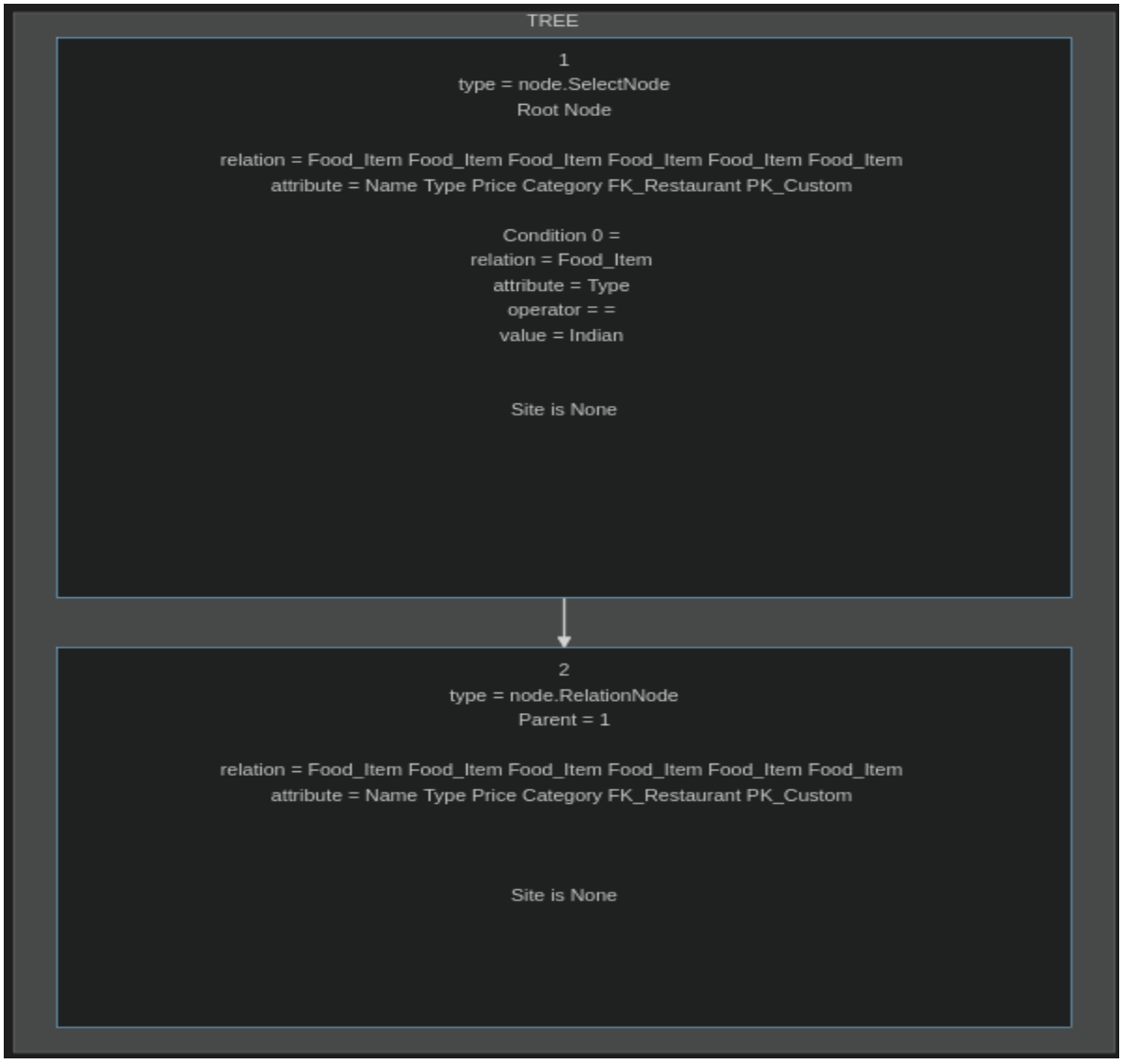
Report Phase 2:- Outlaws [Kartik Garg (2019101060)]

