Report (Phase 2) - Outlaws

By:- Kartik Garg(2019101060)

Task 1: Parsing

For handling basic initial parsing I have used the **sqlparse** module available in python. This module basically takes a sql query as input and will add appropriate new lines and spaces wherever necessary. This library will also capitalize all the keywords used in the given query. For instance a query given as:

“SELECT col1, t1.col2 , col3 ,col4,col5 FROM t1,t2 , t3 , t4 WHERE (a=b) aNd (c=d) and (e=f) AND (a<5 OR a>10) AND (c<5) AND (d<5 OR e>10 OR f>=15) GROUP BY c2, c3, c4 , c5 HAVING (count(c2)>0 OR Sum(c3)=5) AND (MIN(c5)>10);”

Will be parsed as a single string (given below)

```
“SELECT col1,
       t1.col2,
       col3,
       col4,
       col5
FROM t1,
     t2,
     t3,
     t4
WHERE (a=b)
  AND (c=d)
  AND (e=f)
  AND (a<5
       OR a>10)
  AND (c<5)
  AND (d<5
       OR e>10
       OR f>=15)
GROUP BY c2,
         c3,
         c4,
         c5
HAVING (count(c2)>0
        OR Sum(c3)=5)
AND (MIN(c5)>10)“
```

After above kind of parsing, I have a self written **parse function** in **Query** class, which will basically break down this query further and will extract the relevant parts of like what all joins have been mentioned, what all column do i need to extract and everything i need ranging from mere Table Name extraction to the extraction of aggregate operators. The above mentioned query will be parsed into following data structure (data is also shown along with them)

```
relations ['t1', 't2', 't3', 't4']                    // list to store all the relations
                                                       Mentioned in the query (FROM
                                                       clause)

join_conditions {
    'relation1': ['t1', 't4', 't3'],
    'attribute1': ['a', 'c', 'e'],
    'operator': ['=', '=', '='],
    'relation2': ['t4', 't3', 't2'],
    'attribute2': ['b', 'd', 'f']
}                                                      // this is representing 3 join conditions
                                                       Which are t1.a=t4.b ; t4.c=t3.d ;
                                                       t3.e=t2.f
                                                       Extracted from WHERE clause
                                                       conditions, when both the operands
                                                       were some variable name instead of a
                                                       literal value
all_projects {
    'attribute': ['col1', 'col2', 'col3', 'col4', 'col5', 'c2', 'c3', 'c4', 'c5', 'c2', 'c3', 'c5'],
    'relation': ['t2', 't1', 't4', 't3', 't3', 't1', 't1', 't1', 't2', 't1', 't1', 't2'],
    'aggregate_operator': ['', '', '', '', '', '', '', '', '', 'COUNT', 'SUM', 'MIN']
}                                                      // This dictionary stores all the attributes
                                                       that we will need to project. All the
                                                       attributes mentioned here are extracted
                                                       from 3 clauses which are SELECT ,
                                                       GROUP BY and HAVING
aggregates {
    'attribute': ['c2', 'c3', 'c5'],
    'relation': ['t1', 't1', 't2'],
    'operator': ['COUNT', 'SUM', 'MIN']
}                                                      // This dictionary stores the aggregate
```

operators which have been specified throughout the query in various clauses which are **SELECT** and **HAVING**

```
group_by {
    'attribute': ['c2', 'c3', 'c4', 'c5'],
    'relation': ['t1', 't1', 't1', 't2']
}
```

// Dictionary storing the attributes mentioned in **GROUP BY** clause

```
select_conditions [
    {
        'relation': ['t1', 't1'],
        'attribute': ['a', 'a'],
        'operator': ['<', '>'],
        'value': [5, 10]
    }, {
        'relation': ['t4'],
        'attribute': ['c'],
        'operator': ['<'],
        'value': [5]
    }, {
        'relation': ['t3', 't3', 't2'],
        'attribute': ['d', 'e', 'f'],
        'operator': ['<', '>', '>='],
        'value': [5, 10, 15]
    }
]
```

// List of Dictionaries to store the select conditions except the join conditioned mentioned in the **WHERE** clause

Each Item in the list is a separate condition and all these items/conditions were given in **conjunction (AND)** in the original query

Each dictionary/element of the dictionary represents conditions which have been mentioned using **ORs**.

