LACPP Final Project
FooBase : A Simple NoSQL Database

Desislava Stoyanova
Ismail Elouafiq

March 9, 2016

Contents

1 Introduction

This document describes the results of the final project. Databases are a good place for both concurrency and parallelism. FooBase is a simple NoSQL database. It is implemented in Python (version 2.7.6 of the CPython implementation).

The database server uses a telnet connection to receive queries. Data is stored in JSON format in a key/value store. Values will be stored inside a corresponding key. Each value could be retrieved using the corresponding storage key. For simplicity the database handles only one document in a JSON format which will contain data in the form of keys and their respective values.

Additionally, two implementations of the FooBase server are provided in order to solve the common friends problem, one with a basic algorithm implementation and a second one using MapReduce. A comparison of their results is given at the end of this document.

Please note that the server implementation was inspired from Jeff Knupp's implementation on github. An example telnet client is provided within the source code and was taken from an example presented on binarytide.com .

2 Terms and abbreviations

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119. The following terms and abbreviations are used in this document:

- **FooBase** : refers to the database we implemented.
- **NoSQL database** : a database that models data storage and retrieval using a key/value store without a table schema.
- **FIFO Queue** : First In First Out Queue.
- **CRUD** : Create, Read, Update, Delete
- **DB** : DataBase.
- **JSON** : JavaScript Object Notation.

3 Specifications

3.1 Data Format

FooBase stores data as a key/value pair in a JSON format document.

For instance, if we have a database that contains the keys *oranges*, *apples* and *potatoes* with the respective values *10*, *20* and *42*, then the data will be stored in the following format:

```
{
  'oranges'  : '10',
  'apples'   : '20',
  'potatoes' : '42'
}
```

To solve the intersections problem (See section 4 Problem Considered), the value corresponding to a key should contain all the values to be stored in the key separated by a comma ','.

For example, we could have:

```
{
  '1' : '2,3',
  '2' : '1,4,3',
  '3' : '1,2,4',
  '4' : '2,3'
}
```

The data is stored in the *data_store/data.dat* file present in the *data_store* directory. This directory contains also the output of the *generate_intersections* command (See Section 3.3 Available Commands) for both the basic and MapReduce implementations.

3.2 Query Format

When a client sends a query to the DB server the query MUST follow the following format:

```
command key value
```

Where :

- **command** is the command to be executed by the database server (See Section 3.3 Available Commands).
- **key** is the key to which the command is to be executed. If the command requires a key, the key MUST be present in the query, otherwise it SHOULD NOT be present.
- **value** is the value given to the key in the command. This value MUST be present if it is required by the command.

The arguments (*command*, *key* and *value*) should be seperated by a space. For example, to create a new key named *foo* and assign the value *bar* to it, the query should look like this:

```
CREATE foo bar
```

NOTE : Please note that the command name is not case sensitive (i.e.: create, Create and CREATE will correspond to the same command)

3.3 Available Commands

The FooBase server can handle both the basic CRUD commands (*Create, Read, Update, Delete*) and one additional command *generate_intersections*, in the case of a friends database as described in Section 4 Problem Considered, to output the intersections between values that are in the form of sets separated by a comma as described in Section 3.1 Data Format.

- **Create**: Creates a key and a value pair. The key MUST be provided in this case. The value is OPTIONAL, in case no value is provided the value by default is '*None*'.
- **Read**: Reads the value of a key from the database. The client query SHALL only provide the key in this case. The provided key MUST be present in the database.
- **Update** Updates the value of a key. The key MUST be already present in the database. The client query SHALL provide the key. The value of the update is OPTIONAL, if this value is not provided it is set to *'None'*.
- **Delete** Deletes a key value pair from the database. The key MUST be provided in the query. The key MUST be present in advance in the database in order for the delete to happen.
- **Generate_intersections** This command is not implemented in the *FooBaseServer* class. It is implemented in both *FooBaseServerBasic* and *FooBaseServerMapReduce* classes that are derived from it. Running the *test_fbsbasic.py* using *python test_fbsbasic.py* will start a server on **localhost** port number **10000** where generate intersections can be tested, in this case the basic algorithm implementation is used. On the other hand, running the *test_fbsmapreduce.py* using *python test_fbsmapreduce.py* will start a server on **localhost** port number **11111** where generate intersections can be tested, in this case the MapReduce implementation is used.

