TECHNISCHE UNIVERSITÄT DRESDEN

Master thesis

Database Group

Bit Flip Resilient Prefix Tree

Supervisors:

Author:

Mina Ahmadi

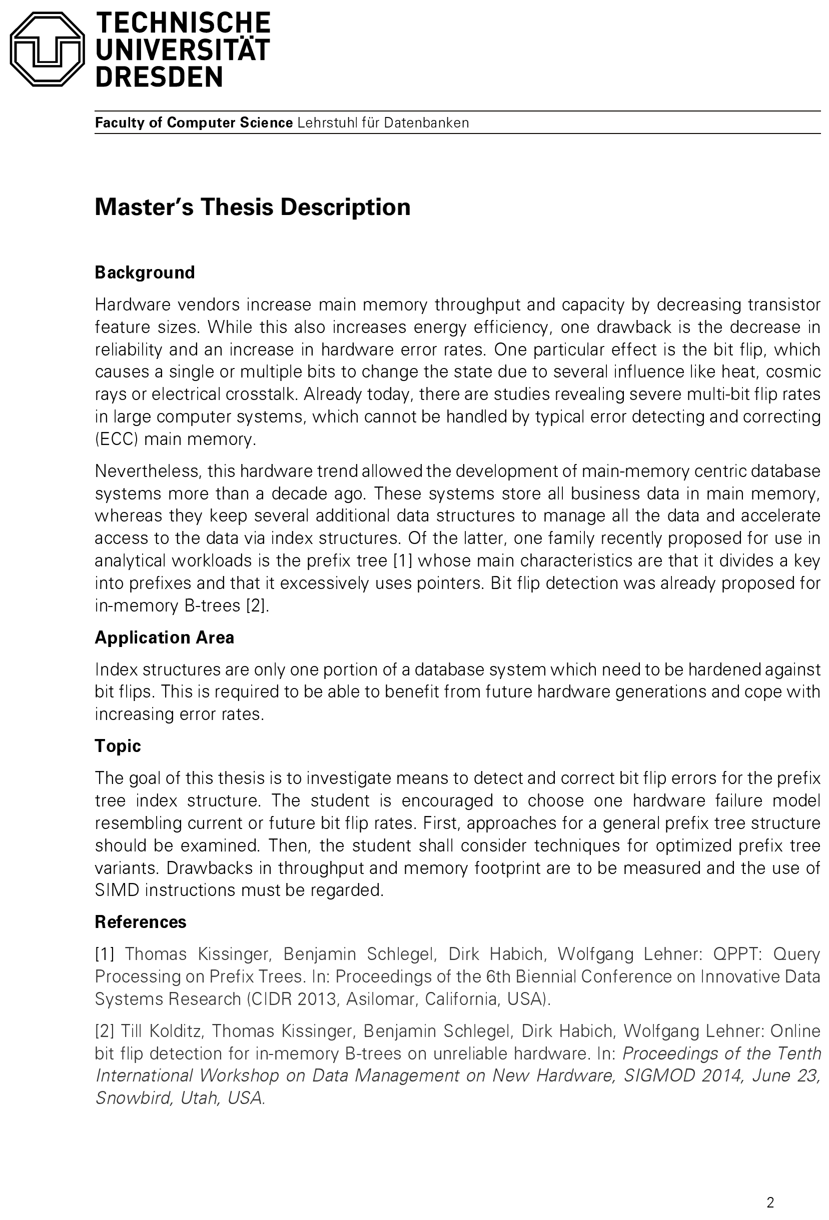
Prof. Dr.-Ing Wolfgang Lehner,

Dr.-Ing. Dirk Habich,

Dipl.-Inf. Till Kolditz

February 2016

Aufgabenstellung



TECHNISCHE
UNIVERSITÄT
DRESDEN

Faculty of Computer Science Lehrstuhl für Datenbanken

Master's Thesis Description

Background

Hardware vendors increase main memory throughput and capacity by decreasing transistor feature sizes. While this also increases energy efficiency, one drawback is the decrease in reliability and an increase in hardware error rates. One particular effect is the bit flip, which causes a single or multiple bits to change the state due to several influence like heat, cosmic rays or electrical crosstalk. Already today, there are studies revealing severe multi-bit flip rates in large computer systems, which cannot be handled by typical error detecting and correcting (ECC) main memory.

Nevertheless, this hardware trend allowed the development of main-memory centric database systems more than a decade ago. These systems store all business data in main memory, whereas they keep several additional data structures to manage all the data and accelerate access to the data via index structures. Of the latter, one family recently proposed for use in analytical workloads is the prefix tree [1] whose main characteristics are that it divides a key into prefixes and that it excessively uses pointers. Bit flip detection was already proposed for in-memory B-trees [2].

Application Area

Index structures are only one portion of a database system which need to be hardened against bit flips. This is required to be able to benefit from future hardware generations and cope with increasing error rates.

Topic

The goal of this thesis is to investigate means to detect and correct bit flip errors for the prefix tree index structure. The student is encouraged to choose one hardware failure model resembling current or future bit flip rates. First, approaches for a general prefix tree structure should be examined. Then, the student shall consider techniques for optimized prefix tree variants. Drawbacks in throughput and memory footprint are to be measured and the use of SIMD instructions must be regarded.

References

[1] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, Wolfgang Lehner: QPPT: Query Processing on Prefix Trees. In: Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR 2013, Asilomar, California, USA).

[2] Till Kolditz, Thomas Kissinger, Benjamin Schlegel, Dirk Habich, Wolfgang Lehner: Online bit flip detection for in-memory B-trees on unreliable hardware. In: *Proceedings of the Tenth International Workshop on Data Management on New Hardware, SIGMOD 2014, June 23, Snowbird, Utah, USA.*

Declaration of Authorship

I, Mina Ahmadi, declare that this master's thesis entitled 'Bit Flip Resilient Prefix Tree' is my own work. All information published and written by another person is acknowledged and referenced in the bibliography.

Dresden, 1st of February 2016

Abstract

Detecting bit flips in index structures of databases is an important issue. Nowadays, index structures reside in memory in order to avoid accessing the disk and make processes of databases faster. Since size of the memories are big enough nowadays, there is no problem to keep the indexes in memory but in case of bit flip occurrence, we would like to detect them to avoid accessing a part of memory which is not allocated or accessing wrong values. When bit flip happens, bits can change from 0 to 1 or vice versa. Bit flips can happen because of multiple reasons. Sometimes bit flips are transient, so they will go away after a reboot, but sometimes they will not change at all.

The index structure which is going to be explored is prefix tree. In this thesis, parity bits are used for the pointers in the tree in the first step. In the second step, memory allocation for the prefix tree is done in a way, so that locating the nodes is possible through some calculations and any bit flips can be detected by any mismatch between the calculation and the existing pointers. In the third step, prefix tree is implemented in a way to reduce the memory accesses but bit flip detection is done in the same way as in the second step.

Based on the results of these implementations, bit flips can be detected by using parity bits if the number of bit flips on a byte is odd. In the implementations on the second and third step, any number of bit flips could be detected. There were some memory allocation issues in both of the methods which resulted in high memory consumption when a lot of keys were inserted into the tree, but a reasonable solution for that is given at the end of the thesis.

Acknowledgement

Firstly, I would like to express my sincere gratitude to my advisor Dipl.-inf. Till Kolditz for the support of my master thesis, for his patience, motivation, and immense knowledge.
Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Dr.-Ing. Wolfgang Lehner and Dr.-Ing. Dirk Habich.

Last but not the least, I would like to thank my family, my parents, my brothers, my sister and my friends for supporting me spiritually throughout writing this thesis and my life in general.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Database systems used to exist in hard disks for a long time. With emerging new technologies, having large memories is not a problem like before. So nowadays, database systems can reside in memory. Having database systems in memory has a great benefit because the latency would decrease significantly.

Memories are always in danger of bit flips. When bit flip is an inevitable issue, it can corrupt instructions or data in memory. This can change the control flow of the program or change the data which is going to be read from memory.

With having index-structures of databases in memory, we should be ready for the corruptions that happen in memory. Having corrupted index structures will lead to getting wrong information or accessing a part of memory which has not been allocated, thus reaching segmentation faults.

Detecting bit flips in memory is a vital task. Different index structures are introduced for keeping table indices in databases. In this thesis, our goal is to find bit flips occurring in one specific type of index structures called prefix trees.

1.2 Objectives

Finding bit flips in prefix trees, which is one type of index structures used for keeping indexes, is the main objective of this thesis. We would like to achieve reasonable performance at the end. We would like to minimize the memory usage and detect as many bit flips as possible. For this purpose, we use a few techniques for bit flip detection and show the result of each technique. We will also do performance evaluation to compare these techniques. The techniques which are used for bit flip detection, are:

- Using parity bits
- Using structure of tree itself to locate nodes and other items in the tree.

1.3 Structure of the Thesis

The rest of this thesis is organized as follows.

The second chapter includes some background information along with introducing important terms. In addition, we introduce and review some of the related work in detail.

In the third chapter, the methodologies which are used in this thesis is given. The strategies that we have explored and employed to reach the objectives of this thesis are explained in detail. We have also included some conclusions of the methods which are used in order to show the pros and cons of each method in the matter of memory usage and bit flip detection.

In the fourth chapter, the implementation chapter, all implementation details of the used methods are given.

Finally, in chapter five, evaluation results and final conclusions are given to show the result of the used methods. Future directions on this subject are provided in this chapter for further investigation by future researchers.

Chapter 2

Background and Related Works

In this section, some basic and important terms is introduced and explained. In addition, a review of the related previous work is provided.

2.1 Background

2.1.1 What is Bit Flip?

Any change in the bits which are stored in the memory is called bit flip. If binary value 1 changes to 0 or 0 changes to 1, then a bit flip has occurred.

2.1.2 Types of Error

Bit flips that happen in the memory are divided into two groups. They can be soft or hard errors.

1. **Hard errors**: If a failure happens in a memory chip, then we might have hard errors. There could have been a permanent electrical failure in memory. The only solution to fix this kind of errors is to replace the memory chip. One thing that could cause hard errors is having system functions over the limit of the memory speed [5].

2. **Soft errors**: These types of errors do not cause any failure in the hardware, however, they will change the data in the memory. Rebooting the computer causes the error to go away. [6]. There are a couple of situations which lead to a soft error.

   - Soft error could happen when radio actives in the chip's material go bad. In that case, some alpha particles will be released which can change the data in a cell [6].
   - Soft error could also happen when data is being processed. Data can be hit by noise and noise can be interpreted as data bits [6].

2.1.3 Soft Errors Detection Mechanisms

There are some hardware and software mechanisms for detecting soft errors which are listed below.

Hardware-based techniques

Some hardware-based multi threading techniques [7, 8, 9] exist which will execute the application multiple times and the outputs are compared to identify errors. This can cause hardware design complexity and high cost and performance overhead [10].

Immunity-aware programming

Immunity-aware programming is used when a firmware is programmed for a system. This will improve the tolerance towards soft errors in modules of the program or in program counter. This technique can be used in the software which is written in the source or in the victim device [11].

RAM parity memory

Storing one bit of data as parity which can be even or odd is another technique for detecting soft errors. Parity bit is usually calculated for one byte of data. While reading data from memory, based on the parity bit which is stored, bit flips can be detected. As an example if the data byte is as follows:

11001000

Then:

Parity odd: 0 Parity even: 1

Parity bit used to be stored in an extra memory chip, but now we have DIMM (Dual in-line memory module) and SIMM (Single in-line memory module) which can be found in parity and non-parity modules [12].

ECC memory(Error Correction Code Memory)

By using error-correcting code or parity bit as redundant data, single bit errors can be detected. The example of ECC can be Single-Error Correction and Double-Error Detection (SECDED). Hamming code allows correction of one bit and detection of two bits. ECC is used in systems were corruption is not acceptable. When systems are not critical, motherboards and processors will not support ECC in order to reduce the expenses. Using ECC memory will be more expensive because producing ECC memory modules and having motherboards and processors which are compatible with ECC memory is expensive. Memory performance will decrease because ECC memory controllers need to do some calculations for error checking [13].

After giving an overview about bit flips and mechanisms for detecting them, we put our focus on the bit flips that happen in index structures of the databases and how they can be detected.

In the next section of this chapter, based on the related works, we give an overview of the index structures and some corresponding detection mechanisms.

2.2 Related Works

2.2.1 A Study of Index Structures for Main Memory Database Management Systems

In [3], different types of in-memory index structures have been explained and compared with each other and a new index structure, the T-Tree, has been introduced. The main goal for an index structure in disk is to minimize the number of disk accesses, while the goal for an index structure in memory is to emphasize on the use of CPU cycles and less usage of memory. Since all the relations are in the memory, it is not needed to keep the values of the attributes according to each index. Instead, the pointer to the values can be kept. By just keeping the pointer, we reduce the size of the index because we don't need to store the attributes according to that index.

Array

One of the index structures that can be used is array. Indexes can be kept in an array. The size of the structure is fixed. The main disadvantage is that the complexity for each deletion or insertion operation is $\mathcal{O}(N)$.

AVL Trees (Georgy Adelson-Velsky and Evgenii Landis)

A binary tree search can be done in AVL tree which is quite fast. If an insertion or deletion causes an imbalance in the tree, a rotation will be done to keep the tree balanced. The disadvantage is the inefficient memory utilization, since each node will hold one item and two pointers to its children.

B Trees

Unlike AVL trees, the depth of a B tree is shorter and fewer number of nodes need to be accessed while searching for an index. Insertion and deletion is also fast. Some database systems uses a type of B-Tree which is called B+-Tree. In this type of tree, all the data is kept in the leaf level. Although this is not an advantage in in-memory index structures, cause it increases memory usage.

