ISSN (Online) : 0974-5645
ISSN (Print) : 0974-6846
Indian Journal of Science and Technology, *Vol 8(4), 364–375, February 2015* DOI: 10.17485/ijst/2015/v8i4/59865

An Analysis on the Performance of Tree and Trie based Dictionary Implementations with Different Data Usage Models

M. Thenmozhi[1*] and H. Srimathi[2]

[1]Department of Information Technology, SRM University, Chennai, India; mozhi_2000@yahoo.com
[2]Department of Computer Applications, SRM University, Chennai, India; srimathi.h@ktr.srmuniv.ac.in

Abstract

Tree and Trie are the Abstract Data Types (ADTs) that provide efficient implementation of ordered dictionary. The performance of a data structure will depend on hardware capabilities of the computing devices such as RAM size, Cache memory size and even the speed of the physical storage media. Hence, an application which will be running on real or virtualized hardware environment certainly will have restricted access to memory and other resources of the real hardware. Further, the time taken for any operation on a data structure rely on the data usage model and the most significant operations/tests are very much depend on the size of the "character payload objects" which we use in dictionary like implementations. In this work, we do an analysis on the performance of Tree and Trie based Dictionary ADT Implementations with different data usage models. We consider data usage models such as a typical electronic dictionary with more than million of words or a typical electronic encyclopedia with large string data elements. In this work, we studied the performance of three popular Tree based Dictionary Implementations rbtree, googlebtree, stxbtree, and three Trie based Dictionary Implementations tommy-trie, tommy-trie-inplace, nedtrie under different hardware and software configurations. Among all, tommy trie is proved to be the best for character payload objects with 16 bytes and 4096 bytes. In some operations/tests googlebtree seems to be better. Our evaluation on different tree and trie structures shows tommy trie implementations perform well irrespective of size of application.

Keywords: Cache, googlebtree, rbtree, stx btree, Tommy trie, Trie

1. Introduction

A key Decision when computing applications is the choices of a mechanism to store and retrieve strings. There are two main types of data Structures available for this task is categorized as In-memory and External Memory Data Structures. Some of the In-memory DS are Array, Linked List, Binary Search Tree, Hash Table. Tree and Trie Data structures provide simple and efficient implementation of an ordered dictionary. This paper evaluates some of the popular in-memory Tree and Trie data structures which are frequently refereed in earlier works and commonly used in ordered dictionary like applications.

Simple in-memory data structures are basic building blocks of programming, and are used to manage temporary data in scales ranging from a few items to gigabytes. For the storage and retrieval of strings, the main data structures are the varieties of hash table, trie, and binary search tree[1].

Different Tree and Trie implementations use different approaches in their design. Here we study the performance of Tree based Dictionary Implementations rbtree, googlebtree, stxbtree, and three Trie based Dictionary Implementations tommy-trie, tommy-trie-inplace, nedtrie with two different string payload sizes. We will evaluate the performance under different hardware and

*Author for correspondence

software configurations. The main tests will be made on these data structures are Insert, Change, Hit, Miss and Remove.

In this evaluation, time is considered as a main metric which will be measured in nanoseconds for above mentioned five operations/tests. All these five tests and operations will be done using keys in Random Mode as well as Forward mode.

2. The Dictionary Adt Implementation with Tree and Trie Data Structure

ADT is a mathematical model for a certain class of data structures that have similar behavior; or for certain data types of one or more programming languages that have similar semantics. A tree and trie are widely used Abstract Data Type (ADT) or data structure implementing this ADT that simulates a hierarchical tree structure, with a root value and sub trees of children, represented as a set of linked nodes.

2.1 The Abstract Data Type/Structure Dictionary

Dictionary is a data structure which can stores key–element pairs. It allows "look-up" or find operation and allows insertion/removal of elements. A dictionary may be unordered or ordered and the key must support equality operator. For ordered dictionary, the key also supports comparator operator which will be useful for finding neighboring elements. In most of the applications, the keys are required to be unique.

Dictionary Examples

(i) Natural language dictionary
    word is key
    element contains word, definition, pronunciation, etc.
(ii) Web pages
    URL is key
    html or other file is element

The dictionaries are also known as associative arrays or maps. In programming, the abstract data structure dictionary is represented by many aggregated pairs (key, value) along with predefined methods for accessing the values by a given key.

Operations of a Typical Dictionary ADT

(i) Dictionary create()
    creates empty dictionary
(ii) boolean isEmpty(Dictionary d)
    tells whether the dictionary d is empty
(iii) put(Dictionary d, Key k, Value v)
    associates key k with a value v
    if key k already presents in the dictionary
    old value is replaced by v
(iv) Value get(Dictionary d, Key k)
    returns a value, associated with key k
    or null, if dictionary contains no such key
(v) remove(Dictionary d, Key k)
    removes key k and associated value
(vi) destroy(Dictionary d)
    destroys dictionary d

2.1.1 Existing Data Structures

2.1.1.1 Binary Search Trees

In a Binary Search Tree (BST), each tree node stores a string and two pointers to left and right child nodes. A search will be done by comparing the root value every time and branch will be taken either right or left. As the search progresses, the number of characters inspected at each string comparison gradually increases.

Although the allocation of strings to nodes is determined by the insertion order, for a skew distribution it is reasonable to expect that common words occur close to the beginning of the text collection and are therefore close to the root of the BST. Assuming the distribution is stable, accesses to a common term should be fast, since the first levels of the tree are usually kept in cache and only a few string comparisons are required. On the other hand, if strings are inserted in sorted order or the distribution changes the behavior of a BST can be extremely poor. The BST is of limited use in practice for vocabulary accumulation. However, it is included in our discussion, since they are common data structure and serve as a yardstick in our experiments.

