a CRLF (where `XXX` is the name of your server), `Content-Length: 12` and two (!) CRLFs (to indicate the end of the headers), and then the string `Hello World!`. Close the connection. The `req` test should now pass, and you should be able to see the string the server is sending if you point any local browser on your machine to `http://localhost:8000/`.

Step #4: Parsing the headers. Now, try to parse the request a bit more carefully. Since we will want to support binary uploads later, it is best to initially read the request as bytes, and to convert the headers to strings later on. Keep reading the data as it arrives, and look for the double CRLF (bytes 13, 10, 13, 10), which indicates the end of the headers. Once you see this, take the bytes you've received up to that point and convert them to strings – for instance, by wrapping the bytes in a `ByteArrayInputStream`, then in an `InputStreamReader`, and finally in a `BufferedReader`. When you read this line by line, the first line should be the request line, and the following lines should be the headers. Look for the relevant headers (in this assignment, only `Content-Length`) and parse them if present. (For now, let's assume that there is no message body, so the content length will be zero.) Then go back to reading the next request until the connection is closed, instead of just terminating your server. If you do this correctly, the `persist` test should now pass.

Step #5: Handling error conditions. Now check for the error conditions (400, 405, 501, 505) and return the appropriate HTTP error code when they occur. Notice that most servers return a body with an error condition, which is what most browsers display when they occur; you may want to do the same, even if you just repeat the error ("404 Not Found") from the status line in the body. If you do this correctly, the relevant `err` tests should pass.

Step #6: Reading message bodies. Now start paying attention to `Content-Length`. If it is present and greater than zero, read that many bytes after the double CRLF, and only then go back to reading the next request. (For now, you can discard this data; it will become important in HW2.) Once you do this correctly,

the `withbody` test should pass. If the header is not present, keep reading until the client closes its half of the TCP connection; keep in mind that the server's half is still open at that point. If you do this correctly, the `connclose` test should pass.

Step #7: Reading files. Next, start sending actual data instead of a dummy response. Take the path from the command line, concatenate it with a forward slash and the URL from the request, and then try to read the corresponding file. Return error codes 403 and 404 if the file isn't readable or doesn't exist; otherwise generate the required headers and send the file data back. Also, don't forget to check for the `..` in the URL! To help with debugging, we suggest that you add log entries that say which URL was requested, and which file the server tried to read. Be sure to use `Stream` classes and not `Writer` classes for the file part, to avoid issues with binary files; if you use a `Writer` to send the headers, remember to flush it before you start sending the file, otherwise the headers may be sent only partially, or not at all. Now the `text` and `binary` tests should pass.

Step #8: Concurrency. Finally, add the thread pool. Instead of having a *single* infinite loop that accepts connections, reads requests, and generates responses, you should now have *two* infinite loops: one (for the original thread) that accepts connections and puts the `Socket` objects into a `BlockingQueue`, and one (for the worker threads) that takes sockets from the queue, reads the request, and generates responses as before. Create the required number of threads and have them execute the second loop, and then have the original thread execute the first loop. Now the `multi` test should pass.

Step #9: Performance. Now, run the `stress` test. This will make a lot of requests; if it fails, check whether your workers are getting a new request after finishing the previous one, and whether you are closing all connections once the server is finished with them. Once this passes, run the `stress2` test. This does the same thing, but with persistent connections and multiple threads. This test will also output the number of requests per second; if you disable logging (by deleting or renaming the `log.properties` file), this number should be well above 1,000. Depending on the computer you have, it can even reach 10,000 or more. This number isn't important for grading, but your server will be used for future assignments and for the project, so it is a good idea to find and fix any performance issues right away.

Final step: Double-check the requirements! During initial development, it makes sense to focus on passing the test cases; 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 `HW1` 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.

Points	Feature(s)	Test case(s)
15 points	Single request	`req`
10 points	Persistent connections	`persist`
2 points	Bad Request error condition	`err400`
2 points	Not Found error condition	`err404`
2 points	Method Not Allowed error condition	`err405`
2 points	Not Implemented error condition	`err501`
2 points	Version Not Supported error condition	`err505`
5 points	GET requests with bodies	`withbody`
5 points	Returning text files	`text`
5 points	Returning binary files	`binary`
7 points	Multiple concurrent requests	`multi`
5 points	Stress test	`stress`
3 points	Stress test with persistence	`stress2`
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

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!

Thread pool (+5 points): Refactor your server to use a *thread pool*, instead of just starting a new thread for each request. There should be one thread that accepts new connections and puts them into a `BlockingQueue`, as well as a fixed number of worker threads that pull connections out of this queue and handle them. The number of worker threads should be based on a constant called `NUM_WORKERS` in your `Server` class, which should be set to 100 in the solution you submit.

Conditional requests (+5 points): Add support for the `If-Modified-Since` header. You can find a description of this header in Section 14.25 of RFC 2616.

Range requests (+5 points): Add support for the `Range` header. You can find a description of this header in RFC 7233. You only need to support `bytes` ranges.

CIS 5550: Internet & Web Systems

Spring 2023

Assignment 2: Dynamic Web Server

Due February 6, 2023, at 10:00pm EST

1 Overview

For this assignment, you will extend your static HTTP server from HW1 to support dynamic content and routes. The API will be based on the API from Spark Framework, a real-world framework creating web applications in Java. Your solution will be used as a foundation for several of the other homework assignments, as well as for the frontend of your search engine at the end.

Spark Framework is a great way to write very compact applications. The following is a complete application that should run on your solution at the end:

```
import static cis5550.webserver.Server.*;

class HelloWorldApp {
  public static void main(String args[]) {
    port(8080);
    get("/hello/:name", (req,res) ->
      { return "Hello "+req.params("name"); } );
  }
}
```

When this code runs and you open `http://localhost:8080/hello/Bob` in a browser window, you should see the response "Hello Bob".

To keep the assignment manageable, we will implement a small subset of Spark Framework's functionality: some of the functions for configuring the server and creating routes, the Request and Response objects, and the actual route-matching code. We will skip most of the complicated features, such as filters or transformers. Also, we will provide some of the boilerplate code for you (specifically, the `RequestImpl` object), since this is tedious to write and you wouldn't learn much from it.

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. The Spark Framework documentation may be useful if you have questions about the API. If none of these steps solve your problem, please post on the discussion group, and we will be happy to help!

2 Requirements

Please start by downloading the HW2 package from http://cis5550.net/hw2.zip. This contains a `README` file with a short questionnaire, an Eclipse project definition (which you can ignore if you are not using Eclipse), a small test suite, three interfaces (`Request`, `Response`, and `Route`), and an implementation of the first interface (`RequestImpl`). Your solution must meet the following requirements:

Server API: You should have a class `cis5550.webserver.Server`, as in HW1. This class should have the following *static* methods:

- `port(N)` should tell the HTTP server to run on port `N`;
- `get(p,L), put(p,L), post(p,L)` should create `GET`, `PUT`, and `POST` routes, respectively; `p` is a path pattern (`String`), and `L` is a lambda that accepts a pair `(req,res)`, where `req` is an object of type `Request` and `res` is an object of type `Response`, and returns an `Object`.

The server should also contain a public static class `staticFiles` with static method `location(P)`. When this method is called with a string `P`, the server should start serving static files from path `P`, just as your HW1 solution did. The web server should start the first time `get`, `put`, or `post` is called. You may assume that `port()` is called at most once, and that it would be the first call. If `port()` is not called, the server should run on port 80.

Route API: The two arguments that are passed to the route handler should implement all of the functions from the `Request` and `Response` interfaces, respectively. The expected behavior is documented in `Request.java` and `Response.java`. Notice that an implementation of the `Request` object has already been provided as `RequestImpl`.

Error handling: If a route throws any exception and `write()` has not been called, your solution should return a `500 Internal Server Error` response. If `write()` has been called, it should simply close the connection.

Miscellaneous: The intention is that you will reuse and extend your `Server` implementation from HW1; you can simply copy over your code to the `HW2` project, and then remove the `main()` method. (Recall that web applications will have their own `main()` method, and interact with your server through calls like `get()` and `post()`.) As in HW1, your solution should be able to handle any headers in any order, it should send valid, correctly formatted HTTP responses, and it should handle multiple requests concurrently.

Packaging: Your solution should be in a directory called `HW2`, which should contain 1) the `README` file from the HW2 package, with all the fields filled in, and 2) 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-hw2.zip` and, from the `HW2` folder within it, run `javac --source-path src src/cis5550/webserver/Server.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.

The HW2 package contains a small test suite in `cis5550.test.HW2TestClient` and `cis5550.test.HW2TestSever`, which includes a subset of the tests we will use for grading. Several of the steps below will reference these tests. To run tests, first run `HW2TestServer` (which contains a `main()` method that defines some routes) and then, without terminating the server, run `HW2TestClient` – e.g., in a separate terminal window. As in HW1, you can run all the tests by invoking `HW2TestClient` without command-line arguments, or you can specify a list of tests to run. However, please keep in mind that *all* the features from Section 2 are required, not just the ones that are covered by the test suite!

Step #1: Copy over your HW1 code. You can simply cut and paste your `Server` class into the HW2 project. If you defined additional classes for HW1, please do not forget to copy and paste these as well.

Step #2: Add the static methods. Add a `public static class staticFiles` *within* your `Server` class, and add a `public static` method `location(String s)` within it. Also, in your main server class, add three `public static void` methods called `get`, `post`, and `put`, which should each take a `String` and an instance of `Route`, and a `public static void` method `port()`, which should take an `int`. Now `cis5550.test.HW2TestSever` should compile, and not complain about missing methods.

Step #3: Launch the instance on demand. Add to the `Server` class 1) a `static` field that can contain an instance of `Server` and is `null` initially, and 2) a `static` flag that is `false` initially. In each of the server API methods, check whether the field is `null`, and, if so, create a `Server` instance and put it into the field. Additionally, in methods other than `port`, check whether the flag is `false`, and if so, set it to true and launch a thread that executes a `run()` method in the server. (This is needed because the server will be handling requests in the background, while the application is free to do other things.) Rename your `main()` method from HW1 to `run()`, but get rid of the arguments that specified the port number and the folder location; instead, make the `port()` and `location()` methods set fields in the server instance, and use the values from these fields. Notice that `location()` may not be called at all; your solution should start serving static files only if and when this method is called. The `static` test in `HW2TestClient` should now pass. (Remember that `HW2TestServer` needs to be running while you use the test client; the client will print a message to remind you.)

