Big Data Management

Fall 2022 Project

Νοτάρης Γιάννης
Αριθμός μητρώου: 7115112200038

Τμήμα Πληροφορικής και Τηλεπικοινωνιών
Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
Αθήνα 2022

Key Value Store

1. Data Creation

In this section the main task is to generate a random key (i.e. a key with arbitrary nesting, requested maximum number of sub-keys etc.). Then we loop for the requested number of keys. To combat this we use `DataCreationUtils.createRandomKey(...)` method.

```
1     public static void createRandomKey(JSONObject key, Integer nesting, Integer
      maxKeys, Integer maxNesting, Integer maxStrLength, Set<String> keyFile,
      Integer numberOfChildren) {
2         nesting++;
3         if (maxNesting < nesting) return;
4         for (int i = 0; i < numberOfChildren; i++) {
5             Object[] randAttr = getRandomAttribute(keyFile, maxStrLength);
6             JSONObject newKey = new JSONObject();
7             int newNumberOfChildren = RandomUtils.nextInt(0, maxKeys + 1);
8             if (newNumberOfChildren == 0 || nesting.equals(maxNesting)) {
9                 key.put(randAttr[0].toString(), randAttr[2].toString());
10            } else {
11                key.put(randAttr[0].toString(), newKey);
12                createRandomKey(newKey, nesting, maxKeys, maxNesting,
      maxStrLength, keyFile, newNumberOfChildren);
13            }
14        }
15    }
```

Listing 1: createRandomKey method

The approach here is to pass a JSONObject to the method and use recursion to fill it. We pass all the required parameters (`maxKeys`, `maxNesting`, `maxStrLength`) and use `nesting` and `numberOfChildren` as helper parameters. First, we increase the nesting (line 2), the first call has `nesting = 0`, to keep track of the nesting level. In the next line we check the maximum nesting requirement. The `numberOfChildren` is a helper parameter because when we create a new node we must know whether it has children or not (lines 8 to 13). If the new node has zero number of children we put a random value as the value (line 9) otherwise we put a new JSONObject as the value and we call the method again with the newNode and the new number of children for the newNode (line 12). The method will terminate if we surpass the maximum nesting or the number of children is zero.

The `getRandomAttribute(...)` method uses the keyFile and returns a random attribute (i.e. the name and the value). For that key it will remove the used attribute to make sure there is no reuse in the same nesting level. It could be further optimised to reuse attributes in different nesting. If the key-file runs out of attributes it returns a random UUID as the name and a random (float, int or string) value as the value.

2. Key Value Client

The client is the simplest part of the project. We use a `ServerStruct` class for each server (it has socket, in/out objects and a isConnected flag). At launch it will index the data from the given file by iterating on every line, shuffling the server list and sending PUT commands up to the replication factor. This way we achieve k-replication.

The main methods here are

(a) `handleCommand(String userCommand, List<ServerStruct> servers, int replicationFactor)`: Determines the type of the command and calls the `executeCommand(...)` accordingly (thas is with the correct server list).

(b) `executeCommand(List<ServerStruct> servers, String userCommand, int replicationFactor)`: Sends the `userCommand` in all the `servers` and returns the response.

(c) `handleCompute(String rightPart, List<ServerStruct> servers, int replicationFactor)`.

We will see how these methods work by analysing each command

(a) `PUT`: The `handleCommand(...)` will be called and will check if the key exists (`checkIfKeyExists(...)`). If the key does not exists it will call `executeCommand(...)` with a shuffled sublist with size equal to replication factor in the first parameter to maintain k-replication.

(b) `GET`, `QUERY`: The `handleCommand(...)` will be called and will immediately call `executeCommand(...)` with the full list of servers.

(c) `COMPUTE`: In this case the flow of the `executeCommand(...)` is completely different. It will immediately return `handleCompute(...)` method to handle the computation there. The `handleCompute(...)` method must be on the client side because it may contain multiple keys that don't lie on the same server. The `handleCompute(...)` method will parse the user command, send the required queries to the server (using the `executeCommand(...)`) for every compute parameter and finally calculate the result on the client and print it to the user.