There are several well-known variants of BSTs that maintain balance or approximate balance, in particular AVL trees and red-black trees. With these techniques, the tree is reorganised on every insertion or deletion, thus ensuring that leaves are approximately at same depth. On the one hand, use of rebalancing ensures that for n nodes there is an O(log n) upper limit to the length of a search path. On the other hand, the rebalancing does not

consider the frequency of access to each node, so common words can be placed in leaves. We have experimented with red-black trees[2], and observed that the overall effect is detrimental, with red-black trees slightly faster only for data with low skew, and standard BSTs faster for typical vocabulary accumulation.

2.1.1.2 Splay Trees

A splay tree[4] is a variant of a BST, in which, on each search, the node accessed is moved to the root by a series of node rotations, an operation known as splaying. Intuitively, commonly- accessed nodes should remain near the root (and in the CPU cache), thus allowing them to be accessed rapidly; the tree quickly adapts to local changes in vocabulary; and the use of splaying guarantees that the a mortised cost of accessing a tree of n nodes is at most O(log n).

BST, a splay tree requires more memory, since an efficient implementation of splaying requires that each node have a pointer to its parent. Also, even a common word such as "the" can be moved far from the root between searches for it; even in data in which it is every seventeenth word or so, it is often found deep in the tree. Moreover, splaying is a rather expensive reorganisation technique. We investigated variations of splaying[2] and found that the most efficient is to only splay after every after k accesses, with say k = 11, thus moving common words close to the root but reducing the total cost of reorganisation. We report experiments with both splaying at all accesses and intermittent splaying with an interval of 4 accesses, as this value of k worked well on our data.

Another form of BST that reorganises on each access is the Randomised Search Tree (RST)[3], in which each node uses an additional number r, initially zero. RSTs are maintaining in order traversal and also heap property which is maintained by the numbers r. So the number in a node is at least as large as both of its children. The tree is then reorganised using rotations to restore the treap property.

In the field of mathematics, the spanning tree T[10] of an undirected graph is a sub graph that includes all vertices of G that is a tree. In general a graph will have several spanning trees. This technique is used in applications where minimum numbers of edges are required with less cost.

Decision tree is another type of Tree, which is used in applications where there are more chances of Decisions and their possible consequences, and outcomes. It is a way of displaying an algorithm[11].

2.1.1.3 Tries and Ternary Search Trees

A trie is an alternative to a BST for storing strings in sort order[5]. A node in a standard trie is an array of pointers, one for each letter in the alphabet, with an additional pointer for the empty string. A leaf is a record concerning a particular string. A string is found in a trie by using the letters of the string to determine which pointers to follow. For example, if the alphabet is the letters from 'a' to 'z', the first pointer in the root corresponds to the letter 'a'; the node T indicated by this pointer is for all strings beginning "a-". In node T, the pointer corresponding to the letter 'c' is followed for all strings beginning "ac-"; the pointer corresponding to the empty string is for the record concerning the single-letter string "a".

Search in a trie is fast, requiring only a single pointer traversal for each letter in the query string. That is, the search cost is bounded by the length of the query string. With a small increase in complexity of implementation, a trie can be substantially reduced in size by omitting nodes that lead to a single leaf.

There are several variant forms of tries with reduced space requirements compared to standard tries, such as Ternary Search Trees (TSTs)[6] and compact tries. In a TST, each node represents a single character c, and has three pointers. The left (respectively, right) pointer is for strings that start with a character that alphabetically precedes (respectively, follows) c. Thus a set of TST nodes connected by left and right pointers are a representation of a trie node. These can be rebalanced on access. TSTs are slower than tries, but more compact.

2.2 Dictionary Implementation with Tree

A tree data structure can be defined recursively (locally) as a collection of nodes (starting at a root node), where each node consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

Tree structures support various set operations including Search, Predecessor, Successor, Minimum, Maximum, Insert, and Delete in time proportional to the height of the tree. Ideally, a tree will be balanced and the height will be log n where n is the number of nodes in the tree. To ensure that the height of the tree is as small as possible and therefore provide the best running time, a balanced tree structure like a red-black tree, AVL tree, or b-tree must be used.

When working with large sets of data, it is often not possible or desirable to maintain the entire structure in primary storage (RAM). Instead, a relatively small portion of the data structure is maintained in primary storage, and additional data is read from secondary storage as needed. Unfortunately, a magnetic disk, the most common form of secondary storage, is significantly slower than Random Access Memory (RAM). In fact, the system often spends more time retrieving data than actually processing data.

B-trees are balanced trees that are optimized for situations when part or all of the tree must be 'maintained in secondary storage such as a magnetic disk. Since disk accesses are expensive (time consuming) operations, a b-tree tries to minimize the number of disk accesses. The worst case height is O(log n). Since the "branchiness" of a b-tree can be large compared to many other balanced tree structures, the base of the logarithm tends to be large; therefore, the number of nodes visited during a search tends to be smaller than required by other tree structures. Although this does not affect the asymptotic worst case height, b-trees tend to have smaller heights than other trees with the same asymptotic height.

2.2.1 Trie

Trie is a graph where each path is a string. When you add a string to the trie you start with the first letter and check if there's a path that start from the beginning point (the root) and goes to that letter, if there is you go to that path and search again from that path, this time the next letter and so on, if you find a letter that doesn't have a path you add it and from that on you add all the other letters. Similarly build the rest of the path since it's a new path. At the end of each word path, you use a special sign to designate it as end of path

Searching the trie is very similar to adding a word. You start with the beginning of the trie and the word, and search for a path with the first letter, and then you go to that path and search again for a path for the next letter. This continues until we find a end sign. The difference between searching and adding is that when you don't find a path to the next letter you just quit. Because it means the word does not appear in the trie.

Suffix tree-it is a way to use trie in order to build something a bit different. You build trie for only one word. But it's not only the one word, it's the word and all the suffixes starting with 1 letter all the way to the whole word (e.g. for Cola the trie will contain the words: Cola, ola, la, a and empty word) this kind of database is used when you want to check if a word is contained in another word, you make suffix tree of the containing word and search the contained word in it.

