Database System Implementation (CSE507) : Homework 1

Anshuman Suri : 2014021
Rounaq Jhunjhunu Wala : 2014089

Implementation Description

All abstract objects like primary memory, secondary memory and buckets are implemented as classes. The bucket and secondary memory classes are common to both linear and extendible hashing. Extendible hashing requires primary memory as well, which is implemented for that type of hashing. The classes created and used for this assignment are:

- *Bucket:* contains a pointer to the secondary memory to which it belongs. Each bucket also contains a pointer to a bucket overflow index, along with the number of records in that bucket and that bucket's size. Trivial functions like getters and setters are implemented for this class.
- *Disk:* contains a vector of buckets, along with a counter to count how many times secondary memory is accessed. This contains basic getter functions.
- *RAM:* contains a vector of overflow buckets (for the case when the hash table cannot fit in main memory), the amount of memory being used by the hash table, a vector of integers to store data, and a pointer to the secondary memory with which it is linked. The constructor initialized RAM with 2 buckets, while the getter and setter functions fetch/set data from either the RAM or from main memory, depending on the size of the hash table (whether it is in RAM or overflowing into secondary memory). Along with 2 more setters, this class also contains a function to double the current directory (as is done by extendible hashing).
- *LinearHash:* With some getter and setter functions, functions to insert data into memory, insert data into a specific bucket, search for data in memory, add overflow bucket, recycle a bucket, get a new overflow bucket, split data (all of these functions are abstract implementations of the actual algorithm). A helper function to display the hash table (with data entries) at any point is also implemented.
- *ExtendibleHashing:* most of the function definitions are same as linear hashing, along with an extra function to force insert data. The functions themselves are according to the algorithm used for Extendible Hashing.

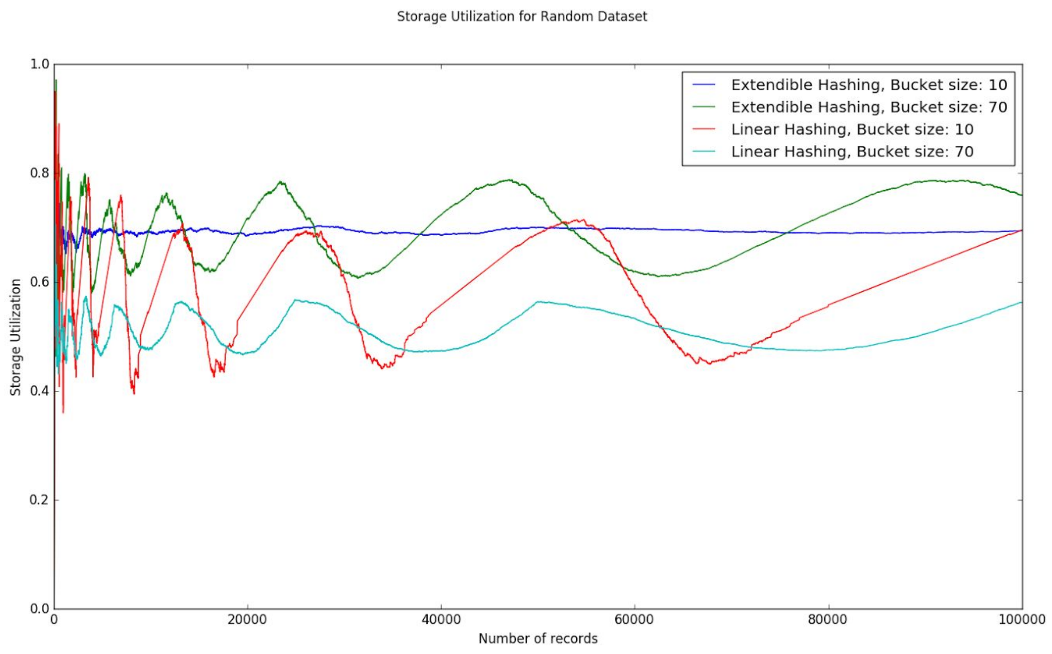
Extendible Hashing

As described by the algorithm, the extendible hash class maintains a hash table in RAM. The hash function is chosen as the the first *x* significant bits, where *x* is the depth. The local depth for each cluster is stored in the bucket for the cluster to which it belongs. For every insertion, we try inserting the entry into the corresponding hash cluster (which we get by hashing with the global depth). If the local depth of that cluster turns out to be the same as the local depth, we rehash the entries in that cluster, that is, the entries for which we get the discrepancy. In case there is no more space to insert any data. If any of the entries go into overflow, we double the hash table and thus the hash function are redefined accordingly (though all the entries are not rehashed). In case the hash table fills out of the RAM class, the overflowing data is stored in spill buckets that are stored in secondary memory. Searching for an element works in a similar fashion, looking at the hash of the global depth and then tracing a route to the correct bucket in which is should be (if it is there at all in the database).