Step #4: Add a routing table. Add some kind of data structure that can store routes. For each route, you will need to remember three things: 1) the method (`GET`, `PUT`, or `POST`), 2) the path pattern, and 3) the handler, which will be an instance of `Route`. Change the `get()`, `put()`, and `post()` methods you created in Step #2 to add an entry to this data structure.

Step #5: Implement `Response`. Next, write an implementation of the `Response` interface – say, `ResponseImpl`. Most of the methods simply set values of the response, so you can have fields for the status code, the body, and the headers, and update them from the relevant methods. Keep in mind that the body may be binary, so the field for it should be a `byte[]` and not a `String`; the `body()` method can convert the string to bytes by calling `getBytes()`. Leave the `write()` method empty for now.

Step #6: Handle dynamic requests. In your `run()` method, find the point after the request has been decoded but before you check for static files. At this point, add code that iterates over the routes and looks for an entry that 1) matches the method from the current request, and 2) has a path pattern that matches the URL in the request. (If there are multiple matches, you can use any of them.) If no entries match, look for static files as before – but if a match is found, instantiate `Request` and `Response` objects (leave the named and query parameter `Maps` as `null` for now), and pass them to the `handle` method on the `Route`. If this throws an exception, return the 500 response as required. If not, set the `Content-Length` header and then write out the headers; then write out the body, if there is any. (Have a look at the `Response` interface for more information about what to return in the body.) Remember to skip the static-file check when a match was found, so you don't accidentally send back *both* dynamic content *and* a static file! Now you can try out your solution with the test suite; the `get`, `post`, `puthdr`, and `error` tests should all pass.

Step #7: Implement `write()`. Now, add the `write()` method. Notice that this is supposed to write out the headers the first time it is called, so this may involve a bit of refactoring. `RequestImpl` has a reference to your `Server`, so you can call methods on that if it is useful. Be sure to keep track of whether the headers have already been written, so you don't accidentally write them out twice. Now the `write` test case should pass. Also, re-run the earlier tests to make sure that the refactoring did not break anything.

Step #8: Implement path matching. So far, your code only invokes routes when the URL matches the route's path exactly. Now it is time to add support for path parameters. Write a function that decides whether a given URL matches a given path pattern: split both URL and path pattern by forward slashes (`/`), then check whether a) both have an equal number of pieces, and b) the pieces are either identical or the path pattern piece is a named parameter, i.e., starts with a colon (`:`). It is useful to generate a `Map` of the path parameters along the way, to avoid code duplication. Then you can simply pass this `Map` to the `Request` object, so it can return it from `params()`. Now the `pathpar` test should pass.

Step #9: Add query parameters. Now add code that looks for query parameters in both relevant places (after the `?` in the URL, and in the body if the content type has the value `application/x-www-form-urlencoded`). If any are found, split them by the ampersand (`&`) and the pieces into name-value pairs by the equals sign (=)), URL-decode the names and values (by passing them through `java.net.URLDecoder.decode()`), then put them into a `Map` and pass it to the `Request` object. Now the `qparam` check should pass as well.

Final step: Double-check the requirements! During initial development, it makes sense to focus on passing the test cases; 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 `HW2` 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!

Multiple hosts (+5 points): Add a public static `host()` method. An application can call `host(H)` to define routes and a static-file location for a "virtual host" `H`. When an application calls `get()`, `put()`, `post()`, or `staticFiles.location()`, these calls should apply *only* to requests whose `Host:` header is set to the argument of the most recent `host()` call (plus optionally a port number, which you

Points	Feature(s)	Test case(s)
5 points	Requesting a static file	`static`
15 points	GET request via a route	`get`
5 points	POST request via a route, with a body	`post`
5 points	PUT request via a route, with extra headers	`puthdr`
10 points	Request with path parameters	`pathpar`
5 points	Exception within a route	`error`
10 points	Using `write()`	`write`
10 points	Query parameters in URL and body	`qparam`
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

should ignore); any calls before the first `host()` call should apply by default – that is, to requests whose `Host:` header does not match the argument of any `host()` calls. For instance, if the application calls

```
get("/foo", (req,res) -> { return "A"; });
host("bar.com"));
get("/bar", (req,res) -> { return "B"; });
host("xyz.com"));
get("/bar", (req,res) -> { return "C"; });
```

then a GET request for `/bar` should return `B` if it has a `Host: bar.com:1234` header, `C` if it has a `Host: xyz.com` header, and a `404 Not Found` if it has a `Host: blubb.com` header. A request for `/foo` should return `A` with a `Host: blubb.com` header, but a `404 Not Found` with `Host: bar.com:1234` or `Host: xyz.com`.

Redirection (+5 points): Implement the `redirect()` method to the response object. If the application calls `redirect(U,c)`, the server should redirect the client to URL `U`, using response code `c` (which can be 301, 302, 303, 307, or 308).

Filters (+5 points): Add two public static methods `before()` and `after()` that each take a lambda that accepts `(request,response)` and returns `void`. The lambda(s) that are provided to `before()` should be called *before* a route is used; the lambda(s) that are provided to `after()` should be called after the route returns. In addition, implement the `halt(X,Y)` in the response object. When this is called while a request is at the `before()` stage, the should return an error response, with integer X as the status code and string Y as the reason phrase (Example: `halt(401, "Not allowed")`), and the relevant route should *not* be invoked.

CIS 5550: Internet & Web Systems

Spring 2023

Assignment 3: HTTPS Server on the Cloud

Due February 13, 2023, at 10:00pm EST

1 Overview

For this assignment, you will extend your dynamic HTTP server from HW2 with support for sessions and HTTPS, and you will deploy it on a real cloud platform, with a valid, CA-signed TLS certificate.

Implementation-wise, this assignment should actually be quite a bit easier than the first two assignments. The main challenge is in the AWS deployment, but we've included detailed steps for this in Section 3 below. Please do not try the deployment step at the last minute! If something unexpected happens, it may take a bit of time to find the problem, and if you need to update your DNS record, the changes can take hours to propagate everywhere, and there is no way you or we can speed this up.

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. The Spark Framework documentation may be useful if you have questions about the API. If none of these steps solve your problem, please post on the discussion group, and we will be happy to help!

2 Requirements

Please start by downloading the HW3 package from http://cis5550.net/hw3.zip. This contains a `README` file with a short questionnaire, an Eclipse project definition (which you can ignore if you are not using Eclipse), a small test suite, and four interfaces (`Request`, `Response`, `Session`, and `Route`). Notice that the `Request` interface is a bit different from the one that was included with HW2. Your solution must meet the following requirements:

HTTPS support: There should be an additional static method called `securePort` in your `Server` class, which should accept a port number. If this method is called before the first route is defined, your server should accept requests *both* via HTTP *and* via HTTPS, on different ports; the port number for HTTPS is the argument to `securePort`, and the port number for HTTP is the argument to `port`, if it has been called, or 80 otherwise. If `securePort` has been called, your server should load the TLS certificate from a file called `keystore.jks`, using a default keystore password of `secret`, which (for HW3 purposes) you may hardcode in your solution.

Sessions: You should implement the additional `session()` method in the `Request` interface. When this method is first called for a given request, your server should check whether the request included a cookie with the name `SessionID` *and* the value of that cookie is currently associated with a `Session` object. If such an object is found, the method should return it; otherwise, it should 1) pick a fresh, random session ID of at least 120 bits, 2) instantiate a new `Session` object, 3) associate this object with the chosen session ID, 4) add a `Set-Cookie` header to the response that sets the `SessionID` cookie to the chosen session ID, and 5) return the `Session` object. If the method is called again while the server is still

handling the same request, it should return the same `Session` object. When the method is never called, no session objects or `SessionID` cookies should be created.

Session expiration: Each `Session` object should have a maximum active interval t_{max}, which can be set with the `maxActiveInterval` method and should be 300 seconds by default. When, for a period t_{max}, the server has not received any requests with the session ID that is associated with the object, the object should "expire", and the server should behave as if the object did not exist. You do not need to delete session objects immediately when they expire, but you should delete expired objects periodically; when the server receives no more requests, all existing `Session` objects should be deleted eventually.

AWS deployment: You should deploy your server on an EC2 instance, such that it can be reached by `https://<yourname>.cis5550.net/`, where `<yourname>` is the login name of your SEAS account. (For instance, my own login name is `ahae`, so I would deploy my server at `ahae.cis5550.net`.) The server's main page should say "Hello World - this is <yourname>", with your own full name instead of "<yourname>". You should obtain a TLS certificate from Let's Encrypt for your domain name, and use it in this deployment. Once the server is deployed, you should open `https://<yourname>.cis5550.net/` in a standard web browser (Firefox, Safari, Chrome, Edge, or something comparable), make a screenshot, and include this screenshot as a file called `screenshot.png` or `screenshot.jpg` (depending on the format of the screenshot) with your submission.

Miscellaneous: The intention is that you will reuse and extend your `Server` implementation from HW2; you can simply copy over your code (except for the four interfaces that were included in the HW3 package) to the `HW3` project. All the original requirements from HW2 continue to apply.

Packaging: Your solution should be in a directory called `HW3`, which should contain 1) the `README` file from the HW3 package, with all the fields filled in; 2) the screenshot, as described above; 3) your `keystore.jks` file with the Let's Encrypt certificate (which you will have to download from your EC2 instance); and 4) 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-hw3.zip` and, from the `HW3` folder within it, run `javac --source-path src src/cis5550/webserver/Server.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.

The HW3 package contains a small test suite in `cis5550.test.HW3TestClient` and `cis5550.test.HW3TestSever`, which includes a subset of the tests we will use for grading. Several of the steps below will reference these tests. To run tests, first run `HW3TestServer` (which contains a `main()` method that defines some routes) and then, without terminating the server, run `HW3TestClient` – e.g., in a separate terminal window. As in HW2, you can run all the tests by invoking `HW3TestClient` without command-line arguments, or you can specify a list of tests to run. However, please keep in mind that *all* the features from Section 2 are required, not just the ones that are covered by the test suite!

Step #1: Copy over your HW2 code. You can simply cut and paste your `Server` class, and any other classes you may have created for HW2, into the HW3 project. Please remember to keep the interfaces that came with the HW3 package, though! Since the `session()` method is new, you will need to add an implementation for it in order to compile the code, but you can simply return `null` for now.

Step #2: Create a self-signed certificate. HTTPS connections require TLS, and, in order to use TLS, your server will need a certificate. For testing, you may want to create a self-signed certificate. You can do this using the `keytool` utility, with is included in the JDK. An example command would be (on a single line):

```
keytool -genkeypair -keyalg RSA -alias selfsigned -keystore keystore.jks
        -storepass secret -dname "CN=Blah, OU=Blubb, O=Foo, L=Bar, C=US"
```

This will create a file called `keystore.jks` in the local directory.

