IMPROVING TIME AND SPACE EFFICIENCY

OF TRIE DATA STRUCTURE

by

NIRMIK KALE

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2018

ACKNOWLEDGEMENTS

I would like to thank Dr. Song Jiang for providing me with an opportunity to work on this project and guiding me throughout the duration. Without his support and guidance this research would not have been possible. I would also like to specially thank him for encouraging me about my other goals and always being supportive. I would also like to thank Dr. David Levine for investing his time and efforts in me. He has been a constant support providing encouragement and guidance throughout my degree. I would also like to thank Dr. Dajiang Zhu for taking time to serve on this committee.

I would like to thank my family, especially my mother, father, aunt, uncle and my cousin who have been a constant support throughout my life. A special gratitude and thank you to all my friends including, but not limited Tanvi Tiwarekar, Kshitij Khakurdikar, Guruprasad Bhavsar, Rishikesh Jadhav, Satwik Kolhe, Abhishek Dhotre, Kedar Nadkarny, Mayur Munot, Ritesh Deshmukh, Tushar Garud and Arpit Agarwal for always being supportive and guiding me through every situation. I would also like to extend my gratitude to Dr. Xingbo Wu and all my other lab mates for their help and being ever so kind and welcoming. Dr. Xingbo has inspired and guided me through many ideas in this thesis. Some ideas in this thesis draw from his work on wormhole [1]. This thesis would not have been possible without either of their support.

August 26, 2018

ABSTRACT

IMPROVING TIME AND SPACE EFFICIENCY

OF TRIE DATA STRUCTURES

Nirmik Kale, MS

The University of Texas at Arlington, 2018

Supervising Professor: Song Jiang

Trie or prefix tree [2] is a data structure that has been used widely in some applications such as prefix-matching, auto-complete suggestions, and IP routing tables for a long time. What makes tries even more interesting is that its time complexity is dependent on the length of the keys inserted or searched in the trie, instead of on the total number of keys in the data structure. Tries are also strong contenders to consider against hash tables in various applications due to two reasons - their almost deterministic time complexity based on average key length, especially when using large number of short length keys, and support for range queries. IP routing table is one such example that chooses tries over hash tables. But even with all these benefits, tries have

largely remained unused in a lot of potential candidate applications , for example in database indexing, due to their space consumption. The amount of pointers used in a trie causes its space consumption to be a lot more than many other data structures such as B+ Trees. Another issue we realized with tries is that even though the time complexity can be of a magnitude far less than some other data structures for short length keys, it can be considerably higher if the keys are of longer lengths. Insertion in a trie can prove to be a repetitive operation for many nodes if the keys are repetitive or have many common prefixes adding to the execution overhead. With this in mind, we propose two optimizations of the trie data structure to address the time and space complexity issues. In the first optimization we present a system that reduces the time for inserts in the trie data structure by up-to 50% for some workloads by tweaking the algorithm. In the second optimization we developed a new version of the trie data structure by taking inspiration from B+ trees, allowing us to not only reduce the space consumption for tries but also to allow features such as efficient range search.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

Chapter 1

INTRODUCTION

With increase in amount of data generated, large sets of small key-value data sets are a common type of data to be stored these days. Even for conventional data stored using either databases or any other data storing methods, key-value stores prove as the meta data structure for faster lookup times. Such key-value data sets are usually implemented as some kind of in memory system to accelerate fast searches. With recent efforts, performance has been approaching the hardware limits for existing data structures being used in such systems. With this is mind, one of the issues faced by such in-memory key-value data sets is the space consumption due to the data structures used for storing the key-value store. As the data increases from thousands to billions, the amount of metadata generated is also increased. In such cases, trying to improve the space efficiency of the data structures used can prove to be a huge benefit for such systems. Although, one needs to keep in mind that such improvements should come with little to no feature reductions in existing systems.

Indexing data structures such as B+ trees and skip lists [3] are used in many major database applications such as MySQL and LMDB for the former and LevelDB [4] for the later. Such ordered indexing data structures are required for supporting features

in databases such as range queries, even when they are hugely slower than indexing structures such as hash tables for point queries.

Another type of system is one where there are huge number of insertions followed by immediate lookups. In such scenarios, if the insertion times are long, the results obtained after immediate lookups can be stale and require a major amount of time for the updated data to be presented. Financial applications are one of the best examples of such requirements.

1.1 Focus and contribution of this thesis

This thesis focuses on improving the efficiency of the data structures used in the applications mentioned above. Improving efficiency is a two-factor process for any data structure – time and space. These two factors usually do not go hand in hand and trying to improve one implicates a sacrifice on the other. But there is, in most situations, a mid-point that can achieve an acceptable improvement as well as sacrifice on both. We try to achieve this mid-point and suggest two approaches to solve two separate problems in key-value data stores. Since a new data structure might not always be the one general solution to all problems and thus replace the existing data structure completely, we work on both, the current data structure and also develop a new data structure.

As an outcome of this thesis, we can achieve a better time efficiency for insertions in the trie and secondly, with the help of the new data structure, we can store the keys in a more space efficient way while still providing existing features, sometimes, with even better efficiency.

The main contributions can be stated as –

1. A new algorithm for faster insertion times for transaction heavy workloads
2. A new data structure to enable more space-efficient ordered indexing

1.2 Thesis organization

The rest of the thesis is organized as follows –

Chapter 2 : Data structures and potential candidates

Chapter 3 : Deeper look into the trie data structure

Chapter 4 : Optimizing time complexity of Trie

Chapter 5 : Optimizing space efficiency of Trie

Chapter 2

POTENTIAL DATA STRUCTURE CANDIDATES

Trying to solve the problems mentioned in Chapter 1 can take many forms of solutions. This study presents two approaches on a specific data structure – the Trie or Prefix Tree. But before we considered working on a trie, we explored our options. Hash table emerges as one of the top candidates to use either by itself or in combination with another data structure. The O(1) lookup time of a well-defined hash table is what makes it one of the most lucrative options to use.

2.1 Hash Tables

Hash tables are one of the most widely used data structures in many different applications in computer science. Hash tables are also the ideal and inspiration for working on better data structures due to its most important benefit, constant lookup time. This is off-course considering that the hash function used in the hash table is a good hash function that does not create a lot of collisions. Section below discusses the hash table data structure with simple example and also its limitations.

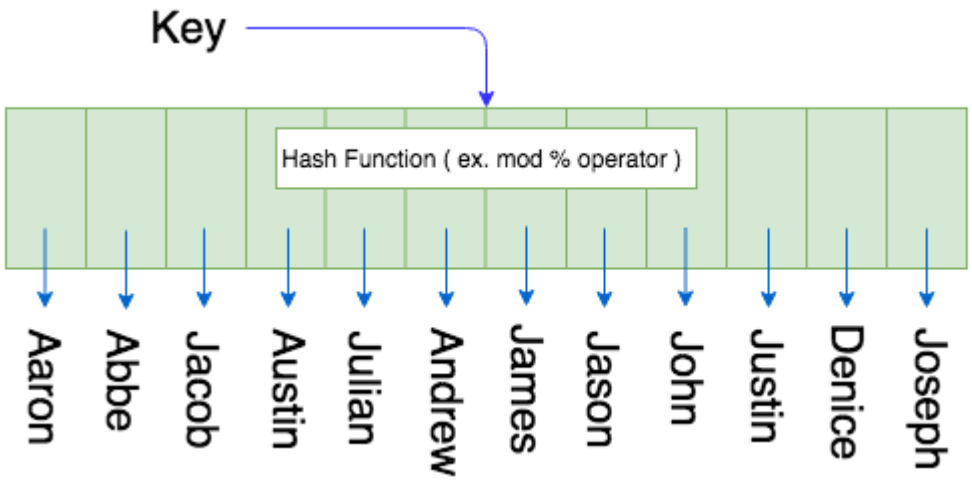

Fig. 1 Hash Table

Fig. 1 represents a simple hash table that stores strings using a hash function. In the case shown in figure, there are no collisions and every new entry in the hash table is mapped to a new hash location. Such a hash function would be known as a perfect hash function.

One of the simplest example of a hash function is the mod operator (%). The mod operator will hash any value submitted to it as an input to a bucket representing the remainder of the mod operation.

For example, if the hash function is defined as –

Hash = Input value % 2

Then the values will be mapped to either bucket 0 or bucket 1

Similarly, if the hash function is defined as –

Hash = Input value % 7

the values will be hashed to one of the buckets from 0, 1, 2, 3, 4, 5, 6

Hashing and hash tables are a very common mechanism used in many computer science applications including indexing. There can be more complex versions of hash tables such as multi-level hash tables that make use of 2 or more hash functions for each level allowing for a larger data holding capacity without as many collisions as a single hash table would occur.

One of the most important drawbacks of a hash table, even a perfect hash function is that a hash table cannot support range queries.

Range queries are queries such as –

- Find all entries starting from H
- Find all entries between H and P
- Find 20 entries starting from H
- Find 20 entries before P

- Etc

The reason for hash tables not supporting range queries is pretty obvious. Since the inputs are not stored in the order of arrival, or the absence of any meta-data recording this information, it is impossible to know if the entry placed in the bucket next to the bucket for H is also the entry that was entered in order after H, or should, according to some sorting mechanism, be the next valid entry.

Another major drawback of hash tables is the size of a hash table when the input data increases. If the size of the hash table, when created is not large enough to accommodate large amounts of data, the hash table can run out of space and result in increasing collisions. The worst-case time complexity for a hash table with too many collisions would be O (n). Multi – level hash tables, as stated before can be one solution, but neither are they an elegant solution, nor are they proof to filling up eventually.