Above two points help to represent the CNF form of the conditions

So this list of dictionaries represent the CNF

```
having_select [
    {
        'aggregate_operator': ['COUNT', 'SUM'],
        'relation': ['t1', 't1'],
        'attribute': ['c2', 'c3'],
        'operator': ['>', '='],
        'value': [0, 5]
    }, {
        'aggregate_operator': ['MIN'],
        'relation': ['t2'],
        'attribute': ['c5'],
        'operator': ['>'],
        'value': [10]
    }
]
```

// List of dictionaries to store the Conditions mentioned in the **HAVING** clause
The structure is similar to the **select_conditions**, just a new attribute in each individual item is added which is to store which aggregate operator was specified in that particular condition

```
project {
    'attribute': ['col1', 'col2', 'col3', 'col4', 'col5'],
    'relation': ['t2', 't1', 't4', 't3', 't3'],
    'aggregate_operator': ['', '', '', '', '']
}
```

// This store the final attributes that we need to project from the query (specified using the **SELECT** keyword)

Above all are the data structures that have been used by the parse function in order to extract all the relevant information related to the query.
Furthermore, i have stored some **META VARIABLES** which will store information like if SELECT * was mentioned then in that case i need to select all the columns and this is represented by the meta variable PROJECT_ALL_ATTRIUBUTES and if this happens then its value is set to 1

META VARIABLES for the given query are listed below along with their values
PROJECT_ALL_ATTRIBUTES = 0
HAVING_CLAUSE_EXIST = 1