Optimization Algorithms/Heuristics Used

Heuristics

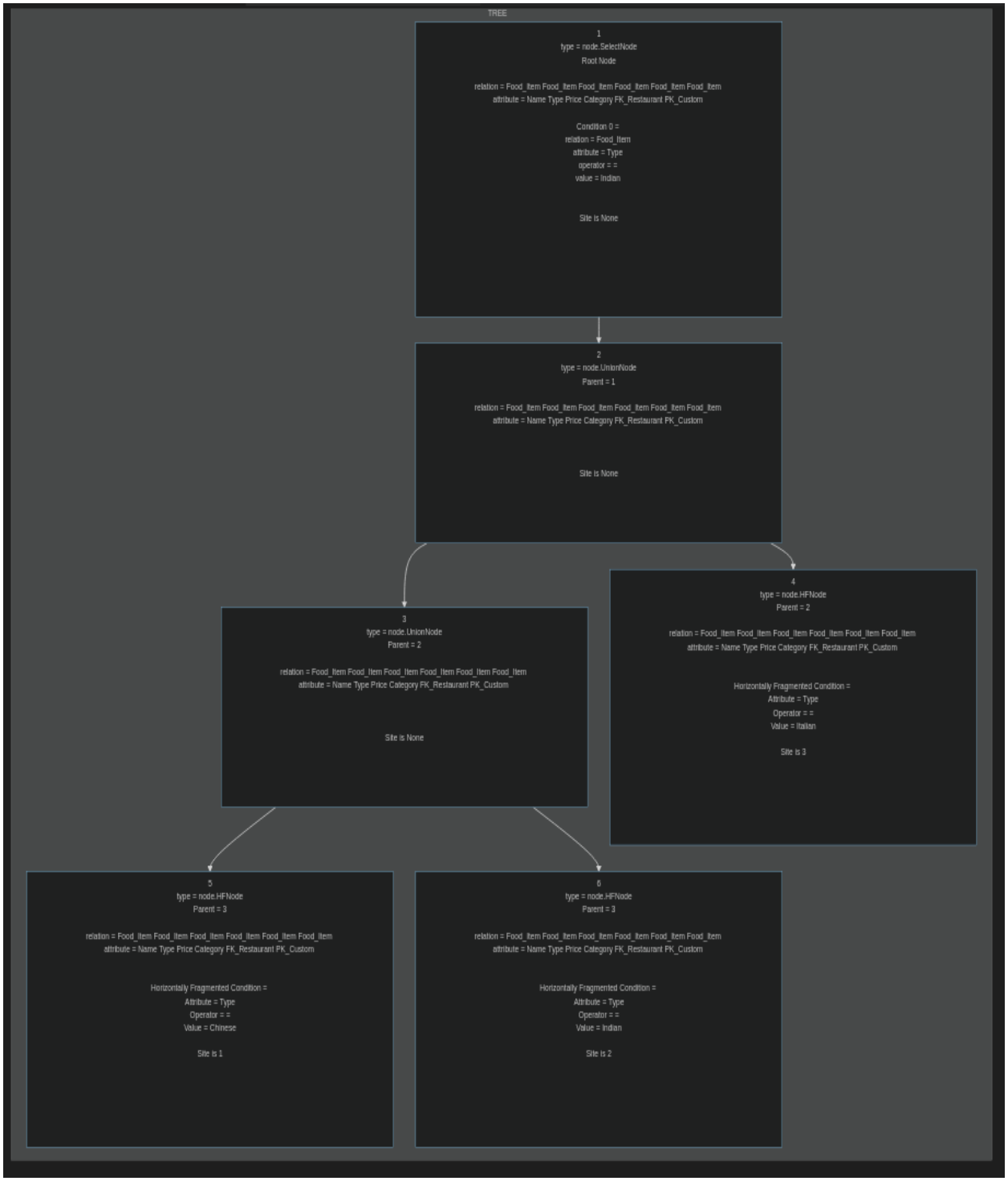
Standard heuristics like pushing the projects and selects down the tree are already present. And not just pushing down for each select condition we also check whether the relation/fragment node on which that select is going to sit on top can logically have the answer or not. So I have a method implemented in my Query class called checkCondition which basically takes two conditions and compares them and outputs whether they can have tuples which are common to both. So if we encounter a condition which is logically contradicting the one which we have for, lets say, Horizontal Fragment Node. In this case we will eliminate this fragment straight away as it cannot have any result tuples inside it. So, this kind of optimization is there.



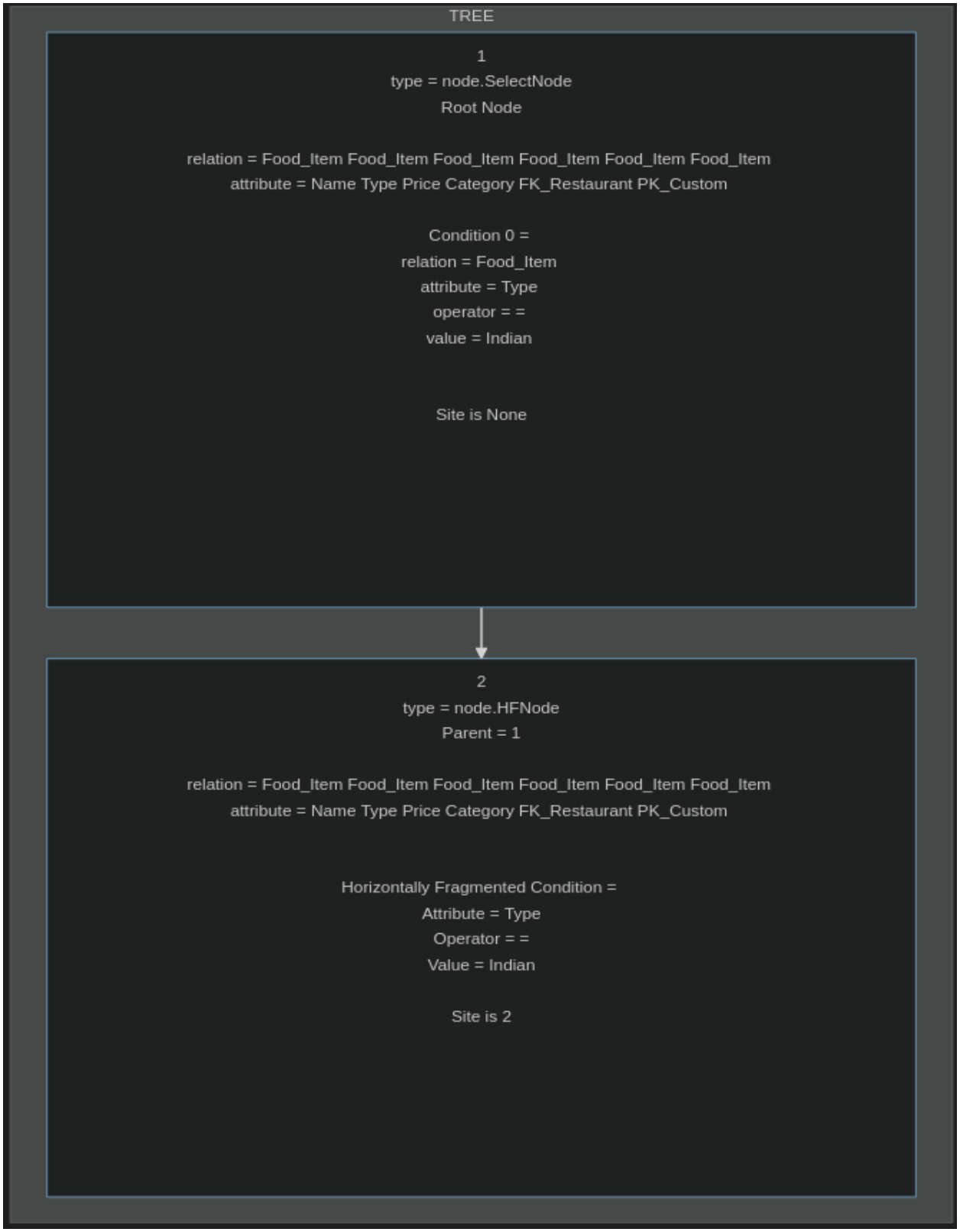
Above attached is the original graph which is made for the query

