By

**Joshua Asogwa**

**SN1200015**

**Big Data Management Programming Assignment Report**

# **Programming Assignment**

# Throughout the course of the assignment, Python was used as the programming language for data creation as well as communication between the key value broker and servers. For this project, a fixed number of three servers were used however this can be increased by duplicating the codes for the servers and changing the port numbers and increasing the number of ports on the server file and making small tweaks in the KV brokers code in terms of ports it tried to establish communicates with. The data creation part was originally written in Jupyter notebook hence its in ipynb format but it can be copied and pasted to any other text editor like python IDLE and Pycharm for testing. The KV Broker and three servers were written using Python and Pycharm respectively. This is because if everything is run on python, an instance of a server run replaces the previous instance and hence makes the KV broker able to only communicate with only one server. To avoid this, PyCharm was used as it can establish parallel running of the three servers without interference. However, the KV broker was run in IDLE python editor. Also different data structures were adopted for the project. Further, assumptions made will be detailed in their appropriate sections below:

N/B: Dictionary and Objects are used interchangeably throughout the assignment and mean the same thing

# **1: Data Creation**

For the purpose of this part of the assignment. I tried creating a function of the format as stated in the assignment.

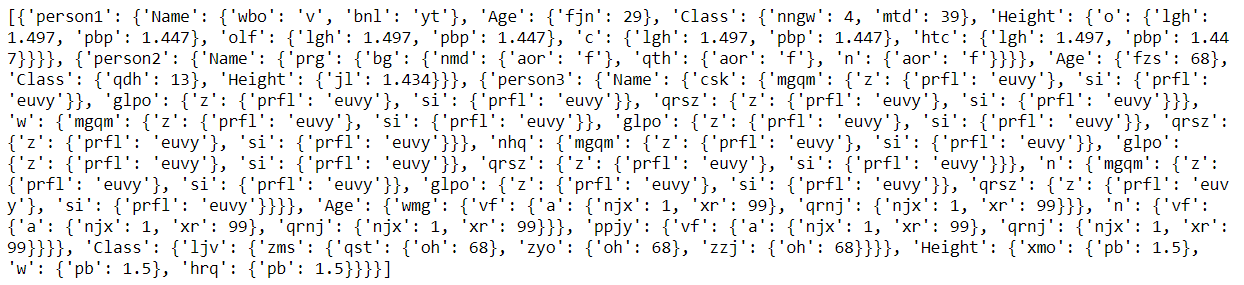
* The function takes input as follows:

**createData (filename, numberOfDictionary, levelOfNesting, lengthOfValue, numberOfKeys)**

* The output is a text file named dataToIndex containing number of person Objects created (input argument to the function is numberOfDictionary) with all the properties specified by inputs to the function including: maximum number of keys, maximum level of nesting of objects in each dictionary, maximum length of each words in the entire key value store.
* The keys used for the project include Name, Age, Class Height, Sex, Score, Address, Food, Best\_Friend, School, Church, Facebook, Special\_Number, All the high-level keys are obtained from the keyfile.txt in a sequential manner to prevent clashes.
* The value type of each key for all levels of nesting below is dependent on the datatype of the high-level key and the three options used include string, integer and float

Assuming the function has an input as follows:

createData ("keyfile.txt",3,4 ,4 ,4) the output is shown below:



*Figure 1: Results from part 1a of the project*

Source: Source code createData

**Methodology for the project**

Multiple looping was done in trying to generate the data required this makes it a little difficult explaining the methodology adopted in the source code. However, for easier comprehension, I decided to break down the process into 5 parts:

**Step 1: Data extraction from the keyfile.txt:**

The high-level keys and corresponding datatypes from the keyfile.txt were extracted and stored in a list of dictionaries with keys as the high-level keys and values as the datatypes for each of the keys.

**Step 2: Generate High level Object:**

Depending on the number of objects (person data) passed into the function to be generated, a list of objects with keys titled person1 up to person[i] is generated with each key having empty values.

