BUC Algorithm

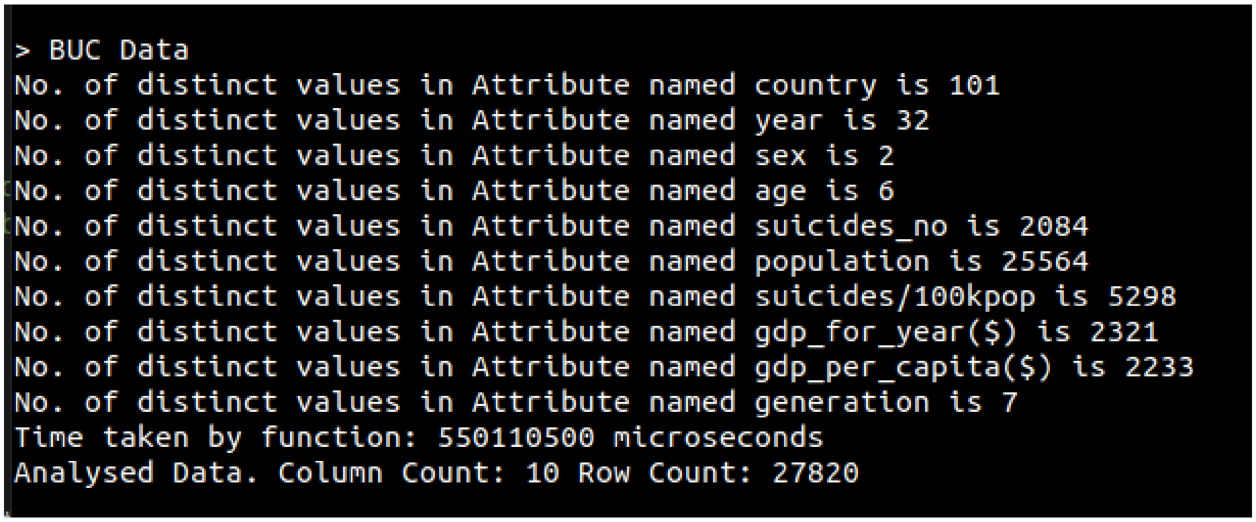
2019111035 – Gunjan Gupta
2020121002 – Pratistha Abrol

Instructions to run the code:

```
> cd BUC/src
> make clean
> make
> ./server
```

This will run the code and display an input command interpreter. Then type
> BUC {filename} .i.e. BUC Data

The output will display the number of unique values in each column and at the end of the program execution, it prints the time.



```
> BUC Data
No. of distinct values in Attribute named country is 101
No. of distinct values in Attribute named year is 32
No. of distinct values in Attribute named sex is 2
No. of distinct values in Attribute named age is 6
No. of distinct values in Attribute named suicides_no is 2084
No. of distinct values in Attribute named population is 25564
No. of distinct values in Attribute named suicides/100kpop is 5298
No. of distinct values in Attribute named gdp_for_year($) is 2321
No. of distinct values in Attribute named gdp_per_capita($) is 2233
No. of distinct values in Attribute named generation is 7
Time taken by function: 550110500 microseconds
Analysed Data. Column Count: 10 Row Count: 27820
```

The image above is for 5 as a min_supp and 8KB the block size.

Code Explanation:

Firstly, the boiler plate is generated to do the paging. It consists of buffermager, cursor, page and executor files.

The semantic parser and syntactic parser are checking if there are any semantic and syntactic errors and print errors accordingly.

After that the file is loaded in the code in rowsAllPage vector to apply operations on it.