Select * from Food_Item where (Type='Indian')

When we break relations into fragments it becomes like the one whose image is on the next page.



But after passing the select condition down, it becomes a lot more simpler graph, image attached on next page


TREE
1
type = node.SelectNode
Root Node
relation = Food_Item Food_Item Food_Item Food_Item Food_Item Food_Item
attribute = Name Type Price Category FK_Restaurant PK_Custom
Condition 0 =
relation = Food_Item
attribute = Type
operator = =
value = Indian
Site is None
2
type = node.HFNode
Parent = 1
relation = Food_Item Food_Item Food_Item Food_Item Food_Item Food_Item
attribute = Name Type Price Category FK_Restaurant PK_Custom
Horizontally Fragmented Condition =
Attribute = Type
Operator = =
Value = Indian
Site is 2

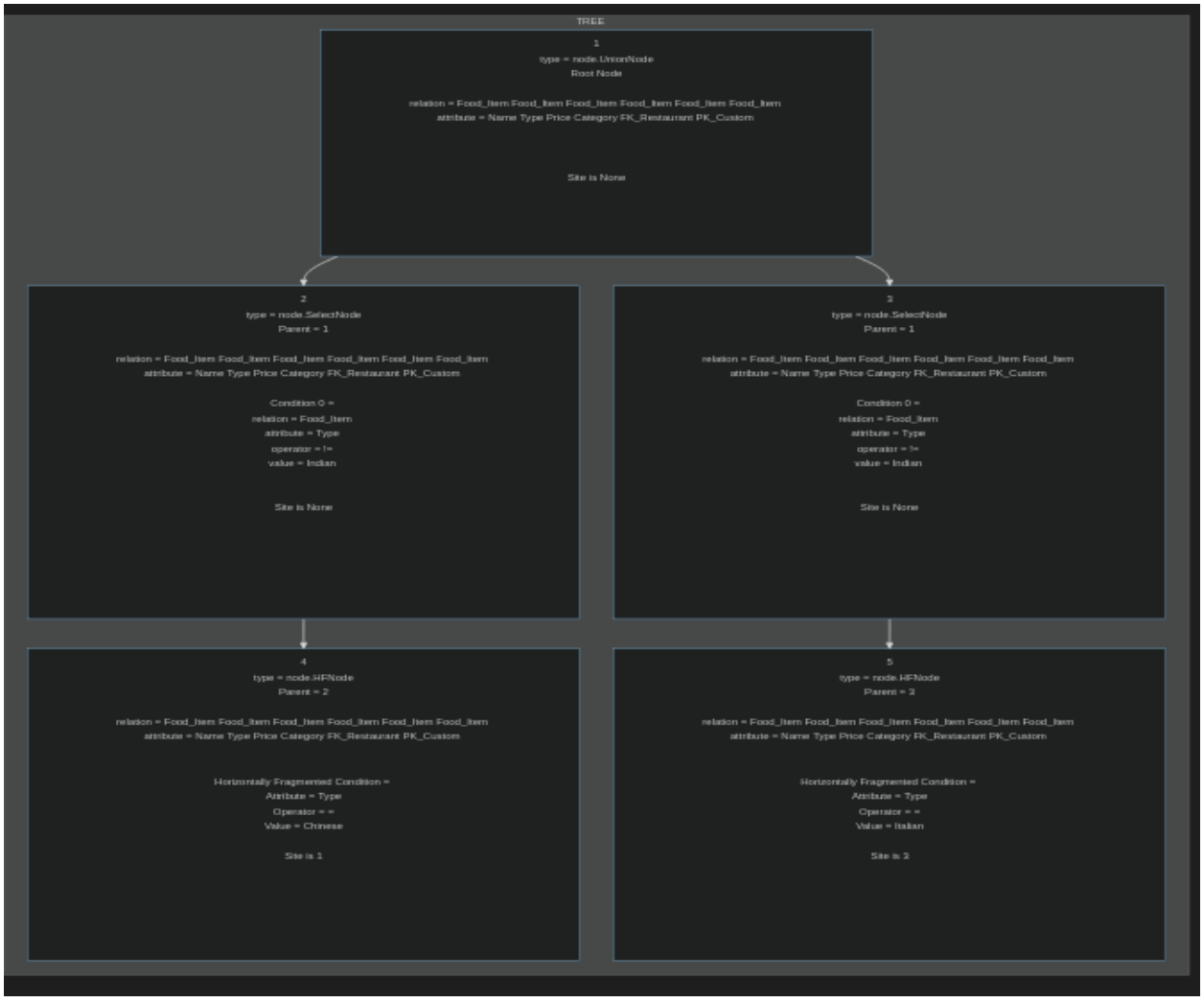
Because we were able to identify from the conditions and decide which fragment will have the final result, we were able to transform a much complex query tree to the one which is much simpler and contains only two nodes.
Above query is executed and gives the following output

```
->> select * from Food_Item where (Type='Indian');

                Name    Type  Price Category FK_Restaurant PK_Custom
0  Special_Shawarma  Indian    100  Non-Veg           AAI        AAP
1   Cheese_Shawarma  Indian     90      Veg           AAI        AAQ
2      Honey_Chilli  Indian    120      Veg           AAK        AAV

Time_Taken (In Seconds): 1.8483760356903076
```

