Big Data Management

Programming Assignment

Name: Orfanidis Eleftherios

ID: CS2200013

Development Environment

Window 10 64-bit, JDK 1.8

Deliverables

A zip that contains the following folders/files:

- src/com/ folder that has the source code of the project
- src/resources/ that has the commons-lang3 apache library.
- keyFile.txt which has 10 key names with their respective data types.
- serverFile.txt which contains information about 3 servers running on localhost, at ports 9000, 9001, 9002
- indexData.txt which contains 1000 generated data from CreateData. If you want to generate your own data you can delete this file and generate a new one.

Compilation

javac -sourcepath src -d build src/com/**/*.java -cp "src/resources/commons-lang3-3.12.0.jar"

Execution

- **CreateData:** java -cp "build;src/resources/commons-lang3-3.12.0.jar" com.main.CreateData -k keyFile.txt -n 1000000 -d 6 -l 11 -m 6
- **KVServer**: java -cp "build;src/resources/commons-lang3-3.12.0.jar" com.main.KVServer -a IP -p PORT
- **KVBroker:** java -cp "build;src/resources/commons-lang3-3.12.0.jar" com.main.KVBroker -s serverFile.txt -i indexData.txt -k 1

The above commands are used for execution in a windows environment. If you want to run the program in a Linux environment then the -cp parameter should be "build:src/resources/commons-lang3-3.12.0.jar" (Compilation and the rest of the execution commands are the same for both environments)

Data Structures

JsonData: This is the class that holds the information about a complex KV type. It consists of a key which is the high level key of the item, along with a list of tuples that represent all the simple KV pairs that belong to the specific key. For the complex children of the item, I create a separate JsonData instance which is stored in the keyValue Trie. For example the following "key1" : { "level": 4 ; "levels" : 7 ; "address" : { "name" : "Panepistimiou" ; "number" : 12 } } will be represented as shown in Figure 1. The most important methods of the JsonData class are the:

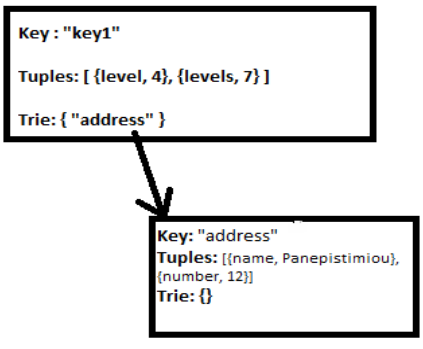

Figure 1

- toString(): which prints the contained data according to the assignment's guidelines

- static fromString(String data) : which generates a JsonData instance from a string.

- getChildKey(String… keys) : which takes as arguments an array of keys and searches for them in the keyValues Trie and in the tuples list if we are at the last given key. This method is used for the QUERY command when we need to search inside a JsonData instance.

Trie: I created a Trie data structure which can hold as payload any class that implements the KVClass interface (has only 1 method getKey() that needs to be implemented by the child class). The trie has a root Node and a list which contains all the high level keys that are inserted in the Trie. Each node has :

- A pointer to its parent.
- A character that represents the current key.
- A HashMap of all the available children of the Node. (Map<Character, Node<E>>)
- A payload pointer which will be null if we are on a non-terminal node.

Suppose we have a trie filled with the following items {KEY1, KA, A} and we want to search for KEY1, we will traverse the nodes as indicated by the red numbers in *Figure 2* and then retrieve the payload.

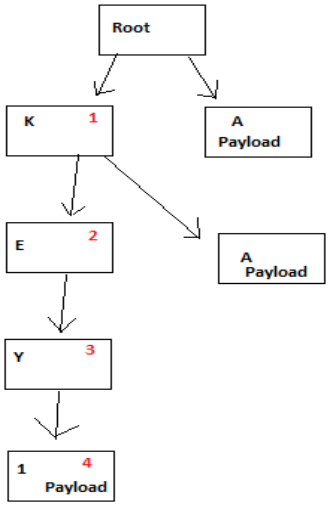

Figure 2

In the same trie, if we wanted to delete the Payload of KEY1 then the next state of the trie would be the one in *Figure 2* but without the nodes 2, 3, 4. Trie's most important methods are:

- get(String key) – Returns the payload or null if the key was not found or if the key stops at a non terminal node.
- delete(String key) – Returns true if it was successful or false if it didn't find the key
- insert(String key, E payload) – Return true if it was successful or false if something went wrong.