Step #3: Open a TLS socket. As a first step, implement the `securePort` method, and then add code to replace the existing server socket with a TLS server socket. You can use code roughly like the following:

```
import javax.net.ssl.*;
import java.security.*;

String pwd = "secret";
KeyStore keyStore = KeyStore.getInstance("JKS");
keyStore.load(new FileInputStream("keystore.jks"), pwd.toCharArray());
KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance("SunX509");
keyManagerFactory.init(keyStore, pwd.toCharArray());
SSLContext sslContext = SSLContext.getInstance("TLS");
sslContext.init(keyManagerFactory.getKeyManagers(), null, null);
ServerSocketFactory factory = sslContext.getServerSocketFactory();
ServerSocket serverSocketTLS = factory.createServerSocket(securePortNo);
```

You should now be able to use `serverSocketTLS` instead of your existing server socket. (This will disable normal HTTP requests for now, but we'll fix this shortly.) Try running a simple test application on your server, like

```
securePort(8443);
get("/", (req,res) -> { return "Hello World!"; });
```

and then open `https://localhost:8443/` in your web browser. (Be sure to include the `https` part!) If you are using the self-signed certificate from Step #2, which is not signed by any CA, you will see lots of warnings in the browser, but it should be possible to override them. For instance, in Safari you would see a warning that says that "This Connection Is Not Private", but you can click on "Details", then on "visit this website", and then confirm by clicking on the "Visit Website" button. At the end, you should see the "Hello World!" message from the test application, and you should be able to view the information in your self-signed certificate by clicking on the little lock icon in Safari's address bar (or whatever equivalent your favorite browser has to this). If the lock icon is missing or you do not get any warnings, something is wrong; chances are that you are still using HTTP. Also, at this point, the `https` test case should pass.

Step #4: Add back HTTP support. In order to support both HTTP and HTTPS, which use different ports, your server will need to listen to two different server sockets. A simple way to do that is to use two separate threads. If you haven't already, you should factor out your server loop into a separate method, say `serverLoop(s)`, where `s` is a `ServerSocket`; at that point, you can simply open both a normal server socket and a TLS server socket, and then launch two separate threads that each invoke this method with one of the two server sockets. At this point, both the `http` and `https` test cases should pass.

Step #5: Add a session object. Next, create a class – perhaps called `SessionImpl` – that implements the `Session` interface. This class doesn't have to do much; all it needs to do is store the various bits of information (the session ID, creation time and last-accessed time, max active interval, and key-value pairs) and implement the various getter and setter methods in the interface.

Step #6: Add session handling. Now you can use this class to add support for sessions. There are four places in the server that need changes. First, you need a data structure that stores the sessions by session ID – probably some kind of `Map`. Second, you'll need to parse the `Cookie` header(s) while reading the

request, and extract the cookie with the name `SessionID`, if it exists; if it does, *and* your `Map` contains a session for that ID, update that session's last-accessed time and make sure that the `session()` method will return it when called. Third, when `session()` is called and there is *not* already a session, you'll need to create a new object with a fresh session ID. Keep in mind that the session ID should be random and have at least 120 bits; for instance, you can pick a set of 64 characters (maybe lower and upper case letters, digits, and two other characters) and then draw 20 of them. Finally, when writing out the headers, you'll need to add a `Set-Cookie` header whenever you've created a new `SessionImpl` object, to send back the corresponding `SessionID` cookie to the user. With this, you should be able to implement the two `attribute` methods from the `Session` interface, and both the `sessionid` and `permanent` test cases should pass.

Step #7: Add session expiration. The final step is to add a way to expire session objects. This requires two steps. First, you'll need a way to periodically expire old sessions that have not been accessed recently; you can do this by launching another thread that sleeps for a few seconds, removes any session objects whose last-accessed timestamp is too old, and then repeats. Notice, however, that this could cause the server to slightly overshoot the lifetime of a session. To fix that, the second step is to *also* check the last-accessed timestamp before updating it, and to simply ignore the object if it has already expired but has not been deleted yet. At this point, both the `expire` test case should pass.

Step #8: Implement a test server. Write a little server that 1) calls `securePort(443)` to set the HTTPS port, and 2) defines a GET route for `/` that returns the required message from Section 2. Test this server locally (`https://localhost/`) with your self-signed certificate, to make sure that it displays the message correctly. If you cannot bind to port 443 on your local machine, you can use port 8443 for testing (in this case, open `https://localhost:8443/` instead), but please don't forget to use 443 in the final version that will be deployed on EC2.

Step #9: Launch an EC2 instance. Log into the Amazon Web Services console and choose EC2 (by clicking on "Services", then on "Compute", and then on "EC2"). Click on "Launch Instance", and then do the following:

- Under "Application and OS Images", choose the default Amazon Linux AMI/.
- Under "Instance type", choose one of the smallest/cheapest instances – say, a `t2.micro` instance. This should be enough for our purposes.
- Under "Key Pair", click on "Create new key pair", choose a descriptive name (perhaps "CIS5550"), and pick RSA and the `.pem` format. Hit "Create Key Pair", which should cause your browser to download a `.pem` file.
- Under "Network settings", make sure that the following *three* options are checked: "Allow SSH traffic from Anywhere", "Allow HTTPS traffic from the internet", and "Allow HTTP traffic from the internet". Please double-check this – if you get this step wrong, chances are that you won't be able to connect to your server later on.
- Under "Configure storage", you can keep the default, which is probably an 8GB gp2 root volume.

In the summary bar on the right, double-check that the number of instances is 1, and then hit the orange "Launch instance" button. Wait a moment, until (hopefully) AWS reports success. Go back to the "Instances" tab in EC2, and find the instance you just launched. When you click on its instance ID, you should see an "Instance summary" that shows, among lots of other things, its public IPv4 address. Write this down. (Do *not* confuse this with the private IPv4 address, which probably starts with `172.` or `10.` and won't be useful for our purposes!)

Step #10: Log into your EC2 instance. Use your `ssh` client to log into the instance you just created. The concrete steps depend a bit on the operating system you are using. Under macOS, you can simply open a terminal and run the command `ssh ec2-user@1.2.3.4 -i CIS5550.pem`, but replace `1.2.3.4` with the public IP address you wrote down in the previous step, and `CIS5550.pem` with a path to the `.pem` file you downloaded above. Be sure to include the `ec2-user` part – don't try to log in as `root`, or using your normal username! You should see a warning that says that the authenticity of the host can't be established, and asks you to confirm that you want to connect. Enter `yes`. You may now see a warning that says that your private key file (`CIS5550.pem`, or whatever name you gave it) is unprotected; if so, run `chmod 400 CIS5550.pem` and try the `ssh` command again. If all goes well, you should see a banner that says "Amazon Linux 2 AMI".

Step #11: Configure your EC2 instance. Next, you'll need to install some software on your EC2 instance – specifically, Java and Certbot. Run the following:

```
sudo amazon-linux-extras install epel -y
sudo yum install java certbot
```

and then hit `y` to confirm when necessary.

Step #12: Set up your DNS entries. You should have received an username and a password for the course infrastructure. Find this information, then go to http://dns.cis5550.net/ and log in. Enter the public IP address of your instance and confirm. If this is the first time you've done this, you should now be able to run `ping xxx.cis5550.net` (with your SEAS login name instead of `xxx`) and see that the name is resolving to your instance's public IP address. (You may not see any actual responses to the pings, because of the AWS firewall.) If you've used this system before, the old DNS entries may still be cached in various places, so it may be necessary to wait a few hours until the changes become effective. In this case, it is important to wait a bit before you proceed, otherwise the Certbot step and/or the final web page access may fail.

Step #13: Generate your certificate. Next, you need to run Certbot to generate a Let's Encrypt certificate. On your EC2 instance, run `sudo certbot certonly --standalone -d xxx.cis5550.net` (again, with your SEAS login name instead of `xxx`), enter your email address when prompted, enter `Y` to agree to the terms of service, decide whether to share your email with the EFF (enter `Y` or `N`), and wait for the verification step to complete. If all goes well, the tool should congratulate you and tell you where the certificate and chain have been saved. If the verification times out, chances are your DNS record is not correct – maybe you haven't waited long enough since the latest update (try again in one hour), maybe you forgot to replace the `xxx` with your SEAS login name, or maybe you accidentally entered the wrong IP address.

Step #14: Convert the certificate. You now have a certificate for your domain, but it is not yet in the format that Java expects. You can convert the certificate by running the following two commands on your EC2 instance (on two single lines; again, remember to replace `xxx` with your SEAS login name):

```
sudo openssl pkcs12 -export -in /etc/letsencrypt/live/xxx.cis5550.net/cert.pem
                    -inkey /etc/letsencrypt/live/xxx.cis5550.net/privkey.pem
                    -out /home/ec2-user/keystore.p12 -name xxx.cis5550.net
                    -CAfile /etc/letsencrypt/live/xxx.cis5550.net/fullchain.pem
                    -caname "Let's Encrypt Authority X3" -password pass:secret
keytool -importkeystore -deststorepass secret -destkeypass secret
                    -deststoretype pkcs12 -srckeystore /home/ec2-user/keystore.p12
                    -srcstoretype PKCS12 -srcstorepass secret -alias xxx.cis5550.net
                    -destkeystore /home/ec2-user/keystore.jks
```

If all goes well, you should now have a file called `keystore.jks`.

Points	Feature(s)	Test case(s)
5 points	GET request via HTTP	`http`
10 points	GET request via HTTPS	`https`
5 points	Session ID generation	`sessionid`
10 points	Session permanence	`permanent`
5 points	Session expiration	`expire`
15 points	Screenshot of AWS deployment	
15 points	Let's Encrypt certificate	
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

Step #15: Copy your server to the instance. Make sure your test server is fully compiled, then use `scp` (or your operating system's equivalent tool) to copy the `.class` files to the EC2 instance. For instance, on macOS you could run something like:

```
scp -i CIS5550.pem -r classes/ ec2-user@xxx.cis5550.net:
```

As before, be sure to replace the name of the `.pem` file and the domain name with the correct values. You may also need to replace `classes` with whatever directory holds your class files. If all goes well, your class files should now be on the EC2 instance.