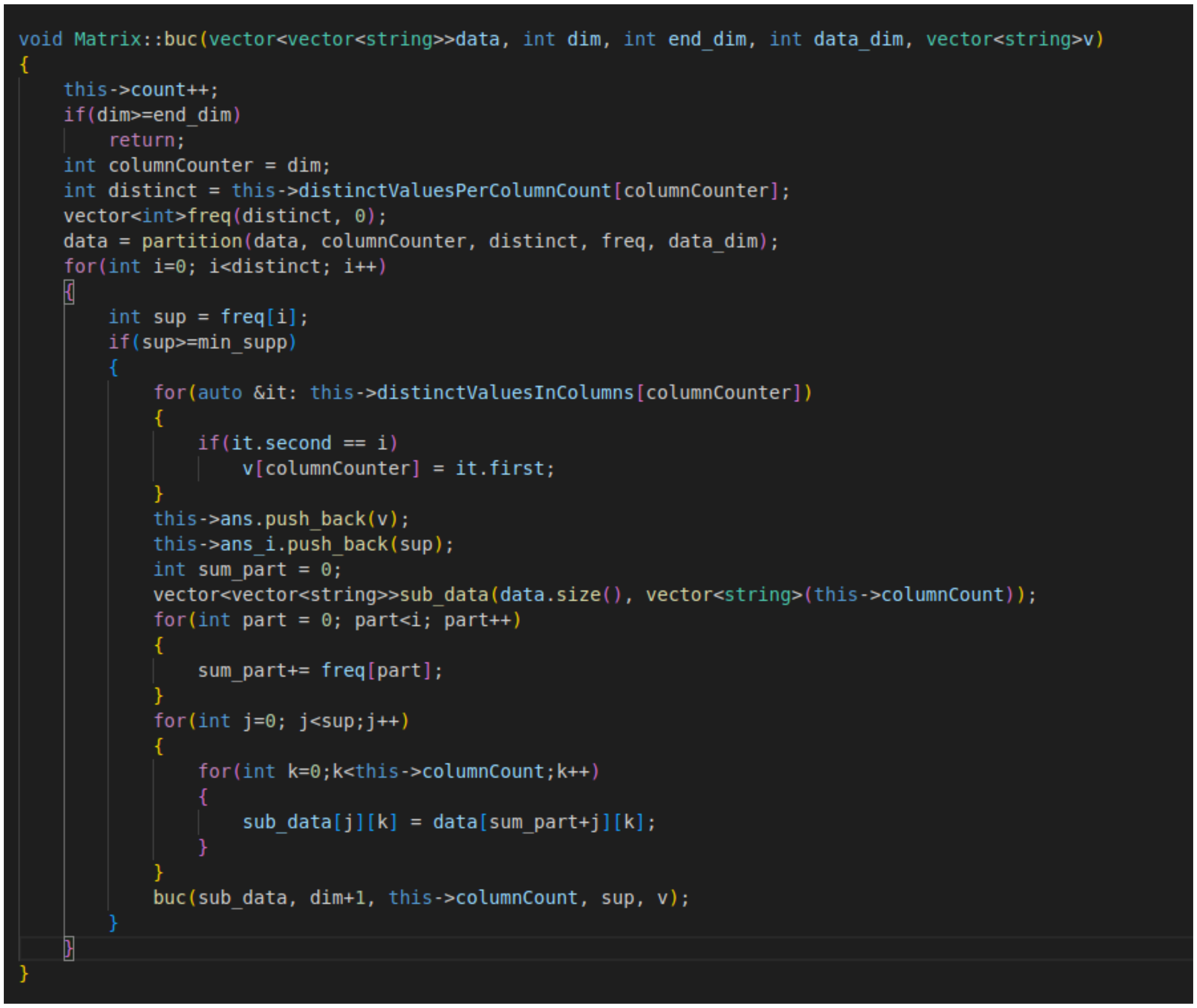
Columns Removed :

We removed

1. country-year column as it is providing redundant information.
2. HDI for year as it has missing values. It can be treated by filling that missing values as 0 but that might not provide accurate results.

Then after, storing the data from a file to 2D vector, the buc function is called.

BUC function:



```
void Matrix::buc(vector<vector<string>>data, int dim, int end_dim, int data_dim, vector<string>v)
{
    this->count++;
    if(dim>=end_dim)
        return;
    int columnCounter = dim;
    int distinct = this->distinctValuesPerColumnCount[columnCounter];
    vector<int>freq(distinct, 0);
    data = partition(data, columnCounter, distinct, freq, data_dim);
    for(int i=0; i<distinct; i++)
    {
        int sup = freq[i];
        if(sup>=min_supp)
        {
            for(auto &it: this->distinctValuesInColumns[columnCounter])
            {
                if(it.second == i)
                    v[columnCounter] = it.first;
            }
            this->ans.push_back(v);
            this->ans_i.push_back(sup);
            int sum_part = 0;
            vector<vector<string>>sub_data(data.size(), vector<string>(this->columnCount));
            for(int part = 0; part<i; part++)
            {
                sum_part+= freq[part];
            }
            for(int j=0; j<sup;j++)
            {
                for(int k=0;k<this->columnCount;k++)
                {
                    sub_data[j][k] = data[sum_part+j][k];
                }
            }
            buc(sub_data, dim+1, this->columnCount, sup, v);
        }
    }
}
```

Steps involved are:

1. The base case of recursion is defined.

```
if(dim>=end_dim)
       return;
```

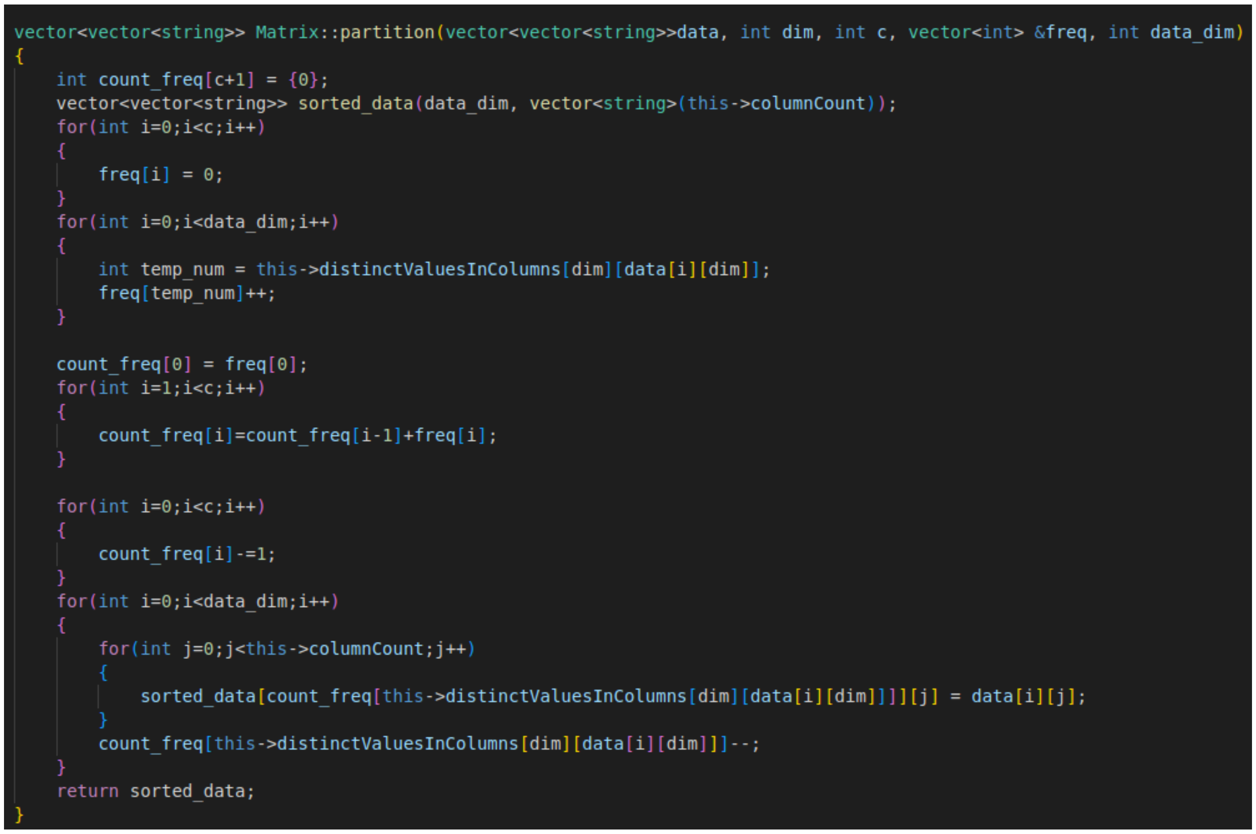
2. The number of unique value in each dimension(column) is calculated.

```
int distinct =
this->distinctValuesPerColumnCount[columnCounter];
```

3. Then the data is count sorted by the function called Partition. It returned the sorted data.
4. After getting the sorted data,
    a. for each distinct value, it’s frequency is checked.
        i. if it’s > min_supp, then the buc is called again on that sub_set of that particular partition..
        ii. else, next partition is checked

It will run till all the partitions are checked from given dimension to last dimension following the minimum support criteria.

Partition function:



```
vector<vector<string>> Matrix::partition(vector<vector<string>>data, int dim, int c, vector<int> &freq, int data_dim)
{
    int count_freq[c+1] = {0};
    vector<vector<string>> sorted_data(data_dim, vector<string>(this->columnCount));
    for(int i=0;i<c;i++)
    {
        freq[i] = 0;
    }
    for(int i=0;i<data_dim;i++)
    {
        int temp_num = this->distinctValuesInColumns[dim][data[i][dim]];
        freq[temp_num]++;
    }

    count_freq[0] = freq[0];
    for(int i=1;i<c;i++)
    {
        count_freq[i]=count_freq[i-1]+freq[i];
    }

    for(int i=0;i<c;i++)
    {
        count_freq[i]-=1;
    }
    for(int i=0;i<data_dim;i++)
    {
        for(int j=0;j<this->columnCount;j++)
        {
            sorted_data[count_freq[this->distinctValuesInColumns[dim][data[i][dim]]]][j] = data[i][j];
        }
        count_freq[this->distinctValuesInColumns[dim][data[i][dim]]]--;
    }
    return sorted_data;
}
```

It is implementing count sort.