2.2 Tries

Trie, also known as Prefix tree is a leading candidate for replacing hash tables. Although hash tables are almost impossible or very difficult to match in performance, there are data structures available that have complexities closer to hash tables than to

other famous data structures on the time complexity scale. And trie is one of the leading ones.

The time complexity for tries is dependent on the length of the keys or data being inserted or searched instead of the total number of keys or data in the data structure. Thus, even though not constant, tries have an almost deterministic time complexity if the average length of the keys is known.

For any given trie –

Time complexity is = O (k)

Where ***k*** is the length of the key.

This makes tries a perfect candidate for keys of short lengths.

For example, if a trie is to store 1 billion keys, of an average length 15,

the time complexity for a search, insert or delete, on an average will be O (15) for the worst case (where the key does not exist for search and delete). Compare this to data structures such as B+ trees and the complexity would be somewhere around O (30), no matter the length of the keys. Once again, obviously a trie cannot beat the absolute deterministic time complexity of a hash table. Another consideration, in this comparison is that the number of keys is larger than the average length of the keys.

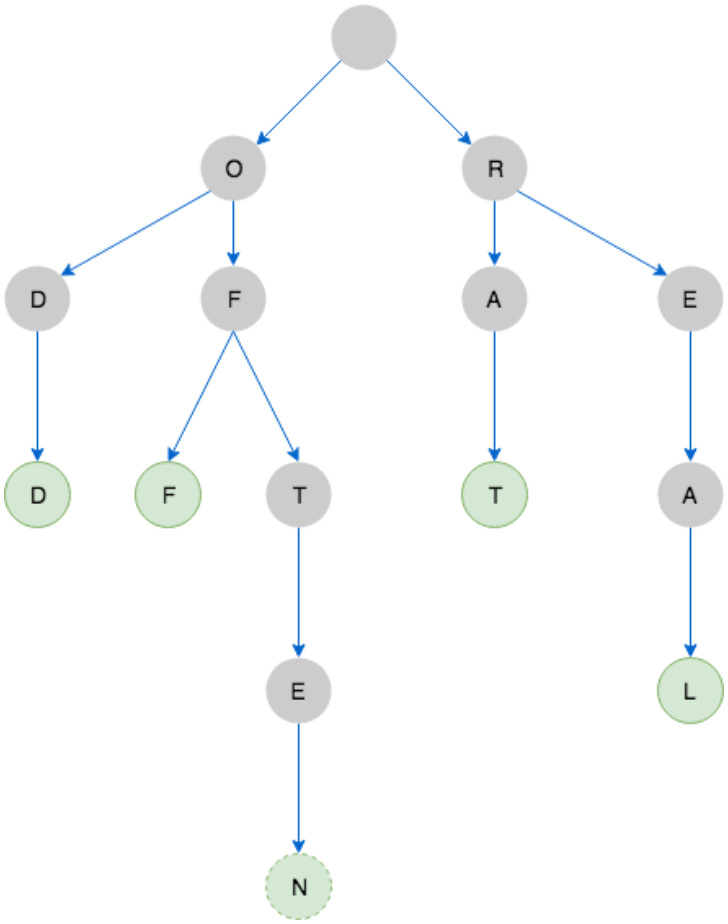

Fig. 2 Trie / Prefix Tree

Fig. 2 shows an example trie data structure.

Each key in the trie is tokenized and each letter in the key (for above example) becomes a token. Thus, the position of any key (and its subsequent value) is solely determined by the key individually itself unlike a B+ tree where the position of key depends on the value of the key and other keys already present in the tree.

2.3 B+ Trees

As mentioned before, B+ trees [5][6] are currently used in quite a few leading database Systems. MySQL with its default engine InnoDB, LMDB all use B+ trees as their data structure of choice for indexing. B+ trees are balanced trees whose height and span depends on the order of the B+ tree and number of elements in the tree. For example, a B+ tree with order ***m*** can hold between m/2 and m keys in each leaf node, thus limiting the height and span of the tree and keeping it balanced.

For any B+ tree, the internal tree is just a means of reaching the leaf nodes. The leaf nodes are the ones holding the actual keys (and their subsequent values).

For a B+ Tree with order - ***m***

$$\boldsymbol{m/2} \leq \text{elements in leaf node} < \boldsymbol{m}$$

The time complexities of a B+ tree not only depend on the number of elements, but also on the order of the tree.

For a B+ tree with order ***b*** and ***n*** elements

Time complexity = $O(\log_{b} n)$

Space complexity = $O(n)$

One of the most important features, as discussed in short before , that b+ trees enable databases to have is range search. The reason b+ trees can perform range searches is because of a couple of reasons. Firstly, that all the keys are stored at the same level in the tree – the leaves. Secondly, all the keys in the leaves are always in a sorted order, inter as well as intra leaves. Meaning that if leaf one contains values 1, 2, 3 then leaf two (its immediate right sibling) will contains values greater than 3, for example may be 4, 5, 6. Thirdly, and most importantly, the part that connects the above two points and makes them useful, is that if all the leaf nodes are connected to each other with forward and backward pointers, it allows us to traverse through the leaves without re-traversing the tree again. Fig. 3 shows an example of a B+ tree of order 3

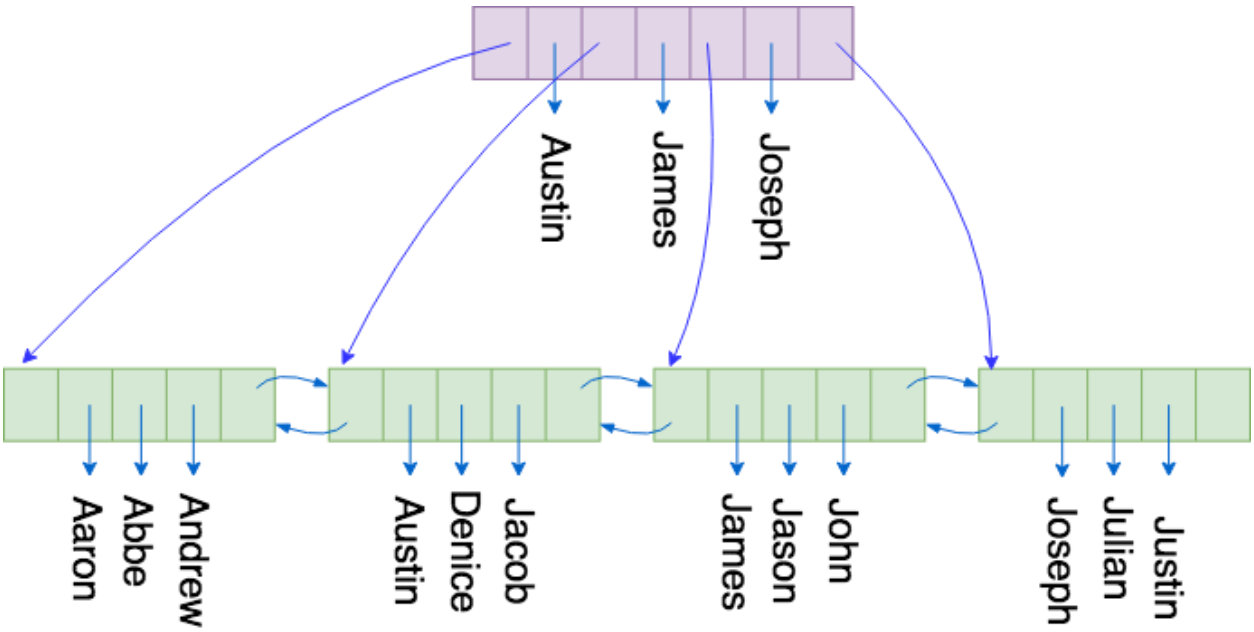

Fig. 3 B+ Tree

As potential candidates, not only were data structures considered individually, but also in combination with each other to improve performance collectively. As will be seen later in the second approach presented, we use a combination of two data structures to enable some features and save space.

As the primary data structure to work on, we selected Trie as the candidate. Some of the potential improvements included using hash tables as an internal data structure in the nodes for finding next paths quickly, etc.

Chapter 3

DEEPER LOOK INTO TRIE DATA STRUCTURE

In the previous chapter, we discussed in brief what the trie data structure looks like. In this section we will iterate over some of those things while taking a detailed look at it and also look at some potential issues.

As depicted in fig. 2 the trie data structure breaks down a key into tokens and stores each token hierarchically one below the other. The trie data structure is most useful in applications that need prefix matching. The Longest Prefix Match (LPM) operation is very common on tries. There can be many versions of a trie, such as a simple integer trie, alphabet trie or a full ascii based trie. We have performed testing on all the 3 above mentioned versions. An integer based trie can basically store only decimal numbers from 0 – 9, meaning that each node in this trie can have 0 – 9 children and similarly the alphabet based trie stores only letters from A – Z or a – z with every node having between 0 – 26 children. The ascii based trie is a more comprehensive version that can store numbers, small and capital letters as well as all ascii based symbols.

