Big Data Management - M111

Programming Assignment

Alexandros Kalimeris - ds1200003

The assignment was implemented in Python

Data Creation

For the creation of the data that will be used for testing the Key-Value Store, I implemented a python script (createData.py). The script supports all the required command line arguements. In order to parse the command line arguements I use the argparse python module. In the case that a command line arguement is missing a default value will be automatically set.

For the creation of each line of data there are several things that should be noted:

- A recursive function that creates a dictionary of dictionaries is used in order to incrementally build each line.
- The top-level keys are named "key0, key1" etc. according to the line number for easier debugging.
- The data creation follows the following procedure more specifically:
  - A random number in [0, m] is chosen (m is the maximum possible number of keys in each value).
  - Then, if the maximum nesting level is not yet reached there is a 30% chance that a value will be a set of Key Value pairs in which case, a random string is chosen as the key name and the recuresive function is again called in order to create the nested set of Key Value pairs.
  - In the other 70% of the cases, where the value of the key is a simple int, float or string, a random key name is picked from the keyFile.txt and then a random value of the respective type is chosen.
  - Keys at the same level of nesting are not possible to be duplicates.
- After creating a dictionary of dictionaries that represents a line of data, since the format that we are asked to create the data greatly resembles JSON, I use the the python json module in order to transform the dictionary to a string.
- Then, a number of string manipulations are applied in order to transform the string to match the format that we are asked.
- Finally the string is written to the specified file, dataToIndex.txt by default.

KV Broker

Regarding the implementation of the KV broker, I followed the following steps:

- The communication between the KV broker and the KV server was implemented using sockets, specifically the python socket module.
- KV broker supports all the command line arguements asked in the assignment, if an arguement is not provided a default value will be used.

- When the KV broker starts its execution it reads the data to be indexed from a .txt file (dataToIndex.txt by default). It also reads the KV servers ips and ports from a .txt file (serverFile.txt by default).
- The broker puts all servers in a list of servers.
- Then the broker tries to connect to each server, if the connection is not succesful an error message will be displayed and the server will be removed from the list of servers. If no server is reached, the execution terminates.
- Then for each line of data, it selects k random servers (k=2 by default). It sends a a PUT command to the selected k servers and then it displays the response (OK, or some error).
- I keep an internal list in KV broker that tracks the top-level keys that have been sent so far, in the case of a duplicate top level key the line will be skipped and a warning will be printed.
- The kv broker now waits for a keyboard input. If it is GET, QUERY, or DELETE (case-sensitive) it will execute the specified command or else an Uknown Command error will be printed, I added an additional command 'exit' that shuts down the broker.
- Before trying to execute each command the server tries again to connect to each server, if a server is down an error message is printed, and it is removed from the servers list. Additionaly, a counter keeps track of how many servers have become offline after the initial indexing of the data.
- **GET:**
  - All keys should be **wrapped inside double quotation marks ""**. eg. GET "key1"
  - A GET request is issued in all available servers in a serial fashion, If the specified key is found the response is printed and we don't look into any more servers. If no server responds positively, NOT FOUND is printed instead.
  - If k or more servers are down, the GET command can still be executed against the remaining servers. However a warning that the results might be unreliable will be printed together with the results of the request.
- **QUERY:**
  - This request works similarly to the GET command.
  - A thing that I should point out is that **double quotation marks are needed for each key in the key path**. eg. QUERY "key1"."inner_key2"
  - Servers that become offline during the execution are handled in the same manner as GET.
- **DELETE:**
  - Again, all keys should be **wrapped inside double quotation marks ""**. eg. DELETE "key1"
  - A DELETE request is issued to all servers. If the delete operation was succesfull in K servers then DELETE OK is printed. If requested key was not found in any server, NOT FOUND is printed.
  - If for some reason, DELETE request is successful in <=k servers, an error message is printed.
  - As far as replication is concerned, if one or more servers are down DELETE is unable to be executed and a warning is printed if the user tries to execute it.

KV Server

Some details about the implementation of the KV Server:

- When KV server is starting up,it binds to an address:port pair specified by the command line arguements and waits for the KV Broker to connect.
- The KV server waits for a request to be sent over the socket. If there is a problem with the request syntax, etc an appropriate error will be displayed. If the request is one of the PUT, GET, QUERY, DELETE then the server follows through with the execution.
- **PUT:**
  - If a PUT request is received the KV server checks its syntax and if it is ok it stores the payload in an internal trie data structure (Trie implementation will be described below).
  - If there is any problem with the insertion there will be an appropriate response else it will respond with OK to the KV Broker.
- **GET:**
  - If a GET request is received the KV server checks its syntax and if it is ok it will perform a search against the internal Trie data structure.
  - If the top-level-key does not exist in the Trie NOT FOUND will be sent to the KV broker, otherwise key value entrie requested will be sent to the KV broker.
- **QUERY:**
  - Similar to GET, a get-like serach will be done against the Trie, if the top-level-key is is found then a dictionary will be returned. If QUERY has only a top-level-key then its results will be the exact same as get.
  - If there are more keys in the QUERY path then they will be searched in the dictionary returned by get in an appropriate manner.
  - If any of the keys does not exist in the Trie NOT FOUND will be sent to the KV broker, otherwise the requested value following the key path will be returned.
- **DELETE:**
  - Issues a delete command against the internal Trie.
  - If the top-lvl-key is found DELETE OK is sent back to the KV broker, otherwise NOT FOUND is sent.
- The server waits for new connections indefinetly, terminate manually with ctrl+C for example.

Trie

A Trie data structure was implemented from scratch for the purpose of storing the data inside the KV servers. The implementation is found at the trie.py file.

- A class called TrieNode is implemented. Is has the following class variables:
  - char: a string that holds the letter that the node represents in the Trie structure.
  - value: the value that is stored in the node if it is a final node. In the context of this assignment, it stores a value(int, float, string) or a dictionary of keys a and values.

    - end: a boolean variable that signifies if the specific node is the end of a high-level key or not.
    - children: a dictionary that holds the letter names of the children nodes as well as the children nodes themselves.
- The TrieNode class implements the following methods as well:
    - put: creates a path of nodes that represent the top level key and stores the respective value in the final node.
    - get: searches a top-level-key against the trie structre. Traverses the path specified by the key and if it ends up in a node with end = True it returns the value stored in that node.
    - query: executes a get query for the top-level-key in the query path. If the object returned is a dict then the next key in the path is used as a key in that dict and so on, until the requested value is found or a key is not found or an integer, string or float is tried to be accessed by the next key in the keys path which case None is returned.
    - delete: traverses the path specified by the key. If it ends up in a node with end = True it means that it is a node that represents a top-level-key, in that case, ends is set to False and the value is set to None. In any other, case None is returned.

General Notes

- Each message that is transmitted between the KV server and the KV broker is prefixed by an unsigned integer that represents the message length. This ensures, that each time either the server or the broker will execute recv() appropriately until the full message is recieved.
- By convention, all keys used in the different requests should be wrapped in double quotation marks.
- As mentioned above, due to the simillarity of the format of the data to the JSON format json.loads, json.dumps and then appropriate transformations are used to handle the data.
- If you want to use your own keyFile, please use 'int', 'float', or 'string' exactly as shown here for the type of each key.
- All servers use 127.0.0.1 as their address.