In depth code documentation has been provided in *README.md*

Graphs

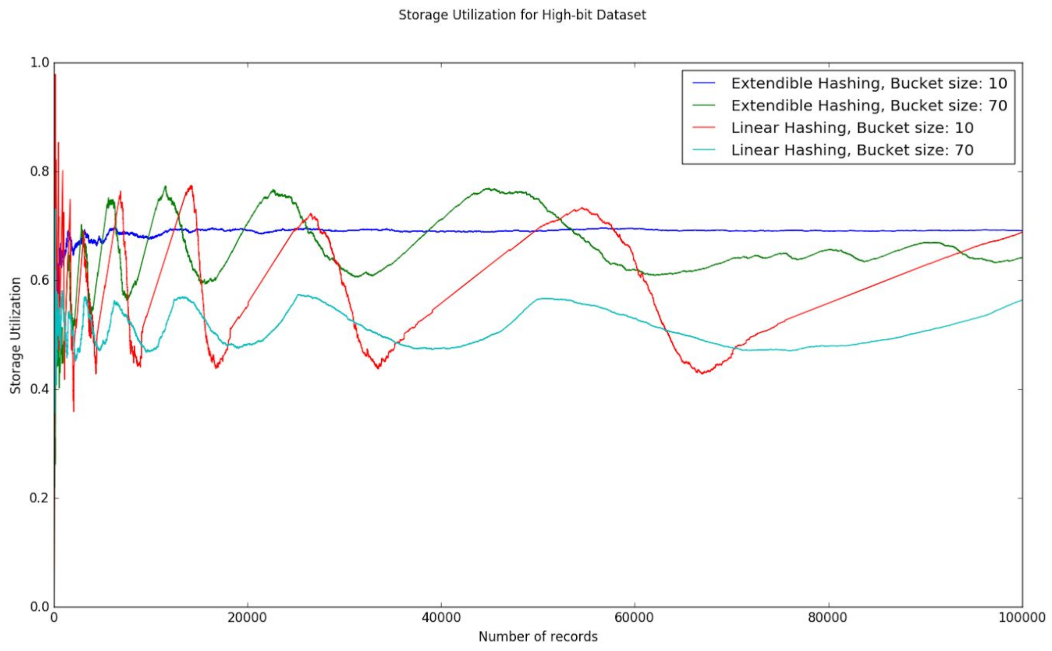
(a) Storage Utilization: *N/(B*b)*



Storage utilization for random-dataset for both linear and extensible hashing, with bucket sizes 10 and 70 for both

On average, the storage utilization is maximum for extensible hashing with a bucket size of 10. Extensible hashing has on an average lower variations in comparison to linear hashing. This is because of the fact that extensible hashing inserts data into buckets and increases its hashes, whereas linear hashing keeps inserting data circularly. Extendible hashing with a lower bucket size gives almost constant storage utilization (as it is limited because of the number of buckets), whereas that with a higher bucket size keeps variating (giving higher accuracy at peaks) while ultimately increasing as we get more and more data (as the overheads get distributed more evenly).