3.1 Internal structure

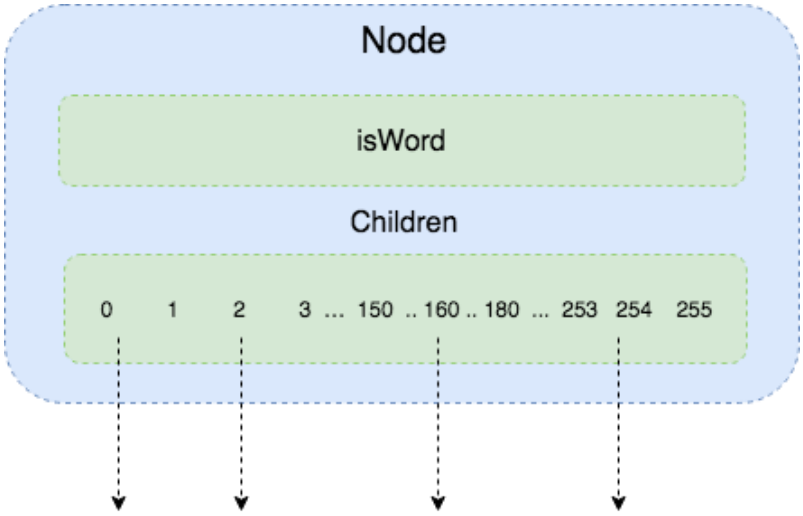

Fig. 4 Internal Trie Node

Fig 4. Shows the internal structure of a node from an ascii trie.

The first parameter is used to detect if a word / key is formed at any point in the trie and helps avoid false matches. This can also be seen in Fig 2 with the last nodes of a key being marked in a different color. Sometimes there might be sub strings of already inserted keys also present as another key, therefore it is not always necessary that a key can be formed only at the leaf node. The second parameter - children, is a list / an array of pointers to child nodes. As mentioned before, for an ascii trie, this number is 256 children (0 – 255). The data or character being represented by any node does not have to be stored in the node since the position of any node determines the value of that

node. For example, a node that is pointed to by a pointer from 97 in the children array in above node, automatically converts to the character ***a*** in ascii, thus defining the value of the node as ***a***. For optimizing the time and space requirements of trie, we have added some more parameters to the node which we will see in later chapters.

3.2 Complexity

3.2.1 Time Complexity

The time complexity calculation for a trie dependent on the key being searched or inserted. In the worst case where the key is not found, the time required asymptotically is to the order of the length of the key.

For a key with length L,

Trie time complexity = O (L)

This is true for insert, search as well as delete operations.

3.2.2 Space Complexity

The worst-case space complexity for a trie would be the total number of nodes stored in the trie. It is pretty rare to have a trie in its worst case for space as it would need no more than 256 unique keys in a trie that start with a unique ascii character.

If the number of nodes is considered to be M,

Trie space complexity = O (M)

3.3 Operations on trie

This section discusses the basic operations on trie, viz, Insert, search and delete.

3.3.1 Insert

```
function SET(root, len, key)
        curr ← root
        i ←  0
        insertStatus ← false
        while ( i < len ) do
                if key[ i ] in curr.children then
                        curr ← curr.children[ key [ i ] ]
                else
                        new ← node()
                        curr.children[ key [ i ] ] ← new
                        curr ← new
                        if i = len – 1 then
                                new.isWord ← true
                                insertStatus ← true
                i ← i + 1
        return insertStatus
```

3.3.2 Search

```
function GET(root, len, key)
        curr ← root
        i ←  0
        found ←  false
        while ( i < len ) do
                if key[ i ] in curr.children then
                        curr ← curr.children[ key [ i ] ]
                        if ( i  = len – 1) and (curr.isWord = true) then
                                found ← true
                i ← i + 1
        return found
```

3.3.3 Delete

```
function DEL(root, len, key)
        curr ← root
        delParent ← root
        delChild ← 0
        i ← 0
        deleted ← false
        while ( i < len ) do
                if key[ i ] in curr.children then
                        if ( delInternal ( curr, key [ i ] ) = true ) then
                                delParent ← curr
                                delChild ← key[ i ]
                        curr ← curr.children[ key [ i ] ]
                        if ( i = len – 1) and (curr.isWord = true) and
                          (delInternal ( curr, key [ i ] ) = false) then
                                deleted ← true
                i ← i + 1
        if ( deleted = true ) then
                free ( delParent.children[ delChild ] )
                delParent.children[ delChild ] ← NULL
        return found
```

3.3.4 Ancillary function : delInternal

```
function delInternal(node, index)
        i ← 0
        while ( i < 256 ) do
                if ( i ≠ index ) and ( node.children[ i ] ≠ NULL ) then
                        return true
                return false
```

The delete operation on a trie is usually more complicated, since they key to be deleted might share a common prefix with another key in the trie. Due to this, the trie needs to be traversed once to find out the last node of the LPM and later traversed again from the remembered node again to delete all nodes.

3.4 The problem

3.4.1 Space Problem

Asymptotically the space consumption of a trie does not seem too bad. With most tries with large amounts of data and sharing a huge amount of nodes (in a real-world scenario) the number M is not very large as compared to N (number of keys). The space complexity can also be less than some other data structures that can take O (N).

Even though asymptotic notation resembles that the space complexity is not too bad, the reality is different. And the main culprit for this is the pointers. For an ascii based trie or a byte trie, there are 256 child pointers in each node. And even though most pointers might not actually point to anything, NULL pointers also account for space. A trie with 1 billion records with each node having 256 pointers could fail quickly as an in-memory data structure.

There have been attempts at overcoming this by using hashing algorithms to store the pointers, but the hash functions need to be good enough and add to total execution time. Having an array of 256 possible positions is the simplest form of hash table that can be implemented. This hash table has no collisions as each ascii character belongs to a unique bucket in the hash table.

3.4.2 Time problem

Since the time complexity of tries, as seen before, depends on the length of the key, for scenarios such as when the length of the keys is short while the number of keys

is fairly large (L <<< N) a trie can perform better than many other data structures such as B+ trees which will depend on the total number of keys.

As seen in examples before, like in fig. 2 , tries can have common prefix paths. The word OFF and OFTEN in the example in fig. 2 have the Longest Common Prefix (LPM) ***'OF'***. If this trie is a part of a system that characterizes in heavy insertions and deletions, such as financial transactions, a unique problem can be observed. Deleting nodes from the trie immediately, and then re-inserting either the same key or a key that had a common prefix that was a part of the deleted nodes, will incur a cost of allocating new nodes and inserting them in the trie. Although this is a normal functioning of the tire, for certain workloads, this can be improved with a few tweaks to the algorithm of *SET* and *DEL* operations.

Fig. 5 below shows an example of a deletion in a trie. If the key OFTEN is deleted from the trie, the nodes T, E and N are deleted.

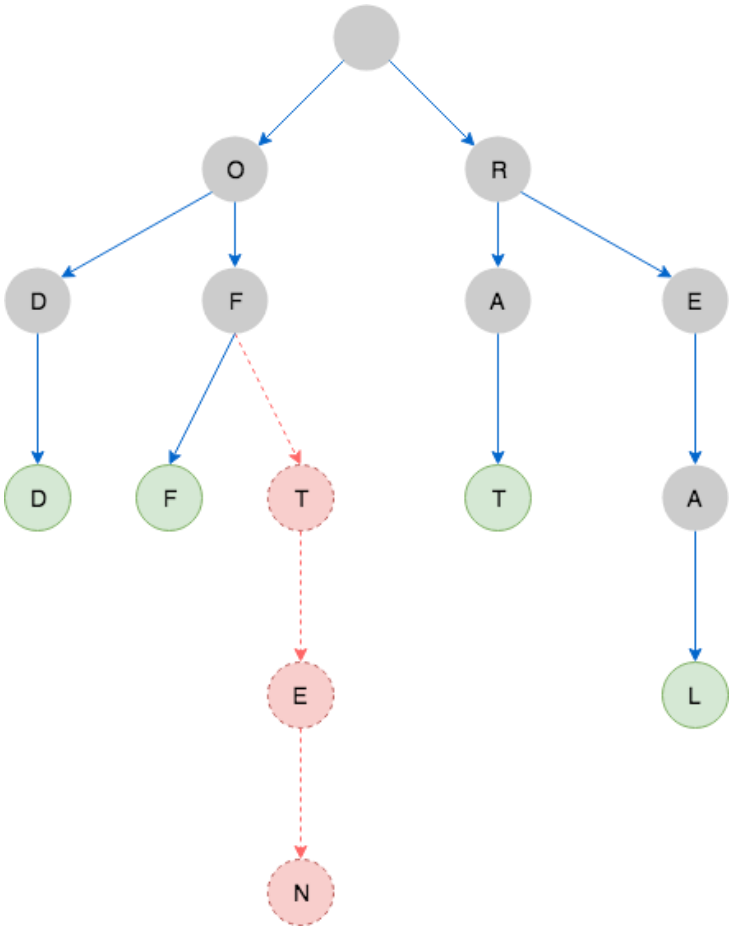

Fig. 5 Deletion in Trie

Below is an example of series of operations to reveal the issue

DEL OFTEN

GET RAT

GET RUTH

SET OFTEN

For the above series of operations, following will be what happens in the trie –

DEL OFTEN

- Find nodes that can be deleted for the key OFTEN
- Delete nodes T, E, N

GET RAT

- Search for key RAT
- Return True

GET RUTH

- Search for key RUTH
- Return False

SET OFTEN

- Look for existence of node O > found > move to next
- Look for existence of node F under O > found > move to next
- Look for existence of node T under F > not found
    - Create a new node > assign it as T child of F
    - Create a new node > assign it as E child of T
    - Create a new node > assign it as N child of E

Chapter 4 discusses the issue uncovered by this series of operations and the approach used to solve the same.

Chapter 4

OPTIMIZING TIME COMPLEXITY FOR INSERTS

The efficiency of insert (SET) operation is important to a very specific workload. A scenario in which the trie has to handle heavy amount of insertions and deletions, some operations become repetitive. This issue has been discussed with example in chapter 3 subsection 3.2. Financial databases or real time systems are potential candidates of such a system.

In order to improve the performance of the insert operations, we implement two strategies.

1. Lazy Deletion with garbage collector
2. Slab Allocation

Benchmark results for these two strategies are shown in later chapters.