If the query is changed to a query

Select * from Food_Item where (Type!=’Indian’)

Now, the final query tree contains the other two fragments, like shown below



And the Output goes like this,

```
->> select * from Food_Item where (Type!='Indian');

             Name     Type  Price Category FK_Restaurant PK_Custom
0           Momos  Chinese     40      Veg           AAJ       AAL
1      Manchurian  Chinese     50      Veg           AAK       AAM
2         Noodles  Chinese     45      Veg           AAK       AAN
3        Chowmein  Chinese     70      Veg           AAK       AAO
4       Marghrita  Italian    100      Veg           AAH       AAR
5  Non_Veg_Loaded  Italian    150  Non-Veg           AAH       AAS
6      Veg_Loaded  Italian    125      Veg           AAH       AAT
7           Pasta  Italian     95      Veg           AAH       AAU

Time_Taken (In Seconds): 4.057796955108643
```

So, this was all about the basic heuristics. Here we pushed the selects and projects down and removed the irrelevant fragments which did not contribute to our final result.

Optimizations

First optimization which we have in our code is that, before making the query graph we find the best join order based on the selectivities provided to us. What we do is, we try all the join orders which are possible for a query and find out the one which has the minimum cost. The code snippet for the same is attached below. Here, we pass a join order to this function and we calculate the cost associated with it. We keep track of which join order gave me the minimum cost and we proceed with selecting that join order.

```
def calcCost(self,joinOrder,relations):
    '''
    Given a join order, join selectivities and relation sizes
    this function is supposed to calculate the cost of their joining
    '''
    cost = 0
    relation_sz = {}
    for x in relations:
        relation_sz[x] = config.relationNumEntries[x]

    for x in joinOrder:
        r1 = x[0]
        r2 = x[0]

        joinSel = config.joinSelectivities[(r1,r2)]
        sz1 = relation_sz[r1]
        sz2 = relation_sz[r2]
        nusz = sz1*sz2

        cost = cost + nusz
        relation_sz[r1] = nusz*joinSel
        relation_sz[r2] = nusz*joinSel
    return cost
```

Other optimizations which make use of join selectivity factors and the allocation information are deployed while handling joins and unions.
So, in **unions** we check the length of the two relations which are taking part in this union. And we will ship the one with fewer rows to the relation with more rows. This adds up to our **second optimization**.
For shipping the relations a helper utility file is written which has 4 functions in it which are dumpTable, copyFromServer, copyToServer, importTable. The first utility function dumps a table as a text file which contains sql statements to generate the exact same table. The second function brings this txt file to my machine. The third function will copy this .txt file from my machine to the server which will be specified. The last utility function will basically import the table from this txt file into the mysql server.
For **optimizing the joins**, we have made use of semi-joins as specified in the description document. So we have to first find out which relation should be shipped to the site of which relation (given both the relations are in separate sites, if they are on the same site then we just execute the normal inner join). To find out this we have designed some preprocessing functions which extract certain information about all the relations which is necessary to calculate the cost.

```
def dumpTable(relname,siteno,ofile = 'dump.txt'):
    '''
    Function which will dump a mysql table using mysqldump command into ofile
    '''
    config.logger.log("utility::dumpTable")

    command = "echo 'use "+config.catalogName+";' > ./Outlaws/"+ofile
    (a,b,c) = config.paramikoConnections[siteno].exec_command(command)
    op = c.read()

    command = "mysqldump -u user -piiit123 "+config.catalogName+" "+relname+" >> ./Outlaws/"+ofile
    # print(command)
    (a,b,c) = config.paramikoConnections[siteno].exec_command(command)
    op = c.read()
    # print(op)
```

```
def copyFromServer(siteno,ofile = 'dump.txt'):
    '''
    Function to copy a file from the Server to my machine
    '''
    config.logger.log("utility::copyFromServer")

    with SCPClient(config.paramikoConnections[siteno].get_transport()) as scp:
        scp.get("./Outlaws/"+ofile,ofile)

def copyToServer(siteno,ofile = 'dump.txt'):
    '''
    Function to copy a file from my machine to Server
    '''
    config.logger.log("utility::copyToServer")

    with SCPClient(config.paramikoConnections[siteno].get_transport()) as scp:
        scp.put(ofile,"./Outlaws/"+ofile)
```

```
def importTable(siteno,ofile = 'dump.txt'):
    '''
    Function to read the dump.txt file
    '''
    config.logger.log("utility::importTable")

    command = "mysql -u user -piiit123 < ./Outlaws/"+ofile
    (a,b,c) = config.paramikoConnections[siteno].exec_command(command)
    op = c.read()
```