Chained Bucket Hashing

This structure is static and its size is fixed. The operations are fast but since size of the structure should be defined first, if the size is too small, performance can be poor but if the size is too large, then memory might be wasted.

Extendible Hashing

Unlike *Chained bucket hashing*, this structure is dynamic. Whenever a new piece of data is going to be added, if the corresponding node has space, it will be added there. Otherwise, the node will split. If a split is not possible on the node, then the directory size has to increase. For example, if the directory has 00, 01, 10, 11 as the prefixes and we have to increase the directory size, then we will have 000, 001, 010, 011, 100, 101, 110, 111 as the prefixes. The problem with this approach is that insertion of one node can cause enlargement of the directory. An example of extendible hashing is shown in figure 2.1. Based on the example which is shown in figure 2.1, when keys 32, 10, 18, 12, 5, 21, 25, 17, 11 and 7 are added, directory has 00,01,10,11 as prefixes. When 6 is added, it has 10 as its two least significant bits, but as we can see, both of the buckets 10 and 00 are pointing to a node which is full. Here, this node will be split and there will be four nodes. Keys which have 00 as the two least significant bits will point to one specific node, same for the keys which end in 01, 10 and 11. Now if 33 is going to be added, it has 01 as the least significant. The node which is pointed to by bucket 01 is full. So now it is the time to increase the directory size to three. [18]

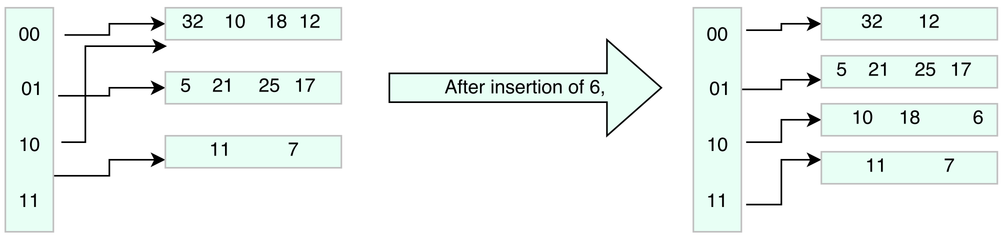


Figure 2.1: Extendible Hashing

Linear Hashing

Linear hashing is also a dynamic structure. The directories are not separate from the keys, but the keys are kept inside the directories. After adding each node, a decision has to be made with regards to splitting of the node. If the result of the following equation is greater than 75%, then the node should be split:

$$\frac{\text{Number of keys}}{\text{Number of buckets} \times \text{Size of the bucket}} \tag{2.1}$$

In Linear hashing, splitting a node is done in a more organized way than *extendible hashing*. In each round after inserting a key, a check is done in order to decide about splitting a node. Hash table grows linearly in this method. In *extendible hashing*, nodes are split when they overflow. In linear hashing, buckets can be set sequentially, so it is possible to locate a bucket's address based on the starting address of the buckets, so no directory is needed.

T Tree

The structure of a T-Tree is a combination of an AVL tree and a B-Tree. It is a binary tree and needs balancing, similar to an AVL tree. Each node can have multiple keys similar to a B-Tree.

The search in T-Tree is similar to the search in a binary tree, but we have to compare the value with the greatest value and the lowest value in the node.

For insertion, the suitable node for inserting a new value should be found. If there is space in that node, we insert the new value and insertion is done. If there is no place for adding the new value, the minimum value in this node will be removed, and then the insertion of new value is done. Later, the minimum value will be added to the tree. Also in an iteration from the node which has the new value to the leaf, the new value should replace the greatest lower bound value for this node.

In case we could not find any node to insert the new value in, then the insertion is done in the last node on that path, if the value fits into that node, otherwise, a new leaf is added and the new value will be inserted into this node. After adding a new leaf, the tree should be checked for its balance.

Based on the tests that are done in this paper, "AVL trees and arrays do not have sufficiently good performance/storage characteristics for consideration as main memory indices" [3], T-Tree seems the best solution for indexing so far.

2.2.2 KISS-Tree: Small Latch-Free In-Memory Indexing on Modern Architectures

In [14], KISS tree is presented which is a latch free and in-memory index structure for minimizing the number of memory accesses. The structure of a KISS tree is similar to a prefix tree with added compression scheme and virtual memory management.

KISS tree structure has three levels. In prefix tree, the key is divided into groups of bits which have equal sizes, but in KISS tree this is not the case. In KISS tree, we have assumed that keys are 32 bits and the size of each group of bits is different and it depends on the level.

- **Level 1: Virtual level** In this level, we get the address of the node in the next level using the 16 most significant bits in the key. Nodes in the next level should be stored sequentially. Since there are no nodes in this level, this level is called the virtual level.

- **Level 2: On-demand level** The next 10 bits of the key defines the bucket in the node in this level. As mentioned earlier, we would like to have the nodes stored sequentially. On demand allocation is done in this level, so we have allocated large amount of memory in virtual address space, but when we write something into the node, a part of physical memory will be allocated for the node. It makes sense to have the node sizes in this level as $4KB$, because pages are allocated in memory in $4KB$ sizes. Each node in this level has 2^{10} buckets. So the address that should be stored in the bucket should have 4 Bytes. For this reason, compact pointer was introduced to reduce the size of pointers from 64 bits to 32 bits.

- **Level 3: Compressed node level** In this level, 6 bits are left from the key, so the number of buckets inside the nodes is 64 buckets. The compression is done on the nodes in this level. 64 bits is located at the start of each node which defines which buckets are used. We do not have to allocate 64 buckets at once.

Operations

- **Update** Assume we are inserting key 42 into the tree. The node in the next level is identified using the first 16 bits of the key. Then, the next 10 bits of the key are used to identify the bucket inside the node. All the pointers in the node are zero, so the node is not physically allocated in the memory. Then we have to ask for a node from memory to insert the bit map and one value. Afterwards, the bit in the bitmap is set which is defined by the third part of the key, which in this case is 42. The value is written after the bitmap in the node. Finally, the pointer in the second level is set with the compact pointer.

- **Read** For reading key 42 from the tree, the node in the next level is identified through the first 16 bit of the key. The next 10 bits are used to identify the bucket. Since the bucket is not zero, we get the actual pointer from the compact pointer which is stored in the bucket. In the third level, the node is found and we check the 42$^{\text{nd}}$ bit of the bitmap in the node. If this bit is set, we have to get the corresponding value from the node.

Memory Management

At the start, virtual memory is allocated for each of the 64 possible node sizes in the third level. So we can have 2^{26} blocks of memory, where each of these blocks have a part of memory which is allocated for 64 possible node sizes. An action which is defined in this work is called RCU (read-copy-update). Based on this action, sizes of the nodes in the third level can change, so if another bit in the bitmap of a node is going to be set as one, then the value will be added to the node and the size of the node will be changed. In this case, until the new node is created, user can get the value from the old node. When the new node is ready, the address of the new node should be set to the corresponding bucket in the second level.

2.2.3 CTrie

A Ctrie [20, 21], is a type of *hash array mapped trie* [17] which is concurrent and lock-free [16]. The methodology of Ctrie is used in KISS tree [14] implementation. All the operations including insert, remove, and search can be done concurrently in a Ctrie. Based on the methods in *hash array mapped trie*, the hash code of a key is calculated. Each node can have up to 32 fragments. The memory will not be allocated for a node with 32 fragments because some of the fragments can be null pointers. At the start of each node, there is a 32-bit bitmap defining which of the 32 fragments are set. Then an array with the size of all the 1s in the bitmap is allocated. An operation called *compare and swap* (CAS) is done on the node whenever a key is going to be inserted on a node [16]. There are some indirection nodes between each node and its sub levels to make sure that the updates are done in the correct way. In Figure 2.2, the initial picture of the tree and picture of the tree after insertion of some keys are given. Whenever there is a key with the same hash code, at least a level should be added into the tree and of course an indirection node is put in between.

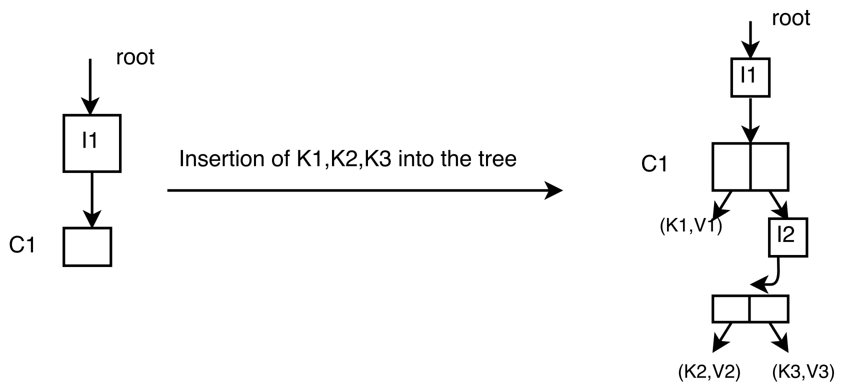


Figure 2.2: C-Trie Insertion

2.2.4 Burst Trie

The Burst trie has been introduced in [22]. In comparison with other structures, this structure is not using more memory than hash tables or other tree structures such as the binary search tree, Splay trees [23], or ternary search trees. In comparison to Splay trees, Burst trie can store 10 GB of vocabulary in 40% less time. Burst trie is more efficient than ternary search tree in memory usage and has a better performance.

Search length in Burst tree structures is higher than hash tables. Burst trie aimed on reducing string comparisons to less than two as well as efficient memory consumption.

The structure of Burst trie contains three main components:

Record has a unique string which acts as a key.

Container has a number of records inside. The data structure used inside a container can be a simple data structure such as a BST or a list.

- Access trie

In Figure 2.3, we can see a burst tree after insertion of "came", "car", "cat", "cave", "cy", "cyan", "we", "went", "were", "west".

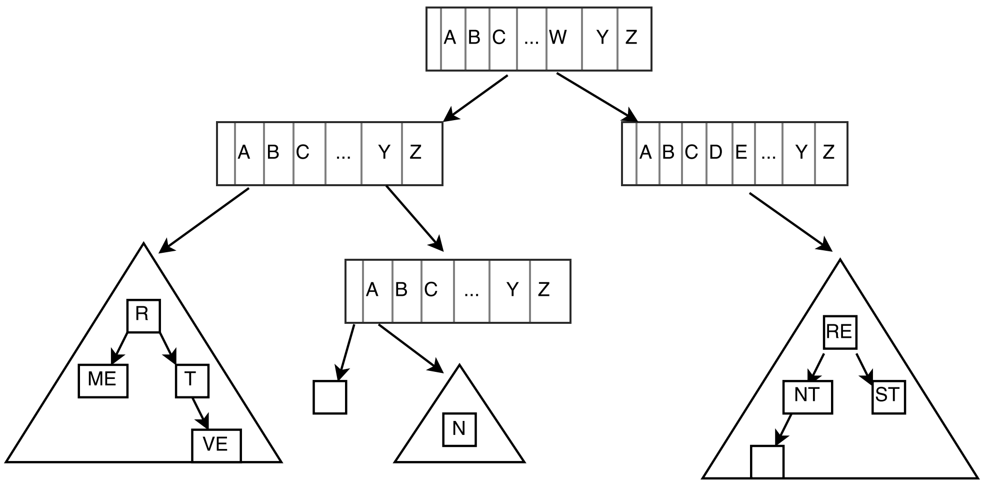


Figure 2.3: Burst trie after insertion of a number of keys

Search

For search in Burst trie, we refer to the root of the trie and find the fragment which has the first character of the word. Then we have to check where this fragment points to. If it points to a node, we have to continue traversing in the same way. At the end if we reach a node which is at level $n + 1$ (n is the number of characters in a word), we refer to the empty string of that node. If fragment points to a container, then we have to search for the remaining characters of the word by doing a binary search within the container.

Insertion

A correct traverse has to be done in order to find the container in which the new key should go. Assume that the target container is at level k. If k is equal to $n + 1$, then the key should be inserted in the empty string of that node. If k is less than n, then the new key should be inserted into the correct container.

Bursting

Bursting happens when a container in level k is replaced by a node and a number of containers which are pointed to from that node. The new containers are located at level $k + 1$. All the records which were in the old container should be pointed to from a fragment in the newly created node.

Experiments done on Burst tries show that they are memory efficient and faster in comparison to Splay trees, binary trees, or tries but still a little less efficient than hash tables.

2.2.5 Online Bit Flip Detection for In-Memory B-Trees on Unreliable Hardware

In this paper [1], B-tree has been chosen for the index structure and various techniques have been introduced for finding bit flips. These techniques can be used in other data structures as well.

Using hardware-based modules has always been an option towards bit flip detection. ECC-DRAM(Error Correction Code-Dynamic RAM) is one of the techniques for detecting bit flips in the memory, though this technique increases memory consumption. The other problem towards this hardware solutions is that they can detect limited number of bit flips. The other technique is TMR (Triple Modular Redundancy). In this technique, there are three replicas of the same data and an algorithm is executed in all the replicas. The results are compared and the final result is decided based on the majority. This method will obviously increase memory consumption and needs redundant computation. Based on this paper, detection of bit flips is accomplished by using the structure of a B-tree and using different levels of redundancy for assuring bit flip detection. In the results, it is concluded that suggested methods can detect more number of bit flip than using the ECC-DRAM for bit flip detection.

In one part of this work, memory modules are heated in order to see the result of heat on them. Based on this experiment, heat is one of the factors in increasing the bit flips in the main memory. The result shows that bit flips increases when heat is increased on the modules. When bit flips are corrected in each run and then experiment is repeated, the result shows less number of bit flips.