Create Data

For data creation I followed the guidelines of the assignment. I create a random JsonData instance according to the program's parameters. CreateData needs the following arguments to run:

- -d : Defines the max depth of the high level JsonData instance (how many nested KV pairs we have)
- -l : Defines the max length of a String value
- -k : Defines the file that contains key names and their respective data types. I used the same format as in the guidelines. (keyName keyType \n etc.)
- -m: Defines the maximum number of simple KV tuples in a JsonData instance.(The max length of the tuples list in a JsonData instance)
- -n : The number of high level JsonData instances created (the number of lines in the output file)

Create data prints each data on standard output, so if you want to save the data to a file, run it with " > indexData.txt " at the end of the command. The high level key names are numbered for easier debugging (key1, key2, key3 etc). All keys and string values have ' " ' surrounding them, but number types don't.

KVServer

Each KVServer holds a Trie<JsonData> and creates a socket on the specified IP/Port. Once the socket is created, it blocks until it accepts a connection (from the KVBroker). KVServer's lifetime is a single socket connection, so if the connection is lost, we need to restart the KVServer and then try to reconnect to it. The Server supports the following commands:

- GET highLevelKey - Returns either the full JsonData instance or 'NOT FOUND'

- QUERY highLevelKey.subsequentKeys - Returns either the tuple associated with the result (if the last key corresponds to a simple KV tuple, then the result will be “address” : “Panepistimioy”) or the JsonData associated with the result (in case the last key corresponds to a complex KV type) or ‘NOT FOUND’.
- PUT complexKVType - Return either ‘OK’ or ‘ERROR’ in case something went wrong during the put operation.
- DELETE highLevelKey – Returns either ‘OK’ or ‘ERROR – KEY NOT FOUND’ if the specified key is not present in the trie.
- PING – Returns ok. This is used by the Broker to check if a specific server is still active (Used to determine if we can run a delete operation on the broker)

KVBroker

KVBroker starts by reading the serverFile and opens a connection with every server specified. Once the connections are established it reads the dataToIndex file and then executes a PUT operation for each line found in that file. For each line I select random K servers from the available list and send a PUT command. After the insertions are complete, the broker blocks and waits for any user input. The user can type any GET, QUERY or DELETE commands which will be executed as described bellow.

- GET/ QUERY commands. We traverse each of the available servers and return the first result that is not ‘NOT FOUND’.
- DELETE command. When we receive a delete command, we need to check that all the available servers are still active. For this we use the PING command to check the state of each server. If any server doesn’t respond, then we show a message to the user that informs him that the command cannot proceed.
- EXIT which closes the connections and exits.

The connections between KVBroker and the KVServers are opened only once (on startup) and closed after the program has finished its execution.

Note: Each socket created from the broker, has a timeout of 800ms. This is done because I don’t want the broker to ‘hang’ during the PING check on the servers, if any server is down or blocked for any reason.

Assumptions

In case we have a duplicate high level key in the Trie, we just overwrite the payload with the new one.

In case we have multiple keys with the same name inside a JsonData instance, the QUERY operation will prioritize search for the key name inside the Trie, and if it is not successful it will look in the tuples list. This is done because querying a Trie is faster than searching for a name in a simple list.

If any server is down, the delete operation will block , but get/query can still be executed. Note that some error prints will be present on the command line if the socket throws IOException (I left them there for debug reasons mostly) , but the operation will continue and will return a result if the specified key is stored in any of the remaining servers.

The simple KVTuples will have datatypes that are specified in keyFile.txt, but the keys of complex types will have a random value to avoid conflicts.