Trie, or prefix tree, is an ordered multi-way tree data structure that is used to store strings over an alphabet. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. Each node contains an array of pointers, one pointer for each character in the alphabet and all the descendants of a node have a common prefix of the string associated with that node. The root is associated with the empty string and values are normally not associated with every node, only with leaves.

A trie is a tree data structure that allows strings with similar character prefixes to use the same prefix data and store only the tails as separate data. One character of the string is stored at each level of the tree, with the first character of the string stored at the root.

To access these information nodes, we follow a path beginning from a branch node moving down each level depending on the characters forming the key, until the appropriate information node holding the key is reached. Thus the depth of an information node in a trie depends on the similarity of its first few characters (prefix) with its fellow keys. Here, while AEROPLANE and TRAIN occupy shallow levels (level 1 branch node) in the trie, CAR, CARRIAGE, CARAVAN have moved down by 4 levels of branch nodes due to their uniform prefix "CAR".

Prefix trees are a bit of an overlooked data structure with lots of interesting possibilities. Trie is an interesting data-structure used mainly for manipulating with Words in a language. Trie has a wide variety of applications in

- Spell checking. Word completion
- Data compression
- Computational biology
- Routing table for IP addresses
- Storing/Querying XML documents etc.

2.3 As a Dictionary

Looking up if a word is in a trie takes O(n) operations, where n is the length of the word. Thus - for array implementations - the lookup speed doesn't change with increasing trie size. It has been used to store large dictionaries of English words in spelling-checking programs and in natural-language "understanding" programs.

Simple spell checkers operate on individual words by comparing each of them against the contents of a

dictionary, possibly performing stemming on the word. If the word is not found it is considered to be an error, and an attempt may be made to suggest a word that was likely to have been intended.

Word completion is straightforward to implement using a trie: simply find the node corresponding to the first few letters, and then collapse the subtree into a list of possible endings. This can be used in auto completing user input in text editors.

2.3.1 Tries and Web Search Engines

The index of a search engine (collection of all searchable words) is stored into a compressed trie[8]. Each leaf of the trie is associated with a word and has a list of pages (URLs) containing that word, called occurrence list.

The trie is kept in internal memory. The occurrence lists are kept in external memory and are ranked by relevance. Boolean queries for sets of words (e.g. Java and coffe) correspond to set operations (e.g. intersection) on the occurrence lists.

3. The Evaluated Tree and Trie Implementations

The following are some of the Tree and Trie implementations which we have evaluated for this study.

3.1 rbtree

rbtree is Red-black tree implementation. It implements left-leaning 2-3 red-black trees as C preprocessor macros, and is used extensively in jemalloc. Most of the similar implementations require approximately four pointer-size fields per node (left child, right child, parent, color), whereas this requires only two (left child, right child + color). The only notable disadvantage is that insertion/removal are ~1.5X slower than the fastest implementations, due to extra overhead for maintaining the current path through the tree.

3.2 B googlebtree

It is a C++ B-tree implementation of Google code project. It is a template library that implements ordered in-memory containers based on a B-tree data structure. Similar to the STL map, set, multimap, and multiset templates, this library provides btree_map, btree_set, btree_multimap and btree_multiset. This C++ B-tree containers have a few advantages compared with the standard containers, which are typically implemented using Red-Black trees. Nodes in a Red-Black tree require three pointers per entry (plus 1 bit), whereas B-trees on average make use of fewer than one pointer per entry, leading to significant memory savings.

Google C++ B-tree containers make better use of the cache by performing multiple key-comparisons per node when searching the tree. Although B-tree algorithms are more complex, compared with the Red-Black tree algorithms, the improvement in cache behavior may account for a significant speedup in accessing large containers.

However the google C++ B-tree containers are not without drawbacks. Unlike the standard STL containers, modifying a C++ B-tree container invalidates all outstanding iterators on that container. For this reason, the library also contains "safe" variations on the four containers: iterators on safe B-tree containers keep a copy of the current key and automatically reposition the iterator whenever it is used.

3.3 C stx btree

The *stx btree* package is a set of C++ template classes implementing a B tree key/data container in main memory. The classes are designed as drop-in replacements of the STL containers set, map, multiset and multimap and follow their interfaces very closely. By packing multiple value pairs into each node of the tree the B tree reduces heap fragmentation and utilizes cache-line effects better than the standard red-black binary tree. The tree algorithms are based on the implementation in Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, Inroduction into Algorithms, Jan Jannink's paper and other algorithm resources. The classes contain extensive assertion and verification mechanisms to ensure the implementation's correctness by testing the tree invariants.

3.4 D tommy-trie

It is a trie data structure well optimized for cache utilization. It is a part of TommyDS in a C library.

TommyDS is a C library of hash tables and tries designed to store objects with high performance.

It claims that it is faster than all the similar libraries like rbtree, judy, googledensehash, googlebtree, stxbtree, khash, uthash, nedtrie, judyarray and others. Tommy is released with a 2-clause BSD license. In this paper, we compared only the hash table implementations of Tommy

library. TommyDS is said to be 100% portable in all the platforms and operating systems. The Tommy containers support multiple elements with the same key.

3.5 E tommy-trie-inplace

It is a trie implementation which is completely inplace. This trie is a inplace implementation and does not need any external allocation. Elements are not stored in order, like tommy_trie, because some elements should be used to represent the inner nodes in the trie. We can control the number of branches and more branches imply more speed, but a bigger memory occupation. Compared to tommy_trie you should use a lower number of branches to limit the unused memory occupation of the leaf nodes. This imply a lower speed, but without the need of an external allocator.

3.6 F nedtrie

It is a binary trie inplace implementation by Niall Douglas. The most interesting characteristic of bitwise tries are insertion, deletion and finds, all take about the same amount of time.