In the other part of this work, some methods are introduced in order to detect bit flips in B-trees.

At first, a basic B-tree is implemented. Each node in the B-tree contains of a number of fields. The first field is the pointer to the parent's node. Then L which defines the level of the node in the tree. The field after that is *fill level* which defines the number of (key,value) pairs that are currently in the node. The final fields are the child pointers. Parameter K in the B-Tree limits the number of key-value pairs in a node. Based on the K value, in each node there can be between K and $2K$ keys and $2K + 1$ child pointers at most. K can vary based on the key size. In the basic B-tree, *fill level* is checked in order to be sure that it is in the boundary of 0 and $2K$. No other checks are done in the basic B-tree.

In TMR B-tree, there exists three replicas of the B-tree and each query will be executed on all replicas and the results will be compared with each other. When the key is found based on the majority, the value according to the keys should also match. There is a possibility that the results of the two replicas are same but both results are actually corrupted.

In order to do an online checking on the B-tree to find the bit flips, EDB-trees (Error Detecting B-trees) are introduced, in which the checks are done while traversing the tree and searching for a node. The nodes are kept one after another in the virtual address space, so the start and the end of the area allocated for the tree in virtual memory is predefined. In case of a pointer corruption where a pointer points to an address out of this boundary, the corruption can be detected. There are some pointer checks that are done while traversing the tree. First, since the node size is fixed to 4KB, the pointers can be checked and the first 12 bits of them should be zero. In second, pointers should point to addresses in the specified area. Third, since the parent address is kept in the child, this address can be compared with the address of the node that we have descended from.

Based on these sanity checks, only some percentage of bit flips can be detected, while there is no check towards the pointers and key-value pairs inside the nodes.

The other type of EDB-tree for bit flip detection is EDB-tree PB. In this type of tree, parity bits are used in order to detect bit flips. For all the node fields, a parity bit has been calculated and put in the most significant bit. In each traversal, node members' parity bits are checked. The disadvantage of this implementation is that since one bit of each member is used for parity bit, the number of possible values for each member will decrease.

The other type of EDB-tree for bit flip detection is EDB-tree CS. For each node, 4 checksums have been added. One of them is the XOR result of the parent pointer, fill level, and tree level. The second one is the XOR result of all the keys. The third one is the XOR result of all the values, and the fourth one is the XOR result of all the pointers. For detecting more corrupted elements, it is better to have checksum for each field in a node, but of course this will cause more memory consumption and higher computational costs. Each of the checksums are validated at different times. Whenever a key or a value is added to the tree, the second and the third checksums should be checked. While traversing the tree, the fourth checksum is checked.

Based on the tests done in this paper, while injecting 1-5 bit flips per 8-Byte word per millisecond in a period of 300 seconds, the results on different types of implemented B-trees is as follows:

In the basic B-tree implementation, since no data correction is done, the number of errors increase with time. When the number of injected bit flips is 5, EDB-tree CS and EDB-tree PB can detect more errors. While injecting 2 or 4 bit flips, error detection of B-tree and EDB-tree PB is almost the same, because even number of bit flips cancel each other. In terms of undetected errors, the result shows that in TMR B-trees, the number of undetected errors is zero.

2.2.6 Efficient Verification of B-tree Integrity

In [4], a number of algorithms for B-tree validation are introduced.

In this paper, the verification of B-trees has been divided into groups. We should verify the relation between nodes and all of the pointers (which can be the child pointers and the pointers between siblings). In-page verification is another part of the verification. There is also verification introduced for the relations between the tables.

In-Page Verification

What matters in this verification is the verification of information in a single node. It can be done by using a physical test or a checksum. Torn-page is the case when a node needs to access multiple sectors but the access is not possible. The detection of torn-pages can be done quite fast.

Index Navigation

In this part, verification of the whole B-tree has been taken into account, so the pointers between the nodes are checked to assure that sibling nodes are correctly pointing to each other. The order of keys inside the nodes should also be verified. For doing these checks, a breadth-first or depth-first traversal can be done in the tree.

Aggregation of Facts

When data pages are read and information is extracted, verification is not done immediately. Instead, facts are extracted and streamed into a matching algorithm, as an example, one fact says that “a leaf node on disk page 5 points to a successor leaf node on disk page 92.”. When a matching fact like “a leaf node on disk page 92 points to a predecessor leaf node on disk page 5,” is found, then verification for these two facts is successful. At the end of the entire matching operation, the B-tree is consistent if and only if all facts have been matched.

The crucial component of this approach is in the selection and representation of facts which are extracted from B-tree pages and are matched in the verification step. For the chain of neighbours, the fact “page x follows page y" is extracted from both pages x and y. For the parent-child relationship, "parent x points to child y for key range [a, b)" is extracted from parents and children. These matches also verify the level of the two pages (leaf pages are level 0) and permit the appropriate flexibility in matching separator keys in the parent and actual keys in the child.

2.2.7 Efficient In-Memory Indexing with Generalized Prefix Trees

In this paper [2], a new type of prefix trees is introduced.

It is mentioned that using trees and hash-based techniques are common ways for update-intensive memory indexing. Using tries is another option but can cause high memory consumption in relevance to the other two structures. Some changes are done in the existing trie structure to make tries less memory consumable. Based on this new structure, different data types with different sizes can be indexed. Furthermore, in order to get a value from the tree, a strategy has been introduced, so that we do not traverse through all the levels in the tree. Another advantage of this structure is to decrease the number of created nodes.

Based on the analysis of existing solutions in this paper, a number of disadvantages have been given for each in-memory index structure. In *sorted arrays*, the disadvantage is that for each update, records should always be sorted. The solution might be to have some gaps between records, but this will cause high memory consumption. In *Tree-based structures*, we can have balanced and unbalanced structures like binary trees. Different number of trees has already been used for in-memory indexing, including: *B-trees, B^+-trees, red-black trees, AVL-trees and T-trees.* All the balanced trees have the $\mathcal{O}(\log N)$ in all operations, where N is the number of records. In *Hash-based structure*, a hash function is used in order to define the place of a key in hash table. If chained-bucket hashing is used, in the worst-case, the complexity will be $\mathcal{O}(N)$. The size of the hash table is also fixed. In other approaches like *linear hashing, modified linear hashing and extendible hashing*, place of the key is defined dynamically. Re-hashing in this approach can have the $\mathcal{O}(N)$. In *trie-based structure*, like prefix trees, a key is divided into some parts. These parts will lead us in the traverse of the tree, in order to find the correct path to the destination. The time complexity in trie-based structure is $\mathcal{O}(k)$, which k is the number of bits in a key. The number of accessed nodes in trie-based structures is independent from the number of records.

Based on the analyses of all the index-structures that are mentioned, another type of index-structure which is a modified version of trie, is introduced. The main concepts of this structure is as follows:

1. Using prefixes with different sizes.

2. In case, there are leading zeros in the prefix of an index, a bypass structure has been implemented to avoid traversing through the tree level by level.

3. A dynamic way for trie expansion.

In generalized trie, prefix size can be different, but prefix size must be a divisor of the key length.

Two properties have been mentioned for this structure.

1. Determinism: There exists only one path for each key in the tree.

2. Worst-case Complexity: If we have a key with size K and prefix size K' , then time complexity is $\mathcal{O}(h)$, where h is height of the tree and $h = K/K'$.

The problems that are seen in this trie structure is with *trie height* and *memory consumption.*

If there is a key with size *varchar(256)* and prefix size 4, then the height of the trie will be 512. So if we have keys with smaller sizes than 256, they have to be padded with zero, so for each iteration we have to go down the tree by traversing 512 nodes. This will cause an increase in memory consumption, memory access and decrease in performance. Plus this fact, if an insertion is going to be done and none of the current prefixes can be reused, then in the worst case, we have to create *h-1* nodes (where h is the height of the tree). The solutions that have been given are *bypass jumper* and *dynamic trie expansion.*

- Bypass jumper: If there exists some nodes which are connected to each other through the 0 fragment in the node, we can keep the address of these nodes in an array. In bypass jumper array, an array is formed which contains pointers to all of the mentioned nodes. By using this array, we do not need to traverse h nodes, but we can jump directly to a node that its k*th* fragment is set where k is not zero.

- Dynamic trie expansion: without using this technique, it is assumed that the trie height is fixed and all the records are kept in the leaf level, so creating a record might lead into creation of h-1 nodes which causes a large memory consumption. The idea is to not to create extra nodes until it is necessary. In this approach we can have records in levels other than just leaf. In order to figure out which fragment in the node is pointing to a record and which of them is pointing to a node, one bit is allocated as a flag bit for each of the pointers in a node. Since there are 16 fragments in each node, two bytes are added at the end of each node.

2.3 Summary

Some of the ideas in this master thesis is taken from the related works. Until now, based on my knowledge, there is no work which is done in order to find bit flips in prefix tree index-structure. From the time that having index structures in memory has become applicable, detecting bit flips in them has turned into an important issue. Existing solutions for bit flip detection can be software-based, OS-based or hardware-based. Finding software-based solutions for bit flip detection is important because hardware-solutions have some limitations in the number of bit flips they can detect and also can cause high memory consumption. Software-based solutions are already given on some index-structures but no solution is given until now for bit flip detection in prefix trees.

By getting ideas from the mentioned related works, some methodologies are given for reaching our goal.

Chapter 3

Methodologies

In this chapter, the methods which are used in this master thesis in order to find bit flips in prefix tree index structure are going to be explained. For each of the methodologies, a brief result is given.

Detecting bit flips in memory is a crucial issue. While keeping database index structures in the memory, the occurrence of bit flips can cause wrong results when we try to get data from the database. There are different solutions that are given for detecting bit flips in different index-structures. Using parity bits or checksums is one of the solutions for bit flip detection.

By using parity bits or checksums, bit flips can be detected, but this leads to higher memory consumption. If we use parity bit for detection, usually we need one bit per byte of data for storing the parity bit. If redundancy bits are needed for the parity bit, then we will need more bits. By using checksum, more bits might be needed.

We would like to find the most efficient way for bit flip detection in prefix trees. For this purpose, some methods are used and evaluated. Using parity bits is one solution. The other solution is to use the structure details of the tree itself to detect bit flips and avoid using redundant data. At the starting point of this chapter, an explanation of prefix tree is given and some details of the prefix tree which is implemented in this thesis is provided. The structure of this master thesis is shown in figure 3.1.

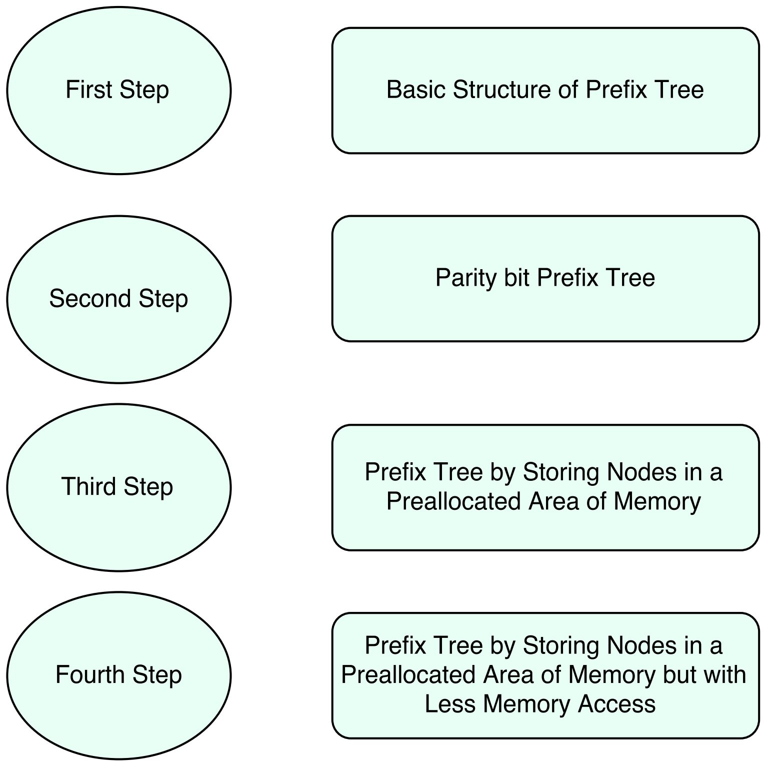

First Step
Basic Structure of Prefix Tree
Second Step
Parity bit Prefix Tree
Third Step
Prefix Tree by Storing Nodes in a Preallocated Area of Memory
Fourth Step
Prefix Tree by Storing Nodes in a Preallocated Area of Memory but with Less Memory Access

Figure 3.1: Overall structure of the thesis

3.1 Basic Structure of Prefix Tree

The index structure that is used in this thesis is Prefix tree. At the start, the key size is assumed to be 32 bytes and the prefix size is 4, so each node can have 16 pointers to 16 children. An initial implementation is done in order to build the basic structure of the tree. So $k = 32$ and $p = 4$. Based on the given assumptions:

- The height of the tree is $\frac{k}{p} = 8$.
- For finding a key-value pair inside the tree, we have to traverse through 8 and only 8 nodes including root.
- Each key is divided into 8 groups of 4 bits. The value of each group can be from 0 to 15. For example if a key is equal to 200, then:

```
0000 0000 0000 0000 0000 0000 1100 1000
 0     0    0    0     0    0   12    8
```

- Based on the value of each of these groups, we can identify the way through the tree to find the correct value. In the figure 3.2, it is shown to how to find the value according to key 200. We assume that key,value pair(200,12) is already added to the tree.