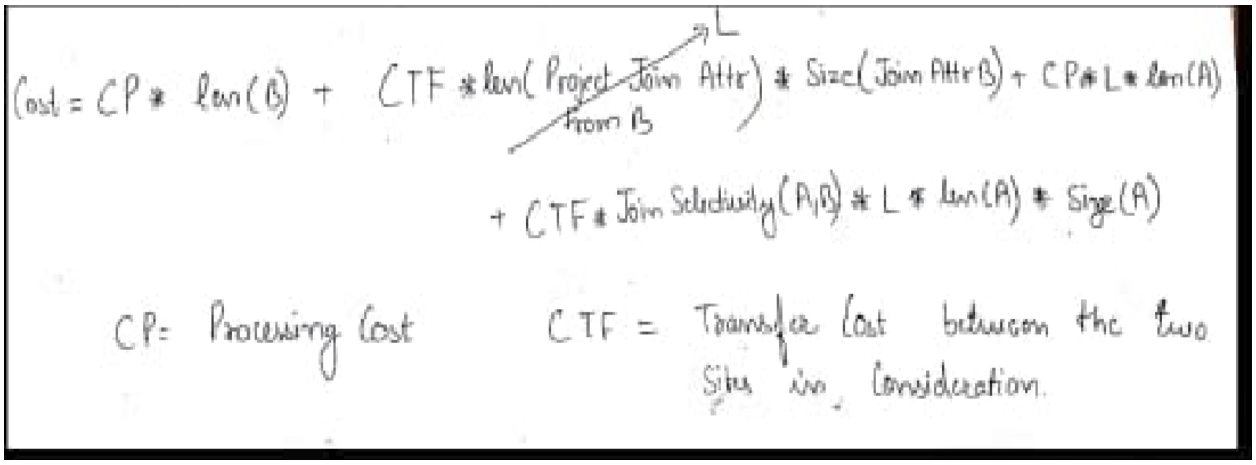
There are mainly two preprocessing function which are:

1. getEntrySizes()
2. getRelationLengths()

Both these functions communicate with each of the server sites and then using the sql's information_schema we get the relation length, in this case it can also be fragments, and the individual column sizes. From individual column sizes we can calculate the tuple size by adding all the size values for each of the column values

And the relation lengths are directly stored in the information_schema.TABLES relation and are retrieved from there only.

Now when we calculate the cost of A join B as (B semi join A) join A, we do it by using the formula given below

