Introduction:

The project mainly deals with implementing relational algebra by parsing the command given. The command can involve any operation like

- Select: The select should be followed by any number of conditions. The condition can be =, !=, >=, <=, >, <. Only those tuples that satisfy those conditions get selected.
- Project: This should be followed by names of the columns which have to be projected. Only those columns are got from the table and are displayed.
- Join operation on two tables like crossjoin, naturaljoin, theta join. Only the table names have to be mentioned in case of natural join. In case of other joins, the condition/s to be satisfied by the columns of the two tables has to be satisfied.
- aggregate functions like min, max, sum, average on any number of columns grouped by one particular column
- Set functions like union, intersection, minus: The operation should be mentioned along with the names of the tables on which the operation is performed. The result will be displayed
- Rename: The name of the new table which will be got as a result and the new names of the columns in that table can be mentioned at the beginning of the command, followed by operations to be implemented.

  The resulting table will be renamed and its columns will be given the names. Only table or only columns can also be renamed.
- Nested operations: All the above mentioned operations can be nested.

Language used:

The language used for the assignment is python. Here a table is seen like a list of columns. Each column is a list of values where the i^{th} value in the column list corresponds to the value of the i^{th} tuple for the given column. The index of the value corresponds to the row index. In other words, in the table the index [x][y] means the value of x^{th} column and y^{th} tuple.

This method of **indexing** has been adopted because it is easier to index the data in this way. The whole tuple can be got from the table if we know the index of one of the values of the tuple in its respective column list. Once we know the index of one of the values in its column list say x, value of index 'x' in every column list belongs to the same tuple.

NESTING:

The schema of every table is saved in a dictionary 'schema' with the name of the table as key. So, schema[tablename] gives the schema of that table. Similarly dic[tablename] gives the name of the text file that has the table.

The operations can be nested as follows:

{ operation1 { operation 2 { operation 3 [tablename]}}}

The parser looks for the innermost set of { } first. It reads the text file which has contents of the table, performs the necessary operations, stores it under another table name into another text file. It then replaces the string including the innermost braces with [new tablename].The function is

recursively called with the new string as input. A global value ‘l’ is used for this purpose. Every time such innermost operation is performed, the value of l is given as the new table name. the new text file is given the name ‘l.txt’ and ‘l’ in incremented. So everytime such operation is performed, a new table gets generated and a new text file is formed. The schema of the new table is saved in schema and the name of the new text file is stored in ‘dic’. The new table name within [ ] acts as the table for the outer operation.

RENAMING:

Renaming is optional. If table is to be renamed, the format is:

[new table name](new column names)<-{ operations}

The string is checked for a ‘ <-‘ symbol. If present, it is split with ‘<-‘ as delimiter.

- The string on the left is parsed and the substring within the [ ] is the new table name. Once all the operations are done, the last final value of ‘l’ is got. Now if the new name is ‘new’, dic[new] is assigned the value of dic[l].
- The string within ( ) is split with comma as delimiter and the new column names are stored in a list ‘ne’.Now the current value of dic[l] (if table is not given a new name) is deleted and
- for f in range(len(ne)):
  schema[l].append(ne[f])

  If the new name is given,say the new name is ‘new’
- for f in range(len(ne)):

schema[new].append(ne[f])

Operations :

Select:

The above operation is used to select only those tuples that satisfy the given condition. The command should be of the form:

{ select (condition) [table name]}

The condition can be of the form:

a. ((cond1) or (cond2) or (cond3)) and so on) where cond1, cond2, cond3 can be of the form:
(column1=… and column2=… and so on)
b. ((cond1) and (cond2) and (cond3)) and so on) where cond1, cond2, cond3 can be of the form:
(column1=… or column2=… or column3!=…and so on)

The conditions can involve =, !=, >, <, >=, <=.

The basic idea here is to initially check whether the conditions are of ‘sum of products’ form i.e. case a, or the conditions are in ‘product of sums’ form i.e. case b.