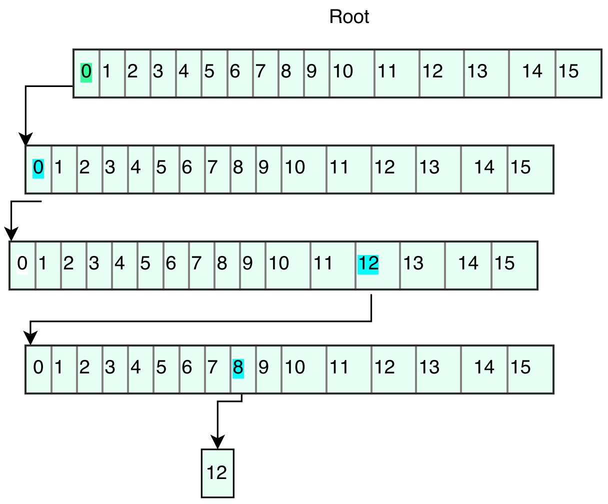


Figure 3.2: Traverse the tree for finding value of key 200

Based on the characteristics that were explained, the structure of a node in the basic prefix tree has only 16 pointers. So for inserting a key-value pair into the tree, prefixes of the key will be calculated and based on the prefixes, we will traverse in the tree. If a node does not exist, we will create the node. Values are connected to the leaf nodes. So if we have inserted the same keys with two different values, values are connected to the leaf via a linked list.

3.2 Prefix Tree with Parity Bits

After building the tree, the main concept that should be taken care of in this thesis is to find the bit flips in the structure.

In the first solution, parity bits is used. As we have already explained, structure of the nodes contains of only pointers, so pointers in the nodes should be protected against bit flips otherwise a wrong value can be retrieved or we might get segmentation fault. For this purpose, parity bit has been calculated for each pointer in the node and 3 more bits is dedicated for redundancy of the parity bit because there is a chance that parity bit becomes corrupted itself.

While inserting keys into the tree, if a new node should be created, its address should be put in the corresponding fragment of the upper layer's node. Parity bit is calculated for that pointer in the fragment and it is set at the end of the node in the right place. After that, three more redundant bits of the calculated parity bit are also set. When a key-value is requested from the tree, during the traverse to reach the target node, parity bit for each of the pointers in this path is calculated and if 3 of the 4 parity bits which had been assigned for the pointer, matches the calculated parity bit, we assume that no bit flip has happened in the pointer otherwise a bit flip is reported.

For inserting bit flips in the tree, we have created bit flips in a certain number of nodes (number of keys to be corrupted is asked from the user) which all of them are in the leaf level. We assume that all the inserted keys into the tree are distinct, so each of the pointers in the nodes in leaf level is definitely related to one specific key-value. So by inserting bit flips in the pointers on last level, we can guarantee that each of the corruptions is related to one and only one key and the number of overall corrupted keys is easy to get. If bit flips would have inserted in the medium levels, then detecting the number of corrupted keys would have not been easy, because multiple keys can descend from a pointer in a node and this would cause difficulty in counting the number of corrupted keys.

Number of bit flips added in a pointer is 3. Parity bits causes high memory consumption since 8 bytes should be allocated for each node for parity bits. The other problem with parity bit is that in case of multiple bit flips happening in a pointer, parity bit is not a good detection mechanism because only odd number of bit flips can be detected by it.

3.3 New Layout for Node Storage in Prefix Tree

This method is implemented for 16 bit keys.

Because of the mentioned reasons in the last level, the second solution is introduced which does not need redundant information on the nodes.

Based on this solution, bit flips are detected by using the layout of the prefix tree. We keep the nodes in a contiguous area inside the memory, so in case bit flip happens in the pointers, it can be detected by knowing the exact address of the following node extracted from the layout (node size is fixed).

Structure of the tree depends on the inserted keys into it. Based on a set of keys, we might have a tree which is very disperse and has a few nodes and based on another set of keys, tree can be dense. Memory is allocated for a full tree at the start. Memory is allocated in partitions where each partition has the size of a memory page (page size is 4KB). Each page can contain 32 nodes, so it is better to have a dense tree in order to gain the best from this memory allocation. In case we have a disperse tree, a lot of memory will be wasted. If we have a reasonable dense tree, this approach is beneficial. So we will assume that we are going to have a dense tree and we will pre allocate the memory for a full tree.

Based on the initial explanation on this method, the implementation has 4 steps:

1. Finding out the density of the tree or insertion of which keys results in a dense tree.
2. Reserving memory for a full tree and creation of the tree with a set of keys which make the tree dense.
3. Insertion of bit flips in the nodes.
4. Evaluation of this method towards finding the bit flips.

3.3.1 Density of Tree

Finding the density of the tree is done on our last implementation of prefix tree, because both of the structures are the same, we used the last method for getting the density and finding out which set of keys will produce a denser tree.

Density of the tree is not a fixed value and it changes based on the inserted keys. We build the tree based on different set of keys. Different set of distributions of keys have been used for this purpose. The number of pointers in each node which are not null is called *node degree*. *Node degree* has been counted and if the average of this value among all the nodes is more than a defined value, we assume that the tree is dense enough. The distributions that are used for this purpose is as follows:

- Simple distribution (in which, keys are multiples of three)
- unifrom int distribution
- Poisson distribution

Details of the implementation and the key distributions which are used are explained in the next chapter. As the result, using the simple distribution leads to a denser tree after insertion of less number of keys. (Implementation of this part is based on 16 bit keys)

3.3.2 Creation of Tree

One thing to mention at the start of this section is that, this tree is going to be created when we have 16 bit keys.

Until now, we have chosen a set of keys to insert into the tree in order to have a dense tree. For memory allocation, since we have a large number of nodes, we can make the nodes into groups. We will have four groups of nodes where each group is equivalent to each level in the tree. So we will keep the address of the root level, first, second and third level in four variables. Second level is divided into 8 blocks where each block has the size of 512*8 bytes (which is equal to a page size). For the third level (which is the last level), 128 blocks of 512*8 is allocated. All the allocations are done using *calloc* in c.

After each insertion into the tree, created nodes will be put in the places of memory which is preallocated for them. If we need to insert four nodes for insertion of a key (this implementation is done on 16 bit keys), then the address of the second node should be calculated and put in the correct fragment in the first node and the address of the third node should be put in the correct fragment in the second node and finally the address of the fourth node should be put in the right fragment in node three. For getting a key from the tree, we can easily traverse to the correct node. The details of the implementation can be found in the next chapter.

3.3.3 Insertion of Bit Flips

After creation of the tree, we will insert some bit flips in a number of nodes. After this insertion, we try to retrieve the keys. Since we know the exact address of the nodes, so whenever we want to get a key, we can go through the nodes by checking the address which is stored in the corresponding fragment of the node. We can calculate where the next node should be and compare it with what is written in the previous node. If these two values are the same, then we will keep traversing to the next node, otherwise a bit flip has been detected.

Insertion of bit flips is done on one or three of the bits of some pointers in the third level. Like what was explained in the last method, having the bit flips in the last level will make counting the bit flips easier, but in this method, we have the corruptions in the middle levels as well. We have kept the address of detected corruptions in an array and whenever another corruption was detected, we could compare the current address with the addresses in the array. If the current address was in the array, then we would not count that bit flip again, otherwise that bit flip is counted and address of the corrupted place is added to the array.

Insertion of bit flips is done by doing XOR bit wise on the pointer with a value that has one or three 1 bits in it and the rest of the bits are 0. In this case, just the bits which are XORed with 1 are flipped.

3.3.4 Evaluation

In this method, all the bit flips according to the pointers are detected, but for the values which will sit in the leaf level, no bit flip resiliency technique is used. Parity bit can be used for the values.

Based on this methodology, all the bit flips in the pointers were detected but as we said before, the only advantage of this method to the last method is that, we do not need to use parity bits. This will reduce the node size, but this solution will lead to high memory consumption when tree is disperse. One problem with this method is that for inserting a key into the tree or getting a key from the tree, a traverse to the leaf level must be done. In this way, we have high memory usage and memory access. In the next approach, this problem is addressed.

3.4 Structure of Prefix Tree with Less Memory Access

3.4.1 Structure of Tree

Another approach is shown for dealing with the high memory access of the last methods. As a brief explanation,in this method, we avoid traversing to the leaf node for getting a value all the time. In this method, values can be kept in any level in the tree, this will lead to low memory access.

In this method, we have reserved memory for a full tree and some additional data in virtual memory using *mmap*, but allocation of a node in physical memory will happen only if there are more than one key descending from a fragment in a node, otherwise, a key-value pair will be stored in physical memory instead of a node.

In this approach, virtual memory is reserved in a number of particles. Reserving memory for each level is done by using mmap and starting address of the reserved place is kept in a variable. So we have all the starting addresses of levels. When reserving memory via mmap, we are not allocating any memory from physical memory, but when we set a value to a part of the reserved memory, then operating system will allocate the size of the mmapped area in the physical memory. Since the mmapped area is large, memory consumption is still an issue in this approach but a better solution for this problem is given in the last chapter. This approach has two objectives:

- How to avoid creating 8 nodes whenever a key is inserted and to avoid keeping the values on only the leaf level.
- How to detect bit flips for this structure.

In this method, some ideas are taken from *generalized trie* [2]. Based on the ideas which are taken from generalized trie, allocating a complete node, whenever we insert a key into the tree is highly memory consuming. In case there is only one child descending from a fragment in a node, we do not need to create a whole node as child of that node. Instead, we can point to a key,value pair. If we have 32 bit keys and 32 bit values, each of these key,value pairs need 8 bytes of the memory while a whole node can take 16*8 bytes (if we have 64 bit pointers).

The other optimization is in the size of fragments in each node. Pointers which are stored in each node can be 32 bit pointers instead of 64 bits, because we can just store the offset other than the whole address since we have the starting address of levels. With this optimization, node size will decrease to half. In order to locate the descending nodes, we can add the offset with the address of the level.

One of the important points in this method is reserving memory for the tree structure. As we said before, for all the levels except of leaf level, one mmap is done on each of them. For the leaf level, because of mmap limitations, reserving memory for the whole level was not possible, so mmap is done 8 times on the leaf level. Based on the information given, mmap function is called 15 times. This reservation is done before inserting any key and creating nodes in the tree.

At the moment we assume that we have 32 bit keys with prefix size 4. Then we will have 8 levels. We could have had prefix size 2 but the number of levels would have increased a lot.

The number of key,value pairs in a level is the same as the number of all the nodes in the same level. For each node in the tree, we need one bit to show whether the node is created or not. If the bit is 1 then we assume that the node is created, otherwise a key-value pair is set instead of the node.

We have allocated memory for all the nodes in a level and also for all the key-value pairs and flag bytes. Based on a calculation which is done, from all the memory which is allocated for a full tree, 11 percent of that is needed for all the key-value pairs and flag bytes which is a reasonable amount.

Based on the give information till now, structure of tree in each level is as follows:

- Root level has one node with 16 pointers. Each pointer is 32 bits.
- First level has 16 nodes. At the end of the level, we have 2 bytes which

flag bits are kept in it (one bit for each node). After that, we have $16 * 8$ bytes. Each of these 8 bytes is for storing one key-value pair.

- Second level has 256 nodes. At the end of the level, we have 32 bytes to store the flag bits and $256 * 8$ bytes for storing the key-value pairs.
- Third, fourth, fifth and sixth level will have $256 * 16$, $256 * 256$, $256 * 256 * 16$ and $256 * 256 * 256$ nodes accordingly. At the end of each of these levels, we will have enough bytes to store the flag bytes and key-value pairs.
- Seventh level will take up a lot of memory, so it has been divided into sub levels. We will have 8 sub levels. Each sub level will have $256 * 256 * 256 * 2$ nodes. At the end of each sub level, flag bytes are stored which take $256 * 256 * 64$ bytes. Starting address of each of these sub levels are kept in variables. At the end of these sub levels, key-value pairs for all the nodes in these sub levels are stored. Since we have $256*256*256*16$ nodes in the seventh level, we need $256*256*256*16*8$ bytes to store these pairs. Starting address to this part is also kept in a variable.

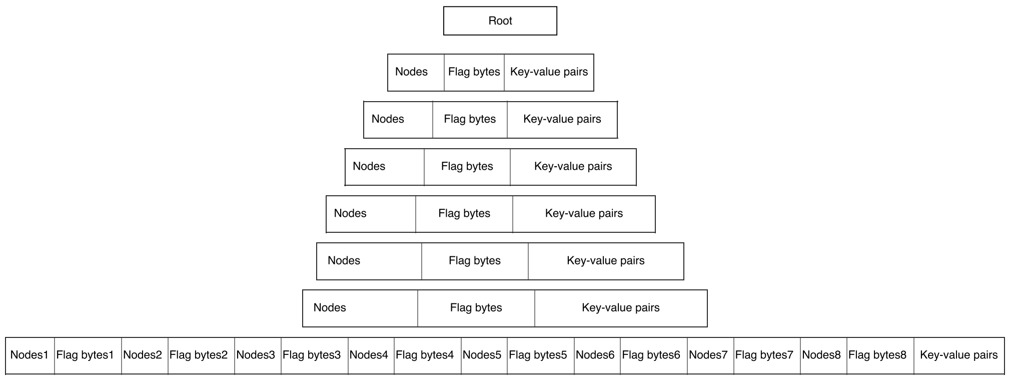


Figure 3.3: Tree structure

3.4.2 Insertion in Tree

Figure3.4, shows the flow of key insertion into the tree.