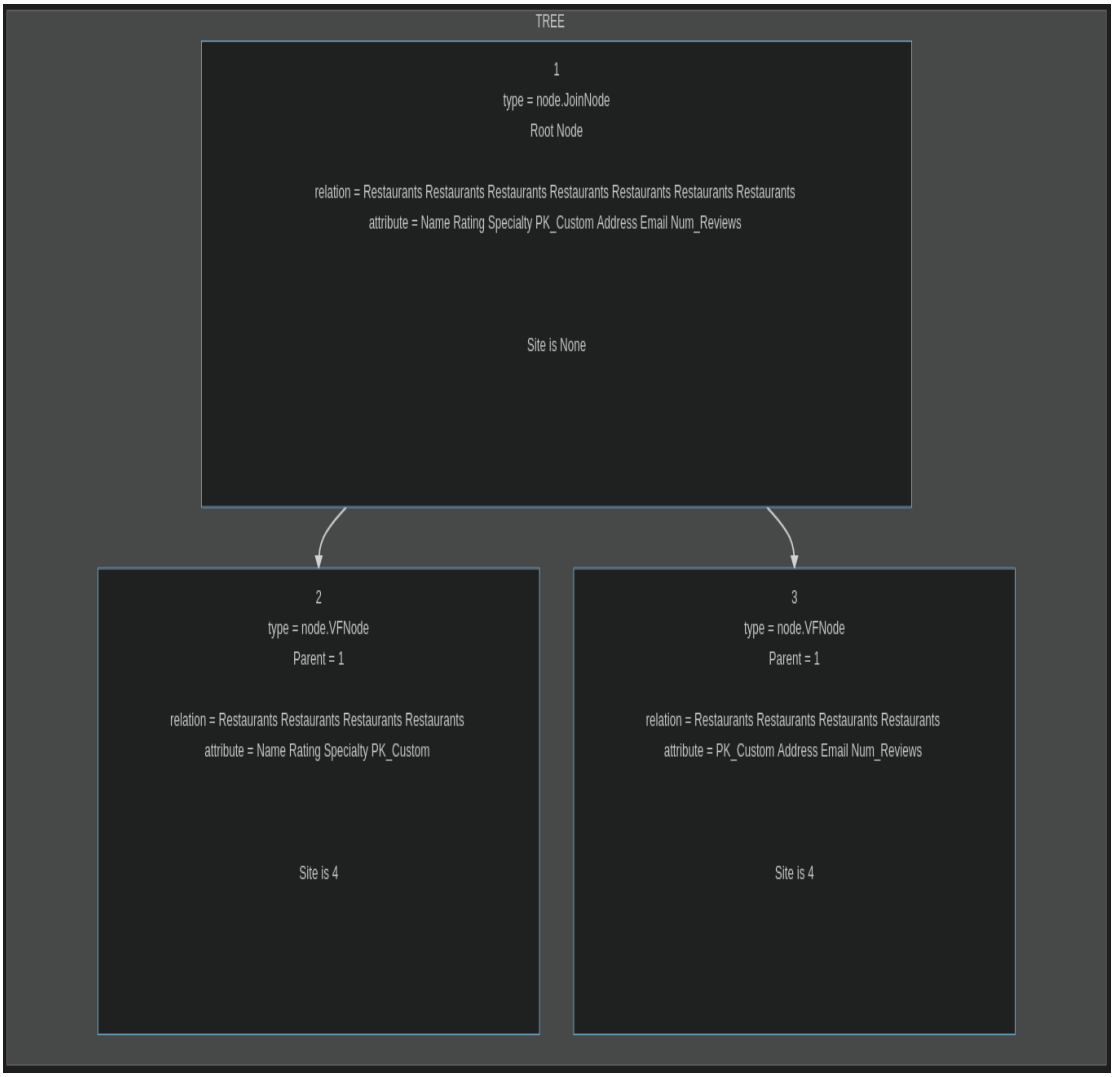


In a similar fashion we calculate the (A semi join B) join B variant and we pick the choice which gives us the minimum cost and we execute it. This was all about our **third optimization**.

Optimized Query Tree and Final Output

It is already attached in the above document for some of the queries. I am also attaching it below in this section for a couple of more queries which will involve use of joins and aggregate operators.