4.1 Lazy Deletion with garbage collector

Lazy deletion is a well-known strategy used by many applications in computer science. Something as simple as a cloud system relies on this strategy to delete

instances a few days after the customer actually deletes it , to enable recovery in accidental situations or something as fundamental as hash tables with open addressing may delete elements with lazy deletion to allow relocation during next search [7].

We implement lazy deletion in the trie for our nodes with two configurable parameters and multithreading

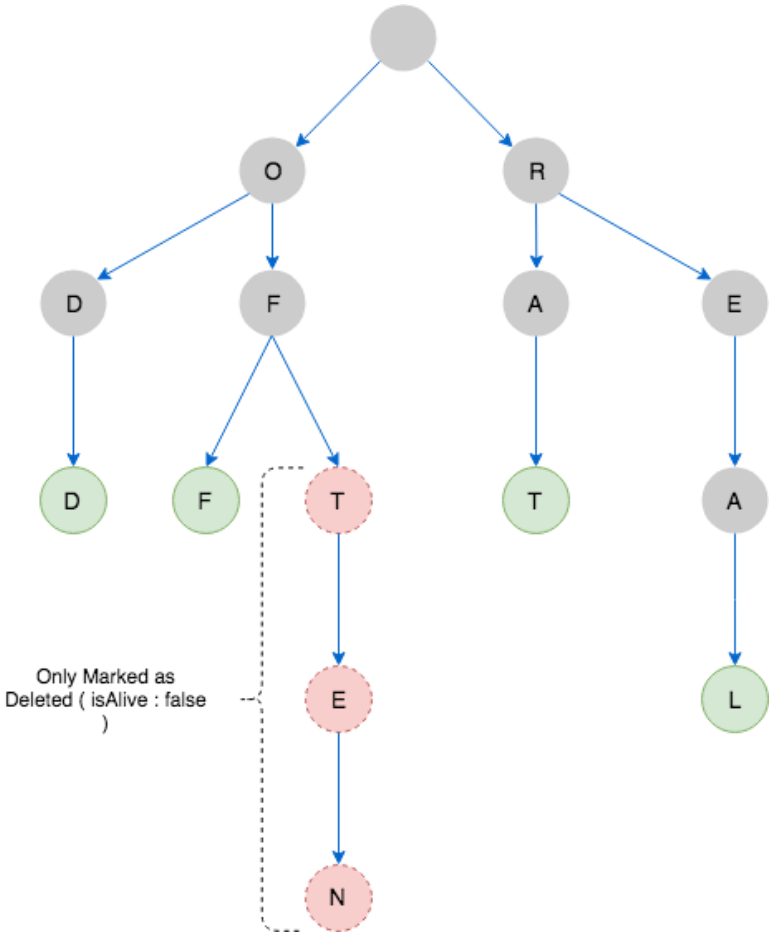

Fig. 6 Lazy Deletion in Trie

Fig 6 shows what lazy deletion looks like for the same set of operations as defined in section 3.3.2

The difference in the above trie is that the nodes T, E and N are not actually deleted from the trie immediately. Instead, they only get marked as deleted. A garbage collector thread later deletes these nodes if some conditions are not met. Following sections describe the conditions for garbage collection.

The conditions mentioned above are two threshold values that decide -

1. If a node should be actually deleted during the garbage collection or not, and
2. When the garbage collector thread should be started.

Later sections detail these parameters in the algorithm . In order to enable efficient lazy deletion and garbage collection, the basic trie node has been modified.

We introduce two new parameters in the node called ***isAlive*** and ***hot_count***

- ***isAlive*** : Boolean value that defines if a node should be counted as a part of the trie or not. Lazy delete operation will set this to false
- ***hot_count*** : The count of hotness of any node. Each time a node is inserted or requested to be inserted (case when node might already be present) the hot

count of the node is incremented by 1. Garbage collector will later use this value to decide which nodes to actually delete

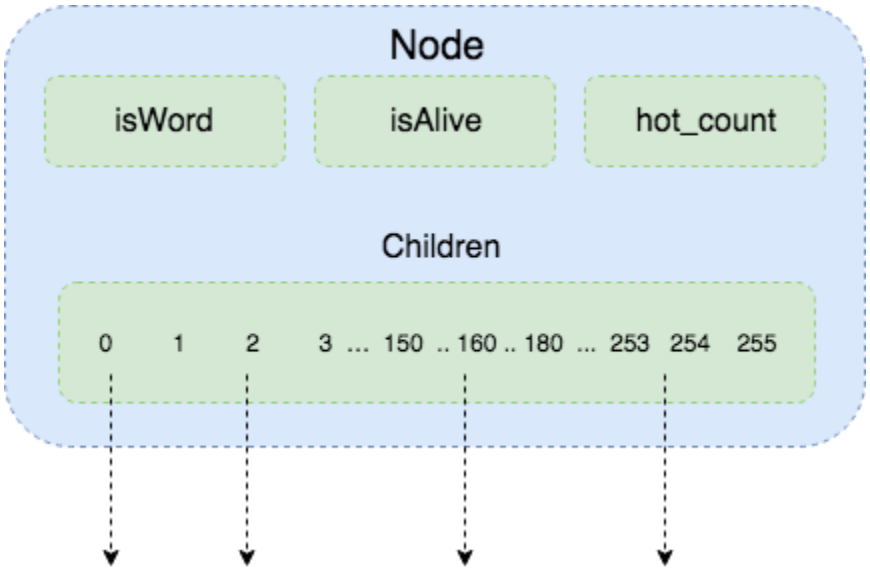

Fig. 7 Modified Trie Node

Fig 7 above represents the modified node structure. Possible values for –

- isAlive : true / false
- hot_count : any integer > 1

The garbage collector is a thread that starts after every ***'X'*** amount of deletions where ***X*** is a configurable parameter. This methodology of garbage collector is known as reference counting strategy [8].

4.2 Slab allocation

Slab allocation is a strategy used in some in memory databases such as Memcached [9]. Slab allocation allows the system to use pre-allocated memory or nodes to avoid initialization time when it is actually needed. In our solution though, we take a slightly different approach than Memcached. Memcached pre-allocates nodes and stores them in a stack with help of a parallel thread. Instead, we do not have any such thread doing slab allocation. We use the nodes from lazy deletion to further delay the process of actual memory freeing.

A stack is maintained that can hold nodes in their initialized states. When the garbage collector traverses through the entire trie and determines that it needs to take a node out of the trie, instead of immediately deleting it, the node will be re-initialized to a blank state and placed on top of the slab-allocation stack. This allows for avoiding re-allocation of nodes during the subsequent inserts. SET operation can check for existence of a node in the slab-allocation stack and pick the node at the top thus saving on execution time. The garbage collector starts freeing the memory when the stack for initialized nodes is full. All the parameters discussed until now are configurable and depending upon the workload, can be tweaked for the best performance.

4.3 Concurrency and consistency on optimized trie

In order to allow for concurrent requests and still maintain consistency, we use locking mechanisms. Although locking mechanisms are not always known as the best solutions for concurrency, they are the best ways to maintain consistency. In our solution, when the garbage collector thread starts, a locking mechanism prevents inserts or updates from happening. With large sizes of tries, locking the trie for the complete duration of the garbage collector operations can become a huge drawback. Therefore, we only set the lock when the garbage collector is performing the actual delete operation on a node or a series of nodes and release the lock as soon as the delete operation is completed. The garbage collector can continue traversing through the trie without the lock set and repeat the process when it wants to delete nodes. With a correct configuration of the parameters mentioned in the sections before, this results in a minimum insertion delay. In fact, as seen in our benchmarks, the performance gain in inserts is large enough to render the delay due to locking irrelevant but still maintaining consistency.

4.4 Operations on optimized trie

This section details the algorithms for SET, GET, DEL operations along with their ancillary functions doDeletes and delInternal.

4.4.1 Insert

```
function SET(root, len, key)
        curr ← root
        i ←  0
        insertStatus ← false
        if ( delLock ≠ true ) then
                while ( i < len ) do
                        if key[ i ] not in curr.children then
                                if (slabSize > 0 ) then
                                        new ←  g_queue_pop()
                                        if ( new ) and ( new.isAlive ) and
                                           (new.hotClount = 1) then

                                                slabSize ← slabSize - 1
                                        else
                                                new ← node()
                                else
                                        new ← node()
                                curr.children[ key [ i ] ] ← new
                                curr ← new
                                if i = len – 1 then
                                        new.isWord ← true
                                        insertStatus ← true
                        else
                                if ( curr.children[ key [ i ] ].isAlive = false ) then


 curr.children[ key [ i ].isAlive ← true
                                else if ( i = len – 1 ) then
                                        insertStatus ← true
                                curr.children[ key [ i ] ].hotCount ←
                                        curr.children[ key [ i ] ].hotCount + 1
                                curr ← curr.children[ key [ i ] ]
                        i ← i + 1
        return insertStatus
```

4.4.2 Search

```
function GET(root, len, key)
        curr ← root
        i ←  0
        found ←  false
        while ( i < len ) do
                if ( key[ i ] in curr.children ) and
                   ( curr.children[ key [ i ] ].isAlive = true ) then
                        curr ← curr.children[ key [ i ] ]
                        if ( i  = len – 1) and (curr.isWord = true) then
                                found ← true
                i ← i + 1
        return found
```

