CSE 507

DBSI HW

Nalin Gupta - 2014065 | Sahar Siddiqui - 2014091

Linear Hashing

Code and Datasets

File Name for Code: main.cc

File names for Dataset: The two datasets were generated by a python script named "gen_db.py" (attached with this report and other codes). The datasets generated were then stored in the following files (attached) and used for both Linear and Extendible Hashing :

a. Dataset-Uniform - db_a.txt
b. Dataset-HighBit - db_b.txt

Implementation Specifications

Assumptions:

1. We are simulating the secondary memory (called "mem" in our code) using a vector of "string" type vectors. These string type vectors simulate the main buckets and follow a fixed pattern as described below:
    a. First element represents the number of empty spaces in the bucket
    b. The next "b" elements (where "b" is the bucket capacity) are the spaces for storing records in the buckets. Initially, all these spaces are initialized with the string "-1" representing that the space is empty.
    c. The last element of the bucket contains the index for the next overflow bucket linked to the current bucket. If there is no other following overflow bucket, this last element will contain the special character ~ denoting the end of the bucket chain.

    Hence, the total size of these string type vectors (buckets) is b+2.

2. After experimenting, we found out that the maximum number of main buckets that would be created while using the datasets that we generated and the bucket capacities

10 and 70, would approximately be equal to 2^14 = 16384. Hence, in order to keep the overflow buckets separate in the vector (secondary memory), we initially create a vector of 16384 arrays and then keep adding the overflow buckets at the end of the vector.
initialize_memory(int b) - Function called from main to initialize to create the initial mem vector as mentioned above.

3. The hash functions are taken as **h modulo** and **2*h modulo** (for the buckets which get split), where h is always a power of two. We initially start with our hash table containing 2^1 = 2 buckets.
4. In our implementation, we maintain all the buckets in such a way that all the empty spaces are at the end of the bucket or at the end of the overflow chain (if any).

.Insertion:

1. **insert (int K, int b) -** Function called from the main function to insert a record(key K) to the hash table. It takes in as input the key “K” and bucket capacity “b”.
2. On reading K, we first find the bucket number for the record to be inserted by using the hash function. If bucket number is less than the Split counter (S), then that means the bucket has already been split, thus the higher modulo function will be used to find its actual bucket location. In case it is greater than or equal to S, the same bucket number will be used for insertion of record.
   _hash(int K, int m) - Function (called from insert function) being used to calculate the hash value. It takes in as input the key and the Modulo value to hash it with and returns the corresponding hash value. In case, the bucket has been split, it uses the higher Modulo value and the lower one, in case it is not.
3. Once we have found the bucket number we go onto inserting the record. Since the empty spaces in a bucket (or overflow chain) can only be found at the end, we traverse the corresponding bucket chain list till the end to correctly insert the record.
   a. If there is no overflow bucket (denoted by ~ in the last index of the main bucket), we will check the number of empty spaces in the bucket. If there is an empty space, we will insert the record there. Otherwise, we will create a new overflow** bucket, insert the record there and update the last character of the main bucket to contain the index of the overflow bucket.
   b. In case, there is an overflow, we will traverse through the buckets using the last characters of the buckets and go till the last overflow bucket. Similar to the first case, if there is space in the bucket we will insert the record there, otherwise create a new overflow bucket.

In order to use the space more efficiently, we make use of a **bitmap in which each element represents whether an overflow bucket at a position is being used in some

overflow chain or not. These positions(or indices) can be directly mapped with the overflow buckets in the mem vector. Every time a new overflow bucket needs to be used, we first check the existing bitmap for a bucket index which is not in use (denoted by '0'). If there exists such a bucket, we do not need to add any new bucket to the vector and can directly use this bucket index for overflow. If there is no such bucket (i.e. all the elements of the bitmap equal to '1'), it means all the existing buckets are in use and hence we create and insert a new bucket to mem vector. Simultaneously a "1" also gets inserted to the bitmap.

insert_helper(int K, int bucket, int b) - Function (called from insert function) to insert the given key K at the location which corresponds to the hash value "bucket". After inserting the record at the appropriate location, the function returns "0" or "1" depending on whether an overflow occurred while inserting a record to help decide whether a split of bucket will take place next or not.