Algorithm for parsing :

1.Check whether it is case a or case b by checking if there is an ‘and’ or an ‘or’ between any closing brackets that are followed by opening brackets.

2.If it is case a, then store the components of every independent condition in a list called orlist. Here

independent condition is that condition which does not have and in it. In other words, all the conditions of the form

‘col1 operator value’ are independent conditions.If any one of these conditions are satisfied by a tuple, it has to be selected. Put all the components dependent conditions of the form (cond1 and cond2 and …) in another list called andlist. Andlist is supposed to be a list of lists. Every list in it corresponds to a set of dependent condition. So for a tuple, to get selected, it should satisfy all the conditions present in at least any such set of dependent conditions.

3. If it is case b, then store the components of every independent condition in a list called andlist. Here independent condition is that condition which does not have or in it. In other words, all the conditions of the form

‘col1 operator value’ are independent conditions.If any one of these conditions are not satisfied by a tuple, it should to be selected. Put all the components dependent conditions of the form (cond1 or cond2 or …) in another list called orlist. Orlist is supposed to be a list of lists. Every list in it corresponds to a set of dependent condition. So for a tuple, to get selected, it should satisfy atleast one the conditions present in at least all such set of dependent conditions.

Example, ((col1=val1) or (col2!=val2 and col3>val3) or (col4<=val4) or (col5<val5 and col6>val6)) belongs to case a. Now:

Orlist=[‘col1=val1’,’col4<=val4’]

Andlist=[[‘col2!=val2’,’ ‘col3>val3’],[‘ col5<val5’,’ col6>val6’]]

Now split the condition into components by using the operator as delimiter. Depending on the operator store the code for the operator in the same list. The codes are:

Operator	Code
=	0
!=	1
<=	2
>=	3
<	4
>	5

Now we have

Orlist=[[‘col1’,’val1’,0],[’col4’,’val4’,2]]

Andlist=[[[‘col2’,’val2’,1],[‘col3’,’val3’,5]],[[‘ col5’,’val5’,4],[‘col6’,’val6’,5]]]

Initially	Finally
‘col1=val1’	[‘col1’,’val1’,0]
’col4<=val4’	[’col4’,’val4’,2]
[‘col2!=val2’,’ ‘col3>val3’]	[[‘col2’,’val2’,1], [‘col3’,’val3’,5]]
[‘ col5<val5’,’ col6>val6’]	[[‘ col5’,’val5’,4],[‘col6’,’val6’,5]]

Now for orlist,for every given column name get the index of the column in the table and store it in another list called ‘col1’.

Store all the values in another list called ‘val1’ .

Store all the sign codes in another list ‘sn’

Do the same for andlist with lists as col11, val11, sn1. Here u have a main list which has lists as values at various indices. But here all the columns indices belonging to the same j^{th} set of dependent condition must be stored in list with index j in main list. And the codes for operator will be stored in sn1 at index j.

So for independent conditions (orlist), for a given index k,

- Col1[k] gives the column index
- Sn[k] gives the code for operator
- Val1[k] gives the value

For every tuple with index i in the table, we can check if table[col1[k]][i] satisfies the operator represented by sn[k] with respect to the value val1[k].

For andlist, similarly, for a set of dependent conditions with index l,for an index k

- Col11[l][k] gives the column index
- Sn1[l][k] gives the code for operator
- Val11[l][k] gives the value

For every tuple with index i in the table, we can check if table[col11[l][k]][i] satisfies the operator represented by sn1[l][k] with respect to the value val11[l][k].

For every tuple, follow the steps given below:

- Check if it satisfies atleast one of the conditions in the orlist.