Step #16: Launch your server and make the screenshot. Now, launch your test server on the EC2 instance. The command will probably be something like

```
sudo java -cp classes cis5550.test.MyTestServer
```

but you'll need to replace `classes` with whatever directory on the EC2 instance contains your class files, and the class name with the actual name of your test server class. Be sure that the `.jks` file is in the current directory, so the server can find it. To verify that the HTTPS port is open, you can open a second terminal window, log into the EC2 instance again, and run `netstat -pant`; this should now show a LISTEN entry for TCP port 443. Now open your domain name (`https://xxx.cis5550.net/`, with `xxx` replaced with your SEAS login name) in a browser of your choice. Verify that the required message appears correctly. Then take the screenshot. On macOS, this should already be in `.png` format; if you are using some other operating system, you may need to convert it to one of the formats we accept. If you are using Eclipse, you should now add the screenshot and your `keystore.jks` file to your project, so they will be included in the `.zip` file you export for the submission.

Step #17: Shut down your instance again. Go to the EC2 web interface again, pick the "Instances" page in the navigation bar on the left, and find the entry for the instance you created. Check the box next to the correct instance (be careful!!), then click on the "Instance state" drop-down list and choose ''Terminate instance". A warning will appear; click the orange "Terminate" button to confirm.

Final step: Double-check the requirements! During initial development, it makes sense to focus on passing the test cases; 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 `HW1` 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!

Add support for multiple hosts (+10 points): This extra credit is only available if you've already done the "multiple hosts" EC on HW2. (If you haven't done that, you can implement it now, but you won't receive additional credit.) Add two additional arguments to your `host` method – the first should be the name of an additional key store file for the new host, and the second should be the corresponding password. Then, generate additional certificates and use the SNI extension to pick the correct one for an incoming connection. You can use the `SNIInspector` class in the `cis5550.tools` package to read the SNI hostname; the file should contain additional instructions. (If you need additional accounts on `dns.cis5550.net` for testing, please contact the instructor.) For any routes that were defined before `host` is called, you should use the original `keystore.jks` file, and these routes should be "default routes" that are used if no SNI hostname is included in the request at all, or if the hostname is not among the ones given as arguments to `host`.

Secure your cookie (+5 points): Have a closer look at the `Set-Cookie:` header and use the security features it provides – including at least `HttpOnly`, `Secure` (if HTTPS is used), and `SameSite` attributes.

CIS 5550: Internet & Web Systems

Spring 2023

Assignment 4: In-Memory Key-Value Store

Due February 20, 2023, at 10:00pm EST

1 Overview

For this assignment, you will implement a simple distributed key-value store. This will be used as the storage system for your crawler and your analytics platform later on.

The basic functionality is quite simple: clients will be able to PUT new values into the store, and GET existing values from the store. The main challenge is coordination: the store will be sharded across multiple worker nodes, and each node will store the values for keys in a certain range. We will use consistent hashing to determine the key ranges: each worker will have an ID, and will be responsible for all the keys between its ID and the next-higher worker ID. As usual, the worker with the highest ID will take care of any keys that are higher than its ID, or lower than the lowest ID. For instance, if there are three workers with IDs `banana`, `fig`, and `pear`, the keys `coconut`, `guava`, and `tangerine` would be stored on the first, second, and third worker, respectively. The key `apple` would be stored on the third worker as well.

This approach requires a way to keep track of system membership. We will use a single master node for this. The worker nodes will periodically check in with the master, and the master will keep a list of currently active workers and their IDs. Clients can download this information from the master, and then contact the relevant workers directly when they want to issue a GET or PUT. This kind of coordination is needed for many kinds of distributed systems, including the analytics framework we will build in later assignments, so we will make this functionality reusable, and factor it out into a separate package.

To keep things simple, we will keep the keys and values entirely in memory for now; we will add persistence in the next assignment. Also, we will not implement any kind of replication, so even a single worker failure will cause some unavailability. All the communication will be done via HTTP, and you should be able to use your dynamic web server from the earlier assignments for this. We will make an example implementation available in case you were not able to complete these assignments.

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!

2 Requirements

Please start by downloading the HW4 package from http://cis5550.net/hw4.zip. This contains a `README` file with a short questionnaire, an Eclipse project definition (which you can ignore if you are not using Eclipse), an interface (`KVS`), and three classes (`HTTP`, `Row`, and `KVSClient`). It also contains a simple implementation of HW3, as a file called `lib/webserver.jar`. Your solution must meet the following requirements:

Generic master/worker: Your solution should contain four classes: `cis5550.kvs.Master`, `cis5550.kvs.Worker`, `cis5550.generic.Master`, and `cis5550.generic.Worker`. The

KVS classes should extend the generic classes, and the latter should contain as much of the functionality below as possible, so it can be reused in HW6. In particular, the generic master should contain 1) a static function called `getWorkers` that returns the current list of workers as `ip:port` strings; 2) a static function called `workerTable` that returns, as a `String`, the HTML table with the list of workers as described below; and 3) a static function called `registerRoutes` that creates routes for the `/ping` and `/workers` routes (but not the `/` route) below. The generic worker should have a static function called `startPingThread` that creates a thread that makes the periodic `/ping` requests, as described below.

Master: Your `cis5550.kvs.Master` should accept a single command-line argument: the port number on which to run the web server. The application should maintain a list of active worker nodes, with an ID, an IP address, and a port number for each worker, and it should have make two functions available via HTTP GET requests. A GET to `/ping?id=x&port=y` should add an entry for the worker with ID `x`, port number `y`, and the IP address the request originated from; if an entry for `x` already exists, its IP and port should be updated. The request should return a 400 error if the ID and/or the port number are missing, and the string `OK` otherwise. When there are k active workers, a GET to `/workers` should return $k + 1$ lines of text, separated by LFs; the first line should contain k, and each of the following lines should contain an entry for a different worker, in the format `id,ip:port`. When a worker has not made a `/ping` request within the last 15 seconds, it should be considered inactive, and be removed from the list. In addition to the above two methods, your master should *also* accept `GET` requests for `/`, and such requests should return a HTML page with a table that contains an entry for each active worker and lists its ID, IP, and port. Each entry should have a hyperlink to `http://ip:port/`, where `ip` and `port` are the IP and port number of the corresponding worker.

Worker: Your `cis5550.kvs.Worker` should accept three command-line arguments: 1) a port number for the worker, 2) a storage directory, and 3) the IP and port of the master, separated by a colon (`:`). When started, this application should look for a file called `id` in the storage directory; if it exists, it should read the worker's ID from this file, otherwise it should pick an ID of five random lower-case letters and write it to the file. It should make a `/ping` request to the master every five seconds, and it should, via a web server that runs on the port number from the command line, make two functions available via HTTP: 1) A PUT to `/data/<T>/<R>/<C>` should set column `C` in row `R` of table `T` to the (possibly binary) data in the body of the request, and 2) a GET for `/data/<T>/<R>/<C>` should return the data in column `C` of row `R` in table `T` if the table, row, and column all exist; if not, it should return a 404 error. In both cases, the angular brackets denote arguments and should not appear in the URL; for instance, a GET to `/data/foo/bar/xyz` should return the `xyz` column of row `bar` in table `foo`. Row and column keys should be case-sensitive.

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

Packaging: Your solution should be in a directory called `HW4`, which should contain 1) the `README` file from the HW4 package, with all the fields filled in; 2) the file `webserver.jar` in a subdirectory called `lib`; and 3) a subdirectory called `src` with *all* of your HW4 source code, including the files we provided, and in the directory structure Java expects (with subdirectories for packages). Please do not include your HW1/2/3 code. Your solution *must* compile without errors if you unpack `submit-hw4.zip` and, from the `HW4` folder within it, run `javac -cp lib/webserver.jar --source-path src src/cis5550/kvs/Master.java` and `javac -cp lib/webserver.jar --source-path src src/cis5550/kvs/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.

The HW4 package contains a small KVS client in `cis5550.kvs.KVSClient` that you can use to issue individual GET and PUT requests. This could be useful for testing, in addition to the test cases we provided.

Step #1: Create the master classes. Create classes called `cis5550.generic.Master` and `cis5550.kvs.Master`, and make the latter extend the former. Create the three required methods in the generic master, and make `getWorkers` and `workerTable` return dummy values for now. In the KVS master, parse the command-line arguemnt and pass it to the web server's `port` function; then call `registerRoutes` (which won't do anything, for now) and finally define a GET route for `/` that returns a little HTML page with a descriptive title (maybe "KVS Master") and that includes the return value of `workerTable`. Your code should now compile; if not, make sure that the file `lib/webserver.jar` is in the classpath. Run the KVS master with argument `8080` (and `webserver.jar` in the classpath – Example: `java -cp lib/webserver.jar cis5550.kvs.Master 8080`) and then open `http://localhost:8080/` in your web browser; the little HTML you created should now show up. If the Master can't bind to port 8080, another program on your machine may be using it; in this case, try a different port.

Step #2: Implement `getWorkers` and the `/ping` and `/workers` routes. In the generic master's `registerRoutes()` method, create the `/ping` route; decode the worker ID and port number, and add them to an internal data structure. The `ping` test should work now. Next, create the `/workers` route, which should return the data from this data structure in the required format. The `ping` test should work now; you can test this with the `workers` test case in `HW4MasterTest`.

Step #3: Implement `workerTable`. In the generic master, implement the remaining method; don't forget the hyperlinks in the table! Run your master again, open `/ping` a few times to populate the table, and then open `/` to make sure that the table looks right. Click on one of the hyperlinks; the browser should complain that the page isn't found, but the address bar should show the IP and port from the worker entry. The `table` test case should work now as well.

Step #4: Implement expiration. If you haven't already, add a field to your worker data structure that keeps track of when that worker last invoked the `/ping` route. In that route, set this field to the current time, and in `getWorkers()`, only return workers whose field is sufficiently recent. The `expire` and `refresh` tests in `HW4MasterTest` should work now.

Step #5: Create the worker classes. Now create classes called `cis5550.generic.Worker` and `cis5550.kvs.Worker`, and make the latter extend the former as before. Create the `startPingThread` method in the generic worker, but leave it empty for now. In the KVS worker, parse the three command-line arguments and pass the first to the web server's `port` function; then call `startPingThread`. Your code should now compile; if not, see the end of the first step for the master classes.

Step #6: Implement the ping thread. In `startPingThread`, create a new thread that does two things: 1) invoke `Thread.sleep` to wait for the required interval (see Section 2), and 2) create a `URL` for `http://xxx/ping?id=yyy&port=zzz` (where `xxx`, `yyy`, and `zzz` are the third command-line argument, a random string of five lower-case letters, and the first command-line argument, respectively) and then call `getContent()` on it; this is a method in Java's built-in `URL` class that will trigger a HTTP request. Now, run the master on port `8080`, and then run the worker with arguments `8081 xxx localhost:8080`. If you now open the master's status page (`http://localhost:8080/`), you

should see an entry for the worker. Wait at least 30 seconds, and then reload the page; the entry should not disappear.