HAVE_AGGREGATES = 1
GROUP_BY_CLAUSE_EXIST = 1
PART_ONE_PROJECT_ALL = 0
SELECT_ALL = 0
HAVE_JOIN = 1

With all above mentioned things, we are done with TASK 1 that is parsing

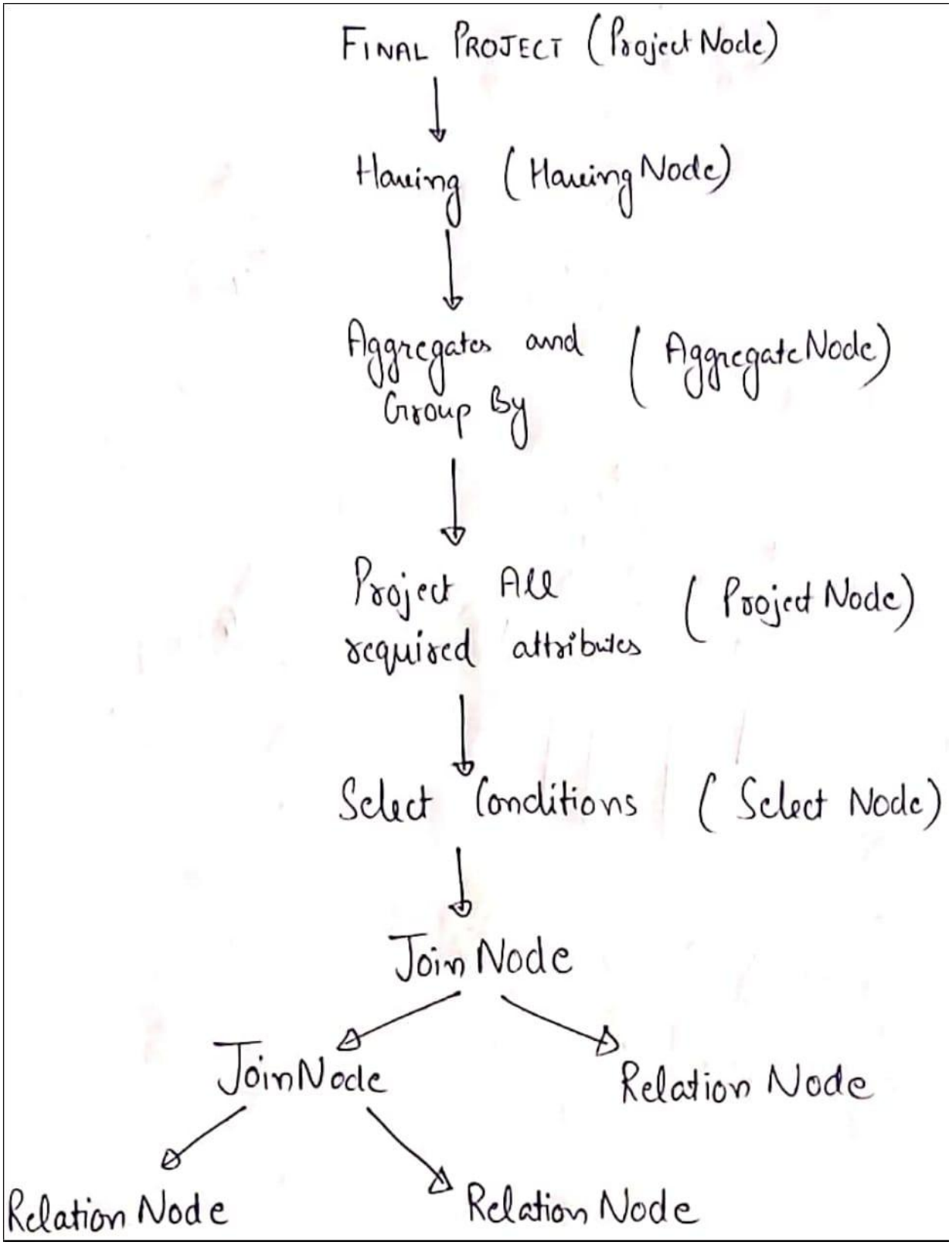
Task 2:Decomposition

Now, for task 2 we have to make a tree structure for which i have created a class Node which has some more classes which basically inherit from this Node class. These different parent and child classes will be used to represent the tree structure.

To visualize the Tree made, i have used mermaid graphs

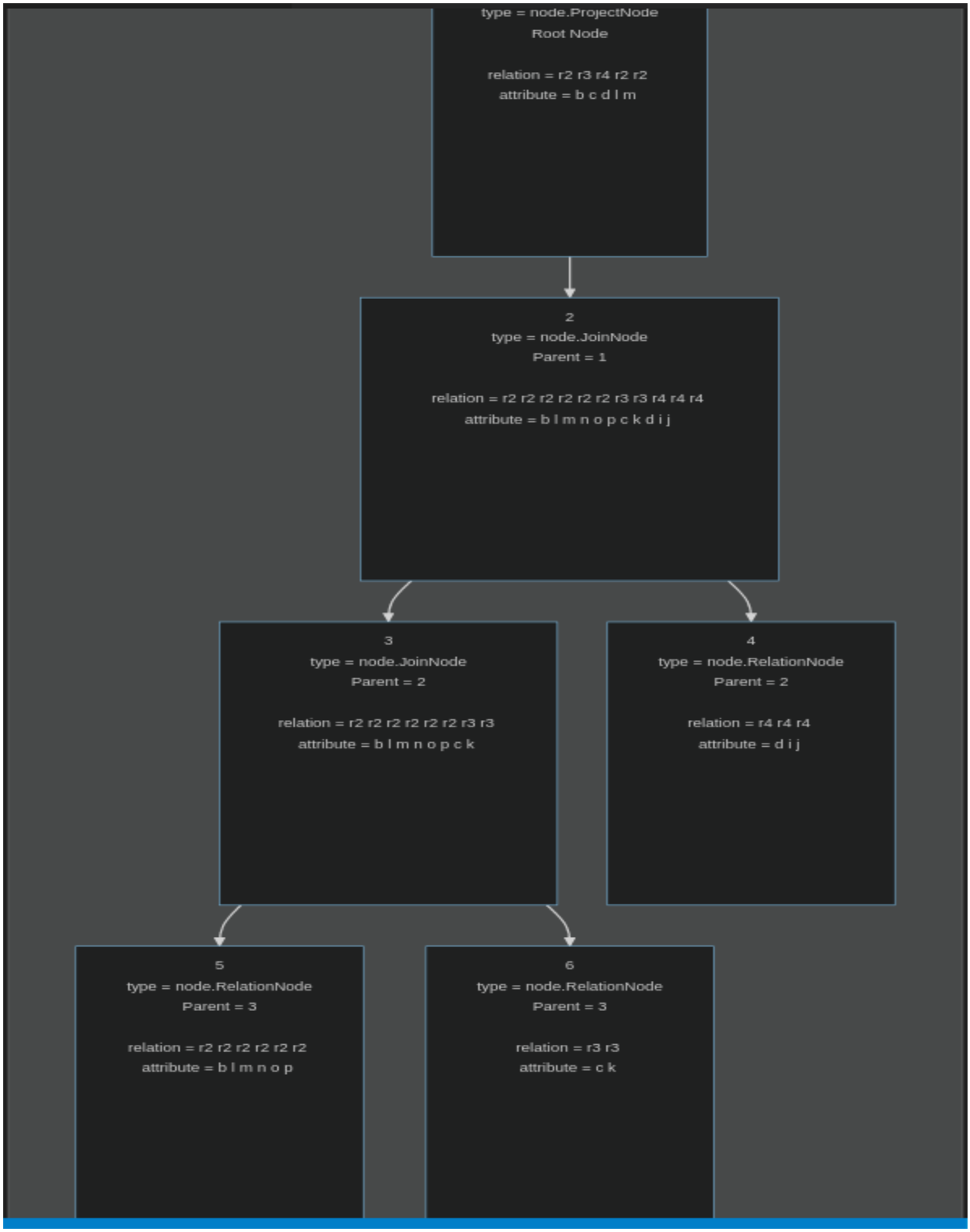
Class structure
| Node
--------->|| ProjectNode // Node to represent the PROJECT operation
--------->|| SelectNode // Node to represent the SELECT operation
--------->|| HavingNode // Node to represent the HAVING operation
--------->|| AggregateNode // Node to represent all the AGGREGATES and GROUP BYs
--------->|| JoinNode // Node to represent the JOINs
--------->|| UnionNode // Node to represent the UNIONs, used in HF and DHF
--------->|| HFNode // Node to represent a Horizontal Fragment in HF and DHF
--------->|| VFNode // Node to represent a Vertical Fragment in VF
--------->|| RelationNode // Node to represent the Initial Non-Fragmented Relations