The steps to the count sort are
1. Calculate frequency of all the distinct values available for a particular dimension.
2. Then, calculate the cumulative frequency of all the distinct values.

3. Then store the data on the basis of cumulative frequencies, so that all the values of dimension, dim, are grouped together, forming a partition.

So, count sort returns the sorted data on the basis of dimension, dim.

Paging:

The results of BUC are stored in vector ans and ans_i which stores the attributes and it’s count respectively.

We check if the total number of columns can be accommodated by a page or not, **if not, then we’ll divide the columns in the following manner.**

Page 1

col 1	col 2	col 3	col 4	col 5	col 6	col 7
row1_1	row1_2	row1_3	row1_4	row1_5	row1_6	row1_7

Page 2

col 8	col 9					
row1_8	row1_9					

Page 3

col 1	col 2	col 3	col 4	col 5	col 6	col 7
row2_1	row2_2	row2_3	row2_4	row2_5	row2_6	row2_7

Page 4

col 8	col 9					
row2_8	row2_9					

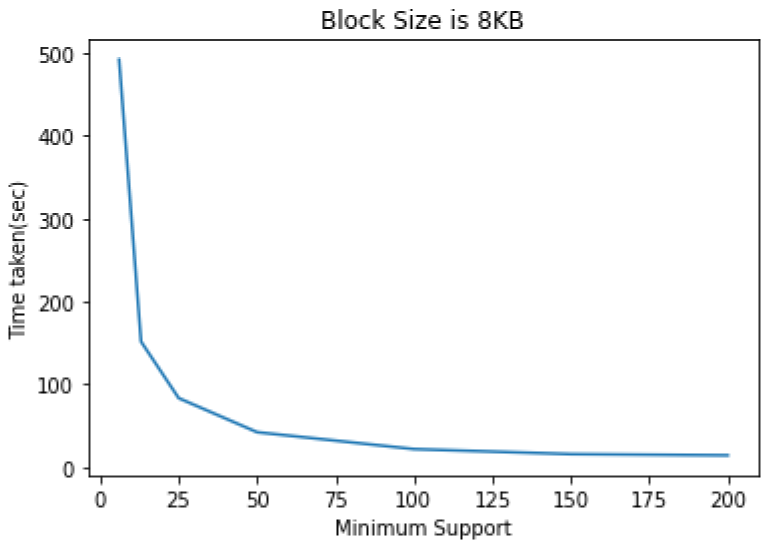
if a page can store all the columns, then a same table is stored

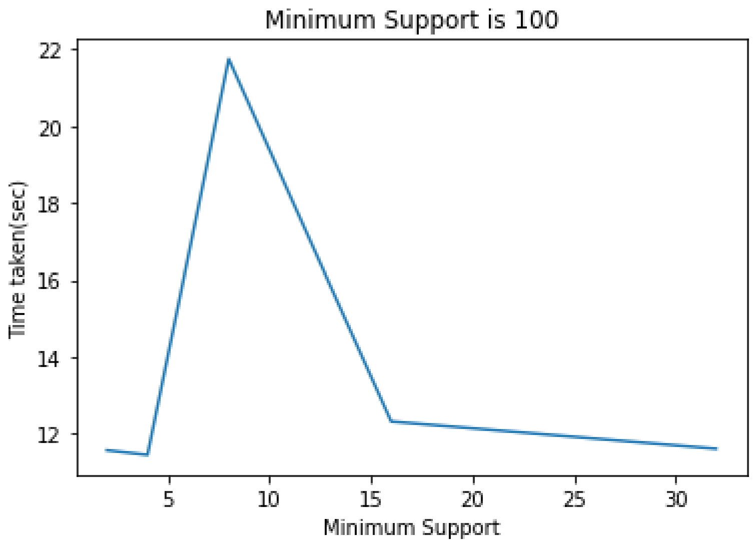
row1_1	row1_2	row1_3	row1_4	row1_5	row1_6	row1_7
row2_1	row2_2	row2_3	row2_4	row2_5	row2_6	row2_7

…… till rown

Results:

The graphs formed are :





Optimizations:

The code can be optimized by dividing some particular columns into continuous intervals, applying it on the column Population as it consists of 25564 unique values, the code takes time to run.

Population has values from 278 - 43805214.

It can be normalized into intervals of 10000s.
1-10000, 10001-20000, ……

This will improve the number of unique values to 4381 at max, which is 10 times lesser than original.

Similarly, some columns can be normalized for lesser run time of the code.