(d) `DELETE`: The `handleCommand(...)` will be called and will check if all the servers are online. If all servers are online it will call `executeCommand(...)`

3. Key Value Server

To implement the server we use 3 classes. The main server class, a client handler and a utils class. The server uses a thread pool to handle multiple users. When a client connects, the server creates a ClientHandler object to handle the client's requests and submits it to the thread pool for execution. We use `ServerSocket` to implement the server but we will focus on how the server handles each command and not on the technical details of the client-server communication.

```java
package org.notaris.tree.trie;

import org.notaris.tree.Tree;

public class Trie extends Tree<TrieNode> {
    public Trie(TrieNode root) {
        super(root);
    }
}
```

Listing 2: Trie Class

Our first concern is to implement the trie data structure. To do that we first create a parameterized Tree implementation that we can extend to implement Trie (listing 2 & 3).

```java
package org.notaris.tree.trie;

import lombok.Getter;
import lombok.Setter;
import org.notaris.tree.TreeNode;

import java.util.HashMap;

@Getter
@Setter
public class TrieNode extends TreeNode<Character, TrieNode> {
    private Object value;
    private Boolean endOfWord = false;

    public TrieNode() {
    }

    public TrieNode(HashMap<Character, TrieNode> children) {
        setChildren(children);
    }
}
```

Listing 3: TrieNode Class

We assume that we have implemented helper methods to the Trie class to insert, find, delete etc words from a trie. The value of a key is held in the last character (`TrieNode`) of the field.

At this point a Trie could theoretically represent the whole database. A Trie by itself can only hold the top level keys. If we want to insert keys with arbitrary nesting we can insert the top level key in the main trie and insert a new trie as the value of that key. With recursion we can create nested trie's to represent any key. This is what `indexJSONObject(...)` method does.

```java
    private static void indexJSONObject(JSONObject jsonObject, Trie mainDB,
    String parentKeyName) {
        Iterator<String> keys = jsonObject.keys();
        Trie parentTrie = new Trie(new TrieNode(new HashMap<>()));
        while (keys.hasNext()) {
            String key = keys.next();
            Object value = jsonObject.get(key);
            if (value instanceof JSONObject) {
                TrieUtils.insert(key, parentTrie);
                Objects.requireNonNull(TrieUtils.find(parentKeyName, mainDB)).
    setValue(parentTrie);
                indexJSONObject((JSONObject) value, parentTrie, key);
            } else {
                TrieUtils.insert(key, value, parentTrie);
                Objects.requireNonNull(TrieUtils.find(parentKeyName, mainDB)).
    setValue(parentTrie);
            }
        }
        if (jsonObject.isEmpty()) {
            Objects.requireNonNull(TrieUtils.find(parentKeyName, mainDB)).
    setValue(null);
        }
    }
```

Listing 4: TrieNode Class

We represent the keys as JSONObjects when necessary (e.g. Listing 4).

Lets say we want to save `key -> [attr1 -> [attr2 -> 5 | attr3 -> 6]]` in the database. We create a instance of a `Trie` and we insert the word `"key"`. The `y` character will have `endOfWord = true` and we put as the `value` a new `Trie` to which we insert the word `attr1`. We put as `value` in the last character of `attr1` a new `Trie` to which we insert the words `attr2` and `attr3` and so on.

At this point we have solved the data structure part and we have to build the server.

The server should be able to accept the following commands. We will see how these methods work by analysing each command

(a) `PUT`: We create a `JSONObject` with the key and save it to the trie that represents the main database of the server exactly like described before.

(b) `GET`, `QUERY`: To find a query we split the given string and use the `find` method of trie to find the top level key, get it's value and continue with the nested keys. The `GET` command is identical but simpler because it doesn't use recursion and only searches the top level key.

(c) `DELETE`: We use recursion to go over every character (`TrieNode`) of the top level key and decide wether or not to delete it. Note that a node is not necessarily deleted if it's a part of a deleted word (e.g. If we have keys named `"k1"` and `"k2"`, character `"k"` exists only 1 time in the trie. We delete k2 but we don't want character 'k' to be deleted). Note that the last character of the deleted key will always be deleted witch will delete the value and everything inside it.