I then start the generation of the Tree using the **generateTree** function in the **Query** class. I proceed on by first creating the leaves which will be nothing but Non-Fragmented relations which are specified in the query. A general structure of the tree is attached below


FINAL PROJECT (Project Node)
Having (Having Node)
Aggregates and Group By (Aggregate Node)
Project All required attributes (Project Node)
Select Conditions (Select Node)
Join Node
JoinNode
Relation Node
Relation Node
Relation Node

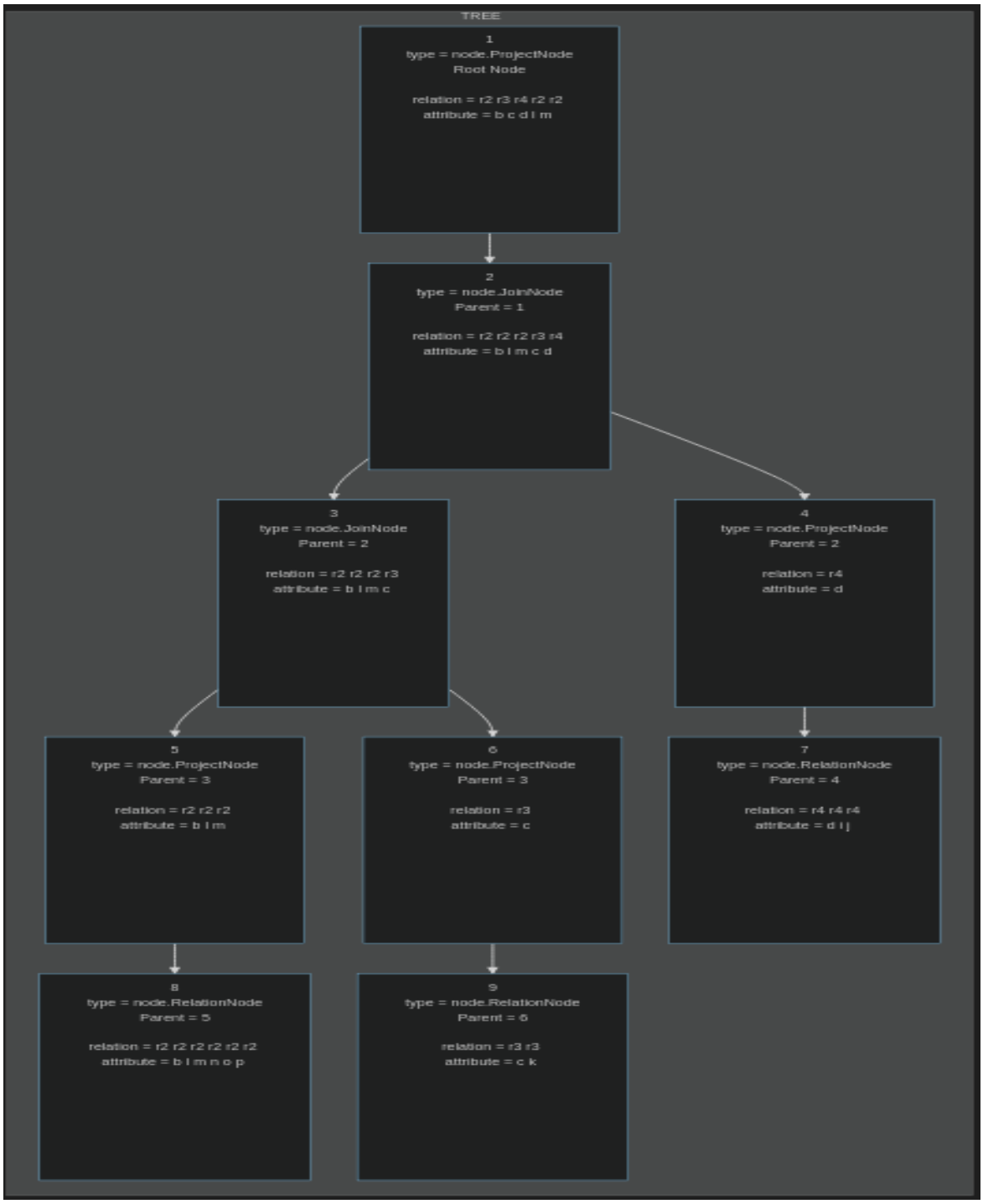
For the above written query the generated tree from the code can be found with the file named as ‘query_tree.md’