3.4 Response Formats

The format of the response sent to a telnet client is as follows:

```
-Response Code: RESPONSE_CODE
-Response Message: RESPONSE_MESSAGE
-Response Value: RESPONSE_VALUE
```

Where:

- **RESPONSE_CODE**, **RESPONSE_MESSAGE** can be either of the following:
  - **'0000'** , "CREATE command successful." In case the **create** command was performed and succeeded.
  - **'0100'** , "CREATE command unsuccessful." In case the **create** command did not succeed.
  - **'0001'** , "READ command successful." In case the **read** command was performed and succeeded.
  - **'0101'** , "READ command unsuccessful." In case the **read** command did not succeed.
  - **'0011'** , "UPDATE command successful." In case the **update** command was performed and succeeded.
  - **'0011'** , "UPDATE command unsuccessful." In case the **update** command did not succeed.

  - **'0010'** , "DELETE command successful." In case the **delete** command was performed and succeeded.
  - **'0110'** , "DELETE command successful." In case the **delete** did not succeed.
  - **'1110'** , "Nothing happened in the database." In case the data has not been modified by the query and the query succeeded.
  - **'1111'** , "Error occured." In case an error occured while performing the query.

- **RESPONSE_VALUE** is the result of the query.

4 Problem Considered

We considered that we want to use this database to store connections between people. We assume that each person is represented by an id number. This number is going to be used to store the connections in the database in the following manner:

- Each **key** refers to a person.
- The **value** contains all the persons to which the current person is connected to seperated by a comma.

An example for this is the following data:

```
{
    '1'  :  '2,3',
    '2'  :  '1,3,4',
    '3'  :  '1,2,4',
    '4'  :  '2,3'
}
```

This would mean that person **1** is connected to both **2** and **3**, person **2** is connected to **1**, **3** and **4** etc. The order of the connections does not really matter.

We will try to solve the common friends problem where we need to find the mutual friends for all the people present in the database.

4.1 Basic Implementation

One version of this solves the problem by using a simple algorithm where we go through all the keys, and for each key we go through every other key and find the intersections between their friends. We improved this version a bit by sorting the keys in advance and only comparing them once so we do not end up comparing both **1** with **2** and **2** with **1** for example. The result of this approach is stored in the *data_store/basic_intersections.dat* file in JSON format.

To test one can run *test_fbsbasic.py* using **python test_fbsbasic.py**. This will start a telnet server in the port **10000** of **localhost**. The **generate_intersections** query can then be sent through a telnet client which can be started by running **telnet localhost 10000**.

4.2 MapReduce Implementation

The second version solves the problem by using MapReduce. In the **Map** step, we do the following we take each pair (X, [N1, N2, ..., Nk]) and send k pairs ((X, N1), [N1, N2, ..., Nk]), ((X, N2), [N1, N2, ..., Nk]), etc. We also make sure to order each pair (e.g: if N1¿X we use (N1, X) as a key instead of (X, N1)). In the **Reduce** step we perform an intersection of the values. If we have for instance for a key (A,B) both the values [B,C,D] and [A,C,E] the intersection would be [C], which is a friend for both A and B. **1** with **2** and **2** with **1** for example.

The result of this approach is stored in the *data_store/mapreduce_intersections.dat* file in JSON format. To test one can run *test_fbsserver.py* using **python test_fbsmapreduce.py**. This will start a telnet server in the port **11111** of **localhost**. The **generate_intersections** query can then be sent through a telnet client which can be started by running **telnet localhost 11111**.

5 Use cases

In the directory where the compressed source code file is extracted, to perform any of the following tasks please go inside the *foobase* folder using:

```
cd foobase/
```

5.1 Performing basic CRUD commands

To try out any of the basic CRUD commands the user must first run the FooBase server.