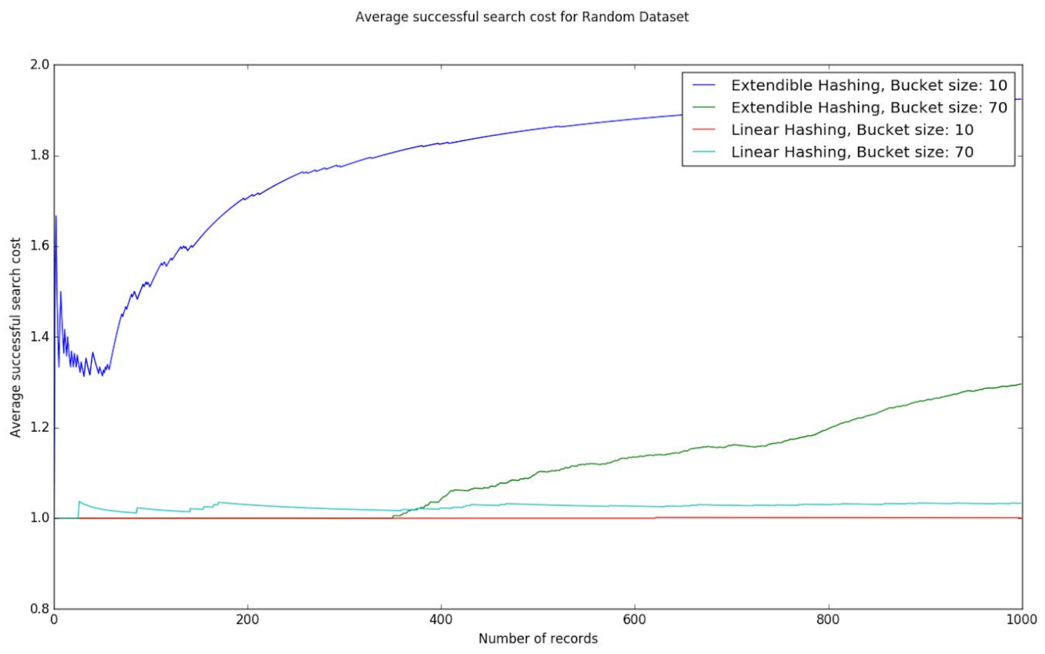
Linear hashing performs better initially, but its performance drops down as the number of records increase (in comparison to extendible hashing). The sinusoidal behaviour of linear hashing is because of the fact that as soon as we fill all the buckets, the performance rises but just after that, we grow the number of buckets, so the utilization falls down drastically.



Storage utilization for highbit-dataset for both linear and extensible hashing, with bucket sizes 10 and 70 for both

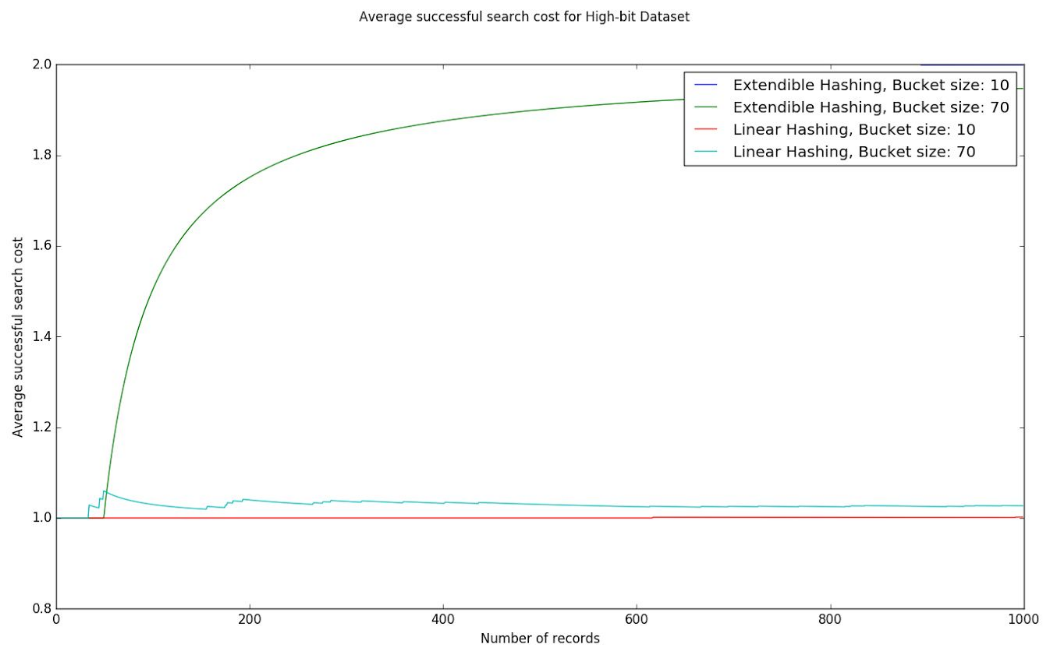
The space utilization graphs for random and high-bit datasets are almost the same in case of linear hashing, as it effectively looks at the least significant bits while hashing. On the other hand, extendible hashing looks at the most significant bits, so as the data grows, we get more clashes (as more and more entries hash to the same buckets), because of which disk utilization drops as the number of records increases.