4.4.3 Delete

```
function DEL(root, len, key)
        curr ← root
        delParent ← root
        delChild ← 0
        i ←  0
        deleted ←  false
        del_cnt_add ←  0
        while ( i < len ) do
                if ( key[ i ] in curr.children ) and
                   ( curr.children[ key [ i ] ].isAlive = true )then
                        if ( delInternal ( curr, key [ i ] ) = true ) then
                                delParent ← curr
                                delChild ← key[ i ]
                                del_cnt_add ←  0
                        curr ← curr.children[ key [ i ] ]
                        del_cnt_add ←  del_cnt_add + 1
                        if ( i = len – 1) and (curr.isWord = true) and
                          (delInternal ( curr, key [ i ] ) = false) then
                                deleted ← true
                i ← i + 1
        if ( deleted = true ) then
                delParent.children[ delChild ].isAlive ← false
                delCount ← delCount + del_cnt_add
        return found
```

4.4.4 Ancillary function delInternal

```
function delInternal(node, index)
        i ← 0
        while ( i < 256 ) do
                if ( i ≠ index ) and ( node.children[ i ] ≠ NULL ) and
                   ( node.children[ i ].isAlive = true ) then
                        return true
                return false
```

4.4.5 Ancillary function doDeletes – garbage collector thread

```
function doDeletes(root)
        curr ← root
        i ← 0
        while ( i < 256 ) do
                if ( curr.children[ i ] ≠ NULL ) then
                        if ( curr.children[ i ].isAlive = false ) and
                           ( curr.children[ i ].hotCount < hot_count_lim ) then
                        if ( slabSize < slabSizeLim ) then
                                delLock ← true
                                initializeNode( curr.children[ i ] )
                                g_queue_push_head(labStack,curr.children[ i ] )
                                curr.children[ i ] ← NULL
                                slabSize ← slabSize + 1
                                delLock = false
                        else
                                delLock ← true
                                free ( curr.children[ i ] )
                                curr.children[ i ] ← NULL
                                delLock ← false
                else
                        doDeletes ( curr.children[ i ] )
```

Chapter 5

L-TRIE – A SPACE EFFICIENT TRIE

As discussed in chapter 3 subsection 3.2, the space consumption of a trie is usually bad. Having 256 pointers in each node consumes a lot of space in the main memory. To address this issue, we present a new variant of the trie data structure named L-Trie (Leaf Trie). Section 2.3 also discussed B+ trees. The L-Trie takes clues from both a trie and a B+ tree. Some rules that are followed in the L-Trie are as follows –

1. Keys are stored only in leaf nodes
2. Leaf nodes are fat nodes with an order $\boldsymbol{m}$
3. Number of keys in a leaf follow the rule –

   $$\boldsymbol{m/2} \leq \text{number of keys} \leq \boldsymbol{m}$$

4. To reach the leaves, we traverse through a trie who's nodes act as anchors to leaves.
5. Searching follows some specific rules that we will see in later sections.

Following section details the structure of an L-Trie.

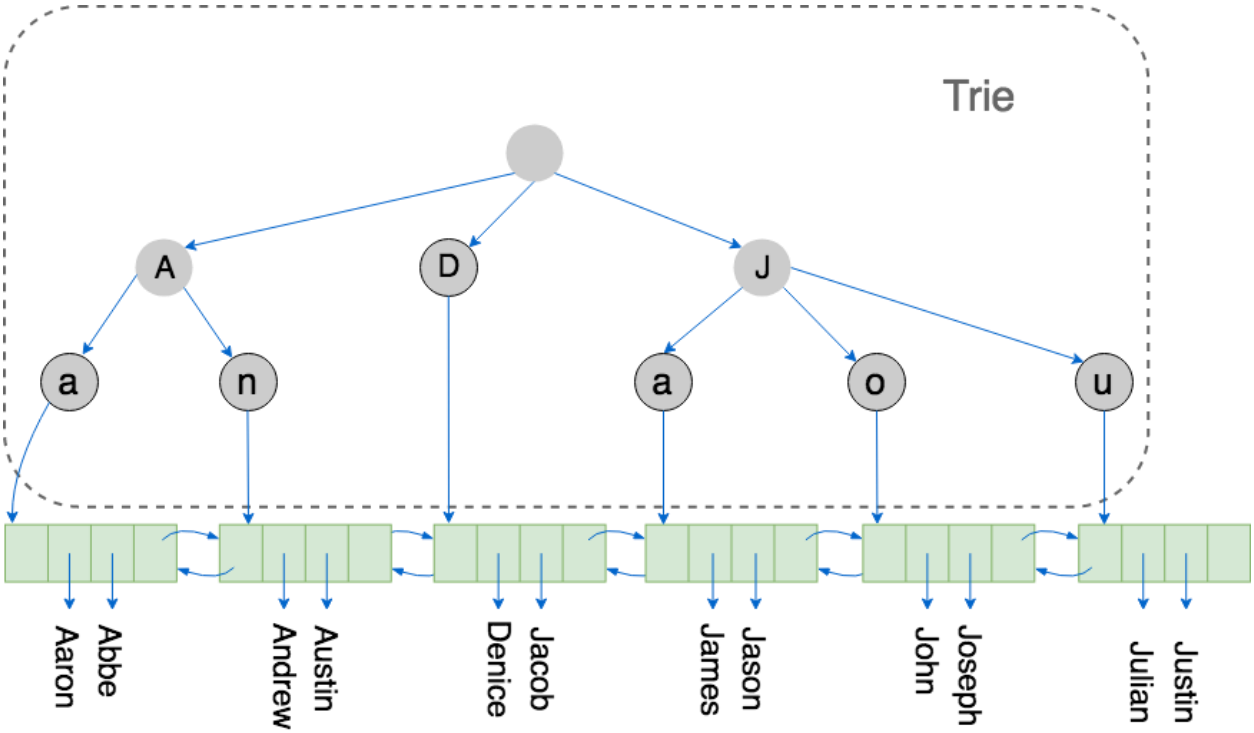

Fig. 8 Example of L-Trie

Fig. 8 shows an example of a L-Trie. The internal tree in a B+ tree is replaced with a trie allowing traversing to the leaves in O (L), where L is the length of the key (in the worst case – where the complete key is an anchor) or the length of the anchor. The trie paths leading to the leaves are called ***anchors***. The leaves are all sorted lexicographically, and all leaf nodes are linked to each other with pointers as next and previous siblings. This allows traversal between leaves without re-traversing through root for range queries.

5.1 Anchors

The anchors in the L-Trie are generated dynamically as insertions happen and serve as paths to trace till leaves. The traversing to leaves via these anchors follows certain rules as listed below –

1. For any leaf node other than left-most leaf node,

   $$left\text{-}key < anchor \leq right\ key$$

   where left-key is any key in the leaf node immediately left to current leaf node (previous sibling) and right-key is any key in the current leaf node.

2. For the leftmost leaf node,

   $$anchor \leq right\text{-}key$$

The initial state of the L-Trie is an empty root node. The first element inserted is inserted in a leaf node directly. A *split-on-demand* during inserts and *merge-on-demand* during deletions strategy is used in the L-Trie. This will be discussed in detail when discussing the operations on the L – Trie.

5.2 Leaves

The leaves of a L-Trie are the nodes that hold the actual keys (and links to values to each key). Therefore, the leaves are basically fat nodes and can be of maximum order m and minimum order $m/2$. All keys in the leaves are always sorted. This enables fast binary search within the leaves with the complexity for this internal search being $O(\log m)$ in worst case, where m is the maximum keys that a leaf can hold. Unlike B+ trees , all the leaves may not be at the same level and the L-Trie can be unbalanced in worst case scenarios. Although, all the leaves are connected to each other with next and previous pointers and allow traversing through the leaves easier.

Due to the rules used to split a leaf during insertion, a value traced to a leaf A might be in the previous node to node A. This scenario only has to backtrack to one previous node in the worst case and does not cost any time overhead asymptotically. In real execution time, this is only two binary search operations, one in each leaf node.

5.3 Operations on L- Trie

5.3.1 Insert

We start with an empty trie root before any keys have been inserted. As mentioned before we use the *split-on-demand* strategy during inserts on leaf nodes. These leaf nodes are fat-nodes which can hold between $\boldsymbol{m/2}$ and $\boldsymbol{m}$ keys. Whenever the key count goes beyond $\boldsymbol{m}$, we split the leaf node and generate necessary anchors using the rules stated in section 5.1. When the very first key arrives, we create a new leaf attached to the trie root and insert the key in the leaf. It is important to remember that a trie root is always an empty node. Similarly in L-Trie, the root never acts as an anchor. It is simply the starting point for the L-Trie. As stated in section 5.2 all keys in the leaf are sorted and this sorting is done after each insert. It is necessary to sort every time after an insert as the order of keys is important if a split operation is demanded as well as for any concurrent search, which uses binary search to search inside the leaf.