It works by constructing a binary tree based on individual bit difference, so higher the entropy (disorder) between key bits, the faster nedtries. Hence it has a complexity for a given key of O(1/DKL(key||average key)) Its worst case complexity - where all keys are almost identical - is O(log2 N). This implies that for well distributed keys that average complexity will always be a lot better than red-black trees, but somewhat worse than O(1) because for obvious reasons, more keys means more unique information and therefore there is always some scaling relation to the number of keys in the tree

There is a C macro based implementation as well as a C++ template-through-the-C-macros implementation are available.

4. The Analysis and Result

In this research we used our customized version of TommyDS[7] benchmark program. We have developed customized benchmark scripts which will run the benchmark tests with different hardware and software Configurations.

In this evaluation we store a set of N pointers to "character/string payload objects" and searching them using a key. This scenario is equivalent to that of a typical dictionary implementation. This scenario is different than the simpler case of mapping integers to integers, as pointers to objects are also de-referenced resulting in additional cache misses in such dictionary like implementations.

This evaluations made are much suitable for implementations like dictionary ADT, that stores information in the objects itself, as the additional cache misses are already implicit.

4.1 The Metric Used for Evaluation

In this evaluation, the important metric is time. Times are expressed in nanoseconds since it is measured for a single element operation/test. The following are the operations/tests made during the evaluation. Since the time is the metric, lower is always considered as better.

4.2 The Important Test Made are

(a) Insert: Insert one element in the container. In this evaluation, the Insert test starts with an empty container and ends after inserting N "character payload objects".

(b) Change: Change is searching, removing and reinsert it with a different key value. That is, find and remove one character payload objects and reinsert it with a different key. In this evaluation, the Change tests operate when N character payload objects are in the containers.

(c) Hit: Hit is searching an object with success. In this evaluation, the Hit tests are made when N character payload objects are in the containers.

(d) Miss: Miss is a scenario where a find operation ends with a failure. In this evaluation, the Hit tests are made when N character payload objects are not in the containers.

(e) Remove: In this evaluation we remove all the N objects and dereference them. So this evaluation ends with an empty container.

The character payload objects are always de-referenced, as we are assuming to use them during the operation. This happens even in the remove case, as we are supposing to deallocate them.

In all these tests, the time is measured in nanoseconds because, we are measuring it for one single operation – like the time needed to insert/delete one single object from the container.

As in the original benchmark[7], all the objects are pre allocated in the heap, and this allocation time is not included in the test. The objects are identified and stored using integer and unique keys. The key domain used is dense, and it's defined by the set of N even numbers starting from 0x80000000 to 0x80000000 + 2*N. The use of only even numbers allows to have missing keys inside the domain for the Change test. In such tests it is used the key domain defined by the set of N odd numbers starting from 0x80000000 + 1 to 0x80000000 + 2*N + 1. Note that using additional keys at the corners of the domain would have pushed tries and trees as they implicitly keep track of the maximum and minimum key values inserted. The use of the 0x80000000 base, allows to test a key domain not necessarily starting at 0. Using a 0 base would have pushed some tries managing it as a special case.

As in the original benchmark[7], in our evaluations, tests are repeated using keys in Random mode and in Forward mode. In the forward mode the key values are used in order from the lowest to the highest. In the random mode the key values are used in a completely random order. In the forward mode of Change test, each object is reinserted using the previous key incremented by 1. In random mode each object is re-inserted using a completely different and uncorrelated key. The forward order pushes tries and trees as they use the key directly and they have a cache advantage on using consecutive keys. The random order pushes hash tables, as the hash function already randomizes the key. Usually real use cases are in between and the random one is the worst one.

4.3 The Parameters of the Evaluation

By default, in this benchmark framework code, the objects contain an integer value field used for consistency checks, and 16 bytes of characters to simulate payload field, and any other data required by the data structure.

For this evaluation we used dictionary word with string payloads of two sizes 16 and 4096. A 16 byte payload to simulate lower stress and a 4096 bytes payload to simulate high stress on the memory.

The size of the dictionary is started with minimum number of elements and increased to a maximum size in a predefined step multiplication factor. And the average performance of the data structures during each size of the dictionary is logged on separate files (one file for each test) for further analysis.

We did the experiments with dictionary size of 100 to 8,00,000 elements (8lakhs or 0.8 Million).

4.4 The Result

4.4.1 Results with Random Mode Operations/ Tests with String Payload Size of 16 and 4096 Characters

The following line charts shows the performance of random mode Insert, Change and Hit operation/tests made with different payload sizes 16 Characters/Element (3 Left column Images) and 4096 Characters/Element (3 Right column Images)

If we compare the above three Left column line charts with the three Right column line charts we can see that two Tommy trie implementations perform good at smaller string object size of 16 as well as larger string object size of 4096. But with the payload size of 4096, the performance of google tree is good in most of the tests and comparatively better than tommy_trie_implace. Rbtree and Nedtrie are performed very poor both in the case of smaller string object size of 16 as well as larger string object size of 4096.

The following line charts shows the performance of Random Mode Remove, and Miss operation/tests made with different payload sizes 16 Characters/Element (2 Left column Images) and 4096 Characters/Element (2 Right column Images)

If we compare the two Left column line charts with the two Right column line charts of the two Random Mode operations Remove and Miss, we can see that both the two Tommy trie implementations perform good at smaller string object size of 16 as well as larger string object size of 4096. googlebtree implementations perform good at smaller string object size of 16. Rbtree and Nedtrie are performed very poor both in the case of smaller string object size of 16 as well as larger string object size of 4096.

4.4.2 Results With Forward Mode Operations/ Tests with String Payload Size of 16 and 4096 Characters

The following line charts shows the performance of Forward Mode Insert, Change and Hit operation/tests made with different payload sizes 16 Characters/Element (3 Left column Images) and 4096 Characters/Element (3 Right column Images)

If we compare the above three Left column line charts with the three Right column line charts we can see that two Tommy trie implementations perform good

at smaller string object size of 16 as well as larger string object size of 4096. But with the payload size of 4096, the performance of google tree also good in most of the tests and comparatively better than tommy_trie_implace. Rbtree and Nedtrie are performed very poor both in the case of smaller string object size of 16 as well as larger string object size of 4096.

