COL362 A3 Report

Student Madaka Jaya Prakash , Somishetty Harsha Vardhan
Entry Number 2020CS10356 , 2020CS10390

1 Data Structures Used :

We used Vectors for in Memory sorting and Priority Queues also for Merging of k file.
Using Vectors and std::sort() function took less time . we also tried using tries which took more time for outputs so shifted to vectors.
True Datatype used was mentioned in the report.
We took the priority queue (A Max Heap based Implementation) available in standard #inlcude <queue> librarly of cpp.
This Priority Queue Comparator is changed for convinience to sort the strings in required fashion and the changes are mentioned in the report.

2 Strategies for Optimising :

2.1 Algorithm used for sorting chunks :

A BUFFER SIZE of 500MB was considered. When reading the string from file a chunk of 500MB is read at once and stored in a vector and sorted the vector chunks are stored in a file temp.0.<runnum> later the vector is cleared.
All these files are individually sorted and stored to disk.
After the completion of first run the merge operations take place when num_merges = 0 then the merge algorithm runs until output file is generated when num_merges ≠ 0 it performs only num_merges merge operation. A merge operation is considered as merging k files if run has more than k files . If the run has less than k files all the merging is considered as one merge.
Sorting Using Vector and using inbuilt std::sort() function took less time than trie and priority queue sorting.
To get more efficient code in time we used std::sort() function and cleared all the data when stored to temp files.
This is because the struct we used for trie creates a node for each char if the strings are random and prefixes are not same then lot of memory wastage takes and also we need to call inorder traversal of trie to get sorted output which can be achieved by stacks.

```
struct TrieNode
{
  struct TrieNode *children[SIZE];
  short count;
  bool end;
  struct node* head;
  TrieNode()
    {
      for (int i = 0; i < SIZE; i++){
            children[i] = NULL;
      }
      count = 0;
      end = false;
      head = NULL;
    }
};
```

Trie Struct

```
struct node{
  short val;
  struct node* next;
  node(){
    val = 0;
    next = NULL;
  }
  node(int x){
    val = x;
    next = NULL;
  }
  node(int x,node* n){
    val = x;
    next = n;
  }
};
```

Node struct in trie struct

```
void insert(struct TrieNode *root,string key,short index) {
  struct TrieNode *ptr = root;
  for(int i=0;i<key.size();i++){
    if(!ptr->children[key[i]]){
      ptr->children[key[i]] = new TrieNode();
    }
    ptr = ptr->children[key[i]];
  }
  ptr->end = true;
  ptr->count++;
  if(ptr->head == NULL){
    node* n = new node(index);
    ptr->head = n;
  }
  else{
    node* n = new node(index,ptr->head);
    ptr->head = n;
  }
}
```

insert function

```
void preorder(TrieNode* node, string arr ,ofstream &v)
{
  if (node != NULL){
    for (int i = 0; i < SIZE; i++) {
      if(node->children[i] != NULL) {
        if(node->children[i]->end){
          for(int j=0;j<node->children[i]->count;j++){
            v << arr+string(1,(char)(i)) << "\n";
          }
        }
        preorder(node->children[i], arr+string(1,(char)(i)),v);
      }
    }
  }
}
```

preorder function

So we used vector for sorting the strings (in memory sorting) and stored these to temp files. We didn't altered input file and stored sorted chunks in temp.0.run files.

2.2 Algorithm for merging files :

Now A Buffer is considered as Write Buffer.
Size of WRITE BUFFER is 500MB.
A different priority queue of inputs tuple<string,int> values is used for sorting.
We are storing the string and file from which the string came.
Compare struct of Priority Queue is given by a new Struct compare in which first we compare strings by ascii values if they are equal then comparision of file index are taken the lowest ascii value string and file with lowest index has higher priority.
The data structure tuple was taken from standard #include <tuple> library of cpp.

```
struct Compare
{
  bool operator()(const tuple<string,int> &a, const tuple<string,int> &b)
  {
    if(get<0>(a) == get<0>(b)){
      return get<1>(a) > get<1>(b);
    }
    return get<0>(a) > get<0>(b);
  }
};
```

Now we are taking first element of all files if present and added them to priority queue.
Until the queue is empty we pop out the minimum element and added a element from the file in which the pop element came from.
These steps are repeated until all files are read.
So, that the priority queue size will be size of files merging at a time.
This implementation gave less time rather than taking particular chunks from a file adding it to priority queue and making the queue size as read buffer size.
The popped elements are added to WRITE BUFFER and when the buffer is full we are writing it the file this is done until all the elments are read in the files that has to be merged.
The number of previous runs were updated using the values of k and prev runs count new runs are made and final output file is generated.

3 Testing :

We used datasets given in piazza and also our own datasets to confrom the working of code tried different values of k and also with different valus of BUFFER sizes.
The optimal value differed from datasets size and also from k we kept the maximum left over available RAM as BUFFER sizes.

3.1 english-subset.txt

Took 500MB as BUFFER SIZE
only one file is generated.

k	Elapsed Time
2	751 ms
4	730 ms
8	740 ms
16	739 ms

3.2 random.txt

Took 500MB as BUFFER SIZE
only one file is generated.

k	Elapsed Time
2	3666 ms
4	3487 ms
8	3482 ms
16	3446 ms

Testing was done on M2 Macbook Air.