(b) <u>Average successful search cost: *bs/s*</u>



Average successful search cost for random-dataset for both linear and extensible hashing, with bucket sizes 10 and 70 for both

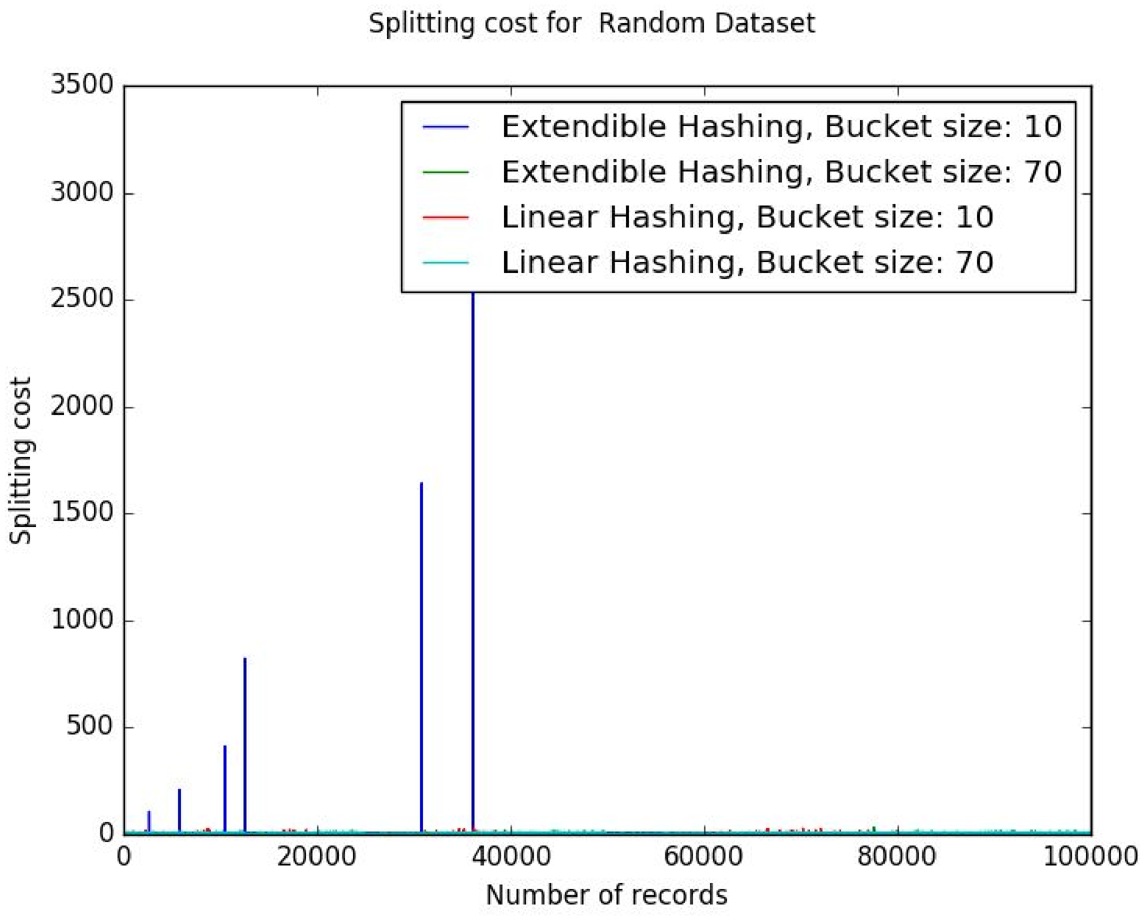
Linear hashing has indirect access, not through the directory. Thus, the average successful search cost in the case of linear hashing is almost independent of the number of the records, and is in fact close to 1 throughout (it is greater than on in the case of overflows, which are very rare). However, extendible hashing works totally differently, because it first looks at the hash lookup table in primary memory. Extendible hashing with a lower bucket size seems to perform worse than a higher bucket size, as the data being random implies that we can fit more records in a bucket, thus leading to lower memory accesses for a higher bucket size.



Average successful search cost for highbit-dataset for both linear and extensible hashing, with bucket sizes 10 and 70 for both