NOTE - An overflow in bucket takes place if the record cannot be accommodated in the main bucket itself.

.Splitting:

1. If an overflow does take place, according to the algorithm the bucket which needs to be split next, gets split. This information is being maintained by a global split pointer "S". NOTE - Split pointer "S", only gets incremented till half of the second hash value ((2*h)/2 - 1 since it is 0 indexed) because only the buckets have not been split til that location. After this point, the split pointer is set to 0 and the hash function value increases in the power of two.
2. **split_bucket(int b) -** Function (called from the insert function after the split condition has been met) splits the bucket being pointed at by "S". The split image bucket is formed at the end of the vector and the second hash function is used to rehash the values of the split bucket, since the new function can hash these values to only the split bucket index and its image's index. For rehashing the values, we traverse through the main split bucket (and it's overflow chain, if any) and place the records in the correct buckets as per the hash values returned by the second hash function. If the values are moved from the split bucket to it's image, that location/index in the split bucket is marked as empty as the number of empty spaces get updated accordingly.
3. Since rehashing would create some empty spaces in between the buckets and our implementation requires the empty spaces to be only at the end, so we need to compress the the bucket (or chain) to have all the records in the beginning.

   update_buckets(int b) - Function used for compressing the buckets. It uses an O(n) algo of shifting all the empty spaces to the end by making use of two pointers. Algorithm for the same is as follows -

a. We will maintain two pointers namely “fast” and “slow” which are both initially pointing to the first element of the main bucket. In implementation, these are structures containing the index of bucket, index of array (within the bucket) and the last character of the bucket which they are supposed to point to.
b. Now, we traverse the bucket (and overflow chain, if any) till the end using the two pointers.
c. If (fast == slow), we check if the element pointed by them is an empty space or not. If it is a record, then both the pointers get incremented and if not then only the fast pointer gets incremented.
d. If (fast != slow), we check if the element pointed by the fast pointer is an empty space or not. If it is a record, then the record gets written to the location pointed by slow pointer and then both pointers get incremented. If it is an empty space, only the fast pointer gets incremented.
e. In this way, at the end we have shifted all the empty spaces to the end.

4. Now, since the empty spaces are shifted to the end, we might have some empty overflow buckets and the end and so the last character of the last filled bucket in the chain need to be updated to ~ and the bitmap needs to be updated to mark the buckets as “not in use” or “0”. Hence we need to traverse through the list once to make the above updations (if any).

Lookup:

1. Find the hash value of key from first hash function. If the value is less than S, find new value from second hash function.
2. Once we have the bucket number from above, find the key in the main bucket (or list)

Time Complexities:

According to our implementation, following are the time complexities -

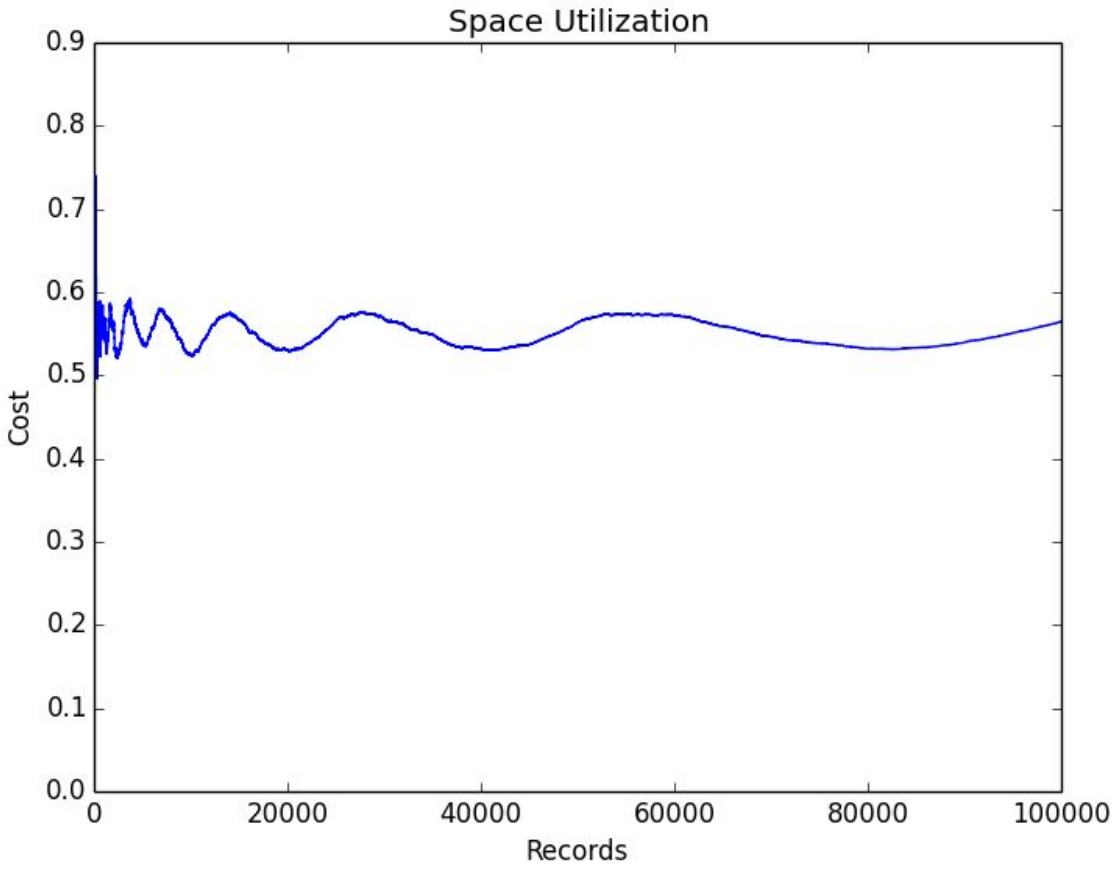
1. Insertion - 0(1) if no overflow | 0(x) if there is overflow where x corresponds to number of overflow buckets.
2. Lookup - 0(n) where n denote the number of elements in the bucket (or chain, if any) corresponding to the hash value

Plots and their Explanation

The plots were generated using a python script “plots.py” (attached with the report)