Step #7: Choose a worker ID. In the KVS worker's `main` method, look for the `id` file and read the ID from it if the file exists; otherwise, create a new one and write it to the file, as described in Section 2. Now repeat the test from the previous step; the worker's entry should show up, but with a random ID this time. The ID should stay the same if you keep reloading the page. Terminate the worker (but not the master) and wait for 30 seconds; the worker entry should disappear. Now restart the worker; the worker entry should reappear, but with the *same* ID as before. Also, the `read-id` test in `HW4WorkerTest` should work. (Notice that we are switching to the second test suite now!)

Step #8: Implement the PUT. In the KVS worker's `main` method, implement the PUT route. We have provided a class called `Row` that can store a row of data; this may seem trivial now but will be useful in HW5, when we'll need to serialize entire rows, using the methods in this class. Start by creating some kind of data structure that maps a `String` (table name) to a map of `String`s to `Row`s. Now implement the PUT. The three arguments (table name, row, and column) should be named parameters (`:xxx`), which you can extract with the `params` method of the request object. Check whether your data structure already contains an entry for the table, and a `Row` with the specified key; if not, create empty ones. Now look up the `Row`, set the value of the relevant column to the request's `bodyAsBytes()`, and return `OK`. For reasons that will become apparent in HW5, it is best to implement a separate function, say `putRow(T,R)`, that puts a given row `R` into a table with name `T`. The `put` test case in `HW4WorkerTest` should now work.

Step #9: Implement the GET. Now, implement the GET route. As before, the arguments should be named parameters, but this time, be sure to return a 404 when the table, row, and/or column are not found. If they are found, set the response's body to the value in the column, using the `bodyAsBytes` method, and return `null` from the route; this should cause the server to correctly return the data in the column, even if it is binary. Again, to prepare for HW5, it is best to implement a separate function, say `getRow(T,K)`, that returns the row with key `K` from a table with the name `T`. The `putget` and `overwrite` test cases in `HW4WorkerTest` should now work as well. If they do, try the `stress` test, which uses multiple threads; if this causes exceptions, check whether the worker is synchronizing accesses to the data structures properly. (Remember that Java's `HashMap` is not thread-safe, but `ConcurrentHashMap` is!) Performance is not important for grading, but good performance will be critical for the project later on, so try to get the throughput at least to a few thousand requests/per second if you can. The number you can achieve depends on the type of machine you have.

Final step: Double-check the requirements! During initial development, it makes sense to focus on passing the test cases; 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 `HW4` 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.

Points	Feature(s)	Test case(s)
5 points	Registering a new worker using `/ping`	`ping`
10 points	List of workers	`workers`
5 points	Worker table	`table`
10 points	Expiration of stale worker entries	`expire`
5 points	Refreshing a worker entry	`refresh`
5 points	Reading and registering the worker's ID	`read-id`
5 points	Individual PUT request	`put`
10 points	GETting a value that was PUT in earlier	`putget`
5 points	Overwriting a value	`overwrite`
5 points	Stress test	`stress`
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

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!

Add a conditional PUT (+5 points): Extend the PUT route so that it accepts two optional query parameters, `ifcolumn` and `equals`. If both are present, it should check whether the column whose name is specified in `ifcolumn` exists and has the value that is specified in `equals`, and it should execute the PUT only if this is the case; if not, it should return `FAIL` instead of `OK`.

Support versioning (+10 points): Instead of keeping only the current value for each cell, keep the previous values as well, and give a version number to each value. The first value that is PUT into a cell should be version 1, and each PUT after that should increase the version by 1. Add a `Version: xxx` header to the PUT and GET responses to specify the version `xxx` that has been assigned or is being returned, respectively, and add a query parameter `version=xxx` to the GET that can be used to request a specific version, instead of the most recent version.

CIS 5550: Internet & Web Systems

Spring 2023

Assignment 5: Key-Value Store with Persistence

Due February 27, 2023, at 10:00pm EST

1 Overview

For this assignment, you will extend the key-value store from HW4 with persistence, as well as a nice user interface and some additional API functions. These will help to get the key-value store ready for the analytics system we will build in the following assignments.

Database management systems use a lot of sophisticated algorithms to make sure that the data is persistent and remains durable after a crash. In this assignment, we won't go that far – we will simply create a file on disk for each persistent table, and, whenever a new row is created, we'll write that entire row to the file. When an existing row is modified, we'll write the entire new row to the file. To save memory space, we'll remember only the offset in the file where the most recent copy of the row is stored. When a node crashes, we can re-create the state of the persistent tables at the time of the crash, simply by starting with empty tables and then reading through each file row by row and inserting each row into the corresponding table, replacing the existing row when necessary. Of course this approach isn't perfect: we'll be storing more information than strictly necessary (especially when there are lots of modifications), we may lose a little bit of information that was written at the moment of the crash, and recovery can take a long time. But it will suffice for our purposes – the main idea is to make sure that, when you crawl the web later on, you won't have to start over if something goes wrong during the crawl.

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!

2 Requirements

Please start by downloading the HW5 package from http://cis5550.net/hw5.zip. This contains a `README` file with a short questionnaire and an Eclipse project definition, which you can ignore if you are not using Eclipse. Your solution must meet the same requirements as in HW4, with the following additions:

Persistent tables: The workers should have a PUT route for `/persist/XXX`, where `XXX` is the name of a new table. If the worker does not yet have a table with that name, it should create an empty "persistent" table, as well as an empty file called `XXX.table` in its storage directory (which is specified using the second command-line argument, as in HW4), and then return the string `OK` with status code 200. If the worker already has a table with that name, it should return an error message and status code 403. The worker should remember which of its tables are persistent.

Logging: Whenever a worker adds or changes a row in a persistent table `XXX`, it should append an entry to the `XXX.table` file. The entry should consist of the output of `Row.toByteArray()` on the relevant row, followed by a LF character (ASCII code 0x0A). When an existing row is modified, the worker should *first* apply the change and *then* serialize the entire, updated row.

On-disk storage: For persistent tables, workers should not keep the full row data in memory; they should only keep an in-memory index that maps row keys to the file offset at which the most recent copy of the corresponding row is stored.

Recovery: When a worker starts up, it should inspect the storage directory (second command-line argument) and look for files with the extension `.table`. For each such file, it should create a persistent table (`XXX.table` should create a table called `XXX`) and rebuild its in-memory index by reading the file line by line, using the `Row` object to deserialize each line, and then inserting the resulting file offset into the index. When the key already exists in the index, the offset should be updated.

Whole-row read: The workers should support a GET route for `/data/XXX/YYY`, where `XXX` is a table name and `YYY` is a row key. If table `XXX` exists and contains a row with key `YYY`, the worker should serialize this row using `Row.toByteArray()` and send it back in the body of the response. If the table does not exist or does not contain a row with the relevant key, it should return a 404 error code.

Streaming read: The workers should support a GET route for `/data/XXX`, where `XXX` is a table name. When this route is invoked, the worker should iterate over all the local entries in table `XXX`, serialize each entry with `Row.toByteArray()` and then send the entries back, each followed by a LF character (ASCII code 10). After the last entry, there should be another LF character to indicate the end of the stream. If the HTTP request contains query parameters called `startRow` and/or `endRowExclusive`, the worker should compare the key of each row to the value(s) of these parameter(s), and send the row only if 1) `startRow` is either absent or the row key is equal to, or higher than, the value of `startRow`; *and* 2) `endRowExclusive` is either absent or the row key is smaller than the value of `endRowExclusive`. For example, if the table has rows with keys `A`, `C`, `M`, and `Q`, and the query parameters are `startRow=B` and `endRowExclusive=Q`, you should return the rows with keys `C` and `M`. If no table with the name `XXX` exists on the local node, the request should return a 404 error code.

Streaming write: The worker should also support a PUT route for `/data/XXX`. When this route is invoked, the body will contain one or multiple rows, separated by a LF character. The worker should read the rows one by one and insert them into table `XXX`; existing entries with the same key should be overwritten. The route should return the string `OK`.

Rename: The worker should have a PUT route for `/rename/XXX`. When this route is invoked, the body will contain another name `YYY`, and the worker should rename table `XXX` to `YYY` (and the corresponding log file from `XXX.table` to `YYY.table`. The worker should return a 404 status if table `XXX` is not found, and a 409 status if table `YYY` already exists. If the rename succeds, the worker should return a 200 status and the word `OK`.

Delete: The worker should have a PUT route for `/delete/XXX`. When this route is invoked and a table with the name `XXX` exists, the worker should delete it, along with the corresponding log file, and return a 200 status and the word `OK`. If no such table exists, the worker should return a 404 status.

List of tables: The worker should have a GET route for `/tables`. This should return the names of all the tables the worker knows about. There should be a LF character after each table name. The content type should be `text/plain`.

Row count: The worker should support a GET route for `/count/XXX`, where `XXX` is a table name. If a table with this name exists, the body of the response should contain the number of rows in that table (as an ASCII string); otherwise, you should send a 404 error code.

User interface: The worker should support GET routes for `/` and `/view/XXX`, where `XXX` is a table name. Both routes should return HTML pages (content type `text/html`). The first route should return a HTML table with a row for each data table on the worker; each row should contain a) the name of the table – say, `XXX` – with a hyperlink to `/view/XXX`, b) the number of keys in the table, and c) the word `persistent` if and only if the table is persistent. The second route should return a HTML page with 10

rows of data; it should have one HTML row for each row of data, one column for the row key, and one column for each column name that appears at least once in those 10 rows. The cells should contain the values in the relevant rows and columns, or be empty if the row does not contain a column with that name. The rows and columns should be sorted by the row and column key, respectively. If the data table contains more than 10 rows, the route should display the first 10, and there should be a "Next" link at the end of the table that displays another table with the next 10 rows. You may add query parameters to the route for this purpose.

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

Packaging: Your solution should be in a directory called `HW5`, which should contain 1) the `README` file from the HW5 package, with all the fields filled in; 2) the file `webserver.jar` 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-hw5.zip` and, from the `HW5` folder within it, run `javac -cp lib/webserver.jar --source-path src src/cis5550/kvs/Master.java` and `javac -cp lib/webserver.jar --source-path src src/cis5550/kvs/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.

The HW5 package does not contain a test suite; instead, you should use the small KVS client (`cis5550.kvs.KVSClient`) from HW4. However, please keep in mind that *all* the features from Section 2 are required, not just the ones that can be invoked by the client!

Step #1: Copy over your HW4 code. You can simply cut and paste your code from HW4 into the HW5 project. Be sure to include the web server library (from the `lib` directory) as well!