During insertions, some keys shave exact prefixes present in the trie as anchors. Insertions for these keys is straight forward. Trace each token in a key to the already existing path of anchors and reach the leaf node where the key can be inserted. But for some keys, there might be a mismatch after matching zero to few tokens within anchors. How the position is found for such keys will be discussed in the search algorithm.

```
function SET(root, len, key)
        curr ← root
        i ←  0
        insertStatus ← false
        splitPoint ← ceil(tree_order/2)
        direction ← bottom
        while ( i < len ) do
                pathFound ←  false
                if ( key[ i ].leafChild ≠ NULL ) then
                        if ( direction = bottom ) then
                                insertInLeaf( curr.leafChild, len, key )
                        else if ( direction = previous ) then
                                if ( curr.leafChild.previous = NULL ) then
                                        createPrevious(curr.leafChild)
                                insertInLeaf( curr.leafChild.previous, len, key )
                        if ( curr.leafChild.no_of_keys > tree_order ) then
                                newLeaf ← new_leaf()
                                if ( curr.leafChild.next ≠ NULL ) then
                                        newLeaf.next = curr.leafChild.next
                                        newLeaf.next.previous = newLeaf
                                curr.leafChild.next = newLeaf
                                newLeaf.previous = curr.leafChild
                                copy_to_leaf(curr.leafChild, newLeaf,splitPoint)
                        old_key ← curr.leafChild.keys[0]
                        new_key ← newLeaf.keys[0]
                        generate_anchors( node, curr, newLeaf, key )
                else
                        if ( curr.children[ key [ i ] ] ≠ NULL ) then
                                curr ← curr.children[ key [ i ] ]
                                pathFound ← true
                        if ( pathFound = false ) then
                                k ← key [ i ]
                                pathFound ←  search_left( curr, k )
                        if ( pathFound = false ) then
                                k ← key[ i ]
                                pathFound ←  search_right( curr, k )
                                direction ←  previous
                        if ( pathFound = false ) then
                                newLeaf ← leaf_new()
                                insertInLeaf(newLeaf, len, key)
                                insertStatus ← true
                i ← i + 1
return insertStatus
```

5.3.2 Search

Searching for a key in L-Trie can be divided into four broad conditions. The search starts like a trie by tokenizing the key and trying to find a path to the leaves using anchors. The four broad conditions as can be seen in the algorithm are -

1. Leaf node is found
2. Anchor node is found and token exactly matches the anchor node
3. Anchor node is found and token does not exactly match any anchor but the trie has an anchor with smaller value
4. Anchor node is found and token does not exactly match any anchor but trie has an anchor with larger value

The search algorithm follows rules based on these four conditions and we shall see them in detail now.

1. Leaf node found

When a leaf node is found, the traversing on the trie stops. This case represents that either the key will be found in this leaf or the previous leaf depending upon which of the three remaining conditions was taken in the earlier traversing of the

trie. The details about these will be mentioned in the following rules. The search in the leaf node is a binary search.

2. *Anchor node found and token matches exactly*

   When a token matches the exact found anchor, continue traversing down the trie by updating current pointer to the anchor just found

3. *Exact anchor not found but smaller anchor exists*

   If condition 2 fails, check for the existence of the closest smaller anchor. If such an anchor is found, traverse to the rightmost leaf of this anchor. This sets the direction variable to bottom, meaning , in the next iteration when condition 1 is met, the binary search in the leaf will happen in the exact leaf that was reached in this step.

4. *Exact anchor not found but larger anchor exists*

   If condition 3 fails, check for the existence of the closest larger anchor. If such an anchor is found, traverse to the leftmost leaf of this anchor. This sets the direction variable to previous, meaning, in the next iterations when condition 1 is met, the binary search in leaf will happen in the leaf previous to the leaf reached in this step.

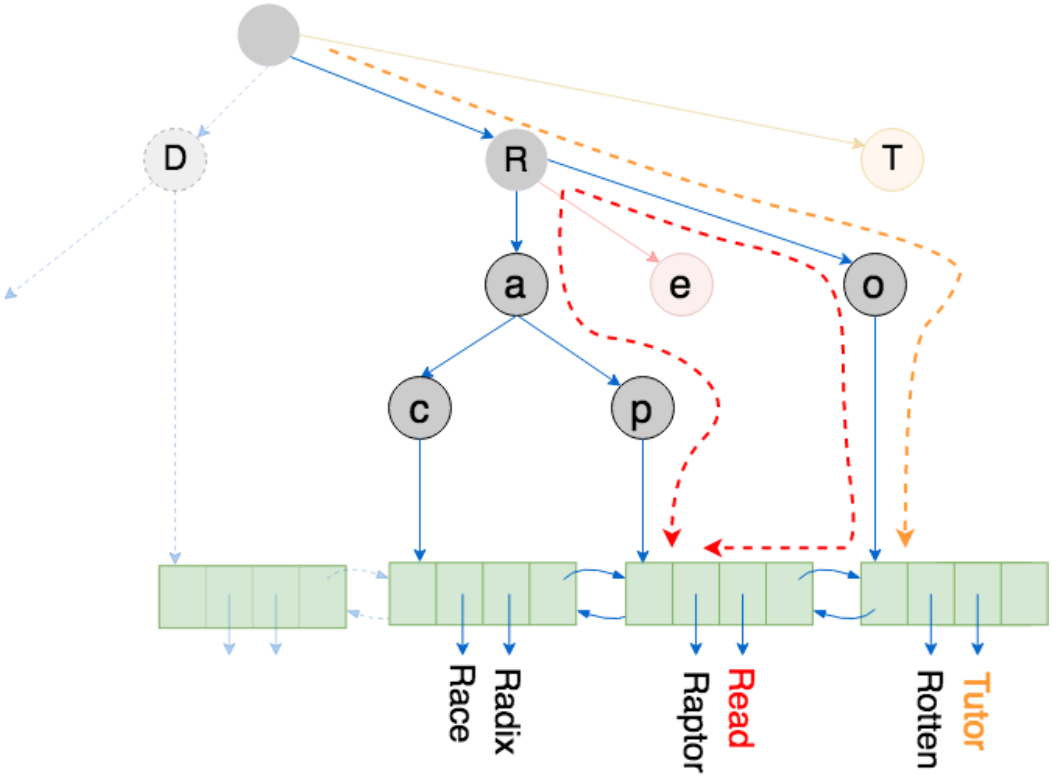

Fig. 9 Searching in a L-Trie

Fig. 9 shows a case where conditions 3 and 4 are in picture.

If we want to search for the key “Read” , we first match the token R with root nodes’ children and find an exact match with the anchor. Following this condition 2, we traverse ahead with R being our current node. After this there are 2 possible paths that can be taken. First path follows condition 3 where, as soon as a mis match occurs (token – e, anchors available – a, o) we search for nearest smaller anchor (a in this case) and traverse to the right most leaf of anchor a. the direction for searching here is *bottom*, so we will perform search for the key in the selected node (node A for reference purposes)

itself. The second path is the path that follows condition 4. In this condition, we search for the closest larger anchor (o in this case) and traverse to the left most leaf of anchor o. The direction for searching here is *previous*, so we will perform a search for the key in the previous node (node A) to the selected node (node B for reference).

Another example of similar kind would be to search for the key " Tutor ". The difference in this case is that right from the beginning token (T) we never find an exact anchor match and thus go for condition 3 straight away. The rest of the process is the same for condition 3. Also worth noting here, is that for this example, there can never be condition 4 unless an insert operation later adds a node greater than R to the root nodes children.

```
function GET(root, len, key)
        curr ← root
        i ←  0
        searchStatus ← false
        direction ← bottom
        while ( i < len ) do
                if ( key[ i ].leafChild ≠ NULL ) then
                        if ( direction = bottom ) then
                                if ( searchInLeaf( curr.leafChild, len, key ) = NULL ) then
                                        if ( searchInLeaf(curr.leafChild.previous, len, key)
                                            = NULL ) then
                                                searchStatus ←  false
                                        else
                                                searchStatus ←  true
                                else
                                        searchStatus ← true
                        else if ( direction = previous ) then
                                if ( searchInLeaf ( curr.leafChild.previous, len, key )
                                        = NULL ) then
                                        if ( searchInLeaf (curr.leafChild.previous.previous,
                                                        len, key ) = NULL ) then
                                                searchStatus ← false
                                        else
                                                searchStatus ← true
                                else
                                        searchStatus ← true
                else
                        if ( curr.children[ key [ i ] ] ≠ NULL ) then
                                curr ← curr.children[ key [ i ] ]
                                pathFound ← true
                        if ( pathFound = false ) then
                                k ← key [ i ]
                                search_left( curr, k )
                                pathFound ← true
                        if ( pathFound = false ) then
                                k ← key[ i ]
                                search_right( curr, k )
                                pathFound ←  true
                                direction ←  previous
                i ← i + 1
return searchStatus
```

Chapter 6

EXPERIMENTS AND RESULTS

The results of the optimizations and L-Trie are presented in this chapter. In order to present the results, comparisons with the original trie data structure and B+ trees are shown so as to allow for fair comparison.

6.1 Data Sets

The data sets used in the performance evaluation are mentioned bellow –

- Small Length Keys –
    - Keys between length 5 – 8 characters
    - Example : Android, Apple, Zebra
- Variable Length Keys –
    - Keys between length 1 – 32 characters
    - Example : &, F-spot, checoslovaquia
- Long Length Keys –
    - Keys with length 36 characters
    - Example : uuids (57be90ea-8611-406f-b3d2-7668752a63bd)

These keys are used in evaluating both the optimizations – Insertion time and L-Trie.

Data-sets used for evaluation as mentioned before, display unique characteristics in order to enable testing the data structure for all types of situations. Some of these include –

- Small value of L
- Large number of common prefixes
- Large value of L with limited amounts of common prefixes
- Etc.

The specific characteristics of the data sets we use are represented in the following table

Data Set	Characteristic
Small Length Keys	Large number of common prefixes
Variable Length Keys	Moderate number of common prefixes
Long Length Keys	Very low number of common prefixes

Tab. 1 Data set characteristics

Testing against such workloads enables uncovering the performance of the data structure in best to worst case and also to understand the scope of the data for which any data structure is suitable.

6.2 Setup used for evaluation

The performance evaluation for all the optimizations and data sets was done on the following configuration –

- Intel(R) Xeon(R) CPU L5410 @ 2.33GHz (8 cores)
- 64 GiB memory
- Arch Linux
- Kernel Version Linux p10 4.15.12-1

6.3 Framework

The performance evaluation of the optimized data structures and the original data structures was done using a common framework designed in house [10]. The framework takes as input a set of keys to insert from a file and a set of operations to be performed.