**Step 3:** **Generate innermost objects:**

Based on the maximum number of keys (m) specified by the user, a random number is selected between one and the number of keys specified by user and a set of one to m objects is generated with each object having a specified number of random keys and corresponding random values Additionally, for each value generated, we check the data type of the high-level key to generate a string, integer or float of specified length by the user

**Step 4: Generate outer objects and handle nesting:**

Based on the number of nesting specified by the user we recursively assign the objects (containing keys and values from previous step) as values to each of the keys in the step 3 repeatedly until maximum level of nesting specified by the user is reached. To avoid having the same keys in nested objects, the function that generates values is rerun during every assignment period. For the level of nesting, a random number is chosen between one and user’s input for each key so that different high-level keys will have values that contain objects with varying levels of nesting.

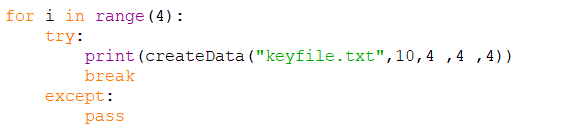
**Step5: Assign the object from part 4 as values to the object value from part1**

For each key in part one containing selected keys from keyfile.txt based on number of keys specified by the user, if the level of nesting is zero, we just create random values depending on the data type of the high-level key and assign it as a value to all the keys selected from the txt file otherwise, the object containing all nested objects generated from step4 is assigned as values to each of the keys from step one.

**Step 6: Error Management:**

Sometimes, the code usually gives an error titled “random.randrange did not receive arguments” when it had these arguments. Its an error that comes from an internal error or something related. So to avoid this, the code was run four times and as soon as there is no error it gives out the output. The code won’t give error four times so it’s a safety margin.

The implementation is shown below:



*Figure 2: Error handling in the project*

Source: Source code createData

# **2: Key Value Store**

For this part of the project, sockets were adopted to establish communication between the KV Broker and all the servers. Therefore the socket library was imported. More details on the implementation of this part of the project is shown below:

# **2a: KV Broker**

For this part, a function was generated titled KV Broker and it takes three arguments as follows:

**KVBroker (serverFile, dataToIndex, numberOfServers):**

The KVBroker tries to connect to the three specified key value servers in the serverFile specified by the server ports and IP address of the laptop as follows IP: 192.168.1.3, and ports 5050, 6050 and 7000. The methodology adopted for the operation of the key value broker is shown below:

**Step 1: Data Collection from the serverFile.Txt and dataToIndex:**

The KV value broker, reads the KeyFile.txt and tries to store the IP and Ports in two separate arrays titled serverIPAddress and serverPortAddress for later use. Additionally, all person’s data in the dataToIndex.txt are read and passed into the variable “data” as a list of objects.

**Step 2: Connection to servers:**

The KV broker tries to establish a connection through sockets with the three servers and in a situation where one of the servers is down, a prompt message is sent stating that one of the servers is down and the KV Broker needs to be rerun when the servers are running. For a connection to happen, the KV Broker sends a connection message and the key value server replies just like a three-way hand shake in TCP protocol as this is the mode of operation, I chose due to nature of the system implemented. A situation where the message is scrambled won’t be good at all. Instead, a few seconds delay for good output is preferred

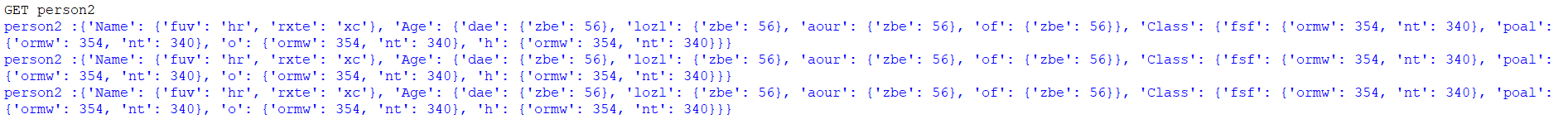
**Step 3: Sending messages to the servers based on the PUT method**