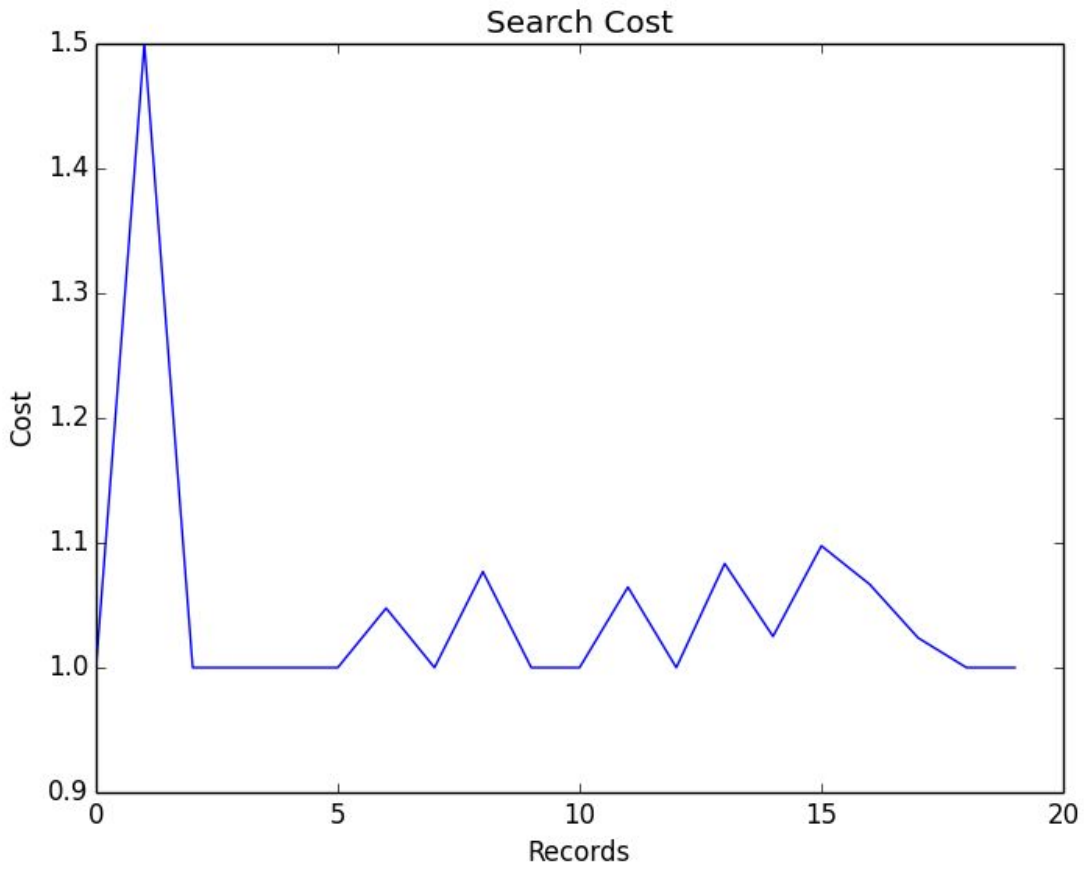
For Dataset-Uniform -

1. Storage Utilization against the number of records in the file



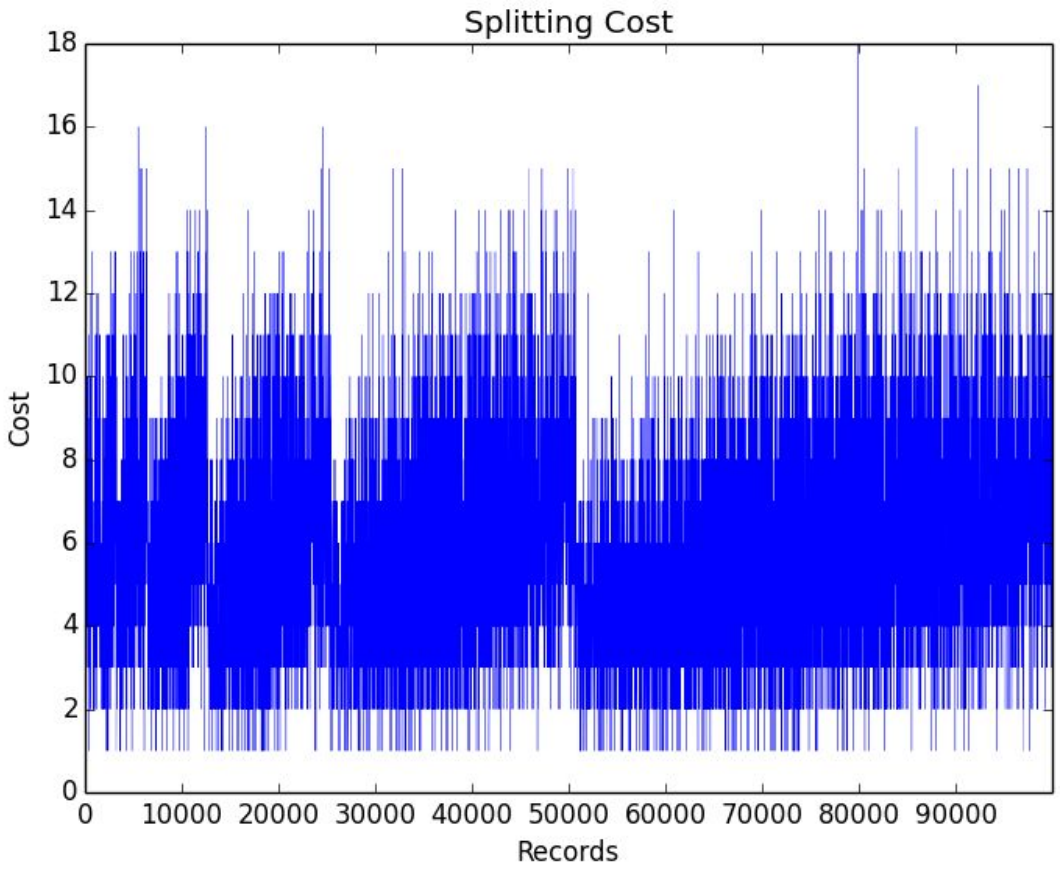
As seen in the plot, the storage utilisation cost only oscillates between the 0.5 and 0.6 and becomes fairly stable as the number of records increase in the number because even though initially the split occurs at random locations (as pointed by the split pointer), as the number of elements increase, the records get evenly distributed).