Step #2: Add the list of tables. The idea of the user interface is to make it easier to see what is going on, both in HW5 and in future assignments that build on it. So this is a good place to start. Add the GET route for `/` first, and return a little HTML page with the required rows. Now the `tablist` test case should work.

Step #3: Add the table viewer. Next, add the `/view/XXX` route, initially without pagination. As in HW4, the table name (here `XXX`) should be a named parameter. Use `KVSClient` to insert a couple of rows into the `test` table, ideally with different columns. Then open `http://localhost:8081/`, click on the link for the `test` table (which should take you to `http://localhost:8081/view/test`), and see whether the rows and columns appear properly. Also, the `tabview` test case should now work.

Step #4: Add persistence. Now add support for logging. Since each persistent table needs its own log, you'll need to store a mapping from (persistent) tables to some object that wraps the log – perhaps a `RandomAccessFile` (since we'll need to get rows from random file positions later on). Don't worry about reading the logs for now; just open a new log when a persistent table is first created, and append a new entry as specified whenever something is added or changed. If you followed our recommendations on the HW4 handout, you should already have a `putRow` helper function; you can just add a write there if the table is persistent. Notice that the `Row` object already contains a `toByteArray()` method for serialization; all you need to do is add the final LF. To test, use `KVSClient`'s `persist` command to create a persistent table, then use the `put` command to generate some load; be sure to both insert new rows

and to change existing ones. You should see a new line appear in the relevant `.table` file for each addition or change. Also, the `persist` test case should now work.

Step #5: Add the index. Next, change your code so that the data in persistent tables is kept only on disk. Change your `putRow` so that, in the case of a persistent table, it writes the actual row *only* to the log and keeps a placeholder row in the in-memory table that contains the file position in some column (say, `pos`). (The behavior for non-persistent tables must not change!) Then change your `getRow` function so that, in the case of a persistent table, it extracts the position from the placeholder row and then reads the actual row from the log disk. The `putget2` test case should now work.

Step #6: Add recovery. Now, add a way for the worker to reconstruct the index from the `.table` files when it starts up. This should be done right at the beginning, before defining any routes, to prevent race conditions. Use the `listFiles()` method on the `File` object to find the files in the storage directory, then pick the ones with the `.table` extension, read each from beginning to end (using the `readFrom()` method in `Row`), and create placeholder rows in memory with the row keys and file positions you are reading. Be sure to keep only the most recent position for each row key! Now the `readlog` test case should work.

Step #7: Add whole-row and streaming read. Now, add the GET routes for `/data/XXX/YYY` and `/data/XXX`. The former should be fairly straightforward, and you can use the `readrow` test case to test it, or simply open the URL directly in your browser (say, `http://localhost:8001/data/test/somerow`); this should produce a readable representation of the data in that row. For the streaming read, use the `write()` method in the response object. Don't try to materialize the entire response in memory and then sending it as a whole – this would consume a lot of memory and could cause problems later, when you work with a large amount of data. Also, don't forget to support the `startRow` and `endRowExclusive` query parameters. Again, test with the `rstream` test case and/or by opening the URL (say, `http://localhost:8001/data/test`) directly in your browser; you should see a response with one row of text for each row of data in the table. If you see only a single giant row, you probably forgot to insert the LFs between the lines, or you are using the HTML content type (try using `text/plain`).

Step #8: Add the streaming write, row count, and renaming. Next, add the PUT routes for `/data/XXX` and `/rename/XXX` and the GET route for `/count/XXX`; all of these should be fairly straightforward. Now the `wstream`, `rename`, and `count` test cases should work.

Step #9: Add pagination. Finally, make the following three changes the table viewer. First, show the rows sorted by key. Second, add a query parameter (say, `fromRow`) that can be used to specify where the table should start; when that parameter is present, skip all rows whose keys are smaller than the specified value. And finally, once you've displayed ten rows of data, check whether there is another key after that; if so, add a "Next" link at the bottom of the page, with the next key as the value of the `fromRow` parameter. To test, you can use `KVSClient` to upload a resonably large amount of data (at least 25-30 rows) and then verify that the viewer can display all of them properly. Make sure that the keys are sorted, and that the overall count is correct; it's easy to accidentally skip a row between pages. The `pages` test case should now work as well.

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:

Points	Feature(s)	Test case(s)
10 points	Writing the logs	`logging`
5 points	Recovering data from the logs	`readlog`
5 points	PUT persistent value, then GET it back	`putget2`
5 points	List of tables	`tablist`
5 points	Table viewer	`tabview`
5 points	Whole-row read	`readrow`
5 points	Streaming read	`rstream`
5 points	Streaming write	`wstream`
5 points	Renaming tables	`rename`
5 points	Creating a persistent table	`persist`
5 points	Counting the rows in a table	`count`
5 points	Pagination	`pages`
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

1. Double-check that your solution is in a `HW5` 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!

Garbage collection (+5 points): Every 10 seconds, check whether the worker has received any requests during the past 10 seconds. If it has not, iterate through all the tables, write a new log file for each table – initially under a different name – that contains only entries for the rows that are currently in the table, and then atomically replace the current log file for that table with the newly written one. This should save space: normally, multiple PUTs to the same row will generate multiple entries in the log, but only the last one really "counts" during recovery.

Replication (+5 points): Have the workers download the current list of workers from the master every five seconds. Whenever a worker that is currently responsible for a given key receives a PUT for that key, it should forward the PUT to the two workers with the next-higher IDs (wrapping around if necessary).

Replica maintenance (+5 points): Add two new operations: one that returns the current list of tables, and another that is analogous to the streaming read but returns only the row keys and a *hash* of each row. Then, every 30 seconds, have each worker invoke the first operation on the two workers with the next-*lower* IDs (wrapping around if necessary), and, for each table that is returned, invoke the second operation on that table, using the key range for which the other worker would be responsible. Then, whenever the worker finds a row it does not have locally, or a row that has a different hash on the other worker than the corresponding local row, have the worker download that row and add it locally (or replace its local copy with it). That way, if a new worker joins, it will automatically acquire all the data it is supposed to replicate. This EC is only available if the "replication" EC is also implemented.

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.

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]`.

CIS 5550: Internet & Web Systems

Spring 2023

Assignment 8: Distributed web crawler

Due March 27, 2023, at 10:00pm EDT

1 Overview

For this assignment, you will build a simple distributed web crawler, based on your Flame engine from HW6+HW7 and the KVS from HW4+HW5. Your crawler should be able to follow redirects, and it should be "polite" to web sites, by implementing the robot exclusion protocol (that is, pay attention to `robots.txt`) and by limiting the number of requests per second it will make to any individual web server.

The crawler will use an RDD to maintain a "queue" of URLs that it still needs to visit, and it will store the downloaded pages in a table called `crawl` in the KVS. Initially, this queue will just contain a single seed URL, which is provided as a command-line argument. The crawler then processes the queue in several "rounds". In each round, it runs a `flatMap` over the queue that does the following for each URL:

1. It checks whether the URL has already been visited. If so, it returns an empty set.
2. It checks whether `robots.txt` has already been downloaded from the relevant host. If not, it tries to download it now.
3. If the host has a `robots.txt` file, the crawler checks whether the file allows visiting the current URL. If not, it returns an empty set.
4. It checks whether at least one second has passed since it last made a request to this host. If not, it just returns the current URL, to be attempted again in the next round.
5. It makes a HEAD request for the URL and enters the HTTP response code – as well as the content type and the content length, if provided – into the `crawl` table. If the response is neither OK (200) nor a redirect (301, 302, 303, 307, 308), it returns an empty set. If the response is a redirect, it returns the new URL.
6. If the response is 200 *and* the content type is `text/html`, it makes a `GET` request and saves the page in the `crawl` table as well. Then it extracts and normalizes all the URLs from the page and returns them.

If there are multiple workers, they will executed the `flatMap` in parallel, so they work on different URLs concurrently. The final result of the `flatMap` is a new RDD of URLs to be crawled in the next round.

For testing, we have provided several "sandboxes" on the course web server. These sandboxes have no links to the outside world, so there is no risk that your crawler will break out of its sandbox and start crawling the entire web.

As in the earlier assignments, please do use Google, answers from the discussion group, 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!

2 Requirements

Please start by downloading the HW8 package from http://cis5550.net/hw8.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), 3) two helper classes (`URLParser` and `Hasher`), and 4) simple implementations of HW3 (`lib/webserver.jar`), HW5 (`lib/kvs.jar`), and HW7 (`lib/flame.jar`). Your solution must meet the following requirements:

Class name and arguments: The `run` method of your solution should be in a class called `cis5550.jobs.Crawler`. It should accept up to two arguments. The first, required argument is the seed URL from which to start the crawl; the second, optional argument is the name of the blacklist. If you are not implementing EC2, please ignore the second argument if it is present.

Output: Your crawler should save its results in a persistent KVS table called `crawl`. This table should contain one row for each URL that has been attempted, including URLs that resulted in redirections or errors. The row key should be a hash of the URL, and there should be a column `url` that contains the url and a column `responseCode` that contains the response code from the HEAD request, or, if the HEAD returned 200, the response code from the GET request. If the HEAD returned `Content-Type` and/or `Content-Length` headers, the corresponding values should be saved in columns `contentType` and `length`, respectively. If the response to the HEAD was a 200 *and* the content type was `text/html`, there should also be a `page` column that contains the body of the GET response exactly as it was returned by the server (same sequence of bytes).

Robot exclusion protocol: Before making any other requests to a given host, your crawler should make a *single* GET for `/robots.txt`. If this file exists, your crawler should parse it and follow any rules for `User-agent: cis5550-crawler`, or, if there are no specific rules for this user agent, any rules for `User-agent: *`. If neither case applies, it should treat the file as if it were empty. It should support three kinds of rules. The first two are `Allow: xxx` and `Disallow: xxx`, where `xxx` is any URL prefix. Your crawler should apply the first rule that matches a candidate URL, and the default should be to allow; for instance, `Allow: /abc` followed by `Disallow: /a` would allow `/abcdef` and `/xyz` but forbid `/alpha`. The third rule is `Crawl-delay: yyy`, where `yyy` is the minimum number of seconds between any two HTTP requests to this host, for different URLs (see below). This number can be a floating-point number (say, `0.05`). The `robots.txt` file should be cached; your crawler should not request it from the same host more than once.

User agent string: Your crawler *must* send the header `User-Agent: cis5550-crawler` in every HTTP request it makes to any web server.