To do this, please use the following command:

```
python fbserver.py
```

This will start a FooBase server on the host and port in the default settins (fbsettings.py). In order to define a different host and port, the user can run fbserver.py by giving the host and port as arguments:

```
python fbserver.py host port
```

To perform one of the CRUD operations, the user must open up a new terminal window and start a telnet client either by using:

```
telnet host port
```

or by using the provided client example

```
python client_example.py host port
```

The user can then use the client to send a command for example:

```
CREATE 1010101 123
```

in the telnet client which would look something like this:

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

```
CREATE 1010101 123
-Response Code: 0000
-Response Message: CREATE command successful.
-Response Value: (u'1010101', u'123')
Connection closed by foreign host.
```

We can see here for instance that the create command was completed successfully and the key **1010101** has been added to the DB with a value **123**

5.2 Testing the Basic intersections generation

To try out the basic implementation of the **generate_intersections** (See Section 3.3 Available Commands) command the user must first run the FooBase basic server.

To do this, please do the following:

```
python test_fbsbasic.py
```

This will start a FooBase basic server on the localhost and port **10000**.

The user must open up a new terminal window and start a telnet client by using:

```
telnet localhost 10000
```

The user can then use the client to send a command for example:

```
generate_intersections
```

in the telnet client which will generate the intersections in the **basic_intersections.dat** file (See Section 6.1 Code Structure)

5.3 Testing the MapReduce intersections generation

To try out the MapReduce implementation of the **generate_intersections** (See Section 3.3 Available Commands) command the user must first run the FooBase MapReduce server.

To do this, please do the following:

```
python test_fbsmapreduce.py
```

This will start a FooBase MapReduce server on the localhost and port **10000**.

The user must open up a new terminal window and start a telnet client by using:

```
telnet localhost 11111
```

The user can then use the client to generate the intersections:

```
generate_intersections
```

The intersections will be generated in the **mapreduce_intersections.dat** file (See Section 6.1 Code Structure)

6 Documentation

6.1 Code Structure

The *foobase* folder contains all the code necessary to try the database. This directory should look something like this:

- **data_store** folder where the data is stored, this folder contains:
  - **data.dat** file that contains the data present in the database.
  - **basic_intersections.dat** which is the output of generating the intersections on the data using the basic algorithm implementation (See 3.3 Available Commands).
  - **mapreduce_intersectiosn.dat** which is the output of generating the intersections on the data using the MapReduce implementation (See 3.3 Available commands).
- **mapreduce** folder containing the mapreduce module implemented in Assignment 5.
- **logging.log** log file containing the logs of the last run.
- **client_example.py** An example telnet client implementation.
- **fbsettings.py** file containing the default settings used by FooBase. These can be modified by hand.
- **fbserver.py** file containing the FooBase server implementation.
- **test_fbsbasic.py** used to test the basic implementation of the intersections generation.
- **test_fbsmapreduce.py** used to test the MapReduce implementation of the intersections generation.

6.2 Notable Classes

The important classes implemented are described in this section.

6.2.1 FooBaseServer

This class represents the DB server. The server is considered as a state machine that can be in either one of the following states: BEGIN, STARTED or CLOSED.

Each instance of the FooBaseServer class is initialized in the BEGIN state. When the *start* method is called the server moves to the start state. The CLOSED state was added for when we stop the server but it has not been used in the current implementation.

When the *start* method is called a telnet server is run and waits for client connections. For each new client a new process is spawn.

A method for handling intersection generation is provided in this class but is only defined in the derived classes: FooBaseServerBasic and FooBaseServerMapReduce.

6.2.2 FooBaseServerBasic

This class defines handling generating intersection queries using a basic algorithm as explained in Section 4.1 Basic Implementation.

6.2.3 FooBaseServerMapReduce

This class defines handling generating intersection queries using the MapReduce implementation as explained in Section 4.2 MapReduce Implementation.

7 Encountered Challenges

7.1 Handling Multiple Clients

We first implemented a basic solution that only handles one client at once. To handle multiple clients at first we simply spawned a process to handle each new client. However the problem with this approach is that all the data stored in the database is being accessed by all the clients.