2. Average Successful Search Cost against the number of records



As seen in the plot, the Search cost normalizes as the number of records keep increasing. This is in direct correlation with the space optimization, since the more the number of records, the more evenly they are distributed. If they are event distributed, we would have to traverse through less number of overflow buckets to retrieve the record.

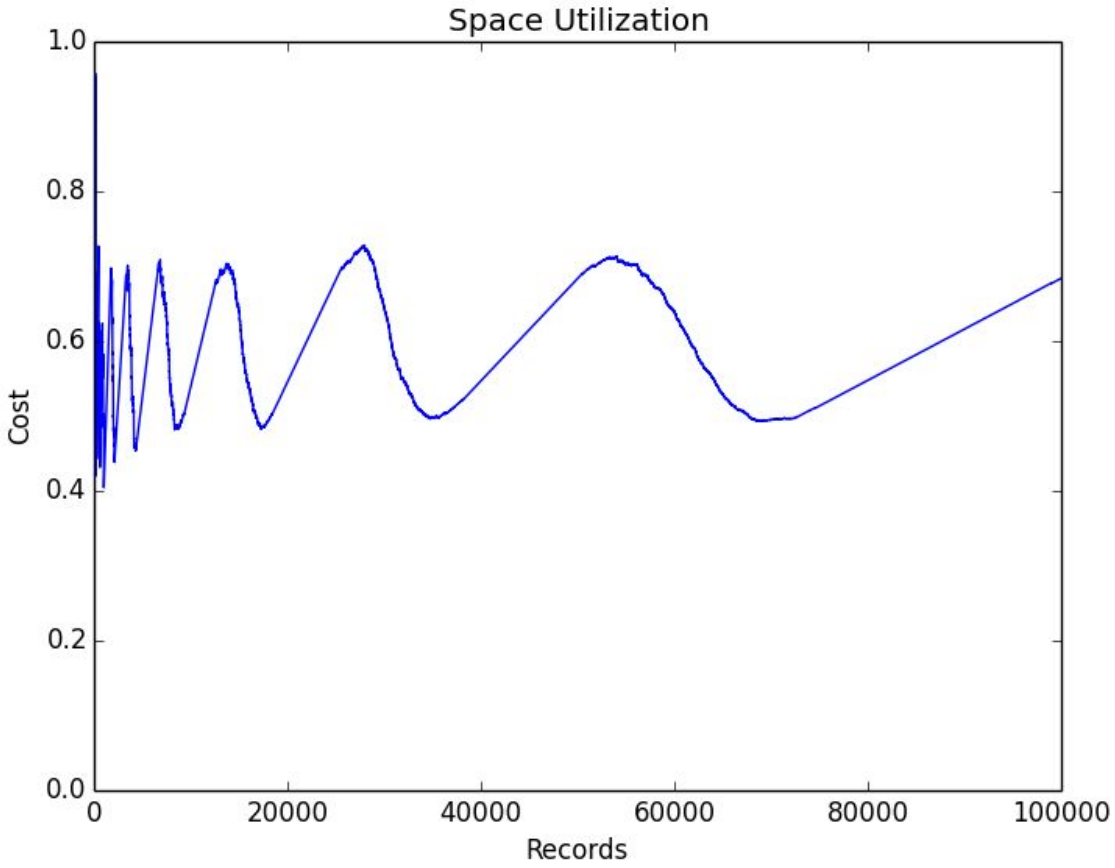
3. Splitting cost against the number of record