Crawl delay: In the absence of a `Crawl-delay` directive in `robots.txt`, there must be a delay of at least one second between any two HTTP requests that your crawler makes to the same web server for different URLs. (It is okay to send the HEAD and the GET for the *same* URL back-to-back.) The only exception is the initial GET for `/robots.txt`, which does not need to be separated from other requests by a delay. If a `Crawl-delay` directive is present, the minimum spacing should be the value in that directive.

URL extraction and normalization: When your crawler has downloaded a page, it should extract and normalize the links in the `href` attribute of any anchor tags (`<a>`). Keep in mind that HTML tags are case-insensitive, that anchor tags can have other attributes (`<a ref="nofollow" href="/foo">`), and that not all anchor tags have a `href` attribute (`<a name="xyz">`). Extracted links should be normalized by adding the host name and port number, if they are not already present, and by converting relative links to absolute links – for instance, on a page from `https://foo.com:443/bar/xyz.html`, a link to `a/b.html` should become `https://foo.com:443/bar/a/b.html`, and a link to `/123.html` should become

`https://foo.com:443/123.html`. If a link to another host has no port number, you should use the default for the given protocol (80 for `http`, 443 for `https`).

URL filtering: Your crawler should ignore any URLs that 1) have a protocol other than `http` or `https`, and/or 2) end in `.jpg`, `.jpeg`, `.gif`, `.png`, or `.txt`.

Redirect handling: If your crawler sees a redirect status code (301, 302, 303, 307, 308), it should add the new URL from the `Location:` header to the list of URLs to crawl.

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

Packaging: Your solution should be in a directory called `HW8`, which should contain 1) the `README` file from the HW8 package, with all the fields filled in; 2) the files `webserver.jar`, `kvs.jar`, and `flame.jar` from the HW8 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-hw8.zip` and, from the `HW8` folder within it, run `javac -cp lib/webserver.jar:lib/kvs.jar:lib/flame.jar --source-path src src/cis5550/jobs/Crawler.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.

The HW8 package *does not* come with a test suite, but we have provided two "sandboxes" on our course web server that you can safely crawl. `simple.crawltest.cis5550.net` contains just ten pages with simple URLs (no subdirectories) that are all allowed by `robots.txt`, and it won't return error codes (at least on purpose) or redirections. `advanced.crawltest.cis5550.net` contains a lot more pages, with subdirectories and disallowed URLs, and it will occasionally redirect your crawler or return interesting error codes.

Step #1: Create the basic job. As a first step, create the `cis5550.jobs.Crawler` class and give it a `public static void run()` with two arguments: a `FlameContext` and an array of `String`s. Check whether the latter contains a single element (the seed URL), and output an error message (using the context's `output` method) if it does not. If it does, output a success message, maybe "OK". Now, compile the job, package it into a JAR file (using the `jar` command), and submit the job using `FlameSubmit` (e.g., `java -cp lib/kvs.jar:lib/webserver.jar:lib/flame.jar cis5550.flame.FlameSubmit localhost:9000 crawler.jar cis5550.jobs.Crawler http://simple.crawltest.cis5550.net/`); you should see either the error message or the success message, depending on what arguments you provide. If the job doesn't run, verify that you've started a KVS master, at least one KVS worker, a Flame master, and at least one Flame worker, in that order; that you've given the correct port numbers to each (the KVS workers need the port number of the KVS master, etc.), and that none of them have crashed or output any error messages. If the class was not found within the JAR file, have a look at its contents (`jar tvf crawler.jar`); there should be an entry for `cis5550/jobs/Crawler.class`. If this is missing, or if the `Crawler.class` is in a different (sub)directory, you did not package the JAR file correctly.

Step #2: Implement the GET. Now that your job is working, you should be ready to crawl! Create an initial `FlameRDD` (perhaps called `urlQueue`) by parallelizing the seed URL and then set up a `while`

loop that runs until `urlQueue.count()` is zero. Within this loop, replace `urlQueue` with the result of a `flatMap` on itself. The `flatMap` will be running in parallel on all the workers, and each worker will be returning the (new) URLs it wants to put into the queue. In the lambda argument, take the URL, create a `HttpURLConnection`, set the request method to `GET`, use `setRequestProperty` to add the required header, and connect it; then check whether the response code is 200. If so, read the input stream, create a `Row` whose key is the hash of the URL, add the `url` and `page` fields, and use the `putRow` from `KVSClient` to upload it to the `crawl` table in the KVS. For now, return an empty list from the `flatMap`, so that the loop will terminate after a single iteration. Run your job with `http://simple.crawltest.cis5550.net/` as the seed URL, then point your web browser to the KVS worker's main page, and inspect the contents of the `crawl` table; you should now see an entry for the sandbox's main page.

Step #3: Extract URLs. The next step is to extract all the URLs from each crawled page. This step (and the normalization step below) will be needed again for the PageRank job in HW9, so it makes sense to factor it out as a separate, static method. Take the downloaded page and look for HTML tags (`<...>`); ignore anything that's a closing tag (`</...>`). Next, split the contents of each tag by spaces; the first piece will be the tag name. Ignore any tags that are not anchors; keep in mind that tags are *not* case-sensitive. Next, look for a `href` attribute; if there is one, extract the URL. To test, you can simply output the extracted URLs from the job (using the `Context` object) and double-check that they look correct, and that all links were found.

Step #4: Normalize and filter URLs. So far, the URLs will be a mixed bag: some will be relative to the base URL of the current page (`foo.html`, `bar/xyz.html`, `../blah.html`), some will be absolute but lack a host name (`/abc/def.html`), some will have the host name but not a port number, some will be links to document fragments (`#xyz`), etc. Start by cutting off the part after the `#`; if the URL is now empty, discard it. If it is a relative link, cut off the part in the base URL after the last `/`, and, for each `..` in the link, the part to the previous `/`; then append the link. If the link doesn't have a host part, prepend the host part from the base URL (which should already have a port number in it); if it does have a host part but not a port number, add the default port number for the protocol (`http` or `https`). Finally, check whether the URL should be filtered out or not (based on the protocol and the extension; see above). Verify that the normalization works properly. For instance, if the following links were found in `href` attributes on `https://foo.com:8000/bar/xyz.html`, then...

- `#abc` should become `https://foo.com:8000/bar/xyz.html`
- `blah.html#test` should become `https://foo.com:8000/bar/blah.html`
- `../blubb/123.html` should become `https://foo.com:8000/blubb/123.html`
- `/one/two.html` should become `https://foo.com:8000/one/two.html`
- `http://elsewhere.com/some.html` should become `http://elsewhere.com:80/some.html`

You should apply a simplified the normalization step to the seed URL as well. Obviously, this won't be a relative link, because there is no page that it can be relative to, but you should add the port number if it is missing, and cut off anything after the `#`, if there is one. A final point: URL parsing is a very error-prone step, so we strongly suggest that you use the helper class (`URLParser`) we provided. Don't try to "roll your own" URL parser unless you are very confident that you can properly handle the many corner cases!!

Step #5: Add the URL-seen test. Now, return the list of extracted URLs from the `flatMap`, so they can be added to the queue again – but don't run your job yet, because the link graph contains cycles, so the same URLs will be downloaded again and again. To prevent this, hash the URL at the beginning of the `flatMap` lambda and check whether the `crawl` table already contains a row with that key. If it does, skip

the rest of the lambda. If you run the job now, it should crawl all of `simple.crawltest.cis5550.net` and terminate; you can verify the results by looking at the `crawl` table using the KVS worker's web interface. There should be entries for 10 pages. We've also included the `crawl.table` file with the framework code in the `examples` directory, so you can have a look at the expected output.

Step #6: Add HEAD requests. Next, add code to make a HEAD request before each GET. You can use a `HTTPUrlConnection` for this as well; just set the request method to HEAD. Inspect the response code and, if present, the content type, and only proceed to the GET if the conditions from Section 2 are met. You can test this by crawling `advanced.crawltest.cis5550.net`, which should return "interesting" error codes once in a while.

Step #7: Handle redirects. When you do the HEAD requests in the advanced sandbox, you will sometimes see a redirect code (301, 302, 303, 307, 308). When this happens, you should a) still save the status code in the `crawl` table under the original URL, so it won't be attempted again, but b) return the new URL (from the `Location:` header) from the `flatMap`, so it will be crawled in the next round. (*Do not* try to follow the redirection chain in the current round; this would violate the requirement that consecutive requests for different URLs be separated by a delay; see below.)

Step #8: Add rate-limiting. So far, your crawler will download the pages as quickly as it can. Next, add the required rate limit. You can create another table, perhaps called `hosts`, in the KVS for this purpose. Have a row for each host name, and a column for the last time this host has been accessed; update this time whenever you make a HEAD or GET, and check it before each request (unless the request is a GET and you've already checked for the corresponding HEAD); if the timestamp is too recent, simply return the current URL from the lambda again, so it is added back to the URL queue and will be attempted again during the next round. When you are crawling multiple hosts, rounds will naturally take more than a second, but during testing they will finish quickly, so you may want to add a brief `Thread.sleep()` after the `flatMap`. (It is time timestamp check that causes the rate limit, not the `Thread.sleep()` – the queue can contain many URLs for the same host!) Notice that there will be a race condition in the event that multiple workers are crawling the same host at the same time; it is okay to ignore this.

Step #9: Implement the robot exclusion protocol. The final step is to respect `robots.txt`. Before making the first HEAD request to a given host, first make a GET request for `/robots.txt`; if the response is a 200, save the result somewhere (perhaps in another column of your `hosts` table). Make sure that there will be only *one* request to download this file from each host, even if the file does not exist or returns an error. Before each HEAD, look up the saved copy (if any) and parse it; be sure to look for rules both under the specific `User-agent` of your crawler (see Section 2) and, if there aren't any, under the wildcard (`*`). Support both `Allow` and `Disallow` rules, and keep in mind that 1) the rules specify prefixes and not entire URLs, and 2) the first match counts. If a URL is disallowed, discard it. `advanced.crawltest.cis5550.net` should have a couple of cases where this happens.

Final step: Double-check the requirements! Please keep in mind that the tests we described above 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 `HW8` 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.

Points	Feature(s)	Test case(s)
40 points	Correct crawl of `simple.crawltest`	
25 points	Correct crawl of `advanced.crawltest`	
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

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!

Content-seen test (+5 points): Extend your crawler to filter out pages that have the exact same content as another page that has already been crawled. You can create additional tables in the KVS for this purpose. In the `crawl` table, do not add a `page` column for pages that are duplicates; instead, add a `canonicalURL` column with the URL of the other page that this page is a duplicate of.