To solve this issue we proposed the following solution:https://preview.overleaf.com/public/khwwdcqwkfgk

- We define a FIFO queue that is shared where we store every change performed on the database.
- The queue is protected by a lock.
- These changes are then stored in the database sequentially.

7.2 Testing MapReduce with a Large Dataset

We needed a large dataset to compare the MapReduce implementation with the basic one. One solution would be to write a client that will generate a lot of data. The solution we chose however was to provide the data directly in the data store folder from the data that has been generated for the Common Friends problem in the MapReduce Assignment (Assignment 5).

8 Results and Comparison

Following are the results for performing the feature of generating intersections (See Section 4 Problem Considered). As shown in Figure 1, the MapReduce implementation takes more time when the problem size is small, this could be due to the overhead that is caused by storing the temporary files to perform the reduce step. However when the number of items in the database increases, we achieve much faster results when using MapReduce whereas the time in the basic implementation grows rapidly.

We achieve a linear speedup as shown in Figure 2 by using MapReduce, considering the basic implementation as a baseline.

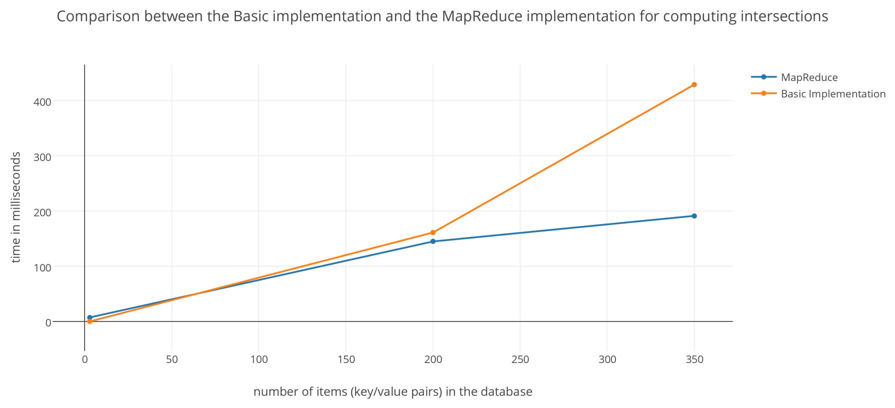

Figure 1: Comparison of time spent by each implementation with increasing problem size.

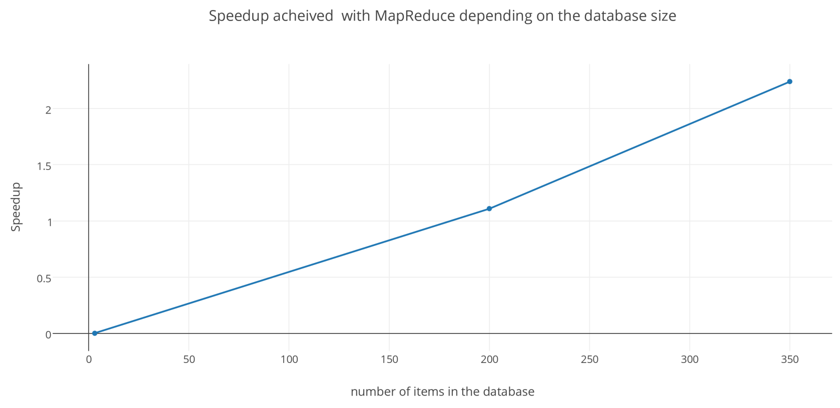

Figure 2: Speedup Achieved with MapReduce.

9 Conclusion and Future Work

Apart from going through the challenges that are encountered by trying to implement a very simple database. Implementing the MapReduce framework in the previous assignment made it easier to include to the project.

Using the MapReduce abstraction was effective in handling the common friends example compared to the simplistic implementation. Storing the data was done using a queue to commit the results of the commands. This approach permits multiple clients to apply changes in the database although it results in a sequential overhead.

Another problem is that although the writing was handled by the queue in the current implementation we did not consider the reading step. This does not cause problems in running the different commands of the database but it surely affects the features of the database itself where a recently updated key might be read by another client. This can be solved in the future by adding a local data set for each client and a commit step.