An approach for improving searching algorithm in big data using hybrid technique

Iman Sadeq Khairi Alzuabidi, Prof. Dr. Kadhim b. S. Aljanabi
Department of computer science, faculty of computer science and mathematics, university of kufa, najaf, iraq
Department of computer science, faculty of computer science and mathematics, university of kufa, najaf, iraq
E-mail: emans.alzubaidy@student.uokufa.edu.iq

Abstract. Search algorithms play an essential role in wide variety of applications in computer science. Different search algorithms can be applied to different data structures. Each data structure has its own advantages and limitations which in turn affect the search algorithm complexity and performance. In the context of Big Data such limitations can be more obvious and noticeable in reducing the performance offered by the data structure and search algorithms applied to it because of big data characteristics such as its huge volume. Also, string data sets can be more challenging. They can cause more overhead in preprocessing. Some string datasets such as personal names may have some skew that has negative effect on hash indexes by increasing the ratio of collisions produced. The work in this paper offers a technique for improving the famous Hash table Data Structure by reducing the collision ratio in hashing in the average case, thus reducing the total number of comparisons made by search algorithm. The advantage of reducing the collision ratio was achieved by building a hybrid approach making use of three common data structures, namely Hash Table, B-Tree and Linked List. The results obtained showed noticeable improvement for time complexity of data searching algorithm over the traditional data structures that were used for comparisons.

Key words: Big Data, Indexing, B-Tree, Hash Table, Linked List, Recurring Keys, complexity.

1. Introduction

Big Data refers to structured and unstructured massive and complex datasets that traditional processing techniques and/or algorithms fail at properly handling its challenges and fulfilling its processing requirements[1]. These properties of big data emphasize the need for search algorithms and data structures used to offer high performance. Since data structures plays an important role in determining search algorithm potential, improving data structures used is a key step towards achieving better performance. Some of the most common data structure used to index datasets are B-Tree and Hash Table. B-Tree is a widely known and used data structure that can be used to index large datasets. B-Tree is a balanced search tree that represent a natural generalization of binary search tree data structure. It consist of a set of nodes and pointers that link them. A node in a B-Tree can have many tree pointers(children) with each two tree pointers separated by a key that represents a means to guide the search through the tree. B-Tree has search complexity of O(Log n) in the worst case[2].
Hash tables are data structure that can be used to search through an indexed dataset. A Hash table is basically a table that uses a randomizing function(called the hash function) to assign each indexed element into a specific location determined by the hash function. Hash Tables are appealing because of the interesting property of offering O(1) search time complexity in best case. on the other hand they suffer from the probability of the occurrence of collisions(two items must be stored in the same place according to the hash function). In this case ,it may be appropriate to try to employ these indexes in a way that reduces some of their limitations to a specific degree. In the context of big data , both of these data structures have some significant limitations. B-Tree’s O(log n) search time complexity can be considered large for large datasets , and Hash Table’s O(1) search time complexity can degrade to O(n) in the worst case due to the occurrence of collisions. Linked List data structure organize elements as sequential group of elements.it uses pointers(links) to keep the order of elements in the list. Linked List works well in insertion and deletion operations, yet it’s search complexity is O(n).

2. Problem statement
Improving existing index data structures by making some changes to these data structures and using hybrid technique in order to reduce the time complexity required to perform searching algorithm on the index.

3. Literature review
(Wu, X. et al., 2019) [3] proposed an ordered index data structure that makes use of the strengths of three data structures, namely : hash table, prefix tree, and b+ tree, to come up with a single fast ordered index that has O(Log L) worst-case time complexity for searching a key with a length of L. publicly available datasets collected at Amazon. com and MemeTracker.org were used.

(Breslow, A. D. et al., 2016) [4] presents the Horton table, a revamped BCHT(bucketized cuckoo hash table) that reduces the expected cost of positive and negative lookups to fewer than 1.18 and 1.06 buckets, respectively, while still achieving load factors of 95%. The remap entries were proposed ,which represent small in-bucket records that allow items that overflow buckets to be tracked and rehashed with one of alternate hash functions, and reduces the majority of negative searches to 1 bucket access.