Below is an example of the input file for keys –

```
AAMSI
Aandahl
A-and-R
57be90ea-8611-406f-b3d2-7668752a63bd
c97ff316-94d6-4ce9-b37a-b22f97b29f37
b7ffb1e6-cd18-4bca-85df-49d9dd84dc9c
```

each new line represents a new key to be inserted or searched or deleted from the data structure.

The specific operations to be done are taken as input from another workload file and look as follows –

```
seqset 0 1580179
seqdel 0 1580179
rndget 0 1580179
rnddel 0 1580179
seqget 0 1580179
```

each line is divided into 3 parts –

- Operation (seq : sequential, rnd : random, set / get / del)
- Start point for keys from keys file
- End point for keys from keys file

6.4 Results for insert optimization

In order to observe the performance gain in this optimization, the workload has to be designed in a specific way. Since the optimization allows for reducing insert times after having had a defined number of deletes on the data structure, the workload has to insert, delete, insert. There may be other operations like get in between.

```
seqset 0 1580179
rnddel 0 1580179
rndget 0 1580179
seqset 0 1580179
seqget 0 1580179
seqdel 0 1580179
```

Above we can see the sequence of operations as mentioned before following the insert – delete – insert rule. Operation on line number 4 is when we can see the insertion time reduce. The garbage collector thread should have performed its operations during the execution of line 3 simultaneously.

The results for this test reveal a huge increase in operations per sec as compared to a conventional trie. Graph 1 shows the comparison between a conventional trie and the optimized trie for all the workloads mentioned before.

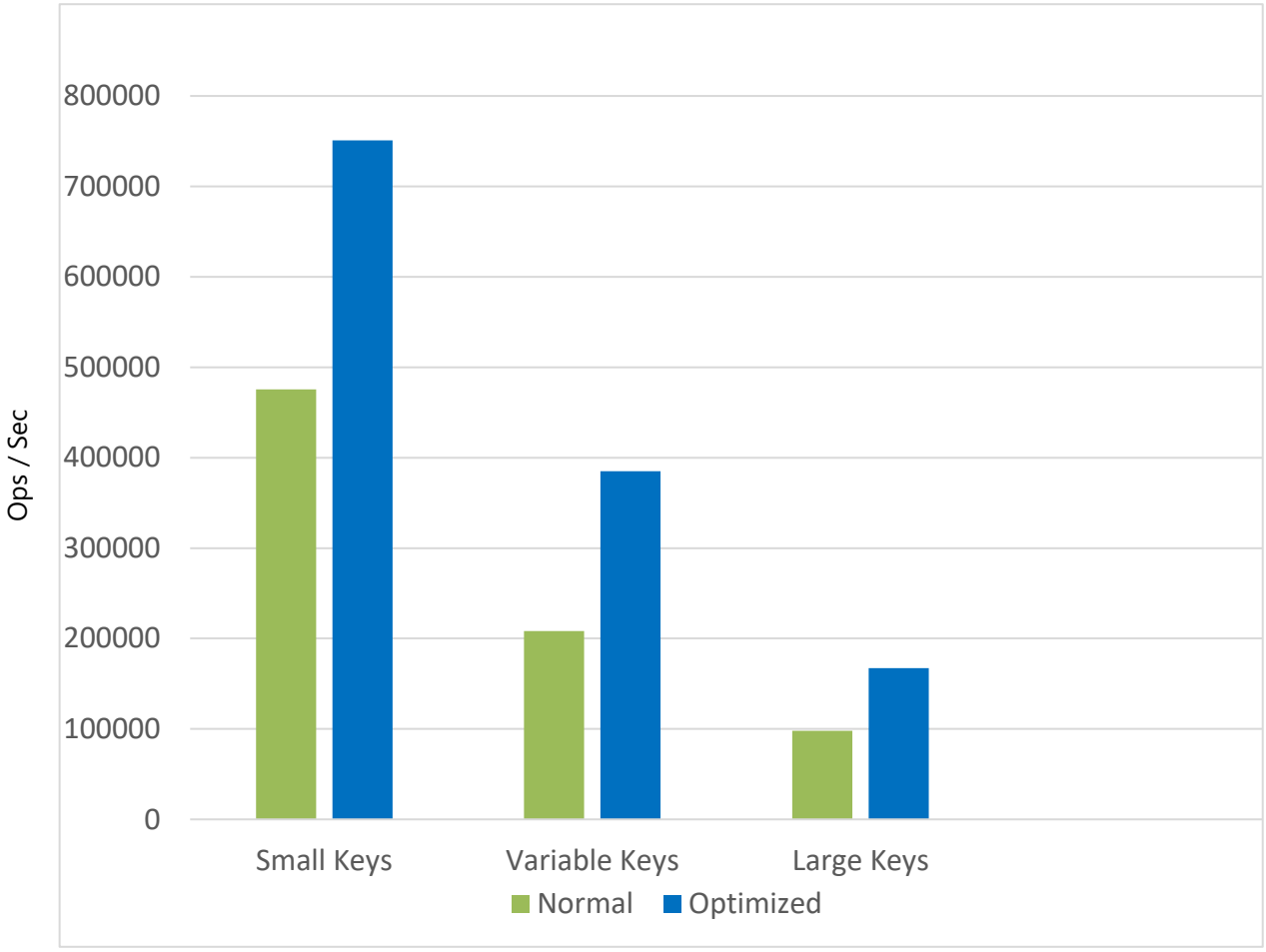

Fig. 10 Performance for time optimization

As seen in Fig. 10, the time optimized version performs almost two times as much as a normal trie with the same set of operations and keys. The performance gain seen in the three types of keys is different due to the nature of keys as seen in section 6.1 and their resulting implications. The performance gain is maximum in small length keys because they share a large number of prefixes. Thus, when there are deletions in the trie, many nodes that were not deleted are reused in either same or different keys inserted later.

In contrast to this, for the large length keys which are uuids, the number of common prefixes is low. Thus the hot_count for each node is not very high resulting in the garbage collector deleting many of these nodes. Also, during subsequent insertions, common prefixes are low and the token needed to be inserted is not already present in the trie (either as alive or as marked deleted). A combination of these factors results in the performance gain being low.

The take away from these evaluations suggests that apart from the system being operations heavy, data – sets with moderate to high number of common prefixes are the best target candidates for this optimization. On the other hand, a system that has low mounts of inserts and deletes or has a lower number of common prefixes (for e.g. in workload 3 – long length keys), this system may not present improved results.

6.5. Results for L-Trie

The main aim of L-Trie is to improve the space efficiency of an L-Trie by building on top of the conventional trie and B+ tree. Apart from performing well in space consumption, we also see a unique result in time utilization when the length of the keys increases and the number of shared prefixes are less. We shall see these results in this section.

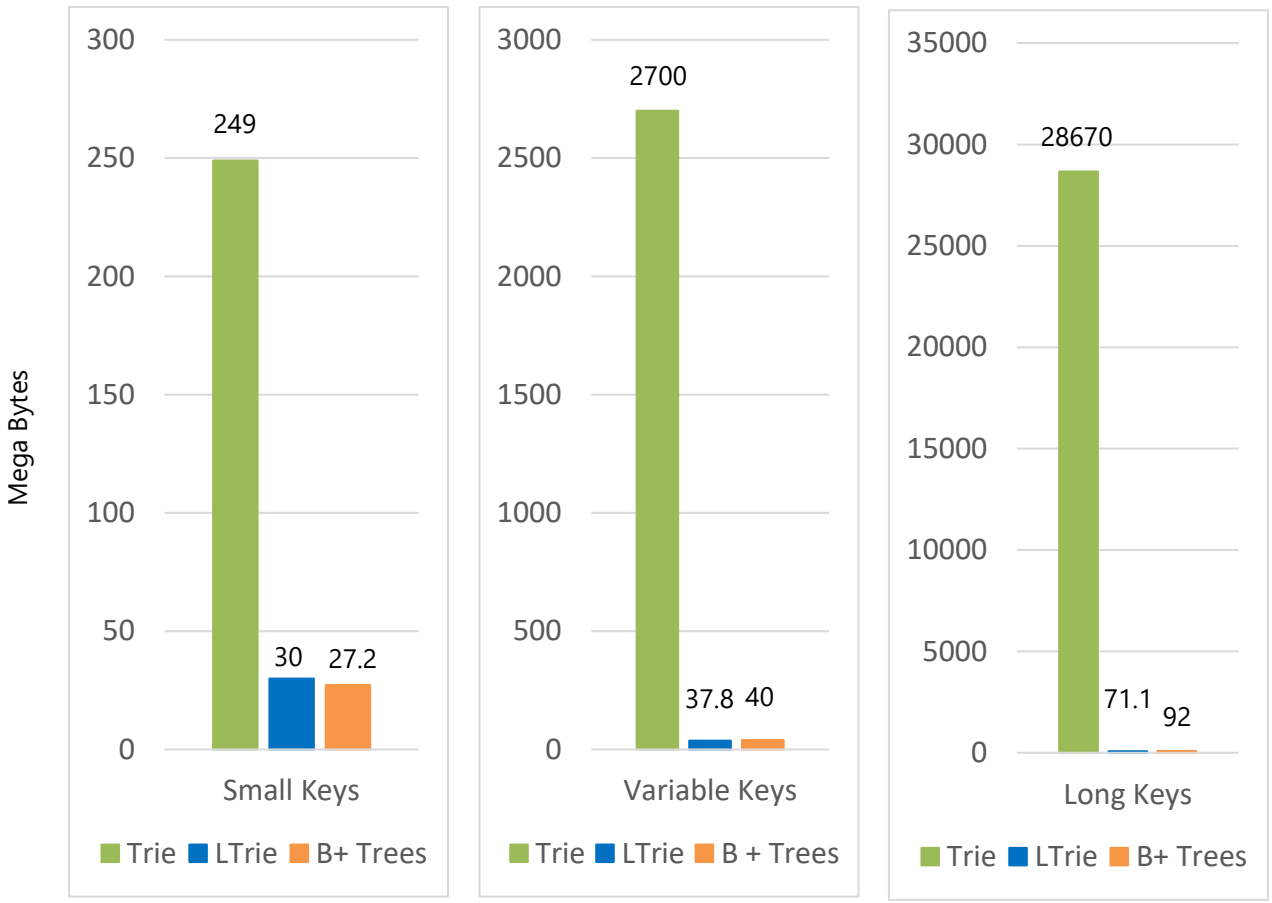

Fig. 11 Comparing space consumption

Fig. 11 shows the space consumption comparison for a conventional trie, L-Trie and B+ Tree. The space consumption for a L-Trie is comparable to that of a B+ tree. The most gain can be seen when compared to a conventional trie. The table below also shows the number of nodes created for a trie vs an L-Trie