The first step for insertion of a key into the tree is to check if the key is the only key which is going to be descended from the corresponding fragment in the node or not (this can be done by checking the value stored in the corresponding bucket, if it is 0, then this key is the first key), then we can set the pointer in this fragment in a way that it points to a place of memory which holds a key,value pair. We know that the key-value pair is stored at the end of each level and after the flag bytes. So for finding the correct address to put the key-value pair, we have to add the number of bytes which is reserved for storing all the nodes in the level and the number of flag bytes and also the correct offset for storing the key-value pair. We put this address in the bucket. Having zero in the bucket can also mean that this bucket is pointing to the first node in the next level. So we have to distinguish between these two conditions. We can do this by checking the flag bit of the first node. If flag bit is set to zero, then we are sure that node is not set and zero in the bucket meant null, otherwise if the flag bit is one, then we confirm that node is set.

If bucket does not have a null value, then we have two conditions.
If we check the value in a bucket in the root, we should see if this value is more than *starting address of firstlevel*$+16*16*4$ (we have 16 nodes in first level. Each of the nodes have 16 buckets and size of each bucket is 4 bytes). In this case, we know that the pointer in that bucket, is already pointing to a key,value pair. So with inserting the new key, we should create a new node and insert the key which was stored in the key-value pair area in the tree and also set the value in the bucket to the address of newly created node. We should free the memory which was allocated for the key-value pair. If the value in the bucket is less than *starting address of firstlevel*$+16*16*4$, then we conclude that a node has already been created, so there must have been multiple children from that bucket in the root. Then we should move traversal to the next level in the tree and do the same check for the next node in the traverse and insert the value in the right place in the tree.

When we reach the sixth level in the tree, checks will be a little different. In the seventh level (the leaf level), we have 8 sub levels as it was explained before. Based on *key1* (which is the value of the least 4 bits of the key), we can define the sub level in which we want to to look for the value. As we said before, flag bytes of each sub level are stored at the end of the sub level. So we should check the flag bit of the corresponding node. If it is one, then we can continue our traversal in the tree and jump to the seventh level. Otherwise, we can look for the key-value pair at the end of the seventh

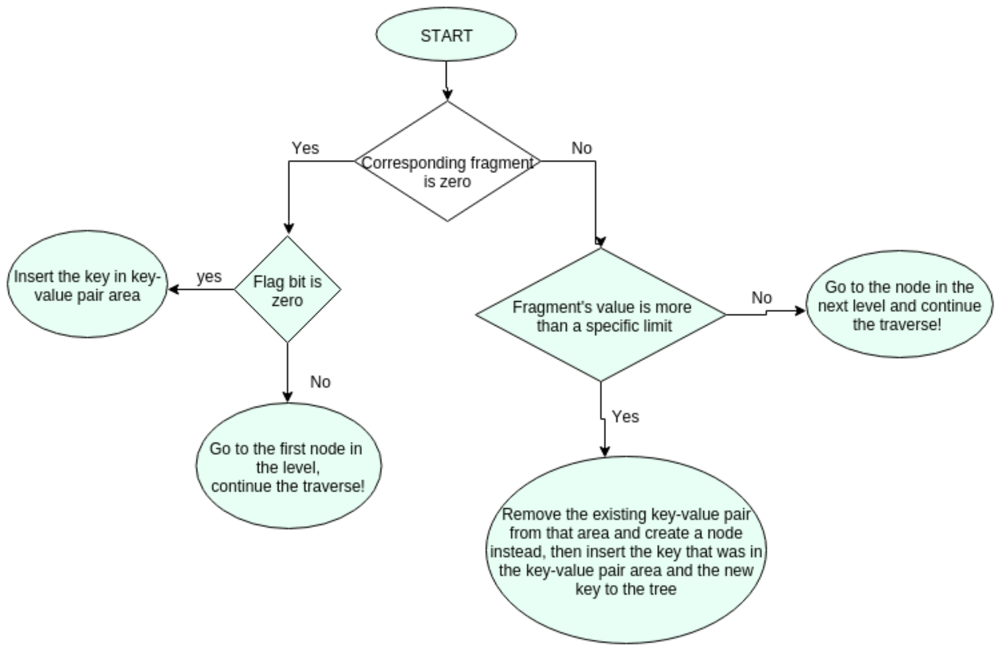

START
Corresponding fragment is zero
Yes
No
Flag bit is zero
yes
Insert the key in key-value pair area
No
Go to the first node in the level, continue the traverse!
Fragment's value is more than a specific limit
No
Go to the node in the next level and continue the traverse!
Yes
Remove the existing key-value pair from that area and create a node instead, then insert the key that was in the key-value pair area and the new key to the tree

Figure 3.4: Insertion of keys into the tree

level. We have already kept the starting address of the area which points to key-value pairs in seventh level. So with some calculations, we can get the correct key-value pair.

Density of the nodes is also calculated when keys are inserted into the tree. More details on this part is given in the implementation chapter.

3.4.3 Insertion of Bit Flips

Insertion of bit flips in the tree is done in different ways.

- Insertion of bit flips in all the levels in the tree in certain bytes.
- Insertion of bit flips in certain number of levels in the tree based on the number of created nodes in the tree.
- Insertion of bit flips on the flag byte, key-value pair or the pointer of the already inserted keys to the tree, in order to be sure that all the bit flips are done on the bytes which are already set.

The methods above have some advantages and disadvantages which the details of them will be covered in the next chapter.

3.4.4 Value Retrieval from Tree

We have considered that bit flips can happen in the pointers, flag bytes or key-value pairs in each level. While searching for a key inside the tree, we check the pointer value in the bucket of a node. We can locate the corresponding node in the next level based on values of *key1*, *key2*, *key3*, *key4*, *key5*, *key6*, *key7* and *key8*. First of all, we should find the corresponding flag bit for the node. Now we have two conditions:

- If flag bit is zero, then we should compare the pointer stored in the bucket with the expected place of storing the key-value pair. If these two are the same, then we should check the key which is stored in the key-value pair with the actual key. If these two are also the same, then we can retrieve the value. We can do another check here. We have the flag bit set to zero, but a bit flip can have possibly happened in this bit, so maybe the actual value of flag bit had been 1. In this case, we check the value stored in the bucket with the address of the expected node in the lower level. In case of equality of these two values, we conclude that flag bit was actually corrupted. If none of the mentioned cases was true, then a bit flip must have happened in the pointer in the bucket.

- If flag bit is set to one, then we should check the value stored in the bucket with the address of the corresponding node in the next level. If these two values are the same, then we can traverse to the next level in the tree to find the value. Otherwise, we assume that a bit flip has happened in the flag bit and it is actually 0 and not 1. So we should check the value in the bucket with the expected address to store the key-value pair. If these two values are the same, then we should check the key which is stored in the key-value pair with the actual key, if these two are also the same, then we retrieve the value and assume that a bit flip has happened in the flag bit.

Figure 3.5 shows a flowchart for details of key retrieval from the tree.

Based on the conditions which were discussed, this solution is bit flip resilient towards the bit flips which happens in the pointers and the flag

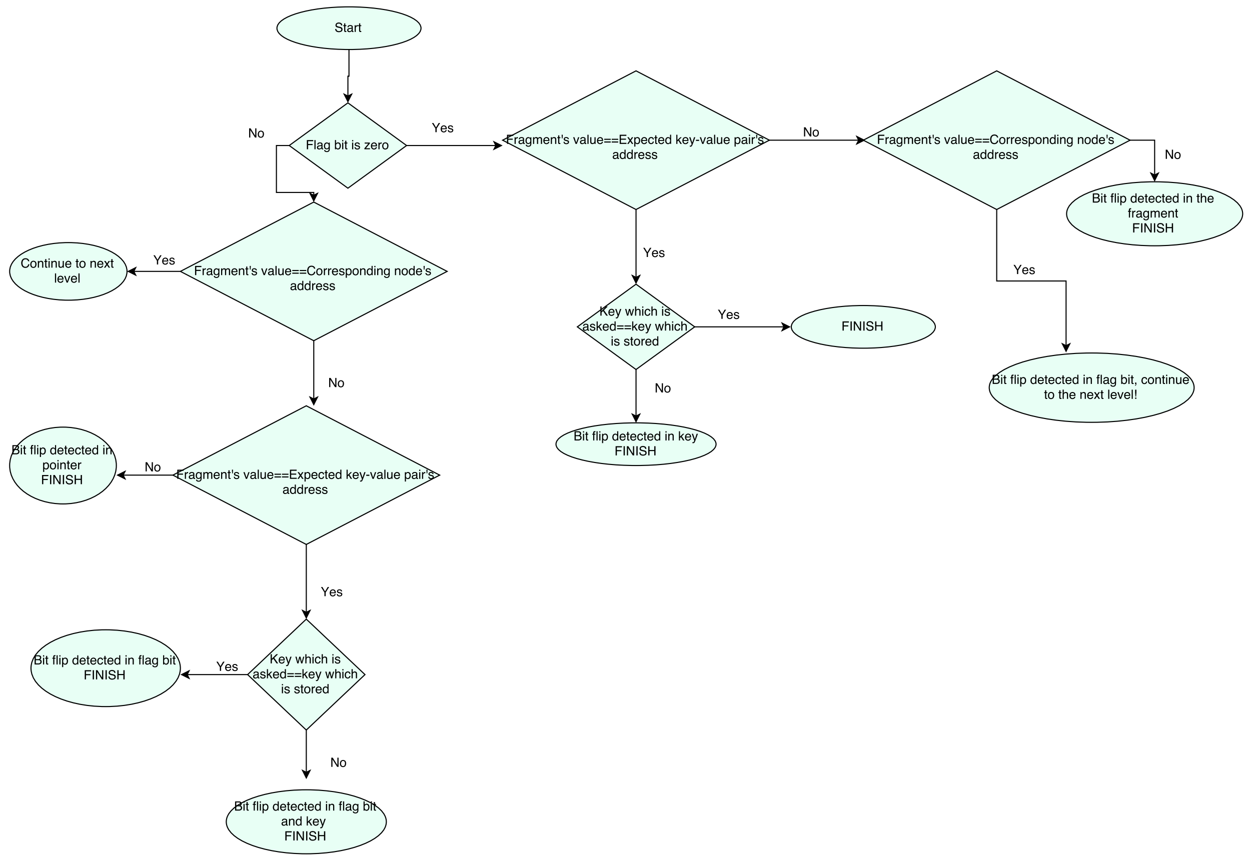

Start
Flag bit is zero
Yes
No
Fragment's value==Expected key-value pair's address
No
Fragment's value==Corresponding node's address
No
Bit flip detected in the fragment FINISH
Yes
Bit flip detected in flag bit, continue to the next level!
Yes
Key which is asked==key which is stored
Yes
FINISH
No
Bit flip detected in key FINISH
Continue to next level
Yes
Fragment's value==Corresponding node's address
No
Bit flip detected in pointer FINISH
No
Fragment's value==Expected key-value pair's address
Yes
Bit flip detected in flag bit FINISH
Yes
Key which is asked==key which is stored
No
Bit flip detected in flag bit and key FINISH

Figure 3.5: Value retrieval from tree

bytes and also the keys. The only bit flip that can not be detected in this method is the bit flip in the value. This can be detected by using parity bits but it has not implemented in this thesis.

3.5 Summary

In this chapter, we covered the details of the methodologies which are used in this thesis. We have used three methods in this thesis. In this chapter, we discussed why did we try these methods one after another. The goals of each of the methods are also discussed. In the next chapter, we cover the details of the implementation of these methods.

Chapter 4

Implementation

Based on the methodologies that has been given in the last chapter, the implementation of them has been explained in details in this chapter.

4.1 Implementation of Basic Prefix Tree

In order to start the implementation part, the first step is to make the initial structure of the prefix tree. It has been assumed that all the keys which are inserted into the tree are 32 bit keys. Prefix length is 4. So at the start, for doing insert operation or get operation, we would like to get the value of the prefixes. For this reason, in four iterations, we get the value of the least 4 bits of the key by doing AND operation between the key and $0000000F$. In each iteration, we have to do right shift on the key for four bits and do the AND operation again. So in each iteration, after getting the prefix value, we keep it in *key* variable. *Key*'s value must be between 0 to 15. In the insert operation and in the first iteration, we have to check the pointer value which is set in the corresponding fragment of the root. If this value is null, then we have to create a new node and set the address of the created node in the correct fragment of the root. New node is created by using malloc function in c. The memory allocated for each node is 16*8 bytes. So the current node is the newly created node. In the next iteration, we will check the fragment's value in the current node and if it is null, we have to do the same action and create a node and set the corresponding fragment of the current node as the address of the newly created node. If node's fragment is not null, then we will just set the current node as the child node and jump to the next iteration. This process will go on for eight iterations. In the last iteration, if the fragment is null, we have to set the fragment to the start of a linked list. If fragment is not null, it means that a linked list is already created and we can just add the value to the end of the linked list.

When we want to get a value from the tree, we have to do the same iterations, but in case, one of the iterations leads to a null pointer, then we assume that key,value pair does not exist.

As it was already said, all the values are inserted into the linked lists which are connected to the fragments in the leaf level. For getting a value from the tree, we have to traverse to the leaf level every time which would lead to high memory access.