(Xia, F. et al., 2017) [5] presents Hybrid Index Key-Value Store (HIKV) , a key-value store that focuses on the idea of building hybrid index in hybrid memory. HiKV uses a hash table and a B+-tree and it selects the respective memory part that each part of the index is to be stored at with the thought of maintaining the consistency of the two indexes. The hash table is built in Non-Volatile Memory(NVM) so as to retain it’s fast index searching. B+-tree on the other hand is built in Dynamic Random Access Memory (DRAM), so as to avoid long NVM writes. HiKV when using single_thread outperforms the state-of-the-art NVM-based key-value stores by reducing latency up to 86.6%, and for multi-threaded performance, HiKV increases the throughput by up to 6.4x under Yahoo! Cloud Serving Benchmark (YCSB) workloads.

(Nguyen, M. K. et al., 2010) [6] proposes a novel type of index structure named B+Hash Tree ,that combine the strengths of B-Tree data structure and the fast search capability of hash based structure. The main idea is using a hash map to improve the B+ Tree in order to allow a constant retrieval time instead of the logarithmic time of B+ Tree. The index was evaluated against existing RDF indexing schemes, And it was showed that a B+Hash Tree is at least twice as fast as its competitors, And that this speed should grow as dataset sizes increase.
4. Data set
In this paper, data was collected from real data sets of resident patients at the public and private hospitals of several cities in Iraq. Collected data belong to several years, and contains more than 30 attributes, including Personal name, age, diagnosis, date, job … etc. data collected contains 607,119 record. With respect to recurring personal names, a high ratio of recurring names was found. The most useful attribute for this research was personal names written in Arabic Language. A sample of the dataset is shown in Table 1 where names and mother names were eliminated to keep privacy of patients. Some Data Preprocessing steps were performed such as a aggregation of data from many excel files into one file, extraction of personal names, elimination of null names (After elimination of null names, Dataset had 60007 names), elimination of Characters not belonging to Arabic language letters (slashes, “parsing signs”, etc). In this paper, Personal names (String keys) were converted into natural numbers using equation 1. Equation 1 can be described as follows[2] :
\[ C_i = C_i + J \times B^L \]

Where Ci is the natural number that corresponds to the string key Ki (Ki being an element of the set of personal names or one single name out of a full personal name), J is the Letter Sequence Of the letter Ki,L (Ki,L means the letter at position L of key Ki) in alphabet beginning from ‘_’=0 and ‘@’=1 …etc., and B is the base. B for example can have one of these values 31, 33, 35, 37.

Table 1 Sample of collected data (attribute “name” was eliminated).

| name  | hlt | hosp | moh1 | staled | iyy | moh | adah | status | age | d | hyp | oxy | hlt | hosp | moh | staled | iyy | moh | adah | status | age | d | hyp | oxy |
|-------|-----|------|------|--------|-----|-----|------|--------|-----|---|-----|-----|-----|-----|-----|--------|-----|-----|------|--------|-----|---|-----|-----|
| 3     | 1   | 1    | 1    | 1      | 1   | 1   | 1    | 1      | 0   | 1 | 0   | 0   | 1   | 1   | 1    | 1      | 0   | 1   | 1    | 1      | 0   | 1 | 0   | 0   |
| 3     | 1   | 0   | 1    | 0     | 3   | 1   | 1    | 0      | 1   | 1 | 0   | 1   | 0   | 1   | 0    | 0      | 1   | 0   | 1    | 0      | 1   | 0 | 0   | 0   |
| 3     | 1   | 0   | 1    | 2     | 0   | 1   | 1    | 2      | 0   | 1 | 0   | 1   | 0   | 1   | 0    | 0      | 1   | 0   | 1    | 0      | 1   | 0 | 0   | 0   |
| 3     | 1   | 1    | 1    | 3    | 0   | 1   | 1    | 1      | 0   | 1 | 0   | 1   | 0   | 1   | 0    | 0      | 1   | 0   | 1    | 0      | 1   | 0 | 0   | 0   |
| 3     | 1   | 2    | 3    | 3    | 0   | 1   | 1    | 0      | 1   | 1 | 0   | 1   | 0   | 1   | 0    | 0      | 1   | 0   | 1    | 0      | 1   | 0 | 0   | 0   |

5. Proposed index
In this section, a description of the proposed index structure is provided, along with basic operations algorithms.

6. Overview of index structure
As mentioned previously, the proposed index structure combines three data structures, namely, B-Tree, hash table, and linked list. We proceed to explain each structure. The B-Tree used has the same structure as an ordinary B-Tree with two differences. The first one is a limited allowed maximum number of levels that need to be specified before inserting any keys into the whole structure. The
second one is that each node in the B-Tree has aside from ordinary B-Tree contents, an array. This array can be expanded under certain conditions into a linked list (at internal nodes) or into a hash table (at leaf node).