Query graph for a smaller query “select b,c,d,l,m from r2,r3,r4 where (b=c) AND (c=d)”



Task 3:Rewriting

After this initial graph generation, I push down the select nodes appropriately to the relations to which they should be applicable. But since for the new query mentioned above we do not have any Select condition which is not a join condition due to which we do not have select nodes in the above query tree and we can only push down the projects. The new optimized query tree will be



For the original query the selection optimized/pushed down query tree can be found as the file named “query_tree_selection_optimized.md”
And the file representing the query tree with pushed down projects and selects is named as “projects_with_selects_optimized.md”

For pushing down selects,
Selects are basically => cond1 AND cond2 AND cond3
So, if all the attributes present in a cond1 are also present in one of the children nodes of that select then this condition will be pushed down to that child node.
This is a gist of how the algorithm works, and more details can be observed from the code. If any questions, kindly feel free to ask me “kartik.ga@students.iiit.ac.in”

Now, when pushing down projects what i do is, i extract the columns present in both the child relation and the project node. And if a join node is involved in between then i make sure to add the attribute involved in the join condition to be also added in the extracted columns list. In this way i design a new list with the join attribute and the attributes which were coming from the child node in consideration and then i create a new project node with these new set of attributes and insert it in the tree.

Task 4:Localization

For performing localization, i am sticking to my fragmentation schema and i will first transform any given information on fragments on some general application to my own schema design.
My schema design contains certain Tables which are mentioned below:

1. Tables(Name,Fragmentation_Type,Number_Of_Fragments)

2. Columns(Table_Name,Column_Name)

3. Horizontal_Fragments(Fragment_Name,Table_Name,Attribute,Operator,Value)

4. Veritcal_Fragments(Fragment_Name,Table_Name)

5. VF_Columns(Fragment_Name,Column_Name)

6. Derived_Horizontal_Fragments(Table_Name,Fragment_Name,Horizontal_Fragment_Name,Direct_Fragment)

7. Sites(Site,User_Name,Password,IP_Address)

8. Allocation(Fragment_Name,Site)

Last two relations are required in the query processing and execution phase, but first 6 are to be used in this phase. Tables relation maintains record of all the relations that are there in my application database and whether they are fragmented or not and if yes then in how many fragments. Then Columns Relation stores the data about the columns that each relation in the application database has. Horizontal_Fragments is the relation to store the data about HFs, like the fragment name and from which table it was cut off and what was the condition which is broken down to 3 parts “the attribute, the operator and the value”. Then Vertical_Fragments stores the name of vertical fragments that have been made from the application database relations. VF_Column stores the column for each vertical fragment mentioned in Vertical_Fragments. Derived_Horizontal_Fragments , this relation stores the data about the DHFs like the table name which is fragmented this way, the name of the fragment and the HF or DHF it depends on. If it is a HF on which it depends on then it is a direct DHF of a horizontal fragment but if not then it is indirectly related to a HF by making use of a chain of DHFs.