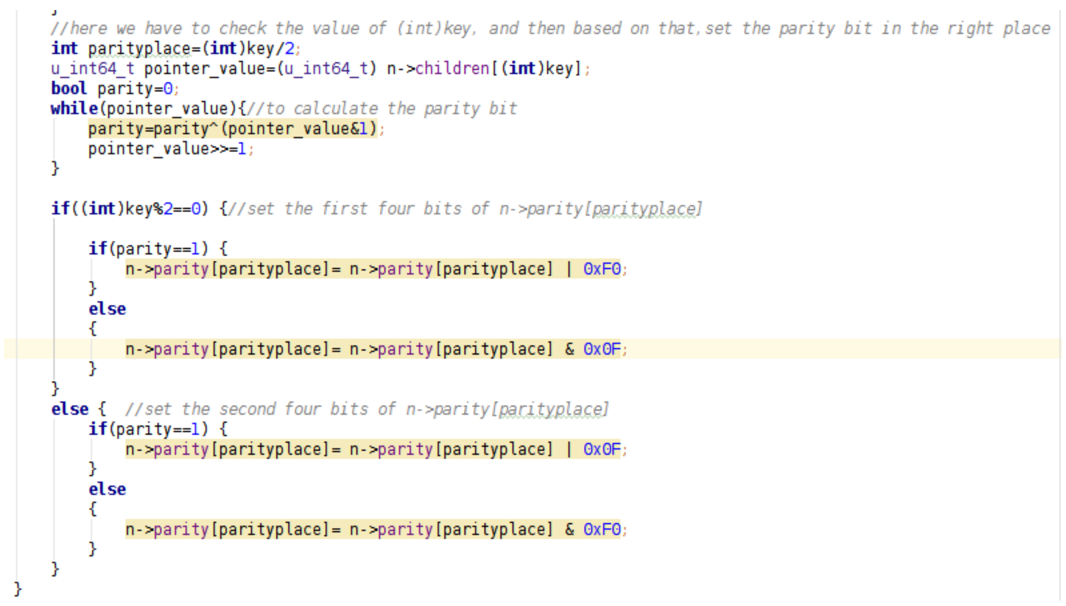
4.2 Prefix Tree with Parity Bits

As discussed, we would like to try different bit flip detection mechanisms. Using parity bits is one of them. In prefix tree, since each node contains only pointers, we should regard the bit flips that happens in the pointers. Here, we calculate parity bit for each pointer and keep 3 redundant bits for that parity bit, in case the parity bit becomes corrupted itself. So by keeping 4 bits extra data for each pointer, we will have $16 * 4bits$ (which is equal to 8 bytes) redundant data for each node. So we will need 136 byte memory for each node:

$$8\text{byte} * 16 + 8\text{byte} = 136\text{byte} \tag{4.1}$$

In the implementation of prefix tree with parity bits, parity bit will be calculated for pointers, while inserting a key into the tree.

Referring to the same example in figure 3.2, while inserting key-value pair (200,12), at the start, a node will be created as a child of root which will be pointed to by fragment 0 in the root. So in 0th fragment in the root, address of the child node will be inserted. Parity bit for this address will be calculated and be put in the correct place in the root. Then we traverse to the created node and the same action will be done. We will continue the rest of the path and we will create the rest of the nodes till the leaf level and set



```
    //here we have to check the value of (int)key, and then based on that,set the parity bit in the right place
    int parityplace=(int)key/2;
    u_int64_t pointer_value=(u_int64_t) n->children[(int)key];
    bool parity=0;
    while(pointer_value){//to calculate the parity bit
        parity=parity^(pointer_value&1);
        pointer_value>>=1;
    }

    if((int)key%2==0) {//set the first four bits of n->parity[parityplace]

        if(parity==1) {
            n->parity[parityplace]= n->parity[parityplace] | 0xF0;
        }
        else
        {
            n->parity[parityplace]= n->parity[parityplace] & 0x0F;
        }
    }
    else {  //set the second four bits of n->parity[parityplace]
        if(parity==1) {
            n->parity[parityplace]= n->parity[parityplace] | 0x0F;
        }
        else
        {
            n->parity[parityplace]= n->parity[parityplace] & 0xF0;
        }
    }
}
```

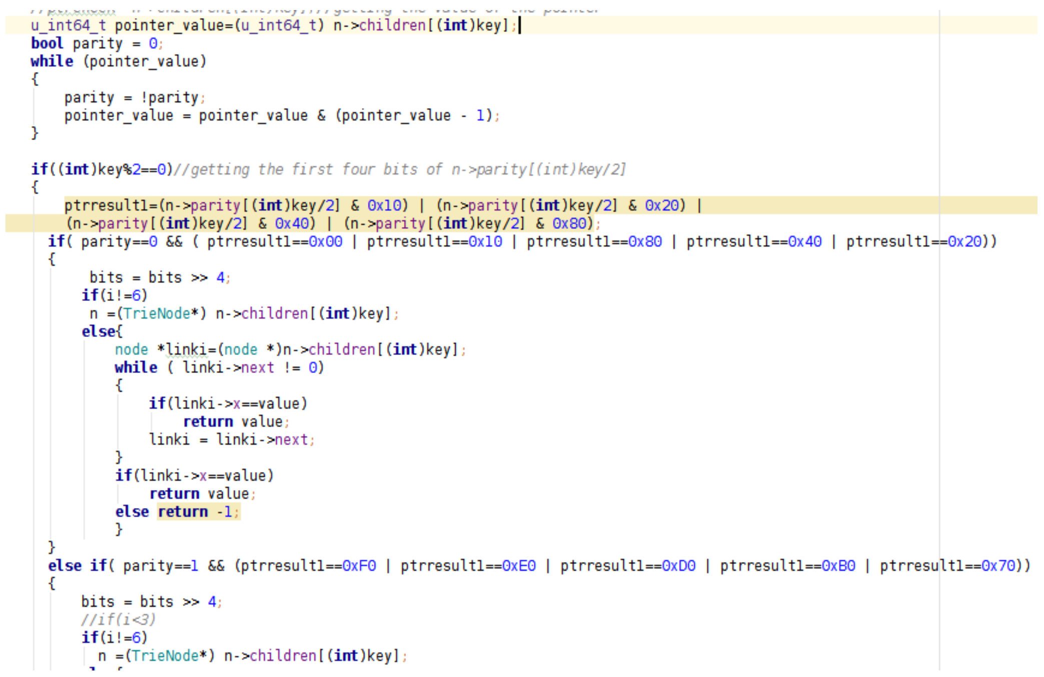
Figure 4.1: Setting parity bits for the pointers

the parity bit for the corresponding pointers. While reaching the leaf node, the value according to the key 200 will be inserted in the linked list pointed to by the pointer at 8th place in the leaf and the insertion is over. A part of the code which shows the insertion of a key into three while calculating the parity bit is shown in figure 4.1.

The parity bits will be checked while trying to get a key-value pair from the tree. If we want to get the pair (200,12) which is already inserted into the tree, we will start the traverse from the root and check the pointer which is located in the 0th fragment in the root node. A calculation will be done in order to get the parity bit of the pointer.

If the result of the parity is 1 and three of the four bits that are assigned for the parity data of this pointer are also 1, then we assume that the pointer is not corrupted, otherwise an error will be printed. This check will be done on all the fragments which lead us to the corresponding leaf. If all the parity checks passes, then the value according to the key will be retrieved. A part of the code which shows the parity checking while trying to get a key-value from the tree is shown in figure 4.2.

If we assume that we have multiple values for one key, then we need to



```
u_int64_t pointer_value=(u_int64_t) n->children[(int)key];
bool parity = 0;
while (pointer_value)
{
    parity = !parity;
    pointer_value = pointer_value & (pointer_value - 1);
}

if((int)key%2==0)//getting the first four bits of n->parity[(int)key/2]
{
    ptrresult1=(n->parity[(int)key/2] & 0x10) | (n->parity[(int)key/2] & 0x20) |
    (n->parity[(int)key/2] & 0x40) | (n->parity[(int)key/2] & 0x80);
  if( parity==0 && ( ptrresult1==0x00 | ptrresult1==0x10 | ptrresult1==0x80 | ptrresult1==0x40 | ptrresult1==0x20))
  {
       bits = bits >> 4;
      if(i!=6)
       n =(TrieNode*) n->children[(int)key];
      else{
          node *linki=(node *)n->children[(int)key];
          while ( linki->next != 0)
          {
              if(linki->x==value)
                  return value;
              linki = linki->next;
          }
          if(linki->x==value)
              return value;
          else return -1;
          }
  }
  else if( parity==1 && (ptrresult1==0xF0 | ptrresult1==0xE0 | ptrresult1==0xD0 | ptrresult1==0xB0 | ptrresult1==0x70))
  {
      bits = bits >> 4;
      //if(i<3)
      if(i!=6)
        n =(TrieNode*) n->children[(int)key];
```

Figure 4.2: Checking parity bits of the pointers

have the mentioned linked lists which are pointed to by the fragments in the leaf level. In this case, for all of the values in the linked lists, parity bit should be set and checked. Otherwise, if we assume that there is one value according to each key, then values can reside inside the fragments of the nodes in leaf level and parity checking is already implemented for those.

For testing this methodology, three steps are done:

- Different number of keys were added to the tree.
- Corruptions are done in the leaf level. Program will ask from the user that how many keys he would like to corrupt. Bit flips are done on the keys which are already inserted into the tree. Based on this solution, 3 bit flips is done on each pointer in the leaf level.
- By using function *getNode()*, we tried to retrieve the key-value pairs of all the inserted keys and all the corrupted keys have been detected but only if the number of bit flips in the addresses were odd. If we have even number of bit flips, detection can not happen.

4.3 New Layout for Node Storage in Prefix Tree

In the last method, we used parity bit for detection of bit flips. In this method, we try to find any corruption in the pointers by knowing the exact place of the nodes rather than using redundant data. In the last method, malloc was called whenever a node was created, so there was no order in the address of the nodes in the memory. In this method, memory for a full tree is preallocated, so whenever a node has to be created, it will be set in its correct place which is known by us. Preallocation of memory for the tree is done using calloc function in c. In each level, calloc is done multiple times in order to reserve memory for multiple blocks in that level. Each of the blocks contain the space for multiple nodes. When a node is created, it is better that most of the other nodes in that block will also be created soon, because memory is already allocated for them. So it is better to have a dense tree which includes more nodes inside it in order to manage memory allocation better. So the first step is to find density of the tree with insertion of different set of keys.

4.3.1 Density of Tree

Density of the tree is calculated based on the number of nodes which are created in the tree after insertion of a set of keys.

A set of keys are inserted into the tree and number of created nodes in the tree is counted. At the start, we inserted keys which were multiples of three. The second set of keys were produced by using *Uniform int distribution*. The third set of keys were produced by using *Poisson distribution*.

A set of results has been produced based on this test which is shown in the following figures.

Based on the charts in figure 4.3, we see the number of added nodes to the tree based on Poisson distribution. As it is shown, more than 30000 nodes should be added to the tree till we reach a full tree.

In figure 4.4, we can see the result for insertion of keys into the tree based on uniform int distribution. As it is shown, around 30000 keys need to be inserted into the tree in order to have a full tree.

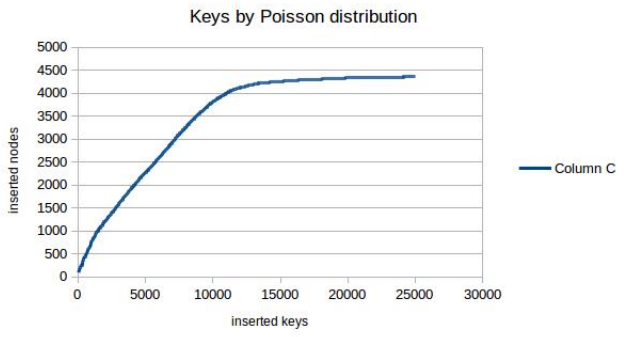


Figure 4.3: Number of created nodes after using Poisson distribution for key creation

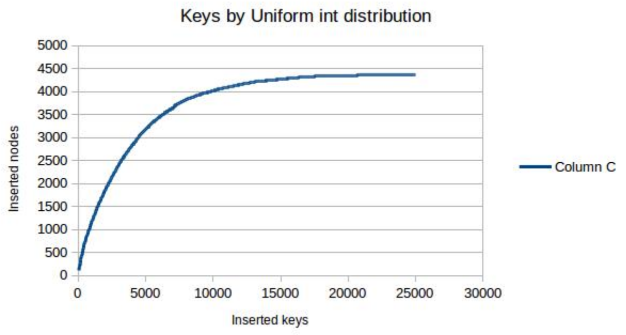


Figure 4.4: Number of created nodes after using Uniform int distribution for key creation

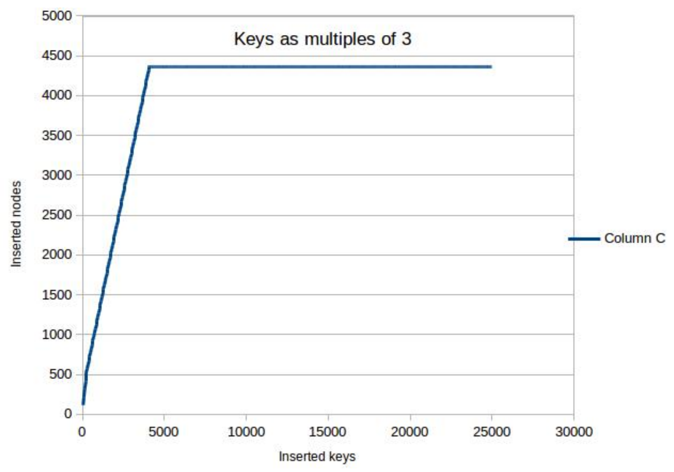


Figure 4.5: Number of created nodes after using keys which are multiples of three

In figure 4.5, we can see the result for insertion of keys which are multiples of 3. As it is shown, around 4000 nodes need to be inserted into the tree in order to have a full tree. This result has a big difference from the other two results. This huge difference is caused by the nature of uniform int distribution and Poisson distribution. Target of these distributions is to provide the random values. By using uniform int distribution and Poisson distribution, more keys are needed to be added in order to have a denser tree. There is no randomness in creating the keys which are multiples of three. Based on this result, keys which are multiples of three are used for creating the tree.