The hash table structure is built under certain conditions at leaf nodes by expanding a leaf node’s associated array as mentioned above so that each leaf node can build its own hash table as needed. Each hash table structure is represented as an array that uses chaining to solve collisions. Each hash table array element has a pointer and therefore can build a linked list to represent the chained list of collided keys. The difference from ordinary hash table structure is that each element in the chained list has aside from its first pointer that helps to create the chained list, a second pointer that can be used to group identical collided elements, therefore reducing the length of the chained list the search has to go through when dataset include some recurring keys.

The third part of the structure is the linked list. One case where linked list is used is the case mentioned above in hash table chained lists. The second one is in internal nodes linked lists. The array associated with each node has in each element two pointers. This array is used to expand under certain conditions into an array that has only one element (at internal nodes), and that element can form a linked list by using the first pointers to form the linked list, this can be referred to as the “first linked list”. Elements are added to this first linked list if they are identical to a key stored in the internal node. The second pointer at each element in the first linked list is expanded as needed into a second linked list that holds values of recurring keys identical to the corresponding key stored in the first linked list where the 2nd linked list was initiated. The goal of using this second linked list is also to reduce length of the first linked list when dataset include recurring keys so as to speed up searching process.

7. Index construction
The structure construction begins by inserting the 1st key into the B-Tree. After that, each key to be inserted is searched for in B-Tree first, if it is found, the new key (or its value) is added to Linked List/Hash Table associated to that node holding key. Note that Each key added to the associated Linked List/Hash Table of a node is added once along with its value, and after that all recurring keys of this key won’t be added to linked list/hash table but instead their associated values will be attached to key occurrence already present in linked list/hash table. If key is not found in B-Tree, it is inserted into it if possible. What makes key insertion into B-Tree not always possible is maximum allowed number of levels.

Once actual number of levels reached maximum allowed number, any further split of full root node during insertion would cause a violation for the condition of keeping levels \( \leq \) maximum allowed levels. Therefore, insertions are allowed when actual number of levels reaches maximum allowed number as long as they do not require splitting root. If they require splitting root, insertion is conducted in appropriate hash table (the hash table associated with the leaf node that \( k \) belongs to its range of keys). The insertion procedure proceeds to find appropriate leaf node and insert \( k \) into its associated hash table. This insertion into hash table requires searching for key since it might exist previously in the hash table, in which case only the new value is attached to the key that is already present in table. If it is not present in hash table, \( k \) and its value are added to hash table.

An example that demonstrates inserting into BT-HT-LL is explained next. Let BT be a B-Tree of order \( P=4 \), and of maximum number of levels \( = 2 \). Let \( S \) be the set of keys to be inserted. \( S \) holds the following keys: \( S=\{1,2,3,4,1\} \). Insertion of these keys in order is depicted in Figure 1. Figure 2 depicts the insertion of 1 into the structure firstly produced in Figure 1.
8. Insertion algorithm in BT_HT_LL

Insertion of key k into BT_HT_LL index follows the following steps:

1. Search BTREE for key k, if found in B-Tree, add k and its value(or its value alone) to Linked List or Hash Table associated with that node holding k.
2. If not found in B-Tree:
   a. Insert k and its value into B-Tree if possible(if no further split after number of levels reached maximum allowed number is required)
   b. If inserting k into B-Tree is not possible, find the node holding appropriate hash table to contain k, search k in appropriate hash table, if found add its value there, and if not add k and its value to the hash table.

9. Searching algorithm in BT_HT_LL

Searching for all occurrences of key k in BT_HT_LL can be described as follows:

1. Searching B-Tree for key k. Then if k is found in B-Tree searching proceed to search associated linked list or hash table for recurrences.
2. If k is not found in B-Tree, searching proceeds to search Chained List at hash table that k is supposed to be stored in if it is stored in the index at all.
10. Deletion algorithm in BT-HT-LL
Deletion in BT_HT_LL depends basically on Ordinary BTree-Delete procedure with preemptive merging found in[2] . Ordinary BTree-Delete procedure is tuned to work with the proposed index. The most important points of difference are as follows :
- In BT_HT_LL, keys can have values list related to them, therefore, once a key is deleted, replaced, or moved to another location, its values list should also be handled according to the operation being conducted on k.
- In BT_HT_LL, if k is not found in BTree, it still might exist in appropriate hash table, therefore, it is searched and deleted from there if it is found in hash table.
- When merging nodes, keys in associated linked lists or hash tables should also be moved to the merged node along with their values lists. In merging leaf nodes, keys that are stored only in hash tables should also be moved to the merged node.