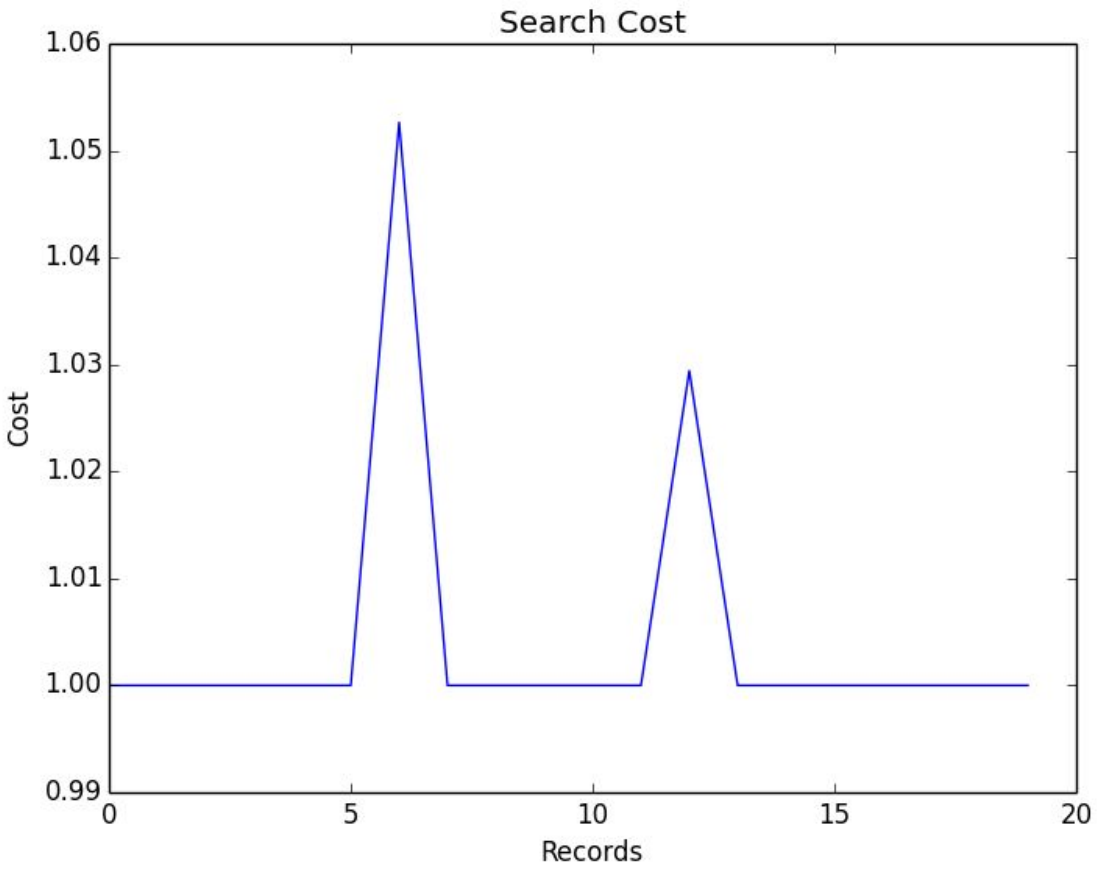
The splitting cost oscillates very frequently with increase in number of records due to the fact that, it is completely random what numbers would be inserted and what would be the existing values in the bucket. Depending on the values, multiple new buckets could have to accessed.