Blacklist with wildcards (+5 points): Add a second, optional command-line argument to your crawler that contains tha name of a KVS table with a "blacklist". When your crawler is started and this argument is present, the "blacklist" table will contain a (relatively small) number of rows that each have a `pattern` column. The entries in this column are strings that may contain the `*` wildcard (Examples: `http*://*/cgi-bin/*`, `*.pdf`, `http://dangerous.com/*`). If your crawler finds a URL on a page it has crawled and the URL matches one of these patterns, it should ignore that URL.

Anchor text extraction (+5 points): When your crawler finds an anchor tag of the form `<a href="xxx">yyy</a>` on a page (possibly with additional attributes other than `href`), it should normalize `xxx`, extract the text `yyy`, and add it to the row for the normalized `xxx` in the `crawl` table, in a separate column with a unique name that must start with `anchor`. You should add only one column for each page from which anchor text was extracted. For instance, if a page `zzz` contains both `<a href="xyz">blah blah</a>` and `<a href="xyz">blubb</a>`, you could add a column called `anchors:zzz` to the row for `xyz` that contains `blah blah blubb`.

CIS 5550: Internet & Web Systems

Spring 2023

Assignment 9: Indexer and PageRank

Due April 8, 2023, at 10:00pm EDT

1 Overview

This assignment is about the final two building blocks for the project: a simple indexer, and an implementation of the PageRank algorithm. Both are meant to take their input from the crawler you built for HW8. The indexer will take the crawled pages, strip out any HTML elements and punctuation, and then build an inverted index that maps each word to a list of pages that contain it. The PageRank job will extract the link graph from the crawled pages and run the algorithm on that graph.

As in the earlier assignments, please do use Google, answers from the discussion group, 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!

2 Requirements

Please start by downloading the HW9 package from http://cis5550.net/hw9.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), 3) two helper classes (`URLParser` and `Hasher`), and 4) simple implementations of HW3 (`lib/webserver.jar`), HW5 (`lib/kvs.jar`), and HW7 (`lib/flame.jar`). Your solution must meet the following requirements:

Class names: The `run` methods of your indexer and PageRank should be in classes called `cis5550.jobs.Indexer` and `cis5550.jobs.PageRank`, respectively.

Indexer: Your indexer should read its input from a KVS table called `crawl`. It should process any rows in this table that have two columns called `url` and `page` (and possibly other columns). Its output should be another KVS table called `index`, which should have a row for each word that occurs in the `page` column at least once; the row key should be the word (in lower case), and the row should have a single column with a comma-separated list of the URLs that contain this word. No URL should appear in this list more than once. Your indexer should filter out any HTML tags, it should remove punctuation (`.,:;!?'"()-`) and any CR, LF, or tab characters, and it should convert all words to lower case.

PageRank: Your PageRank implementation should read its input from the same `crawl` table as the indexer. It should extract and normalize the outbound links on each page (you can reuse your HW8 code for this) and it should run the "improved PageRank" algorithm from the lecture (with decay factor $d = 0.85$) on the resulting graph. It should accept a convergence threshold t as a command-line argument (Example: `0.1`), and it should keep running iterations until the *maximum* change of *any* rank value, relative to the previous iteration, is lower than t. It should output a KVS table called `pageranks` that should contain a row for each (normalized) URL, using that URL as the key; each row should contain a single column called `rank`, which should contain the final rank for that URL.

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

Packaging: Your solution should be in a directory called `HW9`, which should contain 1) the `README` file from the HW9 package, with all the fields filled in; 2) the files `webserver.jar`, `kvs.jar`, and `flame.jar` from the HW9 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-hw9.zip` and, from the `HW9` folder within it, run `javac -cp lib/webserver.jar:lib/kvs.jar:lib/flame.jar --source-path src src/cis5550/jobs/Indexer.java` and `javac -cp lib/webserver.jar:lib/kvs.jar:lib/flame.jar --source-path src src/cis5550/jobs/PageRank.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.

The HW9 package *does not* come with a test suite, but we have provided the output of a crawl of `simple.crawltest.cis5550.net` in `data/crawl.table`; you can load this data into a (single) KVS worker simply by putting the file into its storage directory before it starts up.

Step #1: Load the data (indexer). As a first step, create the class for the indexer, and, in its `run` method, load the data from the `crawl` table. Ideally, we would like a `PairRDD` of (u, p) pairs, where u is a normalized URL and p is the contents of the corresponding page, but we only have a `fromTable` for normal `RDD`s, so you'll have to make a slight detour: first, use `fromTable` and map each `Row` of the `crawl` table to a string `u,p`, where u is the URL and p the page, and then use `mapToPair` to convert this into a `PairRDD` again. (Yes, a `fromTable` for pairs would have been more convenient, but it would have complicated HW7.)

Step #2: Create the inverted index. Next, create the inverted index. This involves two simple steps. First, we need to convert each (u, p) pair to lots of (w, u) pairs, where w is a word that occurs in p. You can use `flatMapToPair` for this. Make sure to filter out all HTML tags, to remove all punctuation, and to convert everything to lower case. Remove any duplicate words, then return a list of pairs. The resulting `PairRDD` will still contain lots of (w, u_i) pairs, with the same word w but different URLs u_i, so we'll need to fold all the URLs into a single comma-separated list, using `foldByKey`. This should produce the required data; you can rename the final `PairRDD` to `index` using the `saveAsTable` method. To test, create a JAR file with your indexer class, then run something like: `java -cp lib/kvs.jar:lib/webserver.jar:lib/flame.jar cis5550.flame.FlameSubmit localhost:9000 indexer.jar cis5550.jobs.Indexer` and inspect the `index` table using the KVS web interface. Additionally, we have included some example output for the "simple" crawltest sandbox in `examples/index.table`.

Step #3: Load the data (PageRank). Now, on to the PageRank algorithm. Create the `PageRank` class, and load the data into a `PairRDD`, just like the indexer did – except that this time, don't make (u, p) pairs, make (u,"1.0,1.0,L") pairs, where L is a comma-separated list of normalized links you found on the page with URL u. We'll call this the "state table" below. If you've followed the suggested approach for HW8, you should already have a static method for extracting and normalizing links, which you can now use. Why

the "1.0,1.0"? We will need to maintain both a current and a previous rank value for each URL (so we can check for convergence). In Spark, you'd use a fancier data structure for this, but in Flame, we just have strings; hence this encoding. It is a good idea to run the job at this point and to inspect the contents of the state table; subtle encoding errors will cause headaches later on.

Step #4: Compute the transfer table. Next, implement a single iteration of the algorithm. The first step is to compute a "transfer table" that specifies how much rank should flow along each link. For each pair (u,"r_c,r_p,L"), where L contains n URLs, compute n pairs (l_i,v), where $v = 0.85 \cdot r_c/n$. In other words, each of the pages that page u has a link to gets a fraction $1/n$ of u's current rank r_c, with the decay factor $d = 0.85$ already applied. You can use a `flatMapToPair` for this. You may want to additionally send rank 0.0 from each vertex to itself, to prevent vertexes with indegree zero from disappearing during the `join` later on. Output the pairs you are emitting to `System.out`, and double-check that they look correct. (They will appear on the worker(s), not on the master.)

Step #5: Aggregate the transfers. Next, we need to compute the new rank each page should get, by adding up all the v_i from the many (u,v_i) pairs for each page u. You can use a `foldByKey` to do this; the result should be a *single* pair (u,$\sum_i v_i$) for each page.

Step #6: Update the state table. At this point, we have *two* `PairRDD`s: the old state table and an aggregated transfer table, both with the page URL as the key. We need to move the "new" rank from the latter to the former. To do this, first invoke `join`, which will combine the elements from both tables by URL, and concatenate the values, separated by a comma. (Technically, the `join` will pair up *each* value for a given key in one table with *each* value for that key in the other table, but in this case both tables should have a single entry for each URL, so the joined table should similarly have only one entry for each URL.) This is not yet what we want – there are more fields in the join result than in the original state table – so you'll have to follow up with another `flatMapToPair` that throws out the old "previous rank" entry, moves the old "current rank" entry to the new "previous rank" entry, and moves the newly computed aggregate to the "current rank" entry. This is also a good opportunity to add the 0.15 from the rank source. Run your job and inspect the results using the KVS client's web interface; the string manipulation for this step is a bit tricky.

Step #7: Iterate, and check for convergence. So far, we have only computed a single iteration, but we need to iterate until convergence. Put the code so far (after the initial load) into an infinite loop; at the end of the loop, replace the old state table with the new one, and then compute the maximum change in ranks across all the pages. This can be done with a `flatMap`, where you compute the absolute difference between the old and current ranks for each page, followed by a `fold` that computes the maximum of these. If the maximum is below the convergence threshold, exit the loop.

Step #8: Save the results. Once the loop terminates, you'll need to do one final step: the data is still in your internal format, with the previous rank value and the adjacency list, so you'll need to convert it to the format that Section 2 requires. To do this, you can run another `flatMapToPair` over the state table; in the lambda, decode each row, then put the rank into the `pageranks` table in the KVS, using the URL as the key and `rank` as the column name. `flatMapToPair` will expect you to return a list of pairs, but you can simply return an empty list; we're effectively using it as a `foreach`, which we did not implement in HW6+7. To test, you can run your solution on the "simple" crawltest sandbox. With 0.01 as the convergence threshold, this should converge in 13 iterations; we've included some example output in `examples/pageranks.table`.

Final step: Double-check the requirements! Please keep in mind that the tests we described above 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.

Points	Feature(s)	Test case(s)
25 points	Indexer	
40 points	PageRank	
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

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 `HW9` 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!

Word positions (+5 points): In the `index` table, produce the same output as specified above, except that a) `:x` is appended at the end of each URL, where `x` is a space-separated list of word posititons, and b) the list is sorted by the number of word occurrences, in descending order. For instance, if the word `apple` is the third and the eighth word on `foo.com` and the second, fifth, and twelfth word on `bar.com`, the (single) column of the `apply` row would be `bar.com:2 5 12,foo.com:3 8`. (Notice how `bar.com` is listed first, because the word occurs three times there, but only twice on `foo.com`.)

Enhanced convergence criterion (+5 points): Add a second, optional command-line parameter to the PageRank job that, when present, specifies a percentage of URLs for which the threshold from the first argument has to be met. For instance, if the job is run with arguments `0.1 70`, it should terminate as soon as at least 70% of the URLs have seen their ranks change by at most 0.1 from the previous round. This is a good way to terminate the job when "most" URLs have converged, without having to continue for the worst-case convergence time.

Stemming (+5 points): Use the Porter Stemmer to stem all the words before adding them to the index. You should add *both* the stemmed version *and* the original version of each word to the index. Search engines use this to boost the score of an exact match, vs. the score of a match after stemming.