11. Indexes used for comparisons
For the sake of conducting comparisons, an additional index was built that represents a part of BT_HT_LL. This additional index is called HashTable-LinkedList. This index is a simple index that consist of a standard hash table that uses chaining to resolve collisions. The difference from ordinary hash table is that inside each chained list, identical keys are grouped together in linked lists as explained in BT_HT_LL. Therefore, each chained list contains unique keys belonging to this hash table location put in one sequence(the chained list) and recurring values for each key are stored aside from this main chained list in the key’s associated linked list. Each unique key in a chained list has a value and associated set of values that belong to it. The application of insertion, searching, deletion is then straightforward. This index can give some informations about ordinary hash table as well since number of keys that were stored during insertion in a chained list can be counted through keys themselves or their values if recurring keys were inserted.

With respect to BTree , [7] describes a BTree that assumes all keys are unique. If keys have duplicates, BTree algorithms can be modified to work with duplicate keys. Insertion was sufficient to conduct insertion of duplicate keys. Searching was tuned by searching recursively through surrounding sub-trees that surround each instance of k in the node where k is first found. Deletion also was tuned and it was conducted by treating each instance of key k as a separate key and deleting it separately, so as long as key k is still found in B-tree, deletion procedure is called again to delete it. This tuned version will be referred to as “Tuned-B-Tree”. Note that in time complexity table , a variable related to number of occurrences of the key may be required to be multiplied by complexity found in table. This was ignored in table because the described tuning of BTree is a straightforward tuning and better, more efficient alternatives may exist. The space complexity described in space complexity table is based on the tuned version.

12. Experiments and results
This subsection describes experiments conducted to evaluate BT_HT_LL. Basically, BT_HT_LL provides a potential improvement on time complexity of B-Tree and Hash Table. Therefore, following experiments were conducted:
1. Average length of chained list in hash tables of BT_HT_LL versus in chained list of standard hash index table. Results of 1st experiment are given in figure 3, figure 4, and they are also shown using tables in table 2 and table 3. Figure 3 shows average Length of Chained List in BT_HT_LL hash tables, while figure 4 shows average Length of Chained List in standard hash table. It can be noticed from results in figure 3 and figure 4 that difference in average lengths of chained lists between the two indexes .
Figure 3 Average length of chained list in BT_HT_LL hash tables. A. Total tables size = n. B. Total tables size = 2n. C. Total tables size = 3n.

Figure 4 Average length of chained list in standard hash table.

Table 2 Average length of chained list in BT_HT_LL hash tables.

| B-Tree Order P | Hash table size = n | Hash table size = 2n | Hash table size = 3n |
|----------------|---------------------|---------------------|---------------------|
|                | Max Levels = 3      | Max Levels = 4      | Max Levels = 5      |
| 10             | 0.6860641667        | 0.673215            | 0.619685            |
| 15             | 0.619685            | 0.335659            | 0.208192            |
| 20             | 0.340321            | 0.3098425           | 0.09253             |
| 50             | 0.226881            | 0.223563            | 0.206562            |
|                | 0.059483            | 0.059483            | 0.059483            |
| 600007         | 0.606923333         | 0.606923333         | 0.606923333         |
| 1200014        | 0.206356            | 0.206356            | 0.206356            |
| 1800021        | 0.130462            | 0.289882            | 0.0884384           |
| 2400028        | 0.335659            | 0.0884384           | 0.0884384           |
| 308576         | 0.220308            | 0.190494            | 0.056279            |
| 4194304        | 0.0884384           | 0.190494            | 0.056279            |
| 500000         | 0.059483            | 0.059483            | 0.059483            |
| 16777216       | 0.177568            | 0.063125            | 0.063125            |
| 2.68E+08       | 0.059483            | 0.059483            | 0.059483            |

Table 3 Average length of chained list in Standard Hash Table.

| Hash Table Size | Average Length of Chained List |
|-----------------|-------------------------------|
| 600007          | 1                             |
| 1200014         | 0.5                           |
| 1800021         | 0.33333                       |
| 2400028         | 0.25                          |
| 308576          | 0.572211                      |
| 4194304         | 0.143053                      |
| 500000          | 0.035763                      |
| 16777216        | 0.008941                      |
| 2.68E+08        | 0.002235                      |

2. Average length of chained list in hash table of HashTable_Linked List index versus average length of chained list of standard hash index table. HashTable_Linked List index is used instead of BT_HT_LL to isolate the effect of storing some keys in B-Tree in reducing collisions in hash tables. This demonstrates the effect of adding Linked List to hash table. Results of experiment 2 are given in figure 5 and table 4. Figure 5 shows average length of chained list in HashTable_Linked List index versus average length of chained list in standard Hash Index table. Due to use of linked lists to group identical keys, searching can stop once finding the first occurrence in HashTable_LinkedList index (and BT_HT_LL), while in standard Hash index table, it has to examine the whole Chained List.
Figure 5 Length of chained list in Hashtable_LinkedList index versus its length in standard Hash Index table.