For Dataset-HighBit -

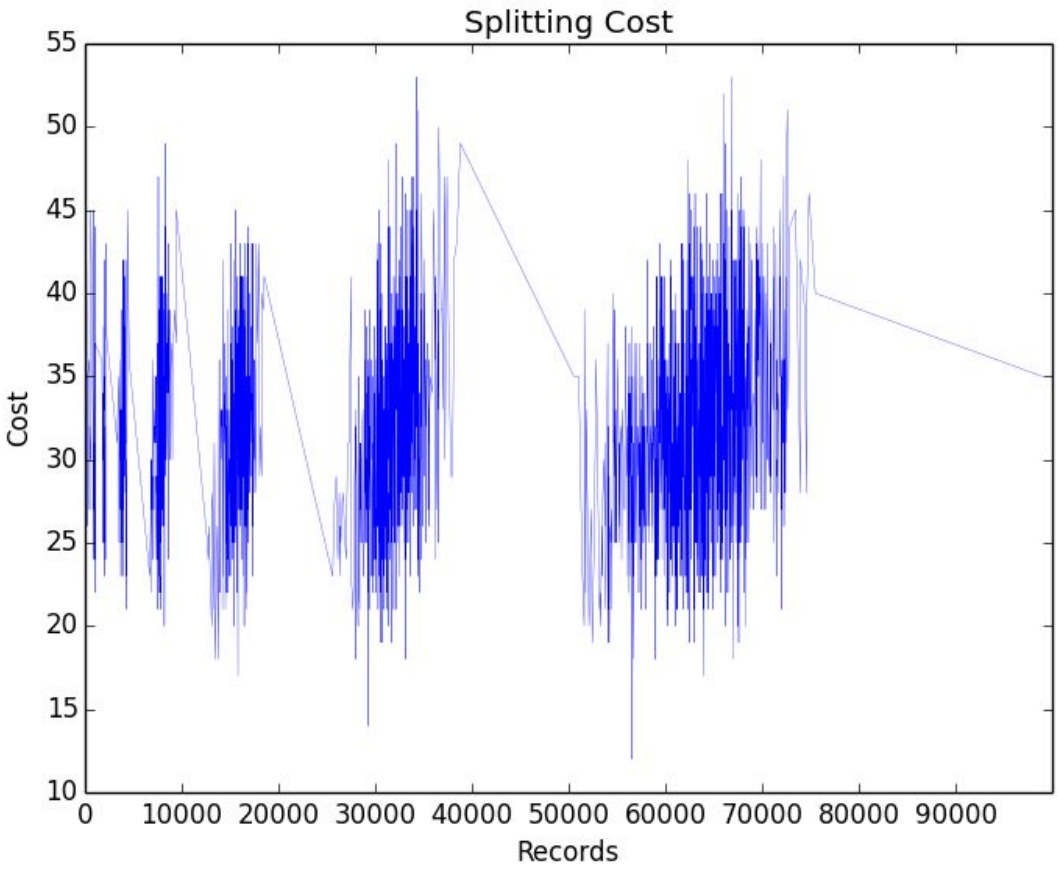
1. Storage Utilization against the number of records in the file



2. Average Successful Search Cost against the number of records



3. Splitting cost against the number of record


Splitting Cost
Cost
Records
55
50
45
40
35
30
25
20
15
10
0
10000
20000
30000
40000
50000
60000
70000
80000
90000