4.3.2 Creation of the Dense Tree

Based on this method, pre allocation of memory is done for a full tree. So we preallocate $16 * 8$ bytes for the root and keep the starting address of the tree in a variable. After that, we allocate $16 * 16 * 8$ bytes for the first level in the tree and keep the starting address of this level in a variable. After that we will keep every 32 nodes in one page. So overall, we have kept starting address of the root, first level,second level and third level in 4 variables. Whenever we want to insert or access a key in the tree, we have to calculate address of the corresponding nodes based on one of those 4 addresses. (This implementation is done for 16 bit keys)

4.3.3 Insertion of Bit Flips

As discussed, we built the tree with the keys which are multiples of three. 20000 keys have been inserted and we chose the keys by reduction of multiples of 54 from 60000 for bit flip insertion. If we assume that third level is the leaf level, then we insert the bit flips in the second level. Inserting bit flips in a fragment is done by using XOR function. So for insertion of one bit flip, we have to XOR the address with a number which has one bit of binary set to one. While inserting bit flips in the second level of these keys, we keep the exact address that is corrupted in an array called *corrupted*. Whenever we want to corrupt a fragment, we have to check the fragment's address with the addresses that are in the *corrupted* array. We do this check in order to avoid corrupting the same address twice. When we try to get these 20000 keys, we know exactly to which addresses we should traverse, because all the nodes in the tree are located in order. So when we reach a node which contains the pointer which does not match with the layout, we check its address with the addresses which are kept in the *corrupted* array. If it matches one of the addresses in the array, we increase the counter *countcorrupt*. At the end of getting all the keys from the tree, we compare the *countcorrupt* with the actual number of corruptions in *corrupted* array. As a result, all of the corruptions have been detected. The difference between detection mechanism in this approach and parity approach is that by using parity bits, only an odd number of bit flips can be detected but by using this new structure, any number of bit flips can be detected and no redundant data is needed.

4.3.4 Evaluation

This solution is almost like the former solution, with this difference that we have not used parity bits for detection, but we have used the structure of the tree to detect bit flips. However, memory consumption is quite high. When size of the indexes increases, number of nodes will increase and will cause to very large memory consumption. If we still want to use this approach to keep the nodes in an order in memory based on breadth-first traverse, then we have to do it in a more memory optimized way.

4.4 Implementation of Prefix Tree with Less Memory Accesses

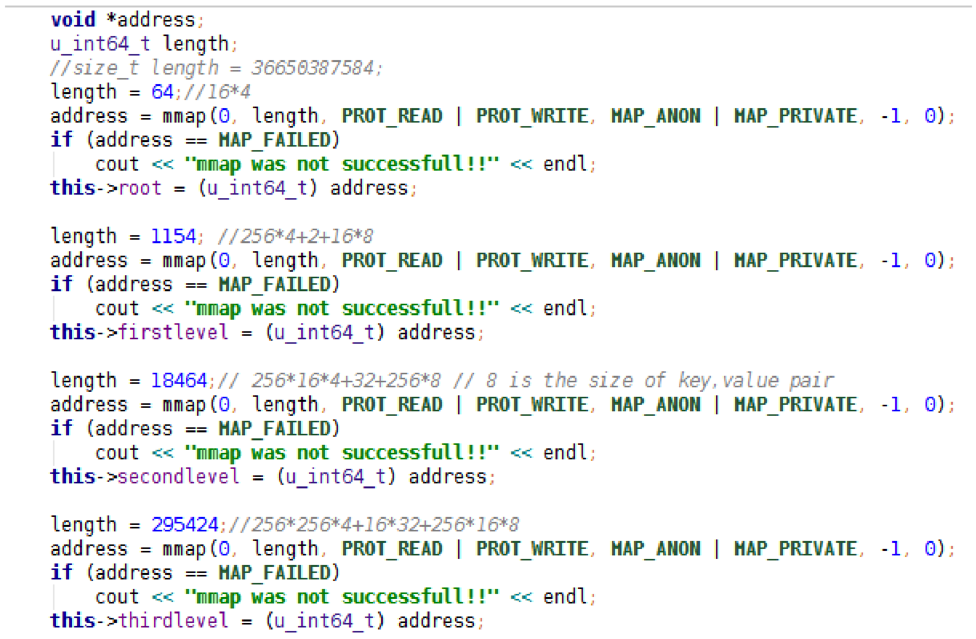
As it was discussed in the last chapter, in this methodology, we will reserve virtual memory for a full tree again (when we have 32 bit keys) plus all the space which is needed for key-value pairs and flag bytes. This is a lot of memory, but as we know, all of it is reserved in virtual memory and it will not cause any allocation in physical memory until we write a value in a part of it. When we attempt to create a node or set a flag byte or key-value pair, then an area which has the size of the mmaped area will be allocated in physical memory. An address in virtual memory will be mapped to an address in physical memory through a mapping table. As we will see in the results, this type of memory preallocation is the disadvantage of this approach.

In figure 4.6, a part of code which allocates memory for root, first, second and third level is shown.

By using *mmap*, we can define a certain number of parameters including:

- where to start the mapping
- length of mapping
- By setting PROT_READ and PROT_WRITE as the third argument, we indicate that pages can be read or written.
- MAP_ANON and MAP_PRIVATE has been set which defines that mapping is not backed by any file and its contents are initialized to zero. [15]

After allocating enough memory for the whole structure, we started inserting key-value pairs into the tree. Both key and value is assumed to be 32 bit. So they can have value of 4000000000 at most. At the worst case, we might need to traverse to the last level in the tree. So we have a for loop with 8 iterations. The only exceptions in these iterations is when we are in the sixth and seventh level. In the sixth level, we have to set the pointers



```
void *address;
u_int64_t length;
//size_t length = 36650387584;
length = 64;//16*4
address = mmap(0, length, PROT_READ | PROT_WRITE, MAP_ANON | MAP_PRIVATE, -1, 0);
if (address == MAP_FAILED)
    cout << "mmap was not successfull!!" << endl;
this->root = (u_int64_t) address;

length = 1154; //256*4+2+16*8
address = mmap(0, length, PROT_READ | PROT_WRITE, MAP_ANON | MAP_PRIVATE, -1, 0);
if (address == MAP_FAILED)
    cout << "mmap was not successfull!!" << endl;
this->firstlevel = (u_int64_t) address;

length = 18464;// 256*16*4+32+256*8 // 8 is the size of key,value pair
address = mmap(0, length, PROT_READ | PROT_WRITE, MAP_ANON | MAP_PRIVATE, -1, 0);
if (address == MAP_FAILED)
    cout << "mmap was not successfull!!" << endl;
this->secondlevel = (u_int64_t) address;

length = 295424;//256*256*4+16*32+256*16*8
address = mmap(0, length, PROT_READ | PROT_WRITE, MAP_ANON | MAP_PRIVATE, -1, 0);
if (address == MAP_FAILED)
    cout << "mmap was not successfull!!" << endl;
this->thirdlevel = (u_int64_t) address;
```

Figure 4.6: Virtual memory allocation for the first, second and third level

to the nodes or key-value pairs in the seventh level. Seventh level has 8 sub levels. So we need to figure out the sub level in which the node is. Also if we are in the last level, then values will sit in the place of pointers (Pointers and values are both 32 bits long).

There are four variables in the implementation named as extr, extr2, extr3 and extr4. These variables will be updated in each iteration and would help us in finding the location of node in the next level, corresponding place for storing key-value pair if it is needed, place of the flag byte and the number of bytes which is needed for storing the flag bytes in each level. At the start of insertion:

- extr = key1 $*$ 4 which shows the location of bucket in the root. (Each bucket has the size of 4 bytes)
- extr2 = key1 $*$ 8 which shows the location of key-value pair at the end of first level. (Each key-value pair has the size of 8 bytes)
- extr3 = key1 which shows that flag bit of the corresponding node should

be in which flag bit at the end of the level.

- extr4 = 2 since we have 16 nodes in first level and we need one flag bit per node, so we need 2 bytes for storing the flag bits in the first level.

In the second iteration, these values will change as follows:

- extr $=$ key1 $* 4 * 16 +$ key2 $* 4$ which shows the location of bucket in the first level.
- extr2 $=$ key1 $* 8 * 16 +$ key2 $* 8$ which shows the location of key-value pair at the end of second level.
- extr3 $=$ key1 $* 16 +$ key2
- extr4 = 32 Since we have 256 nodes in second level, so we need 32 bytes for storing the flag bits in the this level.

4.4.1 Insertion

When we want to insert a key into the tree, the first thing to do is to find out if the flag bit of the corresponding node in the next level is set to one or zero. Based on the variables which were mentioned, we can locate the node and also the place of flag bit. If flag bit is set to one, then we can continue the iteration to the next level, otherwise if the flag bit is zero, we have access to the key-value pair which is already inserted in the key-value pairs' area of that level. We should delete the key-value pair from that area and set the flag bit of the node to one and insert the key that was in the key-value area to the tree. The other calculation which is done in the insertion function is to identify the average degree of nodes in the tree. We need this information in the evaluation chapter. For this calculation, whenever we descend from a fragment of a node for the first time, we will increase the degree of the node which includes that fragment. The map data structure in C++ is used for this purpose. The first in map data structure is the address of the node and the second in map data structure is the degree of that node. Whenever we descend from a fragment in a node, at first we have to check if the address of that node is in the map or not. If the address is in the map, we have to increase the second by one. If the address of the node is not in the map, we have to insert a new pair into the map. For getting the average degree of the nodes in the tree, we have to iterate through the map data structure and get all the degrees of the nodes and add them together and divide this result by the number of nodes in the tree. In the evaluation chapter we will see that how this average degree will change in relevance to the number of inserted keys.

The keys which are inserted into the tree starts from 4,000,000,000 and they are decreased one by one.

4.4.2 Bit Flip Insertion

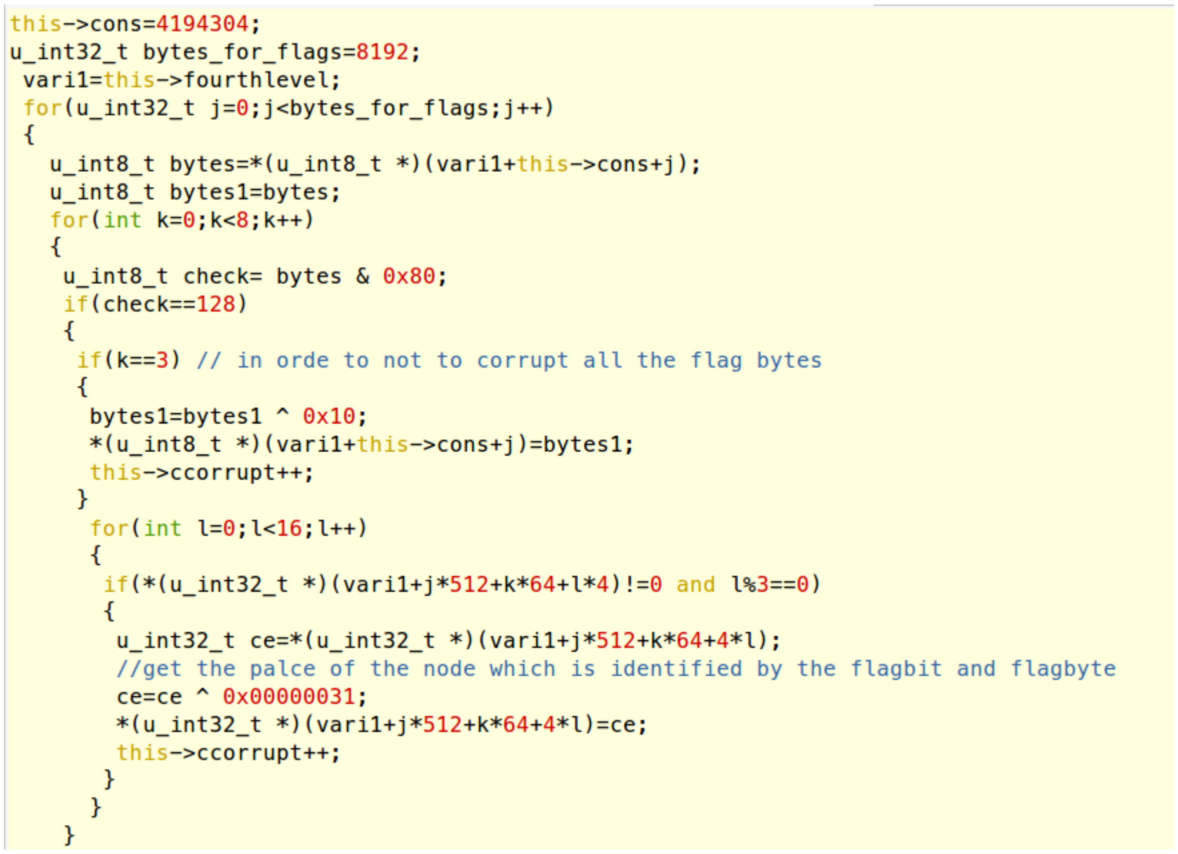
As it was mentioned in the last chapter, three methodologies were used for insertion of bit flips into the tree. The first solution was to insert bit flips in all the levels in the tree, so the number of bit flips in each level were based on the number of bytes in each level. This solution caused a lot of memory consumption. Because we were inserting bit flips in places that we were not sure if they were even set or not and in case the number of inserted keys were a few, we have inserted bit flips in levels that we might not have even had a node there, but by setting bit flips in those levels, we caused that part of virtual memory to be allocated in physical memory which was not actually needed and just leaded to high memory consumption. Instead, I used the second methodology. In this method, after insertion of all the keys into the tree, we count the number of inserted nodes into the tree. We know the maximum number of nodes that can be in each level. By counting the number of inserted nodes into the tree, we can estimate the levels which are filled in the tree and set the bit flips only on those levels. As an example, if number of nodes in the tree is between 300 and 5500, then we will insert bit flips in the second level only. The reason is that, we have 1 node in root level, 16 nodes in first level and 256 nodes in second level and maximum number of nodes in the third level is 4096. Based on this assumption, we are sure that insertion of bit flips in the second level and not in further levels is almost a safe decision. This method can still not recognize some of bit flips but is better than the last solution. In all the methods which were discussed, when bit flips are inserted, we might attempt to corrupt the same pointer twice because it is very obvious that two keys will descend from the same pointer. In this case, we have to avoid corrupting the same pointer twice, because when we corrupt a pointer for the first time, corrupting it for the second time will lead to the original value of the pointer. In order to avoid this problem, I keep the address of the already corrupted places in an array and in case that address is going to be corrupted again, we avoid this action. The same thing can happen while detecting bit flips, we might detect bit flips on an address twice when we refer to two different keys. In this way, we might count this detection twice, while the bit flip has happened only once. So we keep the address of the detected place in an array and whenever we want to check for corruptions we have to check if this address is already in the array or not. If the address is not in the array, we conclude that it is a new detection.