Table 4 Average length of chained list in HashTable_LinkedList index Vs. in Standard Hash Index.

| Hash Table Size      | Average Length of Chained List in Standard Hash Index | Average Length of Chained List in HashTable_LinkedList index |
|----------------------|--------------------------------------------------------|-------------------------------------------------------------|
| 600007               | 1                                                      | 0.68447                                                     |
| 1200014              | 0.5                                                    | 0.342235                                                    |
| 1800021              | 0.333333                                               | 0.228157                                                    |
| 2400028              | 0.25                                                   | 0.171118                                                    |
| 3000034              | 0.572211                                               | 0.391662                                                    |
| 1048576              | 0.143053                                               | 0.097915                                                    |
| 4194304              | 0.035763                                               | 0.024479                                                    |
| 16777216             | 0.008941                                               | 0.00612                                                     |
| 67108864             | 0.002235                                               | 0.00153                                                     |
| 2.68E+08             |                                                        |                                                             |

3. Number of total collisions in all hash tables of BT_HT_LL versus number of total collisions in standard hash index table(or HashTable_LinkedList index). Total number of Collisions found in all hash tables in BT_HT_LL index is shown along with other values in tables 5, 6, and 7. Table 8. shows total number of collisions in HashTable_LinkedList index. Examining these tables, It can be noticed that for some values of p and max levels, count of collisions in BT_HT_LL can drop below count of collisions in HashTable_LinkedList (and standard hash table).

Table 5 Total number of collisions in BT_HT_LL hash tables for total hash tables size ~=n.

| B-tree order p | 10     | 10     | 10     | 15     | 15     | 20     | 20     | 15     | 50     | 100    |
|----------------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| B-Tree Max number of Levels | 3     | 4     | 5     | 3     | 4     | 3     | 4     | 5     | 3     | 2     |
| B-Tree Actual number of Levels | 3     | 4     | 5     | 3     | 4     | 3     | 4     | 5     | 3     | 2     |
| Hash table size | 600    | 600    | 600    | 600    | 600    | 600    | 600    | 600    | 600    | 600    |
| Number of Hash Tables(even null hash tables) | 1000   | 1000   | 10000  | 3375   | 50625  | 8000   | 160000 | 759375 | 125000 | 10000  |
| Total Size of Hash Tables | 600000 | 600000 | 600000 | 597375 | 556875 | 600000 | 480000 | 759375 | 500000 | 600000 |
| Count of Collisions in All Hash Tables | 415259 | 446789 | 412380 | 421132 | 425929 | 364974 | 212569 | 223444 | 245956 | 201305 |
| Average total number of keys in Chained List between All Hash Tables | 0.994949 | 0.964999 | 0.79353 | 0.98318 | 0.89826 | 0.90966 | 0.35699 | 0.21636 | 0.36482 | 0.30963 |
| Average Length of Chained List between All Hash Tables | 0.60694 | 0.66092 | 0.53270 | 0.67321 | 0.60611 | 0.61968 | 0.20635 | 0.12625 | 0.20819 | 0.1762 |
| Average Collisions in Chained list Between all Hash Tables | 0.69209 | 0.74464 | 0.6873 | 0.70497 | 0.76855 | 0.60829 | 0.44285 | 0.29424 | 0.49191 | 0.33550 |
| Average number of Collisions in All hash tables | 415.259 | 44.6789 | 4.1238 | 124.779 | 8.41341 | 45.6217 | 1.32855 | 0.29424 | 1.96764 | 20.1305 |
### Table 6: Total number of collisions in BT_HT_LL hash tables for total hash tables size~2^n.