1. Select * from Restaurants;
    a. Query Tree

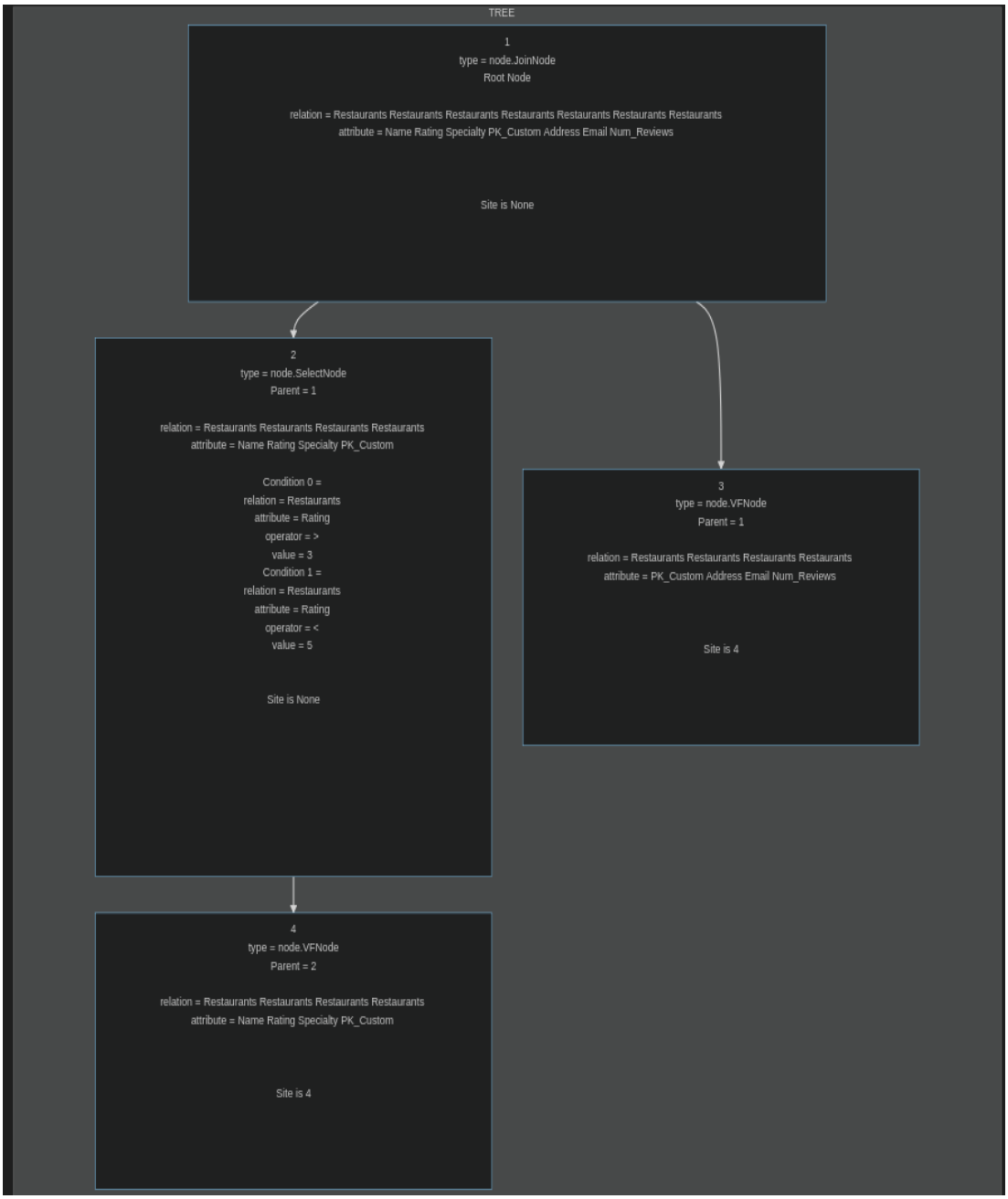


    b. Result

```
->> select * from Restaurants;

         Name  Rating   Specialty PK_Custom       Address           Email  Num_Reviews
0     Dominos       4       pizza       AAH    Gachibowli  domi@gmail.com           45
1  SKShawarma       3    shawarma       AAI           DLF   sks@gmail.com           38
2    WOWMomos       5       momos       AAJ  Indira_Nagar    wm@gmail.com           43
3          h9       4   manchurian       AAK   Garden_Road    h9@gmail.com           50

Time_Taken (In Seconds): 0.9893982410430908
```

Both the Restaurants Vertical fragments are stored at the same place and they are JOINED before proceeding further. This is an application where JOINS are used.

2. select * from Restaurants where (Rating>3) AND (Rating<5);
    a. Query Tree



    b. Result

```
->> select * from Restaurants where (Rating>3) AND (Rating<5);

      Name  Rating    Specialty PK_Custom      Address           Email  Num_Reviews
0  Dominos       4        pizza       AAH   Gachibowli  domi@gmail.com           45
1       h9       4   manchurian       AAK  Garden_Road    h9@gmail.com           50

Time_Taken (In Seconds): 1.2753527164459229
```

An application of WHERE clause

3. select count(Name),sum(Num_Reviews),Rating from Restaurants Group By Rating;
    a. Query Tree
       The file thirdQuery.md contains the query graph
    b. Result

```
->> select count(Name),sum(Num_Reviews),Rating from Restaurants Group By Rating;

   COUNT_Name  SUM_Num_Reviews  Rating
0           2             95.0       4
1           1             38.0       3
2           1             43.0       5

Time Taken (In Seconds): 2.1644561290740967
```

Application of GROUP BY clause

4. select count(Name),sum(Num_Reviews),Rating from Restaurants Group By Rating Having (count(Name)>0) AND (sum(Num_Reviews)>40);
    a. Query Tree
       The file fourthQuery.md contains the query graph
    b. Result

```
->> select count(Name),sum(Num_Reviews),Rating from Restaurants Group By Rating Having (count(Name)>0) AND (sum(Num_Reviews)>40);

   COUNT_Name  SUM_Num_Reviews  Rating
0           2             95.0       4
1           1             43.0       5

Time Taken (In Seconds): 2.227605104446411
```