The following line charts shows the performance of Random Mode Remove, and Miss operation/tests made with different payload sizes 16 Characters/Element (2 Left column Images) and 4096 Characters/Element (2 Right column Images)

If we compare the above two Left column line charts with the two Right column line charts of the two Random Mode operations Remove and Miss, we can see that both the two Tommy trie implementations perform good at smaller string object size of 16 as well as larger string object size of 4096. google tree implementations perform good at smaller string object size of 16. Rbtree and Nedtrie are performed very poor both in the case of smaller string object size of 16 as well as larger string object size of 4096.

4.4.3 Comparison of Random Mode and Forward Mode Operations/Tests with High String Payload Size 4096 Characters

The following bar charts shows the performance of random mode (3 Left column Images) and forward mode (3 Right column Images) Insert, Change and Hit operation/tests made with High payload size of 4096 Characters/Element.

If we compare the above three Left column bar charts with the three Right column bar charts of Random mode and Forward Mode operations Insert, Change and Hit, It is obvious that the performance of tommy trie with high payload size is significantly better than all other compared methods. In some tests the performance of googlebtree is comparatively better than tommy_trie_implace, Rbtree and Nedtrie. The two methods Rbtree and Nedtrie are performed very poor both in random as well as forward mode operations with string object size of 4096.

The following bar charts shows the performance of Random Mode (2 Left column Images) and Forward Mode (2 Right column Images) Remove and Miss operation/tests made of payload size of 4096 Characters/Element

If we compare the above two Left column bar charts (Random Mode) with the two Right column bar charts(Forward Mode) of operations Remove and Miss, It is obvious that the performance of tommy trie with high payload size is significantly better than all other compared methods. In some tests the performance of googlebtree comparatively better than tommy_trie_inplace, Rbtree and Nedtrie. The two methods Rbtree and Nedtrie are performed very poor both in random as well as forward mode operations with string object size of 4096.

4.5 Observation and Finding

- For a dictionary with small word size of 16 bytes (string payload size) as well as 4096 bytes, and around a million words, the two Tommy trie implementations competed the other four implementations googlebtree, stxbtree, Rbtree and Nedtrie.
- At the dictionary size of 8,00,000 words of 4096 byte words, without any doubt, tommy_trie is the competing method since it gave best performance in most of the tests. If we compare the right and left line chars of Figure 1, 2, 3 and 4, we can realize it.
- In some tests with 4096 bytes, even the performance of googlebtree seems to be better than tommy_trie_inplace implementations.
- The two methods Rbtree and Nedtrie are performed very poor both in random as well as forward mode operations with string object size of 4096 as well as 16. If see the line chars of Figure 1, 2, 3 and 4 and the bar chars of Figure 5 and 6 we can obviously see it.
- Undoubtedly, Tommy trie implementations are much suitable for ordered dictionary implementation with small word size as well as larger word size.

5. Conclusion

Hash table implementations of TommyDS[7] claims that it is designed for high performance and faster than all the similar libraries like rbtree, judy, goodledensehash,. In our previous evaluation with different hash tables, we found that the Tommy DS implementation of hash table performed poor in dictionary implementation with big word size.

But, without any doubt, the performance of the trie implementations of Tommy DS was significantly better than trie and tree implementations.

In this work, we have evaluated some of the tree and trie implementations. In future we may address the