| B-tree Order P | 10  | 10  | 10  | 15  | 15  | 15  | 20  | 20  | 20  | 50 | 100 |
|----------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|----|----|
| B-tree Max number of Levels | 3   | 4   | 5   | 3   | 4   | 3   | 4   | 4   | 3   | 6  | 3  |
| B-tree Actual number of Levels | 3   | 4   | 5   | 3   | 4   | 3   | 4   | 4   | 6  | 3  | 2  |
| Hash Table size | 1200 | 120  | 12  | 355 | 23  | 150 | 7  | 1   | 9  | 120 |
| Number of Hash Tables (even null hash tables) | 1000 | 10000 | 100000 | 3375 | 50625 | 8000 | 16000 | 1000000 | 125000 | 10000 |
| Total Size of Hash Tables | 1200000 | 1200000 | 1200000 | 1198125 | 1164375 | 1200000 | 1120000 | 10000000 | 12500000 | 12000000 |
| Count of Collisions in All Hash Tables | 340616 | 372139 | 357159 | 344411 | 361585 | 298076 | 293105 | 222340 | 193942 |
| Average Actual number of keys in chain List between all Hash Tables | 0.49747 | 0.48249 | 0.39676 | 0.49020 | 0.42960 | 0.45483 | 0.15299 | 0.15539 | 0.16214 | 0.15481 |
| Average Length of Chained List between all Hash Tables | 0.34032 | 0.33046 | 0.30984 | 0.33565 | 0.28988 | 0.30984 | 0.08843 | 0.09133 | 0.09253 | 0.0881 |
| Average Collisions in Chained list Between all Hash Tables | 0.28384 | 0.31101 | 0.29763 | 0.28745 | 0.31054 | 0.24839 | 0.17241 | 0.20096 | 0.19763 | 0.16161 |
| Average Collisions in All hash tables | 340.616 | 37.2139 | 3.57159 | 102.047 | 7.14242 | 37.2595 | 1.20690 | 0.20096 | 1.77872 | 19.3942 |

### Table 7: Total number of collisions in BT_HT_LL hash tables for total hash tables size~3^n.

| BTree Order P | 10  | 10  | 10  | 15  | 15  | 15  | 20  | 20  | 20  | 50 | 100 |
|----------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|----|----|
| Btree Max number of Levels | 3   | 4   | 5   | 3   | 4   | 3   | 4   | 3   | 4   | 3  | 2  |
| Btree Actual number of Levels | 3   | 4   | 5   | 3   | 4   | 3   | 4   | 3   | 4   | 3  | 2  |
| Hash Table size | 1800 | 180  | 18  | 533 | 35  | 2   | 225 | 11   | 14  | 180 |
| number of Hash Tables (even null hash tables) | 1000 | 10000 | 100000 | 3375 | 50625 | 759375 | 8000 | 160000 | 1250000 | 10000 |
| Total Size of Hash Tables | 1800000 | 1800000 | 1800000 | 1798875 | 1771875 | 1518750 | 1800000 | 1760000 | 1750000 | 1800000 |
| Count of Collisions in All H Tables | 302606 | 330024 | 321682 | 304243 | 323075 | 204273 | 267492 | 185313 | 211262 | 191447 |
| Average Actual number of keys in chain List between All Hash Tables | 0.33164 | 0.32166 | 0.26451 | 0.32649 | 0.28231 | 0.10818 | 0.30322 | 0.09736 | 0.10423 | 0.10321 |
| Average Length of Chained List between All Hash Tables | 0.22688 | 0.22030 | 0.17756 | 0.22536 | 0.19049 | 0.06312 | 0.20656 | 0.05627 | 0.05948 | 0.05873 |
| Average Collisions in Chained list Between all Hash Tables | 0.16811 | 0.18334 | 0.17871 | 0.16913 | 0.18233 | 0.13450 | 0.14860 | 0.10529 | 0.12072 | 0.10635 |
| Average Collisions in All hash tables | 302.606 | 33.0024 | 3.21682 | 90.1460 | 6.38172 | 0.26900 | 33.4365 | 1.15820 | 1.69009 | 19.1447 |

### Table 8: Total number of collisions in the hash table of HashTable_LinkedList Index.

| Hash Table Size | 600007 | 1200014 | 1800021 | 2400028 | 408576 | 4194304 | 16777216 | 67108864 | 2.68E+08 |
|----------------|-------|---------|--------|--------|--------|-----------|---------|---------|---------|
| Count of Collisions in the Hash table | 302632 | 252150 | 232817 | 222409 | 260204 | 208910 | 194363 | 190581 | 189652 |

### 13. Results analysis

Analysis of the results obtained in subsection 8 is given in this subsection. Following the same order of listing Results in subsection 8, analysis of these results in given next.

