جامعة الأخوين

ⵜⴰⵙⴷⴰⵡⵉⵜ ⵏ ⵓⵅⴰⵡⴰⵢⵏ

AL AKHAWAYN UNIVERSITY

School of Science & Engineering

Data Engineering and Visualization

CSC 3356 **SPRING 2022**

Project #2

By

Zakaria Elaouene

Israe Abdellaoui

and Imane Boudrâ

Project statement:

In this project you are required to build on project # 1 as follows:

a. Build the join using pipelined double hashed join for scalability.
b. Ensure idempotency of your pipe.
c. Identify scaleup and idempotency testing scenarios and run them, the comment the results.

What is pipelined double hashed join?

Double-pipelined hashed joins are a kind of join that calculates the joins on both sides of the hash table at once. The only reason it is not more popular is because it requires an extra level of indirection, and usually one additional disk cache as well. Otherwise, any other possible advantages on seek times or CPU loading are lost, making double-pipelined hashed join a poor optimization for most tables.

Even though double-pipelined hashed join is not a very effective form of optimization, it is an interesting example of how to optimize joins.

A standard way to process a join is to create the hash table and then process the table in order from left to right. the bottleneck is when the join algorithm processes half of the table and then reaches a row where one or more key values are not yet in the hash table, the table is currently half empty.

In this case, after loading the half-empty row, the left branch of the join algorithm must load all rows from the right branch that are not yet in the hash table. These rows have to be loaded together with all other remaining rows for those keys, because if non-matching rows are loaded separately from their matching counterparts, then the algorithm will have to iterate through those rows again. This can result in significant performance loss on large tables.

As a result, the bottleneck of double-pipelined hashed join is when half of the table is processed. The solution is to check if there are any unprocessed rows when the join algorithm reaches the middle of its progress. If there are, then both branches can continue processing at once and load matching rows separately. This way, no one must load non-matching rows again after they have been loaded together with their matching rows.

How to ensure idempotency of your pipe?

If a given pipeline operation generates data that may change over time, then it is important to ensure that the pipeline is idempotent. This means that replicating the

operation repeatedly should generate exactly the same result. And this can be achieved by designing pipelines, so they are always run against fresh data.

A great example of that is processing log data. A log file is a constantly growing series of messages. If you want to continuously process those messages to provide insights, then you need to ensure that your pipeline is idempotent: if you process two different log files in the same way, should they give the same results or not?

If you don't care about what happens to the message after you have processed it, then it doesn't matter how many times you create and process them – they will always be identical. On the other hand, if you do care about the message's life cycle “perhaps because you process a message and then delete it, or you are generating a new message based on what was processed” then reuse of the same message may lead to different results. If that happens, then those results are not idempotent.

That's why idempotent operations are important, they prevent re-run of an operation on a potentially different result.

1- Hashing the files in python:

Hashing is the process of taking input, running it through an algorithm, and converting it back into usable data. When hashing, you essentially take a string of text and statistically match each character with an empty slot in a hash table.

For us, we need to hash file names, or in other words, process a list of filenames into a single string. There are many ways to do this using a simple Python script.

```
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
```

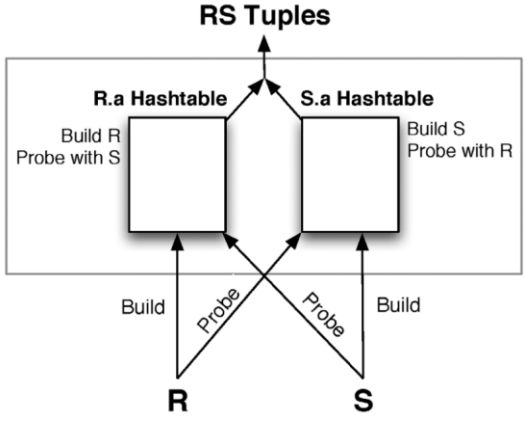
2- Double pipe function:

The function doublepipe() create two strings to hold a portion of the column names from each file. This will help us identify and merge the two datasets together as we iterate through both files asymmetrically depending on the flag.

```
    #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
```

3- probAndInsert function:

this function is used for both inputs it always puts the tuple into the hash table belonging to the specific input and probes the other inputs if the input tuple is say from **R** then it insert into **R** and probe **S** and vice versa, if it's element from **S** we insert into **S** and probes **R,** of course the order here doesn't matter you can whether we first insert and then probe or the other way around.



This algorithm is done using a simple python script:

```
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 finally, we add the results using:

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

The file used for this project are the following:

- myFile0.csv

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
```

- myFile1.csv

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
```