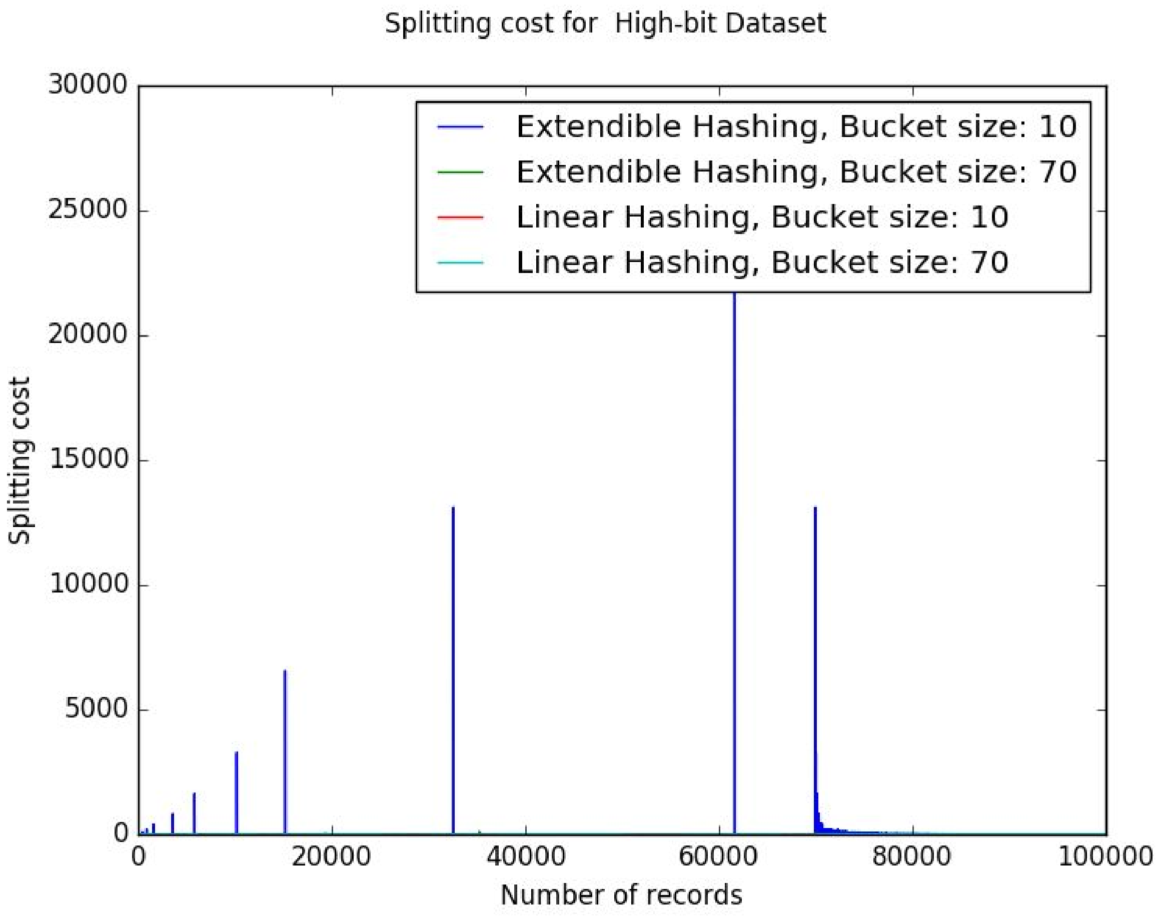
Linear hashing gives performance similar to the random dataset, as it is not really dependent on the nature of data when it comes to it being high bit or random (as there is no indirect access through directories). However, the performance for extendible hashing is much different. Both bucket sizes in the case of extendible hashing give exactly the same performance, as the hash uses high order bits, which are mostly set here (and the data isn't random either). Because of the data being high-order bits mostly, the number of access for each search grows much faster than that in the case of random data. This is because of the nature of the hash function, which results into more overflows and rehashing/growth of hash tables when the data is all high-bit.

(c) <u>Splitting cost:</u>



Splitting cost for random-dataset for both linear and extensible hashing, with bucket sizes 10 and 70 for both

The splitting cost for linear hash is much lower that for extendible hash, as we create at most one bucket (or an overflow bucket, in the worst case). On the other hand, extendible hashing includes doubling the hash table and adding more buckets accordingly, which leads to an extremely high splitting cost. As the nature of the dataset is random, high order numbers may come before, leading to splits in directory, thus leading to a higher splitting cost. The splitting cose is almost independent of the bucket size for linear hash. However, extendible hashing with a higher buckets size performs much better than a lower bucket size (in fact, close to linear hashing in performance). It is almost 1/7th, as we can include more data in buckets.



Splitting cost for highbit-dataset for both linear and extensible hashing, with bucket sizes 10 and 70 for both

The splitting cost for linear hashing follows a similar tried, being very small and almost constant. The splitting costs for extendible hashing are high just like random data. However, the high cost splits happen towards the end, as the data is not random and we get splits as the data gets more and more high ordered. The splitting cost here, again, is much more for a smaller bucket size than a larger bucket size, because of the same reason.