Israe Abdellaoui
Imane Boudra
Zakaria Elaouene

Project 2

Data Engineering and Visualization
Dr. Tajjeeddine Errachidi

Project 2

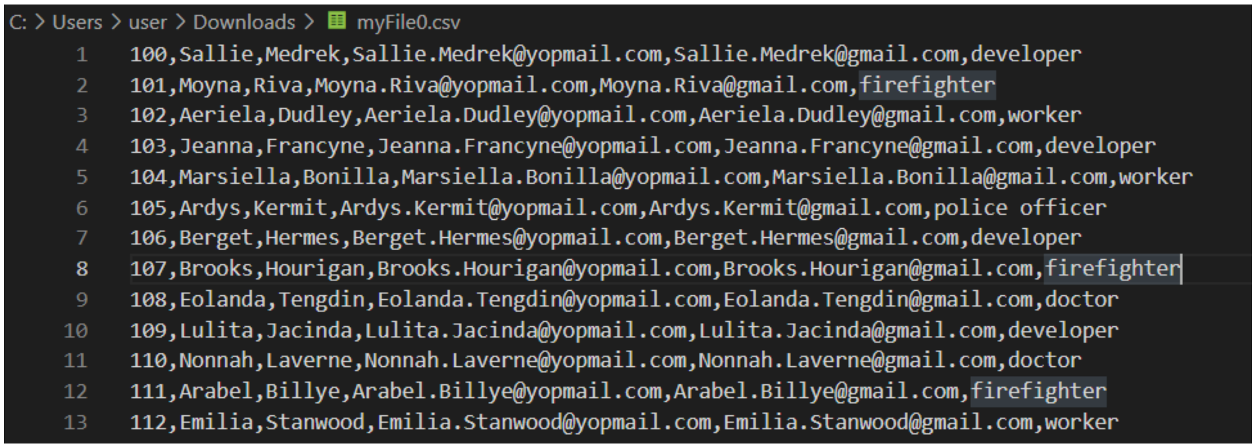
Step 1: Build the Join Using a Pipelined Double Hashed Join

Data Sources & Target Schema:

→Data Sources We're Using:

We're using two csv files with 1M entries each. The first csv contains 6 rows while the second contains 4 rows. The two csvs can be joined using the 'id'.

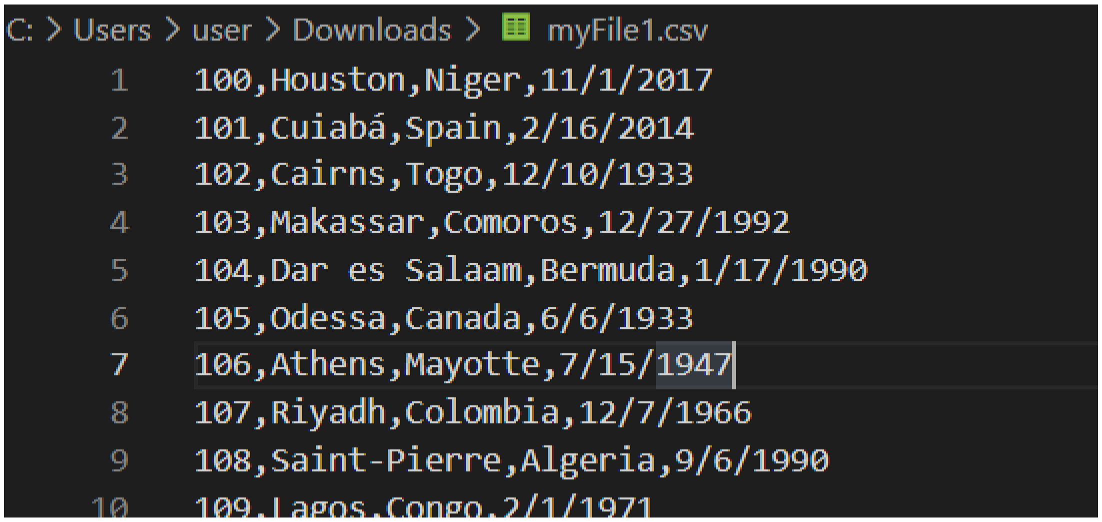
DS1: id, first name, last name, email, email2, job