Data Set	Trie nodes	L-Trie internal nodes	L-Trie leaf nodes
Small Length Keys	135881	201	161
Variable Length Keys	1420649	1222	913
Long Length Keys	14902561	850	577

Tab. 2 Node counts

The data represented in the table shows the difference in the number of nodes created in a conventional trie and L-Trie and that the difference is huge. These results are generated with the max size of leaf nodes (order of L-Trie) $\boldsymbol{m}$ = 7000. Changing this value can produce better or worst results. For e.g. increasing the value of $\boldsymbol{m}$ too much results in increasing the time required for binary search inside the fat leaf node. It also results in increased time for insertions as sorting is necessary at every insert. On the other hand, if the size of $\boldsymbol{m}$ is set too low, the result is increased internal nodes (anchors) and will bring the L-Trie closer to a conventional trie as the value is reduced more. It is important to understand the input data to be stored in an L-Trie and configure the necessary parameters accordingly. This is also true for a B+ tree.

Not only does L-Trie perform better in space consumption, but also in execution time when the length of the keys increases. Although, for smaller length keys, a conventional trie may perform better. As seen in fig. 12 we can see the performance tradeoff due to the bsearch inside a fat node of size 7000 for small and variable length keys. Long length keys which are the biggest problem for a trie expectedly perform bad in a conventional trie while L-Trie performs a lot better. Another factor that can be seen in L-Trie is that the variation (delta) between the three data sets is low allowing for a more consistent performance.

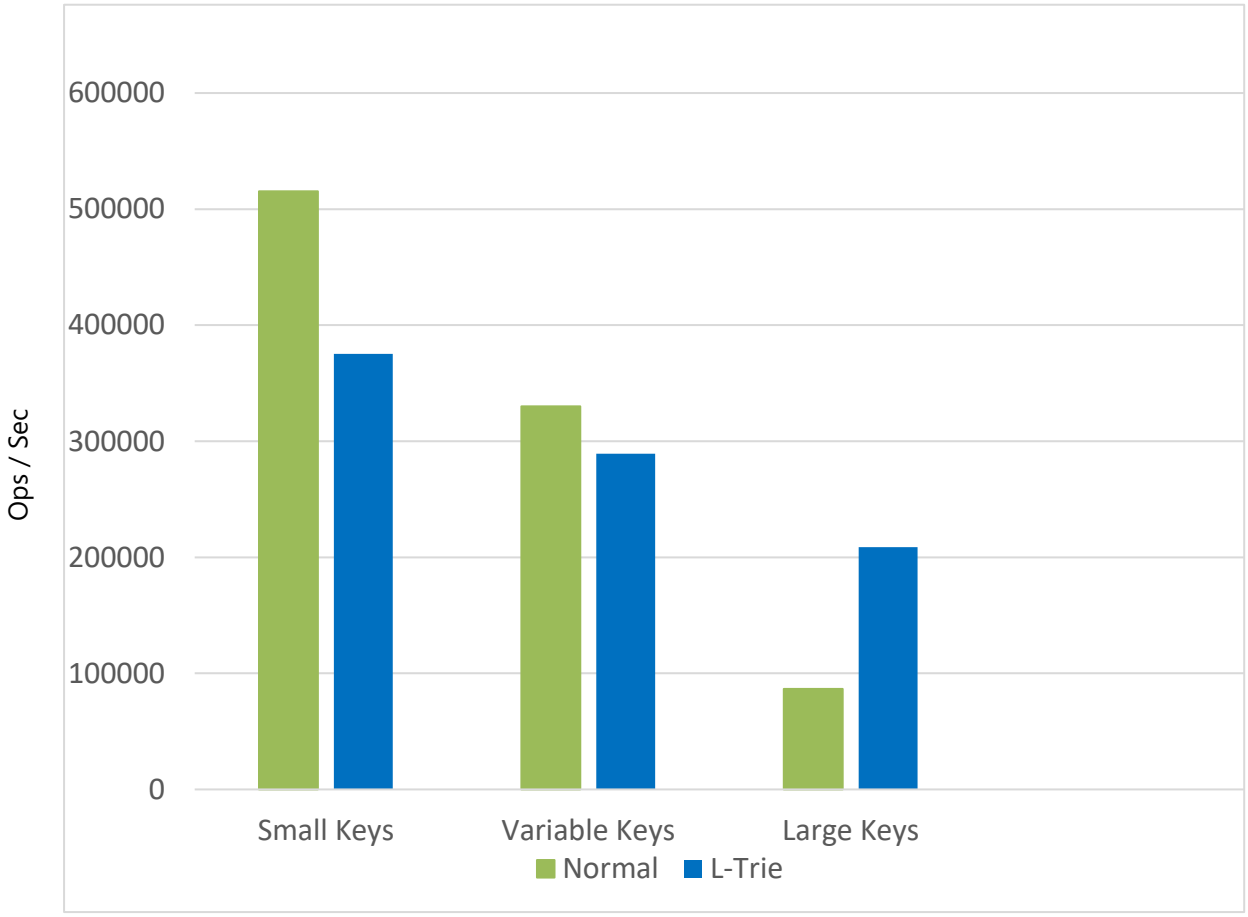

Fig. 12 Comparing time efficiency

Asymptotically, the time complexity of L – Trie can be calculated as –

$$O(L + log\, m)$$

Where L is the length of the anchor (length of key in worst case) and m is the number of keys in a leaf node (order of L – Trie). $Log\ m$ is the complexity of bsearch inside the leaf node.

Chapter 7

CONCLUSION AND FUTURE WORK

This thesis presents two optimizations to the trie data structure after having studied various data structures and evaluating the potential candidates. It presents a optimization that enables reduced insertion times. This enables transaction heavy workloads to increase its performance. Secondly this thesis presents a new data structure L – Trie that builds on principles from B+ tree and trie. L – Trie allows for much better space consumption making it possible to have the trie data structure as a potential candidate for many applications such as indexing. The time consumption of an L-Trie is also more consistent than many other data structures and allows for better performance even with large keys.

The work in this thesis can be extended to allow for better space and time consumption. The insert optimized version of trie can be made more robust by allowing for concurrent operations without the use of locks. The garbage collector for this version can also be optimized for better performance. Future work for L- Trie can be achieved by balancing the leaf nodes during insertions like B+ trees. B+ trees shift elements between leaves if their siblings have capacity. This can be extended to the L-Tries to reduce the number of leaves and save even more space.

REFERENCES

[1] Xingbo Wu, Fan Ni, Song Jiang, "Wormhole: A Fast Ordered Index for In-memory Data Management", *Ph.D. Dissertation, UT Arlington, 2018*

[2] Edward Fredkin, "Trie Memory", *CACM, 3(9):490-499, September 1960*

[3] William Pugh, "Skip lists: a probabilistic alternative to balanced trees", *Communications of the ACM, 1990*

[4] Leveldb: A fast and lightweight key/value database library by google. https://code.google.com/p/leveldb/.

[5] Rudolf Bayer and Edward M. McCreight , "Organization and Maintenance of Large Ordered Indices." *Acta Informatica 1: 173–189,1972*

[6] Douglas Comer, "The Ubiquitous B-Tree", *ACM Computing Surveys 11(2): 121–137, 1979*

[7] Celis Pedro, Franco John, "The Analysis of Hashing with Lazy Deletions", *Computer Science Department, Indiana University, Technical Report CS-86-14, 1995*

[8] Wilson, Paul R. "Uniprocessor Garbage Collection Techniques". *Proceedings of the International Workshop on Memory Management. London, UK: Springer-Verlag. pp. 1–42. ISBN 3-540-55940-X. Retrieved 5 December 2009. Section 2.1.*

[9] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook.", *In Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), pages 385–398, Lombard, IL, 2013. USENIX*

[10] Xingbo Wu, "TrieToy : A Trie framework", available - https://goo.gl/H5fgoS

BIOGRAPHICAL STATEMENT

Nirmik Kale was born in Ahmednagar, Maharashtra, India. He received his bachelor's degree in computer engineering from K.K.Wagh Institute of engineering education and research under University of Pune, MH in May 2014. Thereafter, he worked as a Cloud and Research Engineer at ESDS Software Solutions from June 2014 – July 2016. In fall of 2016, he started pursuing his master's degree in computer science from University of Texas at Arlington. During summer 2017 he worked as a cloud and research intern at BodHOST Ltd., New Jersey. He received his master's degree in August 2018 from University of Texas at Arlington – Computer Science department. His research interests include, but are not limited to - operating systems, Linux, algorithms, cloud and virtualization.