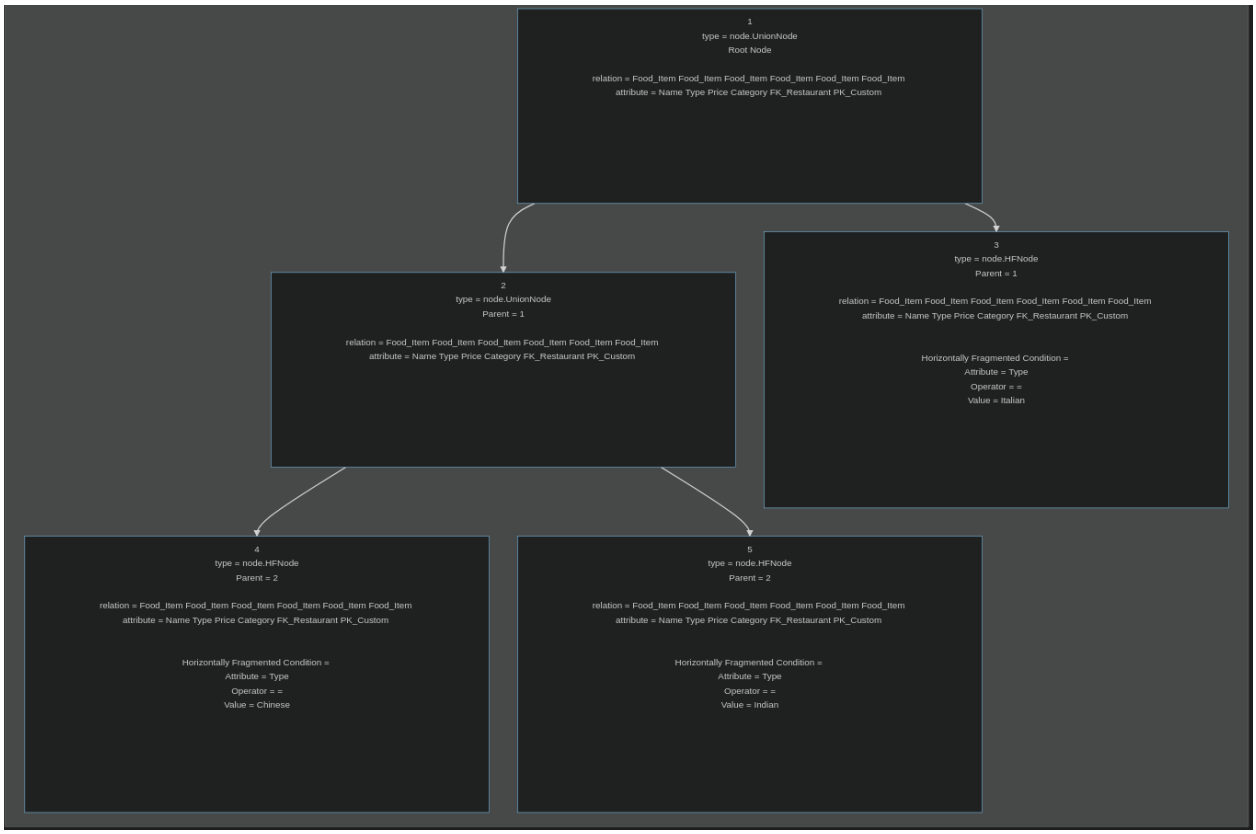
I have written a utility function to preprocess the given CSV to extract the data in the format I require.

Now firstly for each of the horizontally fragmented relations, I replace it with a union of its fragment. Lets say a relation R is fragmented as R1,R2 and R3
Then R = (R1 U R2 U R3)
But it tree form it is represented as
R = (R1 U R2) U R3, basically taking union of first two and then taking union with the third one.
Similarly for vertically fragmented nodes, i replace them with join nodes of their fragments in a similar fashion i replace the relation node with their HFs taken 2 at a time.