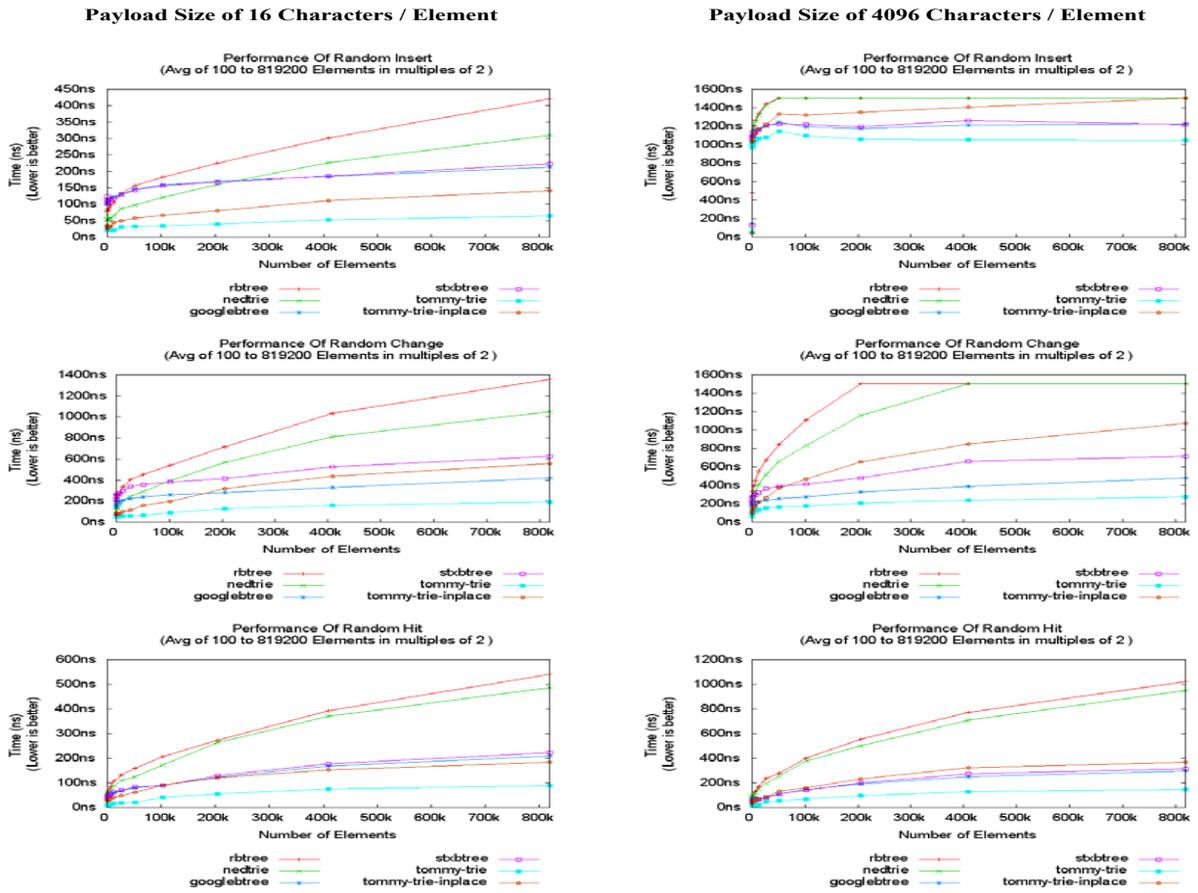

Figure 1. Comparison of Random Mode Insert, Change and Hit Operations/Tests.

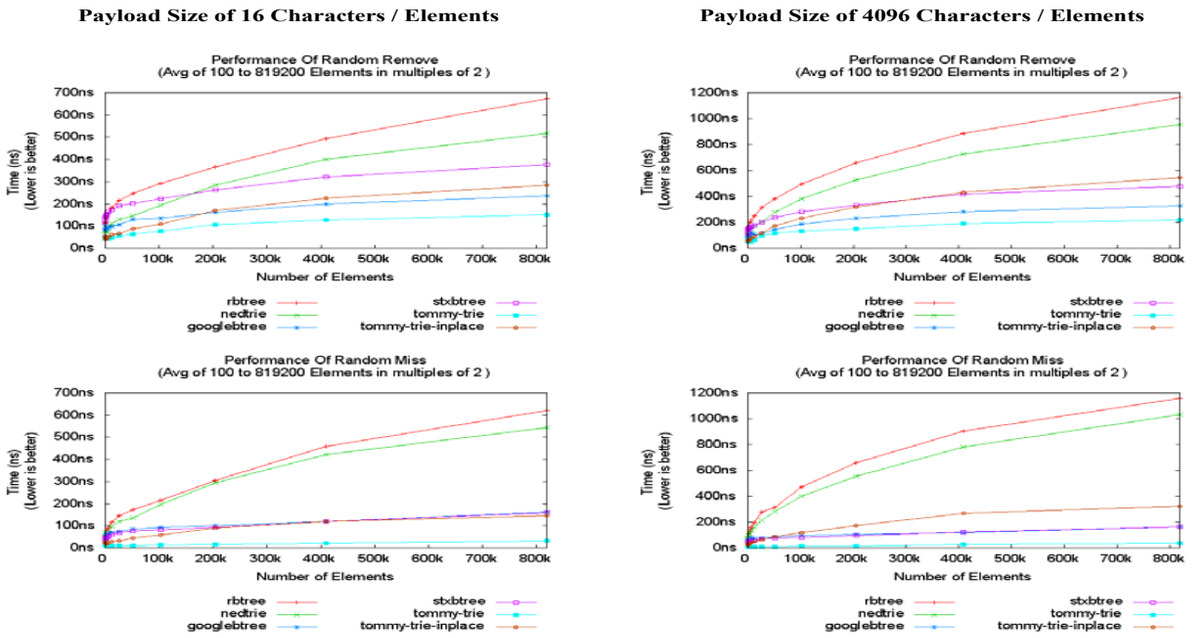

Figure 2. Comparison of Random Mode Remove, and Miss Operations/Tests.