1. Experiment 1 showed Average length of chained list in hash tables of BT_HT_LL versus of chained list of standard hash index table. Number of keys searching procedure (and deletion procedure) have to examine is effected by Length of Chained List. In BT_H_LL, identical keys are grouped together, therefore, once a key is found, values related to that key are associated with it and searching process stops, while in standard Hash Table, searching have to continue till last element since recurring keys may appear anywhere. In Both indexes however, searching may
need to examine all elements (this happens always at standard hash table, and it could happen in BT_HT_LL if key is not present for example or is present as the last element), therefore Length of chained list can effect searching process in BT_HT_LL too. From results, BT_HT_LL gave smaller average length of chained list compared with standard Hash Table Index for the three sizes \((n, 2n, 3n)\) used of hash tables. Several factors affect how BT_HT_LL index distributes keys among hash tables and effect collisions and lengths of chained lists, these factors include:

- Use of B-Tree alongside hash tables in BT_HT_LL can significantly reduce keys stored in hash table and therefore, reduces average Length of chained list.
- recuring keys in chained lists of hash tables are put into linked lists aside from the chained list, this can reduce average length of chained list.
- Not all hash table size is used necessarily since several hash table can be null. This can increase collisions.
- Storing recuring keys of internal B-Tree nodes keys within Linked Lists attached to nodes increases the number of keys that can be accommodated in BTree nodes.
- These mentioned points contains Several factors that compete with each other. However, to reduce collisions, it might be required to increase \(p\) first, then maximum number of levels to get to a desirable outcome.

5. In experiment 2, results given in figure 4 demonstrate effectiveness of grouping identical items together in Linked Lists aside from the chained list on reducing Length of chained list and as a result number of comparisons in searching and deletion procedures on average.

6. Tables 5,6,7, and 8 show that number of total collisions in Hash tables of BT_HT_LL can be reduced bellow that number in standard Hash Index table by changing B-Tree order \(P\) and maximum levels. Several factors affect the total collisions count in BT_HT_LL including:

- B-Tree stores part of the keys apart from the hash tables.
- BT_HT_LL stores recuring keys of internal nodes’ keys aside from the B-Tree nodes in associated Linked Lists which allows BTree nodes to free some space for other keys.
- BT_HT_LL may use limited amount of available total size of hash tables due to having several tables being null, which can increase collisions.
- BT_HT_LL uses Linked List to decrease length of chained list in hash table.

14. Time and space complexity analysis

Time and space complexities are important measures that demonstrate some useful informations about an algorithm. Time complexity denotes how output of an algorithm is related to the size of its input and hence the speed of executing the algorithm. Space complexity denotes space required to implement the algorithm. Table 9 and table 10 summarize time and space complexities of various indexes including Proposed index. Table 11 demonstrates the notation used to describe time and space complexities in table 9 and table 10 and in explanation coming next. It is worth noting that regardless of specified maximum levels, BT_HT_LL’s BTree actual number of levels cannot be bigger than \(\log(\text{Unique } n)\) compared to \((\log n)\) in standard BTree, where \(\text{Unique} - n < = n\). This is due to storing recuring keys of keys held in BTREE aside from BTree nodes.

| Index                  | Insert | Search | Delete |
|------------------------|--------|--------|--------|
|                        | Best   | Average| Worst  | Best   | Average| Worst  | Best   | Average| Worst  |
| BT_HT_LL               | \(t^*\text{Act}\) | \(\text{LenCL}\) | \(\text{Maximum}[t^*\text{Unique}_n,n]\) | \(t^*\text{Act}\) | \(\text{LenCL}\) | \(\text{Unique}_n\) | \(t^*\text{Act}\) | \(\text{LenCL}\) | \(n\) |
| HashTable_LinkedList   | \(O(1)\) | \(O(1)\) | \(O(\text{Unique}_n)\) | \(O(1)\) | \(O(1)\) | \(O(\text{Unique}_n)\) | \(O(1)\) | \(O(1)\) | \(O(\text{Unique}_n)\) |
| Tuned-B-Tree           | \(O(\log n)\) | \(O(\log n)\) | \(O(\log n)\) | \(O(\log n)\) | \(O(\log n)\) | \(O(\log n)\) | \(O(\log n)\) | \(O(\log n)\) | \(O(\log n)\) |
| Hash Table             | \(O(1)\) | \(O(1)\) | \(O(n)\) | \(O(1)\) | \(O(1)\) | \(O(n)\) | \(O(1)\) | \(O(1)\) | \(O(n)\) |
Table 10 Space complexity of various indexes. Size of input is $n$.