- If yes, store it in another table and check for the next tuple
- Else, check if it satisfies all the conditions of at least one set of dependent conditions.
- If yes, store it in another table and check for the next tuple
- Else, check for the next tuple

Project:

The project command should be of the form:

{ project (column1,column2,…) [tablename]}

Algorithm:

- Get the string within the () and split it with ‘,’ as delimiter. Now we have all the column names in a list.
- Access the schema for the table. Get the index of all the column names in the table’s schema and store all the indices in a list col=[]
- I=0,m=0
- For tuple with index i:

    For every j in col:

        Result[m].append(table[j][i])

    m=m+1

    i=i+1

  Now, result can have some repeating values.

- For i in range(0,length(table[0]):

```
Flag=0
For j in range(i+1,length(table[0]:
        For k in range(0,length(table)):
                If a[k][i]!=a[j][k]
                        Flag=1
                        break

        if Flag==1:

                break

if Flag==0:

        append the ith tuple to solution table
```

JOIN:

Natural join has the following format:

{ natural join [table1,table2] }

Algorithm:

- Get the string inside []
- Split it with ‘,’ as delimiter
- Save it in a list
- Now get the schema of table1,table2 into 2 lists l1,l2
- For i in l1:

```
        For j in l2:
                If i==j
                        l=index of column i
                        k=index of column j
```

- For every tuple in table1(index is i):

For every tuple in table2(index j):
If table1[l][i]==table2[k][j]
Append the whole tuple of table1 to result
Append the tuple of table2 without column[k]

For theta join:The following format is used

{ select ((table1.col11=table2.col12) or (…) or (…)) join [table1,table2] }

For parsing:

- Get the string inside [] and split it to get the table names
- Check whether the condition set is ‘sum of products’ or ‘product of sums’.
- Depending on the above result, classify the conditions into independent and dependent sets
- Split every condition into LHS,RHS. On both sides, split again into table names and column names with ‘.’ as delimiter. Store the operator in another list sn[].
- Now save the column indices of the two tables in different lists l1,l2

If it is sum of products,

- For independent conditions, follow the below procedure
- For i in range(table1[0])

For j in range(table2[0])

For k in range(len(l1))

If table1[k][i],table2[k][j] satisfy sn[k] atleast once

Append i^{th} tuple of table1 and j^{th} tuple of table2 into resulting table

For dependent condition, follow

- If all the conditions from at least one set of conditions are satisfied, add the tuples of the 2 tables to resulting table

If it is product of sums,

- For independent conditions, follow the below procedure
- For i in range(table1[0])

    For j in range(table2[0])

        For k in range(len(l1))

        If table1[k][i],table2[k][j] satisfy sn[k] everytime

    Append i^{th} tuple of table1 and j^{th} tuple of table2 into resulting table

    For dependent condition, follow

- If all the conditions from at least one condition from all sets of conditions are satisfied, add the tuples of the 2 tables to resulting table

CARTESIAN PRODUCT:

{ crossproduct [table1,table2]}

Normally, cross product is not used without a select condition on that product. So, instead of finding the product and then applying select operation, the parser detects the

select operation used immediately before the crossproduct and converts it into THETA JOIN.

Ex: { select ((table1.c1=tab2.c2) and (table1.c3!=tab2.c4)) { crossproduct [table1,table2]}

This is interpreted like:

{ select ((table1.c1=tab2.c2) and (table1.c3!=tab2.c4)) join [table1,table2]}

So the table1,table2 are extracted and the tuples that satisfy the conditions are only selected.Thus the space and time needed to compute the actual Cartesian product is saved.

AGGREGATE FUNCTIONS:

{ aggregate (groupingcolumn)->(sum:col1,max:col2,min:col3,…) [tablename] }

or

{ aggregate (sum:col1,max:col2,min:col3,…) [tablename] }

In the first case, the grouping column is specified. So, all the tuples with the same value for that column will be grouped and the aggregate function is applied to each of such group. The steps are:

- Save all the unique values of grouping column in a list called check
- Split the string within () with ‘,’ as delimiter.
- Now again split the strings in above lists with ‘:’ as delimiter. Save the indices of columns used in a list called ‘cn’ and names of used aggregate functions in list called ‘fname ‘

```
1.for j in range(len(cn)):
2.          if fname[j]=='count':
             for i in range(len(check)):
                count=0
                for z in range(len(table[0])):
                   if table[num][z]==check[i]:
                      count=count+1
                solution[j+1].append(count)
  3.        if fname[j]=='max':
             for i in range(len(check)):
                maxi=-9999
                for z in range(len(table[0])):
                   #print table[0][z],i,j,z
                   if table[num][z]==check[i] and int(table[cn[j]][z])>maxi:
                      maxi=int(table[cn[j]][z])
                solution[j+1].append(maxi)
4.          if fname[j]=='min':
             for i in range(len(check)):
                mini=999999999
                for z in range(len(table[0])):
                   #print table[0][z],i,j,z
                   if table[num][z]==check[i] and int(table[cn[j]][z])<mini:
                      mini=int(table[cn[j]][z])
```

```
            solution[j+1].append(mini)
5.      if fname[j]=='avg':
        for i in range(len(check)):
            summ=0
            count=0
            for z in range(len(table[0])):
                if table[num][z]==check[i]:
                    count=count+1
                    summ=summ+int(table[cn[j]][z])
            av=summ/count
            solution[j+1].append(av)
6.      if fname[j]=='sum':
        for i in range(len(check)):
            summ=0
            for z in range(len(table[0])):
                if table[num][z]==check[i]:
                    summ=summ+int(table[cn[j]][z])
            solution[j+1].append(summ)
```

In second case, grouping column is not mentioned. So the aggregate function is calculated on the whole table with all tuples in one group.The above mentioned algorithm is used except the existence of list called check

SET OPERATIONS

{ union [table1,table2] }

{ intersection [table1,table2] }

{ minus [table1,table2] }

The following steps are common for all the three.

- Get the string inside [ ].Split it and store the table names in a list
- Read the tables from text files into lists table1,table2
- For union, initially write the tuples present in table1 only and then write the whole of table2.

```
for i in range(len(table1[0])):
    f=10
    for j in range(len(table2[0])):
        m=0
        v=0
        for k  in range(len(table1)):

            if table1[k]==[]:
                v=1
                break
            if table1[k][i]!=table2[k][j]:
                m=1
                break
        if m==0:
```

```
            f=11
            break
        if f==10:
            write the tuple into resulting table as it is not present
in the other table
```

- Now write the whole table 2 into result.

```
    for i in range(len(table2[0])):
        for j in range(len(table2)):
            solution[sol].append(table2[j][i])
        sol=sol+1
```

- For intersection

```
for i in range(len(table1[0])):
        f=10
        for j in range(len(table2[0])):
            for k  in range(len(table1)):
                m=0
                if table1[k]==[]:
                    v=1
                    break
                if table1[k][i]!=table2[k][j]:
                    m=1
```