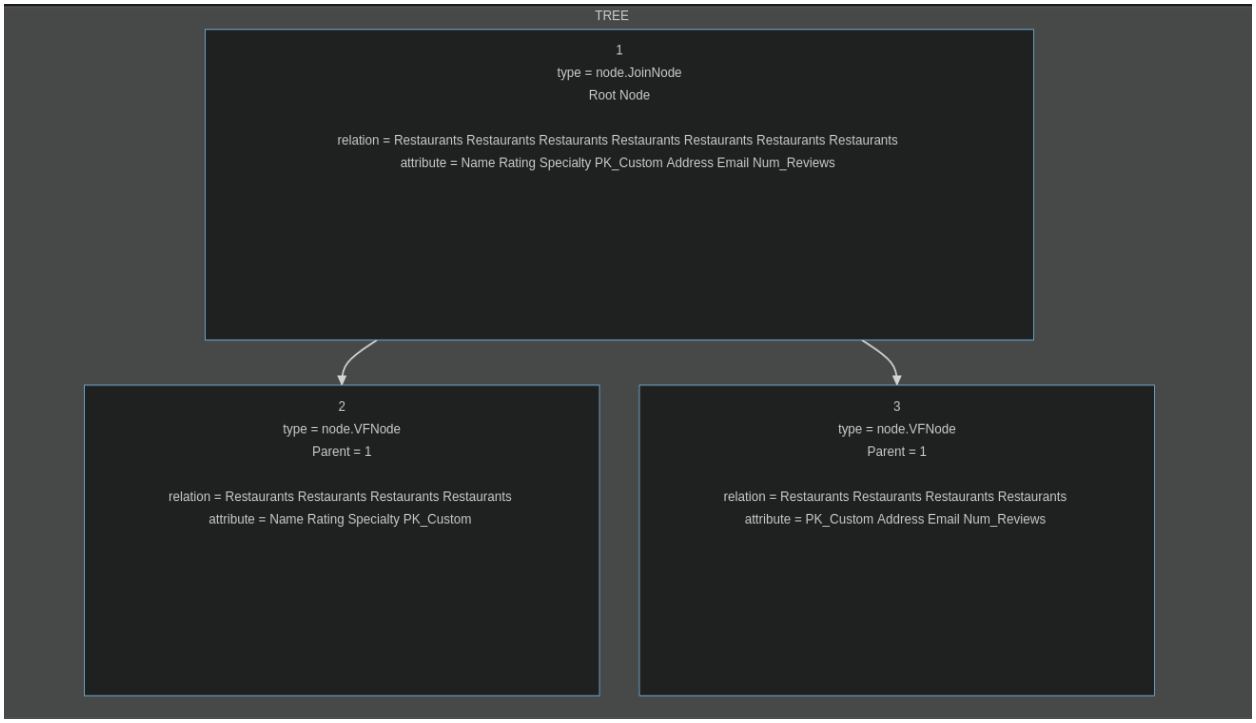
After this thing all I need to do is to push the selects and projects down and remove those fragments which contribute nothing to the result.
Now for pushing selects down the union node, they must go to each HFNode and we must check for all the conditions given in conjunction. If any one of the conditions fails then we must remove that HFNode from the graph as it just gives me an empty answer which we know logically. So i basically first push all the conditions down and remove the nodes which i need to remove and then in the remaining graph i add the select nodes above each HFNode with all the conditions which i had initially and then once again process to remove the condition which are not necessary and this time we are sure that this HFNode will have a certain result because if it had not then it would have never

been a part of the remaining graph. Now projects are just simply pushed down to each HFNode(above it) or to the Select Node which is directly above a HFNode.