| Index                | Best                                      | Average                                  | Worst                                      |
|----------------------|-------------------------------------------|------------------------------------------|--------------------------------------------|
| BT_HT_LL             | $2n$ (All hash tables are null)           | $8sst(P^{Act}) + 3n + 2/3sst$            | $12sst(P^{Act}) + 5n + sst$                |
| Tuned-B-Tree         | $n$                                       | $n$                                      | $n$                                        |
| HashTable_LinkedList | $\min\{6S-2n, 12S\}$                     | $10S$                                    | $12S + 2n$                                 |
| Hash Table           | $5/2 S$                                   | $6S + 11/18S + 1/6n$                     | $\max\{25/3 S + n, 9S - 1/3n\}$          |

Table 11 Notation used in table 9 and table 10.

| $k$  | The key                        |
|------|--------------------------------|
| $P$  | B-Tree Order = number of children at node |
| $t$  | $P/2$                          |
| $n$  | Dataset size                   |
| Unique-n| Number of Unique keys in $n$    |
| LenCL| Length of chained list in hash table, max of LenCL is $n$. |
| LenLL| Length of Linked List associated with internal nodes, max of LenLL is $P-1$. |
| Max  | Maximum allowed number of levels in B-Tree |
| Act  | Actual number of levels in B-Tree |
| sst  | Size of single hash table. This is used in BT_HT_LL since it contains multiple hash tables. |
| $S$  | Size of the hash table counting by indexes. Used when the index is designed to contain one hash table only such as HT_LL. |

15. Conclusion

From the result obtained in tables 2 through 8, it can be concluded that:

1. BT_HT_LL improves time complexity due to grouping identical keys together, limiting searching in B-tree by limiting its maximum levels to reduce B-Tree searching time, and allowing use of hash tables to continue searching.

2. By looking at expected and counted total collisions from experiments and count of identical keys found in real data, Grouping of identical keys in hash table-LinkedList has significant impact on time efficiency of the hash table searching. Since this grouping is also used in BT_HT_LL, it can be impactful on BT_HT_LL as well.

16. Recommendations

From the results obtained in tables 2 through 8, It can be recommended that:

1. For small amount of data, HashTable-LinkedList Indexing technique can be used since basic operations are simple to implement, and this index can be sufficient for this data since $n$ is small and worst case is acceptable.

2. For moderate and big size of data, BT_HT_LL has several features and tunable parameters that can improve time efficiency of B-Tree and Hash Table.

References

[1] Taylor-Sakyi, K. 2016. Big data: Understanding big data. arXiv preprint arXiv:1601.04602.
[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. 2009. Introduction to algorithms.
MIT press.

[3] Wu, X., Ni, F., & Jiang, S. 2019, March. Wormhole: A Fast Ordered Index for In-memory Data Management. In Proceedings of the Fourteenth EuroSys Conference 2019 (p. 18). ACM.

[4] Breslow, A. D., Zhang, D. P., Greathouse, J. L., Jayasena, N., & Tullsen, D. M. 2016. Horton tables: fast hash tables for in-memory data-intensive computing. In 2016 {USENIX} Annual Technical Conference ({USENIX} {ATC} 16) (pp. 281-294).

[5] Xia, F., Jiang, D., Xiong, J., & Sun, N. 2017. HiKV: a hybrid index key-value store for DRAM-NVM memory systems. In 2017 {USENIX} Annual Technical Conference ({USENIX} {ATC} 17) (pp. 349-362).

[6] Nguyen, M. K., Basca, C., & Bernstein, A. 2010. B+ Hash Tree: Optimizing query execution times for on-Disk Semantic Web data structures. SSWS2010, 96-111.

[7] Elmasri, R., & Navathe, S. 2010. Fundamentals of database systems. Addison-Wesley Publishing Company.

[8] Galakatos, A., Markovitch, M., Binnig, C., Fonseca, R., & Kraska, T. 2018. A-tree: A bounded approximate index structure. arXiv preprint arXiv:1801.10207.

[9] Helmer, S., Neumann, T., & Moerkotte, G. 2003, November. A robust scheme for multilevel extendible hashing. In International Symposium on Computer and Information Sciences (pp. 220-227). Springer, Berlin, Heidelberg.

[10] Adamu, F. B., Habbal, A., Hassan, S., Malaysia, U. U., Cottrell, R. L., White, B., ... & Malaysia, U. U. 2016. A survey on big data indexing strategies (No. SLAC-PUB-16460). SLAC National Accelerator Lab., Menlo Park, CA (United States).