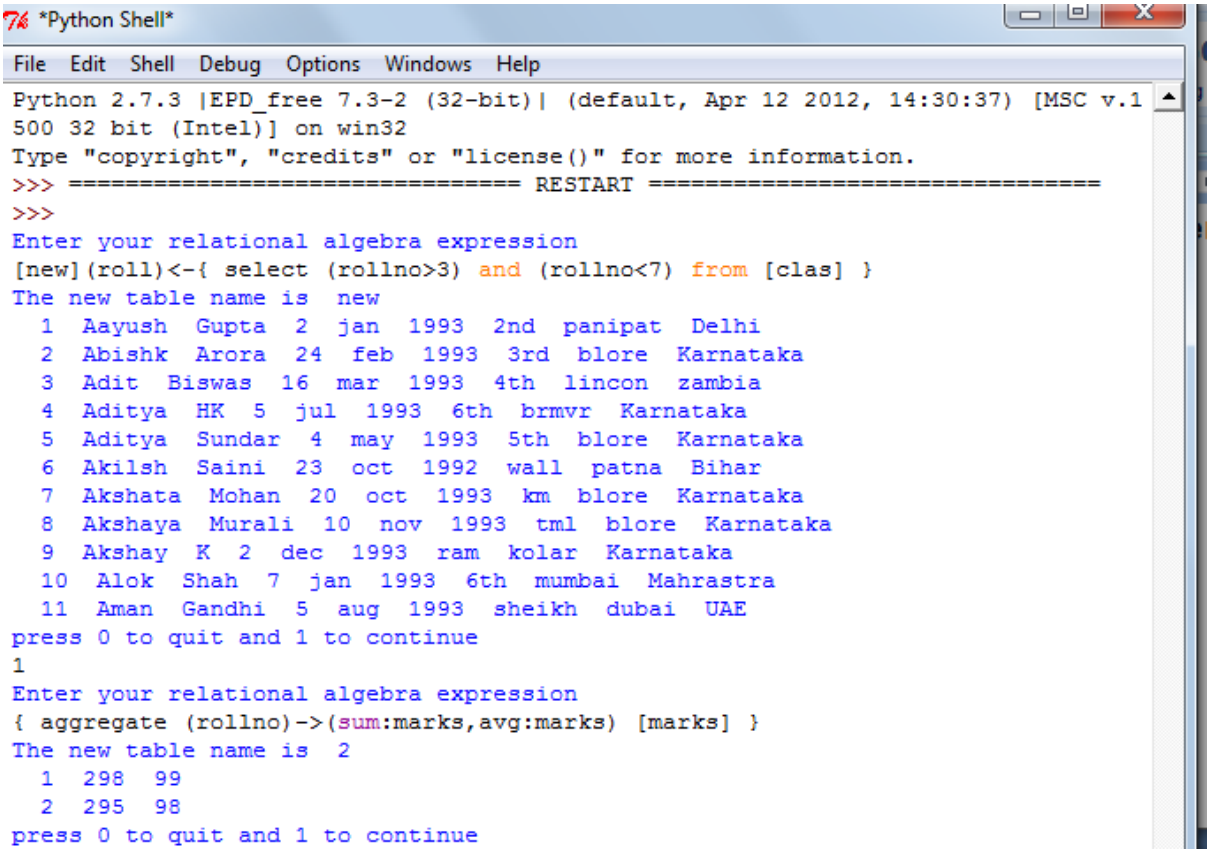
```
            break
        if m==0:
          f=11
           break
    if f==11:
        for s in range(len(table1)):
            add the tuple to resulting table as it is common to
both tables
```

- The implementation of minus is same as intersection except that the tuple should be added if flag==10 instead of 11.

Challenges Faced:

Indexing the data in order to divide it on the basis of rows or columns was difficult

Implementing join was not easy as it is a kind of select operation where the condition involves the values of two different tables which should also be joined later.


Python Shell
File Edit Shell Debug Options Windows Help
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Enter your relational algebra expression
[new](roll)<-{ select (rollno>3) and (rollno<7) from [clas] }
The new table name is new
1 Aayush Gupta 2 jan 1993 2nd panipat Delhi
2 Abishk Arora 24 feb 1993 3rd blore Karnataka
3 Adit Biswas 16 mar 1993 4th lincon zambia
4 Aditya HK 5 jul 1993 6th brmvr Karnataka
5 Aditya Sundar 4 may 1993 5th blore Karnataka
6 Akilsh Saini 23 oct 1992 wall patna Bihar
7 Akshata Mohan 20 oct 1993 km blore Karnataka
8 Akshaya Murali 10 nov 1993 tml blore Karnataka
9 Akshay K 2 dec 1993 ram kolar Karnataka
10 Alok Shah 7 jan 1993 6th mumbai Mahrastra
11 Aman Gandhi 5 aug 1993 sheikh dubai UAE
press 0 to quit and 1 to continue
1
Enter your relational algebra expression
{ aggregate (rollno)->(sum:marks,avg:marks) [marks] }
The new table name is 2
1 298 99
2 295 98
press 0 to quit and 1 to continue