C: > Users > user > Downloads > myFile0.csv

```
100,Sallie,Medrek,Sallie.Medrek@yopmail.com,Sallie.Medrek@gmail.com,developer
101,Moyna,Riva,Moyna.Riva@yopmail.com,Moyna.Riva@gmail.com,firefighter
102,Aeriela,Dudley,Aeriela.Dudley@yopmail.com,Aeriela.Dudley@gmail.com,worker
103,Jeanna,Francyne,Jeanna.Francyne@yopmail.com,Jeanna.Francyne@gmail.com,developer
104,Marsiella,Bonilla,Marsiella.Bonilla@yopmail.com,Marsiella.Bonilla@gmail.com,worker
105,Ardys,Kermit,Ardys.Kermit@yopmail.com,Ardys.Kermit@gmail.com,police officer
106,Berget,Hermes,Berget.Hermes@yopmail.com,Berget.Hermes@gmail.com,developer
107,Brooks,Hourigan,Brooks.Hourigan@yopmail.com,Brooks.Hourigan@gmail.com,firefighter
108,Eolanda,Tengdin,Eolanda.Tengdin@yopmail.com,Eolanda.Tengdin@gmail.com,doctor
109,Lulita,Jacinda,Lulita.Jacinda@yopmail.com,Lulita.Jacinda@gmail.com,developer
110,Nonnah,Laverne,Nonnah.Laverne@yopmail.com,Nonnah.Laverne@gmail.com,doctor
111,Arabel,Billye,Arabel.Billye@yopmail.com,Arabel.Billye@gmail.com,firefighter
112,Emilia,Stanwood,Emilia.Stanwood@yopmail.com,Emilia.Stanwood@gmail.com,worker
```

DS2: id, city, country, date



C: > Users > user > Downloads > myFile1.csv

```
100,Houston,Niger,11/1/2017
101,Cuiabá,Spain,2/16/2014
102,Cairns,Togo,12/10/1933
103,Makassar,Comoros,12/27/1992
104,Dar es Salaam,Bermuda,1/17/1990
105,Odessa,Canada,6/6/1933
106,Athens,Mayotte,7/15/1947
107,Riyadh,Colombia,12/7/1966
108,Saint-Pierre,Algeria,9/6/1990
109,Lagos,Congo,2/1/1971
```

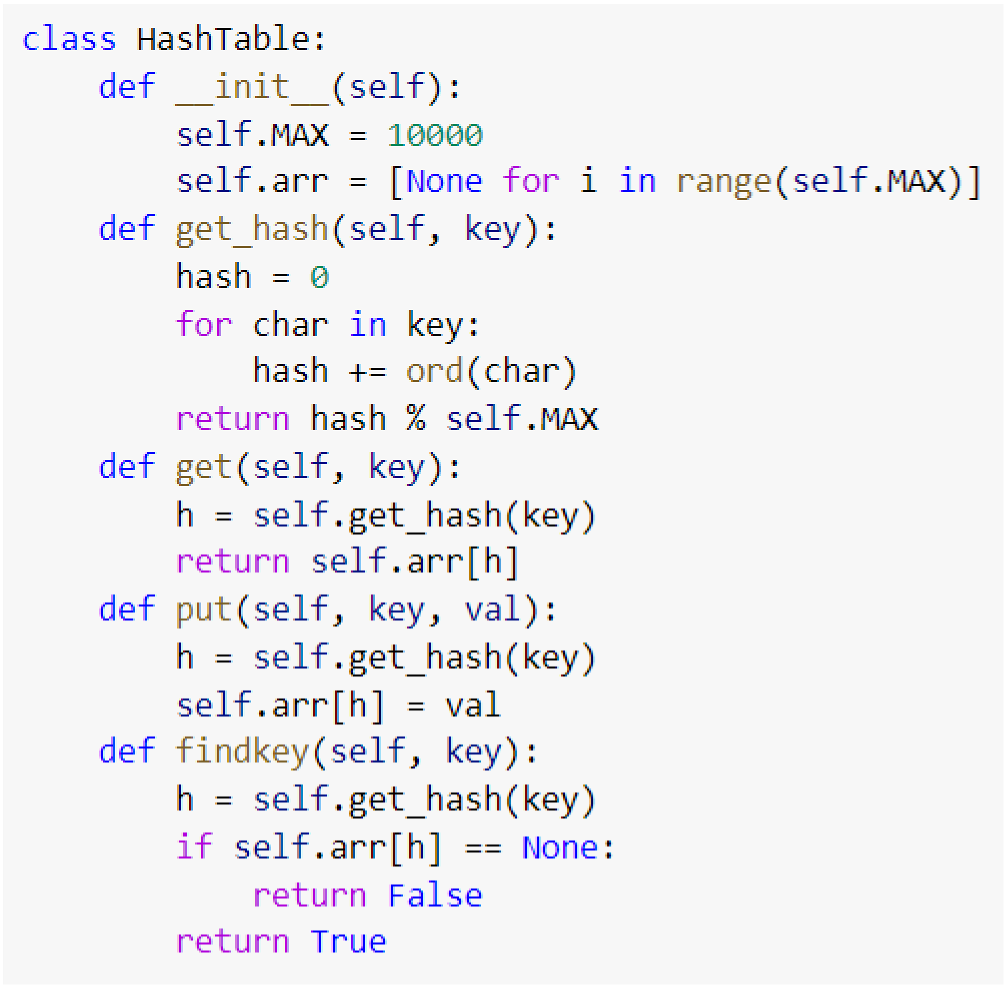
→ Target Schema:

Data	
PK	Id
	First Name
	Last Name

	Email
	Email2
	Job
	City
	Country
	Date

The ‘HashTable’ Data Structure:

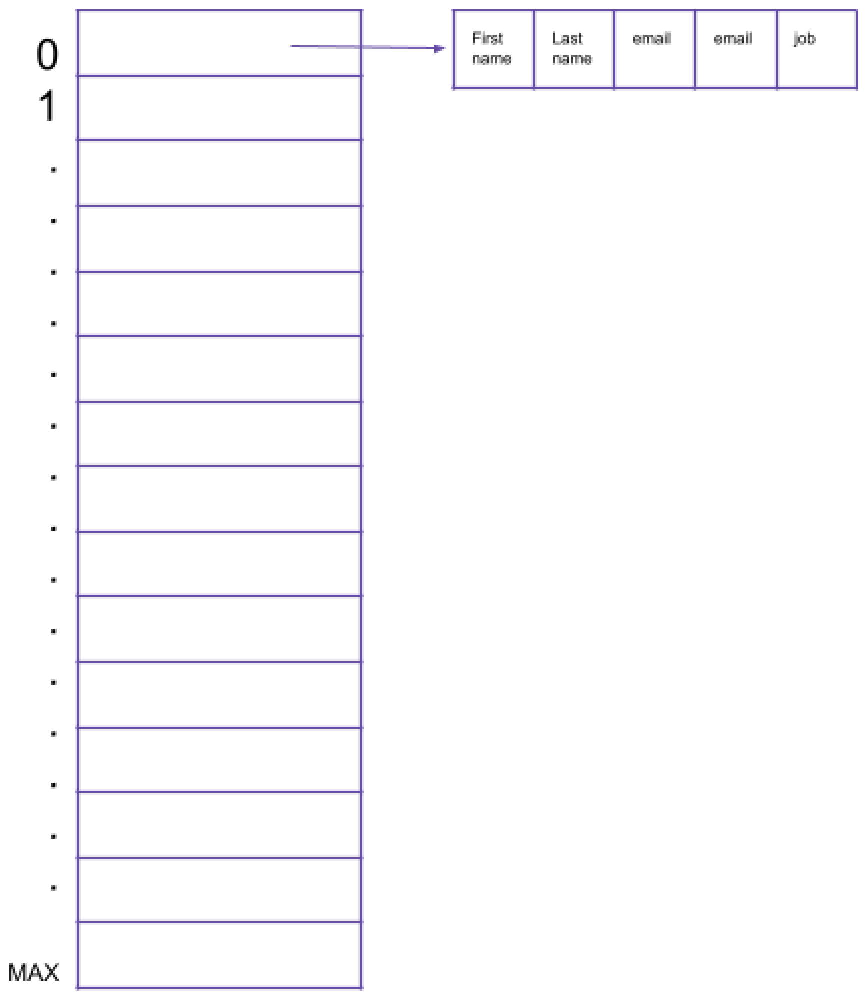
To implement the double pipelined hash join, we need to use 2 hash tables; one for each csv file. To that end, we defined our own ‘HashTable’ class like the following:



```
class HashTable:
    def __init__(self):
        self.MAX = 10000
        self.arr = [None for i in range(self.MAX)]
    def get_hash(self, key):
        hash = 0
        for char in key:
            hash += ord(char)
        return hash % self.MAX
    def get(self, key):
        h = self.get_hash(key)
        return self.arr[h]
    def put(self, key, val):
        h = self.get_hash(key)
        self.arr[h] = val
    def findkey(self, key):
        h = self.get_hash(key)
        if self.arr[h] == None:
            return False
        return True
```

→ The ‘MAX’ attribute:

Our hashtable is simply an array of size ‘MAX’ where each cell contains an array that holds the data. We used the ‘MAX’ attribute to limit the size of the hash table and simulate the case of ‘the file being too large that it cannot be held in the RAM all at once’.