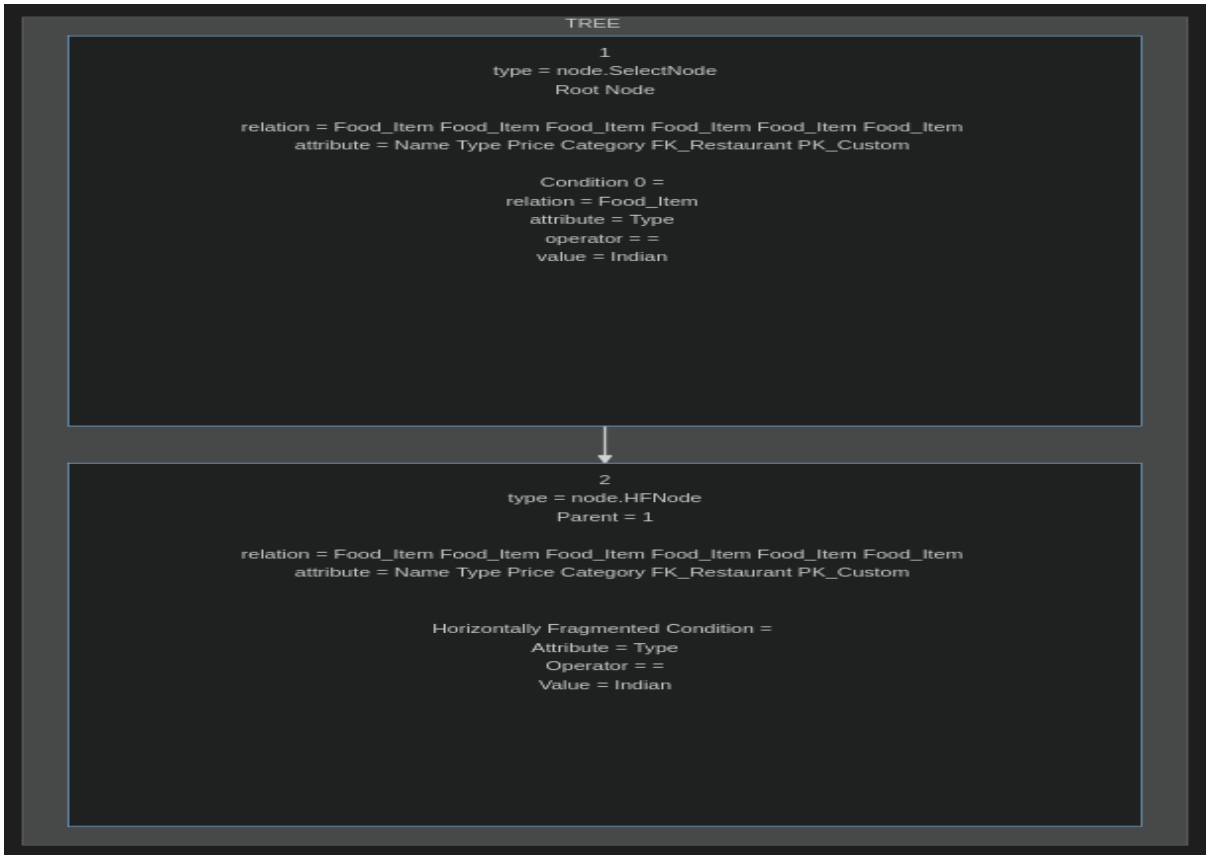
Now, if we talk about pushing selections down the JoinNode for the VFs then this will be the same as we did before. Like if one of the kids of this join node has all the attributes involved in the condition then that condition will be moved down to that kid and removed from the current node. Above mentioned condition is the one mentioned which will be mentioned in conjunction in CNF form. In this way selects are moved down, after that we have projects to move down. Now for the project to move down, we basically need the key to be projected from both the kids so that we can have the join in the end. So I first separate the given attributes in two lists, list1 and list2 where list1 contains the attributes belonging to kid1 and list2 contains the same for kid2. Now if these lists do not contain the key attribute then that is added to these lists and the project is pushed down to each fragment.
For ex, the query select * from Food_Item gets broken to [HF]



Similarly if we had, select * from Restaurants then we have [VF]



For select * from Food_Item where (Type='Indian')



Directory Structure:

changes.md => File mentioning the changes we have made since Phase1
complete.md => File which represents the final query tree after localization
config.py => File maintaining some global variables
documentation.md => This file mentions the assumptions we made and other little details of the code
fragmented_select.md => File represented the query tree when only the selects were pushed down after replacing nodes with their fragments
fragmented.md => File representing the part where we have just localized the relation but have not yet passed down the projects and selets
general_query_syntax.txt => It is common file that we keep to help us in coding, basically to write notes or points whichever are required while writing code
generalStructure.jpeg => Image showing the general structure of the query tree
input.py => File responsible for managing the inputs that we receive in the code.
logs.txt => File used to maintain logs of each function which is visited during the processing of that query.
node.py => File containing the parent class Node and all the inherited classes which make it easier to construct the tree
optimized.md => File containing the query tree right before we localize the relations.
original.md => Original tree which is made without any localization or optimization
preprocess.py => File containing function which will read data for both application database and for the fragmentation schema. It will further process the fragmentation schema data to the form which I have mentioned above.
project_with_selects_optimized => One of the above mentioned queries optimized.md file
query_tree_selection_optimized => For the above mentioned query, the query tree which is optimized by pushing the selects down
query_tree.md => Original query tree generated for one of the above mentioned queries.
query.py => File processing all the functions of a particular query, all the functions to push down selects and projects are written here.
selection_optimized.md => File which will have the query graph which is optimized by pushed down the projects
server.py => File which has the main loop
trace.py => File responsible for writing the logs.txt
readme.md => same as the documentation.md
report.pdf => Report which i have made for phase2