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.