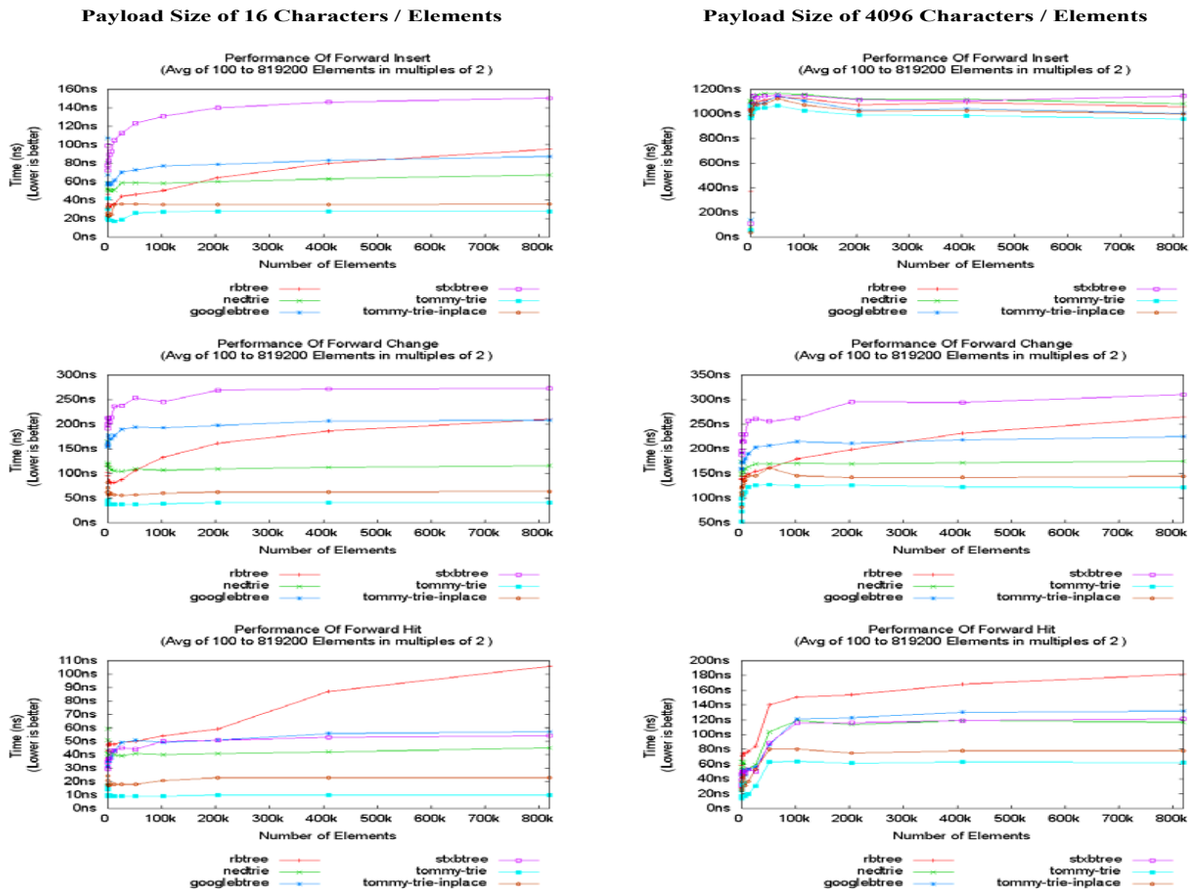

Figure 3. Comparison of Forward Mode Insert, Change and Hit Operations/Tests.

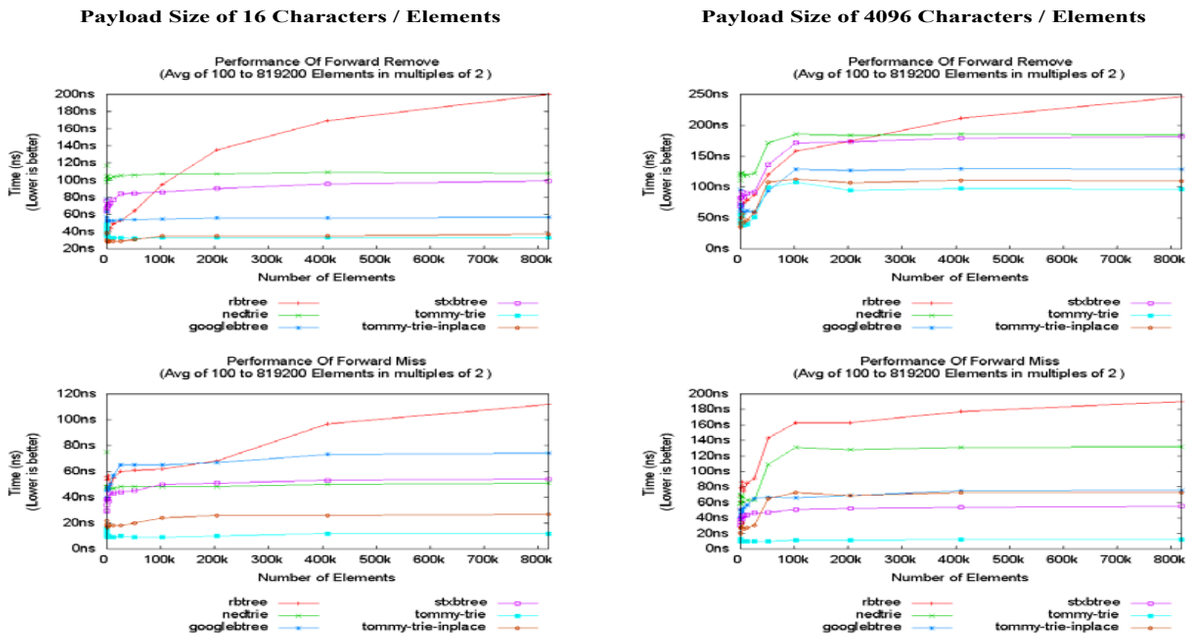

Figure 4. Comparison of Forward Mode Remove, and Miss Operations/Tests.