→ How the Data Structure Works:

To insert a row in hashtable h1, we pass the id (in the 'key' variable) and the rest of the rest of the row data (as an array variable) to our defined function 'put'. The function hashes the id and stores the data using the hash value as the hashtable index.
Once stored, we can use:

- 'Get_hash' to get the hash value of a key (the id in our case)
- 'Find_key' to check if a cell in the hashtable is empty or not
- 'Get' to get the data corresponding to a certain id.

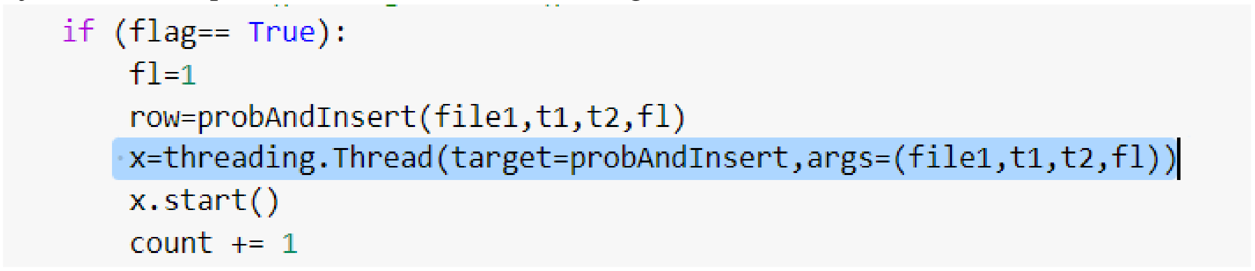
The Algorithm for the Pipelined Double Hashed Join:

→ Reading from the Files:

The algorithm keeps reading alternatively between the two files. It reads one row from each file and inserts it into one of the hashtables (if no join is possible). Once the hashtable reaches its limit, its contents are flushed to disk (in our case: inserted in another table) and it is initialized for further use (in our case, we empty all its cells).

→ The Use of Threads for Probing and Inserting in the Hashtable:

Once data is read from a data source, a thread is started. The thread is responsible for checking if a join should be performed or not, and committing the insert to the database.



```
if (flag== True):
    fl=1
    row=probAndInsert(file1,t1,t2,fl)
    x=threading.Thread(target=probAndInsert,args=(file1,t1,t2,fl))
    x.start()
    count += 1
```

→ Constant Checking of Whether the Hash Tables are Full or Not:
We keep a count of how many rows are inserted in each hashtable. Once it reaches its limit, we write the content of the hash table to another one (to simulate flushing it to disk) and we use the hashtable for new data.
→ Inserting in the Database:
Once a record is read from one of the files, its primary key (the 'id') is checked against the hash table of the other file. If a record with a similar 'id' is found, data from both sources (from the file and from the hashtable) is joined and inserted into Postgres database.
→ Limitation of the join:

- If a new record has an 'id' that has already been joined, that record will be left in the hashtable until it is full and flushed to disk.
- If a record hasn't been joined and the part with which it can be joined arrived way later, we will not be able to join them as the older record would have already been flushed to disk.
    - One way to deal with this is to periodically bring back the hash table that is in disk and check it against the other hash table.

Step 2: Ensure Idempotency of the Pipe:

Failure of Insertion to the Database:

→ Writing to a Log File:
The idea: Do not commit to the database using 'conn.commit()' until you write to a log file.
In other words, just before the 'commit()' line of code, insert the following:

- Open the log file (if not opened)
- Append to it the 'id', which is the primary key.
- Only commit after checking that the write operation was done successfully (using a try-catch statement)

Recovery:
This way, if the network or database connection fails, we would have written in the log the 'id' of the record on which it fails.
To recover, we need to do the following:

- Open the log file
- Read the last line which contains an id
- Look for the id in both hash tables, join the data, and try inserting again

Limitation & Alternative:

- Accessing the log file before each database insert is costly. Therefore, we can choose to append the 'id' to a variable in our code and periodically write the last 'id' value to the log file. This will increase the performance. However, the information will be lost if the server holding the code crashes.

Step 3: Scaleup Scenarios:

→ The Hash Tables get Full:

As discussed previously, we use a max attribute to identify when the hash tables are full, and then we flush the tables to the disk and work with reinitialized ones.

→ Collision:

As the number of records increases, collisions will be more likely to happen. In our program, what will happen is that, when two values collide, the later one will override the first one. However, this is not an optimal situation. One can either implement a more ‘appropriate hash function’ that will make collision less likely or use chaining or open addressing.