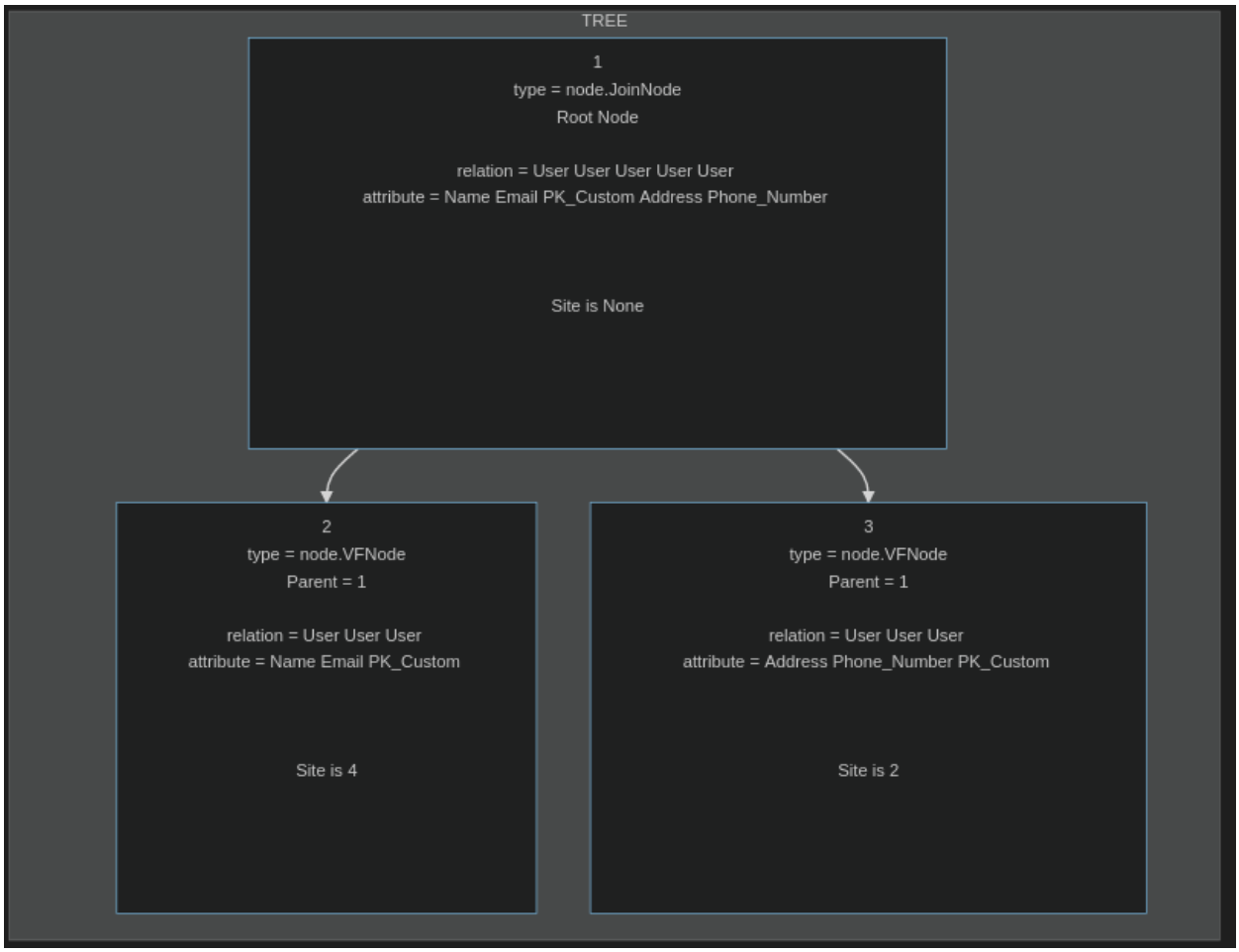
Application of HAVING along with GROUP BY

5. select count(Name),sum(Num_Reviews),Rating from Restaurants where (Num_Reviews>43) Group By Rating Having (count(Name)>0) AND (sum(Num_Reviews)>40);
    a. Query Tree
       The file fifthQuery.md contains the query graph
    b. Result

```
->> select count(Name),sum(Num_Reviews),Rating from Restaurants where (Num_Reviews>43) Group By Rating Having (count(Name)>0) AND (sum(Num_Reviews)>40);

   COUNT_Name  SUM_Num_Reviews  Rating
0           2             95.0       4

Time Taken (In Seconds): 3.6412532329559326
```

Application WHERE along with GROUP BY and HAVING clause

6. Select * from User;
    a. Query Tree



    b. Result

```
->> select * from user;

      Name              Email PK_Custom   Address Phone_Number
0    Kartik  krtikgrg@gmail.com       AAA     Sunam   9478077895
1  Aaradhya     aardg@gmail.com       AAB     Delhi   9971352631
2  Priyansh         p@gmail.com       AAC     Delhi   9876543210
3   Harshit         h@gmail.com       AAD  Amritsar   1234567890
4   Aaditya         a@gmail.com       AAE    Jaipur   4561237890
5  Shreyash         s@gmail.com       AAF     Delhi   6543219870
6     Ayush        ag@gmail.com       AAG    Jaipur   7896541230

Time_Taken (In Seconds): 3.6980063915252686
```

This is an application of inter-site Join, because the fragments for the User table is stored in two separate sites.

Directory Structure

changes.md => File mentioning the changes that have been made since Phase1
complete.md => File representing the final query tree after localization
config.py => File maintaining some global variables which in some sense form the configuration of the system.
data.txt => data that has been added to the sql tables at each site
documentation.md => This file mentions the assumptions that were made while writing the code and also contains some other little details of the code.
executor.py => This file contains the executor class which does nothing but just iterates over the query tree using **DFS** and executes each node.
general_query_syntax.txt => It is a common file that we maintain to help us in coding basically to write notes or points whichever are required.
initial_query_terminal.png => This is a screenshot which shows some commands which were run in succession in the same terminal session.
input.py => File responsible for managing the inputs which we receive in the code.
logs.txt => File which maintains the record of all the functions which were called during the execution of a query. It is a common file for all the queries which are run in a single session.
node.py => File containing the parent class node and all the inherited classes which make it easier to construct and execute the query in form a nice tree structure.
optimization.py => This file contains the optimizer class, which contains nothing but all the functions which are optimizing my query execution.
Post_query_terminal.png => A screenshot in succession to that of initial_query_terminal.png
preprocess.py => File containing the functions which will read data from both application and fragmentation schema. It will further process the fragmentation schema to our chosen form. The file also contains other functions now which help us to retrieve the number of tuples in each relation along with each column's size for each relation. There are a couple of more functions which are present in this file.
query.py => File processing the functions of a particular query, all the functions to push down selects and projects including the execution (which will in turn call the executor class method) are present in this file.
server.py => File containing the main loop.
trace.py => File responsible for writing the logs.txt
utility.py => File containing the functions to transfer a relation from one site to another site
report.pdf => File which contains the explanation of the code and the directory structure