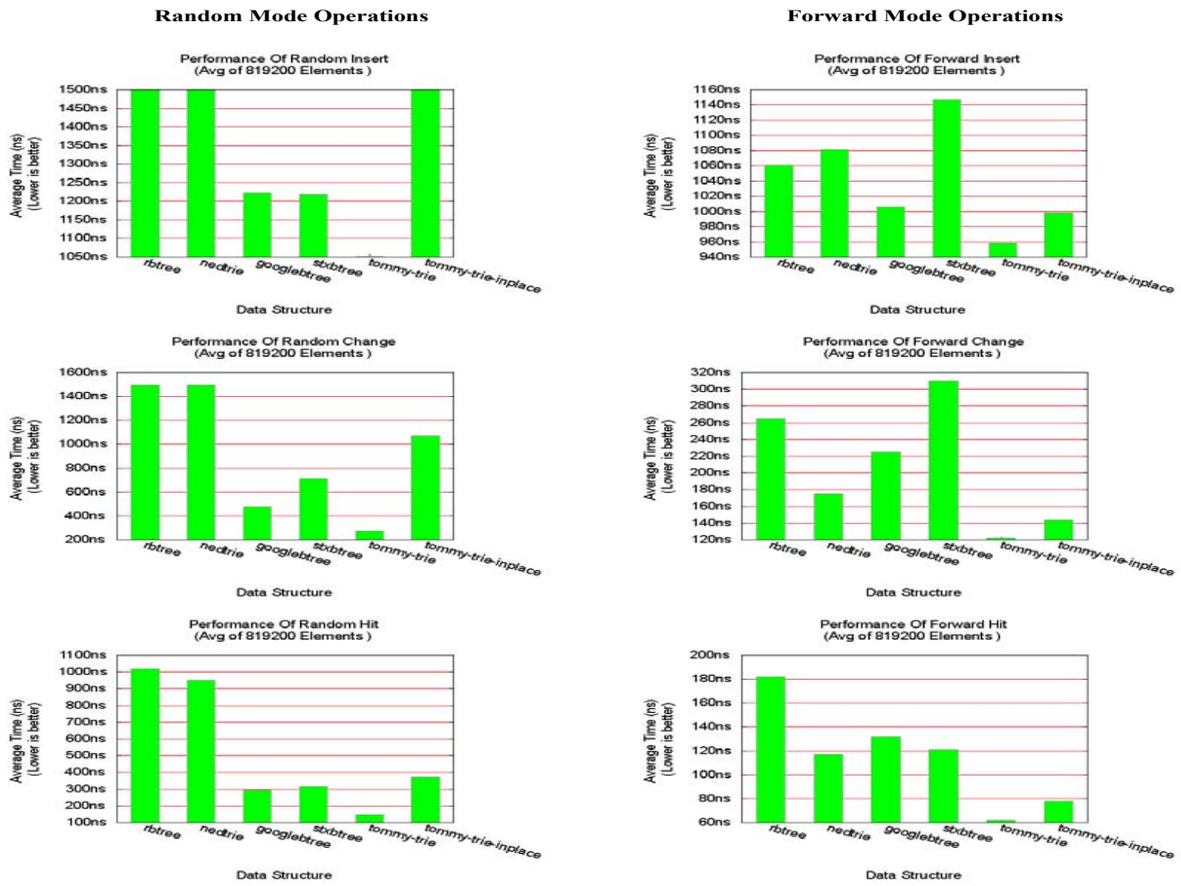

Figure 5. Comparison of Random Mode and Forward Mode Insert, Change and Hit Operations/Tests.

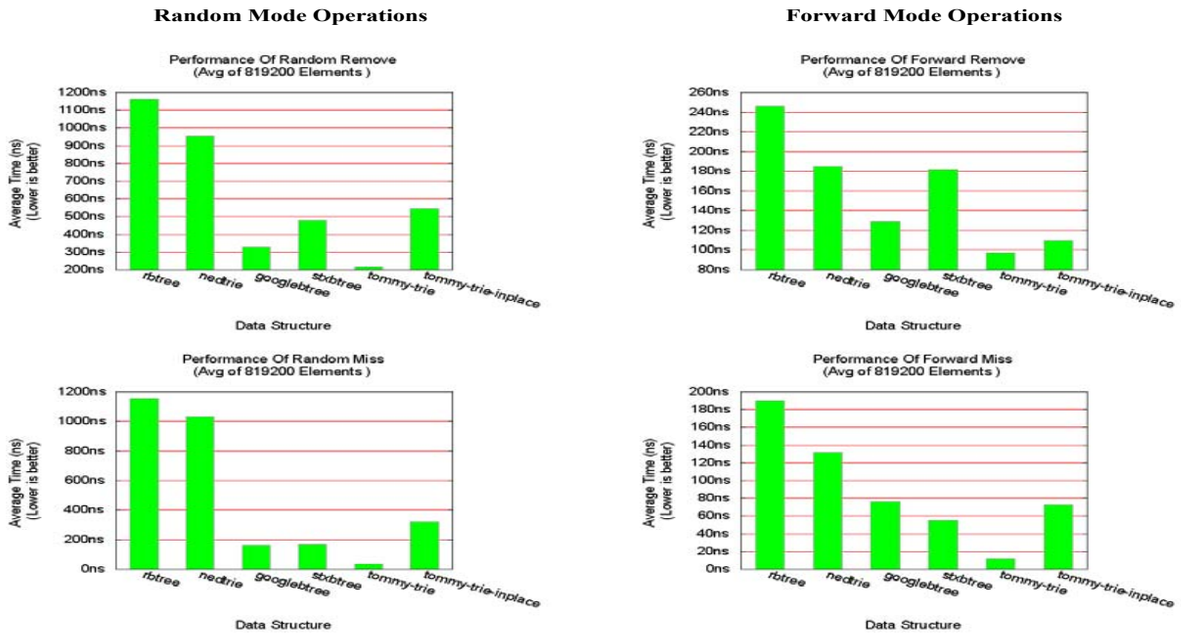

Figure 6. Comparison of Random Mode and Forward Mode Remove, and Miss Operations/Tests.

performance issues of these abstract data types on virtual and cloud computing architecture.

An existing hash table data structure are computationally efficient but uses large number of pointers to manage string payload objects and other type of huge data objects. Use of pointers in such dynamic data structure with huge data payloads increases cache inefficiency as they lead to RAM very often. Redesigning classical String based data structures for cache friendly operation may be needed for efficient performance. Future works may address the ways to improve the existing hash table implementations for cache friendly operations. Future works may address the ways to improve a selected data structure for achieving improved performance.

6. References

1. Askitis N, Zobel J. Redesigning the String Hash table, Burst Trie, and BST to exploit cahe. JEA. 2011 Jan; 15.
2. Williams HE, Zobel J, Heinz S. Splay trees in practice for large text collections. Software-Practice and Experience.
3. Martinez C, Roura S. Randomized binary search trees. Jour of the ACM. 1998 Mar; 45(2):288–323.
4. Sleator DD, Tarjan R. Self-adjusting binary search trees. J ACM. 1985; 32:652–86.
5. Morimoto AK, Sato T. An Efficient implementation of trie structures. Software Pract Ex. 1992 Sep; 22(9):695–721.
6. McCreight EM. A space-economical suffix tree construction algorithm. J ACM. 1976; 23(2):262–71.
7. Mazzoleni A. Tommy Benchmark, and Tommy DS 1.8, A high performance C library of Hash tables and Tries. Availaible from: https://github.com/amadvance/tommyds
8. Purdin TDM. Compressing tries for storing dictionaries. In: Berghel H, Talburt J, Roach D, editors. Proceedings IEEE Symposium on Applied Computing; Fayettville: IEEE: 1990 Apr. p. 336–40.
9. Jayalakshmi R, Baranidharan B, Santhi B. Attribute based Spanning Tree Construction for Data Aggregation in Heterogeneous, Wireless Sensor Networks. Indian Journal of Science and Technology. 2014 Apr; 7(S4):76–9.
10. Alzahrani AS, Qureshi MS. Privacy preserving optimized rules mining from decision tables and decision trees. Indian Journal of Science and Technology. 2012 Jun; 5(6):2831–4.