Once KV Broker is run, the PUT function automatically loop through all the individual person objects in the dataToIndex file and for each object, generate N random numbers between 1 and 3 based on the replication factor chosen as input to the broker, each person’s data is sent to a server that corresponds to this number. A string “PUT” is sent before the actual object data so that the servers can know what to do with the data. The data is converted to a string and finally bytes in UTF-8 format and sent to the server where its converted back to strings, evaluated, processed sand stored. After each object is sent, a disconnection message is sent to the server to give room for another message to be sent and also for the server to process the current message sent before the next one is received. Once the server receives the bytes and processes it, it sends a message showing message uploaded to server back to the KV Broker. As soon as all data has been sent to the respective servers and received message has returned for all objects sent, a message saying “OK data has been uploaded to all servers” is sprinted to the user. If one of the servers are not running it also sends a message stating one of the servers are not running and exits. The PUT query is not allowed to be inputted manually by the user. If its tried a message is printed stating this.

**Step 4: GET**

After the data has been uploaded to all servers, the broker requests the user to input a command. The GET function queries all the servers with the requested query and prints out all the information received from three servers and in a situation where one of the servers are down. It outputs one of the servers are down. If a server doesn’t have an information it prints “NOT FOUND, Data not in server”. It also prevents the user from sending wrong queries, e.g GET person.Name, GETperson1, GETS person etc.. Due to some time constraints, I was unable to implement the logic that, collate the information from the servers and outputs one information at a time or stating that we might not get the required information as one of the servers are down. However, the KV Broker receives all the required information from the servers if the servers have it and prints it out to the user as shown below:

e.g., The code works like this if three servers have all the requested information



*Figure 3: Results from GET part of the project*

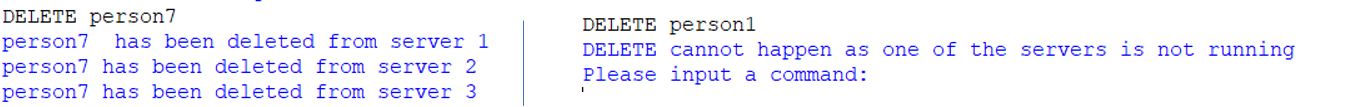
Source: Source code KV Broker

**Methodology**

The KV Broker simply receives an input from the user in text form of the form “GET highLevelKey”. Then the message received from the server is printed out to the user

**Step 5: DELETE**

The DELETE statement sends data of the form “DELETE person1” to all the servers. If a wrong query is sent it outputs the error in query. Also if any of the servers are down, it sends a message stating that one of the servers are down and delete cannot happen. Example:



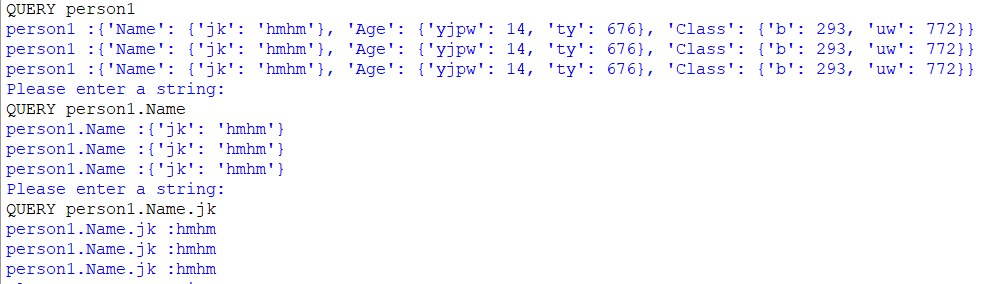
*Figure 4: Results from DELETE part of the project*

Source: Source code KV Broker

If any server is down and this is sent – DELETE person7, It states that delete cannot happen as one of the servers are down. If personsss is not in any server’s database and its sent like this -DELETE personsss, it outputs “NOT FOUND, Data not in database”.

