CSC 5356 01 - Data Engineering and Visualization

Double pipelined Hash join

Teammates: Zakaria elaouene and Imane Boudrâ

Intro:

In this project we use postgres database; we can see the limitations of the relational approach in comparison of the no SQL approach, it has a lot of issues cited below

double pipelined hash join :

Steps:

- Read a row in the first csv file (file1)

- Compare the index of this row with all the indexes of the hash table of the second file (file2): t2

- If there is a match, join the information of that input (file1) and the hash table of the file2 (t2) into the database

- Whether there is a match or not, insert that input into the hashtable of the file1: t1

- Then read the next input but this time of the file 2, and redo the same step, which means to compare the index with t1, and join if there is a match, after that insert that input of file2 into the hashtable t2

Code that illustrates those steps:

1. We define a hashtable as a class, where the attributes are a table of size 10000

Our hashtable is an array with an index (key) and value

We define 4 functions in our class:

-The one that will add value to the table: set

-The one that will give us the value when we provide a key: get

-The one that will hash the keys, we will use it inside get and set also to hash the key before looking for it in the array

-The one that will search is a key exists or not in a table, it will return false if not

```python
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
```

2. we define a function doublepipe():

In this function, we will implement the double pipelined hash join method, we will iterate to read the files until we reach the end of file. We need to go back and forth between the two csv files to read the inputs asymmetrically, so we will use a flag that after each iteration between the csv files will take the true and false value. When the flag is true, the function will process the first file, when it is false, it will process the second file

```
    #while f.hasnext() and g.hasnext():
        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
            if (count==10000):
                #flush to dik
                t1=[0]*10000
        else:
            fl=2
            row2=probAndInsert(file2,t2,t1,fl)
            y=threading.Thread(target=probAndInsert,args=(file2,t2,t1,fl))
            y.start()
            count2 += 1
            if (count2==10000):
                #flush to dik
                t2=[0]*10000
        flag = not flag
```

In each iteration, the files should be read, compared, joined and their inputs inserted, this will lead us to the next step

3. probAndInsert()

In this function, we read the row of the file and we compare the index to the hashtable of the second file, if they match we join

```
with open(fil) as fil:
        csv_reader = csv.reader(fil, delimiter=',')
        row = next(csv_reader)
        index1={row[0]}
        column2={row[1]}

#prob
    #hashIndex=t1.get_hash('index1')
    if(t2.findkey('index1')!=False):
d = {'id': [index1], 'firstname': [column2], 'city': [t2.get('index1')]}
        df = pd.DataFrame(data=d)
        df.to_sql('HashJoin',con=conn, if_exists='append', index=False)
        conn = psycopg2.connect(conn_string)
        conn.autocommit = True
```

and then we insert into the input hashtable

```
 t1.put('index1',column2)
```

Assessing problems:

- To avoid waiting for the whole file to be inserted into the hashtable before doing the next operations, we iterate from one file to another, and this is the principle of the double pipelined hash join

- Even with this solution, we still have to wait for the index to be compared with all the indexes of the hashtable, so we used threads for the probAndInsert() function:

```
x=threading.Thread(target=probAndInsert,args=(file1,t1,t2,fl))
x.start()
```

- Ram issue: at some point, the ram on which are saved the hashtables will be full and the system will crash, the solution is that our hashtable is an array of size 10000 and we have a counter that add +1 whenever we insert an input into the hashtable, when the limits are reached, we flush the hashtable into a disk and we make it persistent. This solution is limited because if an index need to be compared with an index already persisted into disks, it will not find it

- Our hash function can lead to collisions:

```
def get_hash(self, key):
        hash = 0
        for char in key:
            hash += ord(char)
        return hash % self.MAX
```

Which means two indexes can have the same hashed function. In this case there is the probing solution