**Step 6: QUERY**

The QUERY function works very similarly to the GET function. The only difference is in the nature of the string attached before the actual data stored in an object is sent to the servers. We get results from all servers as follows.



*Figure 5: Results from QUERY part of the project*

Source: Source code KV Brokers

# **2b: KV Server**

In this part, once the three servers are run in different ports, they start listening for connections from the KV broker which functions like a client. Once the server receives a message from the KV Broker, it checks if the message is correct. i.e if the message is PUT, QUERY, GET, DELETE or DISCONNECT. If it isn’t it prints ERROR and specifies the type of error. If the request is correct, it carries out the necessary functions. The various functionalities carried out by the server is shown below:

**Part 1: PUT function**

In the server, a trie data structure is defined via the class with attributes add, delete and search as these will be used later on. The trie data structure is instantiated to a variable where all received high-level keys will be stored. As soon as the server receives a message from the KV Broker it’s going to be a PUT message. The following steps are carried out:

* The server receives the message in a string format and evaluates the object part in string to a dictionary format and stores in a variable temporarily.
* The server checks to see if this object already exists in the already existing database through the high level key.
  + If it exists, then we just replace the new object with the previous one.
  + If it doesn’t, then we append the object to the list and add the high-level key of this object to the Trie data structure already instantiated.
  + This helps adopt the Trie data structure and also handle the nesting
* After this, a message is sent back to the broker that the message has been uploaded to the server

**Part 2: GET function**

When the server receives a GET string in front of the object sent from the KV Broker the following processes are carried out:

* It checks to ensure that the query does not have a full stop because it means we are trying to access a child object which isn’t appropriate for the GET function.
* If there is no problem with the query sent, the KV Broker tries to access the high-level key of the object sent. Once this is accessed, it checks the Trie data structure to see if the key already exists through the search attribute already defined in the earlier steps
  + If it exists, then we try to access the list of objects to find the index of the high-level key and access the value of this high-level key which could either be a singular value or an object. This result is forwarded to the key value store in the form: person: {Result}
  + If it doesn’t exist, then a message is sent to the KV Broker that this data does not exist in the server.

**Part 3: QUERY**

* When the server receives the QUERY request from the key value broker. A similar process used by the GET method is used to ensure that the query is in correct form otherwise an appropriate error is sent to the server. After this, queries of this form person.name.randomKey are all extracted and appended to a list. A similar process used by the GET method is adopted to check if the high level key is already existent in the servers memory i.e the first element in the list created
  + If it exists, then we try to find the index of the high-level key in the variable that contains all objects in the server. This location is appended to the variable in string format. The result will look like this receivedData[indexOfHighLevelKey] where “receivedData” is the variable name
  + After this, the content of the list that contains the requested query from KV Broker is concatenated to the variable name holding all received data. In the end the result will be like this “ receivedData[0][person][Name][randomKey1]”.
  + After this, the string is evaluated and this gives the actual data in memory. This result is queried back to the KV Broker
  + Otherwise, from step 1, if the high-level key doesn’t exist in memory, then we send a message to the KV Broker that message doesn’t exist in memory

**Part 4: DELETE**

* When the server receives the DELETE request from the key value broker. A similar process used by the GET method is used to ensure that the query is in correct form otherwise an appropriate error is sent to the server. After this, the server checks if the high-level key is already existent in the servers memory (trie data structure and list containing all objects).
  + If it exists, then we delete the high-level key from the trie data structure through the delete attribute previously defined. Additionally, we try to find the index of the high-level key in the variable that contains all objects in the server in a list. The entire data which is usually an object is popped from the list.
  + On the other hand, if the data is not found, we simply send a message “NOT FOUND, Data is not in server” back to the key value broker.

**Part 5: DISCNONNECT Message**

Once the server receives this message from the KV Broker, it simply disconnects and starts listening for a new query