In the last method, some checks are done in order to make sure that all the nodes, flag bytes or key-value pairs which are corrupted are actually set in the tree.

At first, a specific level is chosen to be corrupted. We do not corrupt multiple levels, because that can cause problem in detecting the number of corruptions. The reason for that is that when nodes in a level are getting corrupted, corrupting nodes or key-value pairs in the lower level might not make sense, because we might never access those nodes or key-value pairs based on the corruption of upper level.

After choosing the level for corruption, the flag bytes at the end of the level are checked bit by bit, if a bit is 1, then the corresponding node in that level is found. When a flag bit is set to 1, it means that the node is created but it does not define which of the fragments in that node is set, for this purpose, there is a for loop which loop through the node and would check the fragments in that node. If a fragment is zero, it means that it is not set. If a fragment is not zero, it means that the fragment is set and corruption will be done on it. Some of the flag bits and key-value pairs are also chosen for corruption. Figure 4.7 shows a piece of code for doing this corruption.

Using the last method for bit flip insertion, will lead to detection of all the bit flips, because bit flips are inserted in the places which we are sure are set. The disadvantage of this approach is that, the run time is quite high. The reason is that, for insertion of bit flips, we have to iterate through the levels and put the corruption in a certain key-value pair, flag byte or pointer. All these iterations for all the keys that we want to corrupt will lead to a very high execution time.



```
this->cons=4194304;
u_int32_t bytes_for_flags=8192;
 vari1=this->fourthlevel;
 for(u_int32_t j=0;j<bytes_for_flags;j++)
 {
   u_int8_t bytes=*(u_int8_t *)(vari1+this->cons+j);
   u_int8_t bytes1=bytes;
   for(int k=0;k<8;k++)
   {
    u_int8_t check= bytes & 0x80;
    if(check==128)
    {
     if(k==3) // in orde to not to corrupt all the flag bytes
     {
      bytes1=bytes1 ^ 0x10;
      *(u_int8_t *)(vari1+this->cons+j)=bytes1;
      this->ccorrupt++;
     }
      for(int l=0;l<16;l++)
      {
       if(*(u_int32_t *)(vari1+j*512+k*64+l*4)!=0 and l%3==0)
       {
        u_int32_t ce=*(u_int32_t *)(vari1+j*512+k*64+4*l);
        //get the palce of the node which is identified by the flagbit and flagbyte
        ce=ce ^ 0x00000031;
        *(u_int32_t *)(vari1+j*512+k*64+4*l)=ce;
        this->ccorrupt++;
       }
      }
    }
```

Figure 4.7: Insertion of bit flip in key-value pairs, nodes and flag bytes in the tree

4.4.3 Retrieving Values from Tree

For getting a key and its corresponding value from the tree, the first thing to do is to check the pointer value which is stored in the bucket of the node. If the pointer is null, then we should insert the key-value pair at the end of the next level. If pointer is not null and is less than a variable called *cons*, then we establish that this bucket is pointing to a node rather than a key-value pair, so we should continue the iteration to find the corresponding value. Otherwise, bucket is pointing to a key-value pair and we can retrieve the value. *Cons* actually holds an offset. For the first level (the level after root level), *cons* is 1024 (16*16*4). All the nodes in first level are stored from address 0 to address 1024. Flag bytes and key-value pairs are stored after this address (assuming that we are dealing with the offset address). In the second level, *cons* will increase to 1024*16. *Cons*, the number of nodes, number of flag bytes and number of key-value pairs increases every level. These values will help us to get the place of a key-value pair or a node. While retrieving values from the tree, we will check for any bit flip in the flag byte, key-value pair or a pointer. The details of this check is already explained in last chapter.

4.5 Summary

In this chapter, we covered the details of the implementation of different methodologies in this thesis. We started with the implementation of a simple prefix tree, then we added parity bits to the nodes of this tree in order to detect bit flips. After that we implemented the tree in a reserved part of memory in a way that all the nodes are after each other. In the last part of the implementation, tree is structured in a way to decrease memory accesses. In all of these implementations, we made the tree with the capability of detecting bit flips.

Chapter 5

Evaluation and Future Work

In this chapter, results and evaluation of the used methods in this thesis is provided. At the end of the chapter, future works has been discussed.

5.1 Evaluation

In this chapter, we have evaluated the following methods:

- Prefix tree with the ability to detect bit flips by using parity bits and having high memory access.
- Prefix tree with the ability to detect bit flips by using the structure of the tree in order to detect bit flips instead of using parity bits.

In order to evaluate the mentioned methods, I have provided a number of charts to show the results of the experiment.

In the first set of pictures, we have compared number of created nodes in the tree. When we insert keys into the tree, nodes are created and number of created nodes increases whenever we increase the number of keys. The result in the two solutions are quite different. In the first solution, number of created nodes increases rapidly when we insert keys into the tree since as it is already said, every value according to a key should be inserted in the leaf level. So we need all the nodes in between in order to traverse to the leaf.

In contrast, in the second solution, value can be inserted in any level in the tree and we do not need to traverse to the leaf level. While inserting a key into the tree, if we need to descend from a bucket in a node, if the bucket was null till now and it is the first time that it is going to be set, then we set the bucket with a pointer that points to a key-value pair in the next level. In this case, we will not create a node until it is really necessary. The result in the graphs and tables shows the difference in the number of created nodes in the first and second solution.

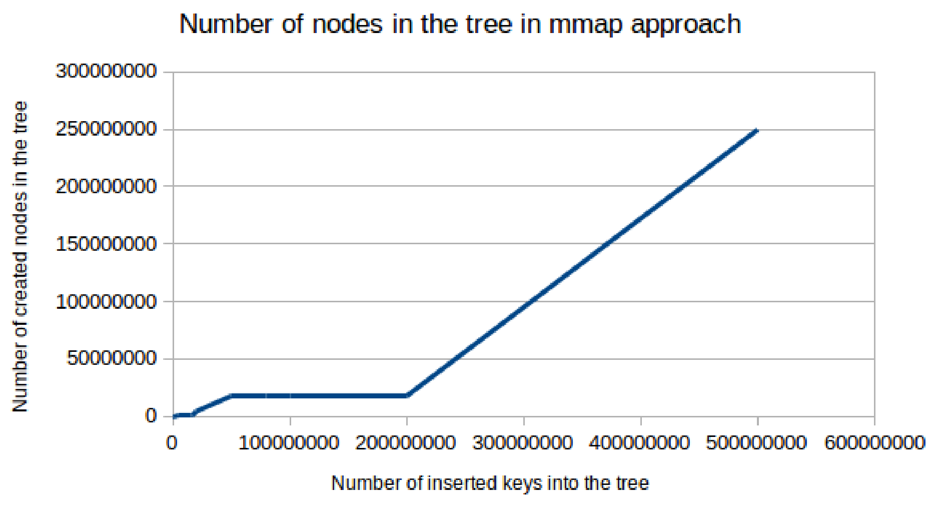


Figure 5.1: Number of created nodes based on the number of inserted keys in the new structure of tree with less memory access

1000	10,000	100,000	1,000,000	5,000,000	15,000,000	17,000,000	50,000,000	100,000,000
273	14368	38833	69905	1118481	1118481	1341265	17895697	17895697

Table 5.1: Number of created nodes based on inserted keys in the new structure of tree with less memory access

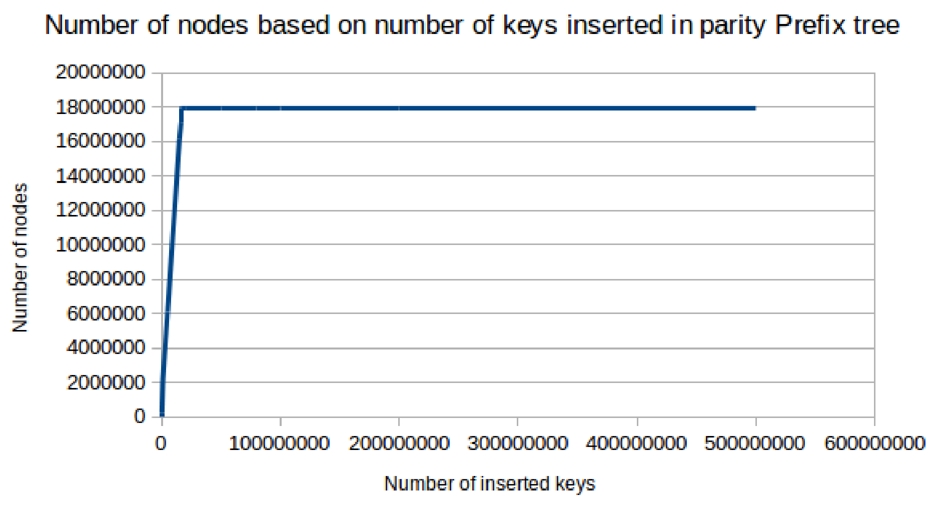


Figure 5.2: Number of created nodes based on the number of inserted keys in the tree with parity bits and high memory access

10,000	100,000	1,000,000	5,000,000	15,000,000	17,000,000	50,000,000	100,000,000
4237	269905	2069905	6118481	16118481	17895697	17895697	17895697

Table 5.2: Number of created nodes based on inserted keys in the tree with parity bits and high memory access

In the second set of pictures, we have compared average node degrees in the tree. In the prefix tree with parity bit implementation, we can see that density of the tree is low most of the time, since in every round, a lot of nodes are added into the tree. This will cause a low degree in average. However, in the other implementation of prefix tree, the average degree is quite higher. Since we add nodes into the tree only if it is necessary. In this case we will have a denser tree, but as it is obvious in the graph, the value for the density changes in every round since the number of created nodes are changing.

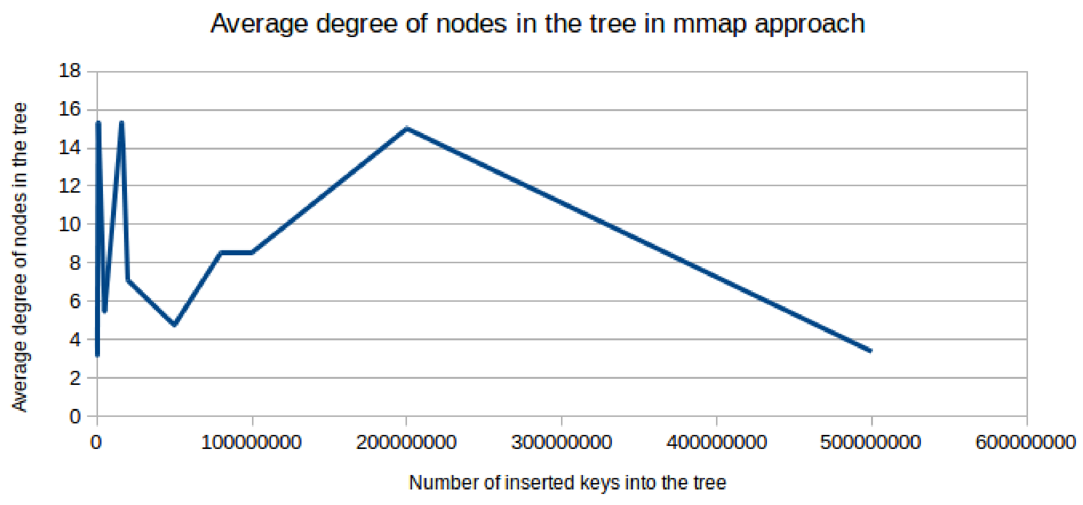


Figure 5.3: Average degree of nodes based on the number of inserted keys in the new structure of tree with less memory access

1000	10,000	100,000	1,000,000	5,000,000	15,000,000	17,000,000	50,000,000	100,000,000
4.65	3.2	3.57	15.3	5.47	14.41	14	4.75	8.5

Table 5.3: Average degree of nodes based on the number of inserted keys in the new structure of tree with less memory access

1000	10,000	100,000	1,000,000	5,000,000	15,000,000	17,000,000	50,000,000	100,000,000
1.23	1.29	1.37	1.48	1.81	1.93	1.949	3.79	6.58

Table 5.4: Average degree of nodes based on the number of inserted keys in the tree with parity bits and high memory access

In the third set of pictures, we have compared memory consumption in both of the solutions. The way that memory usage increases in the first

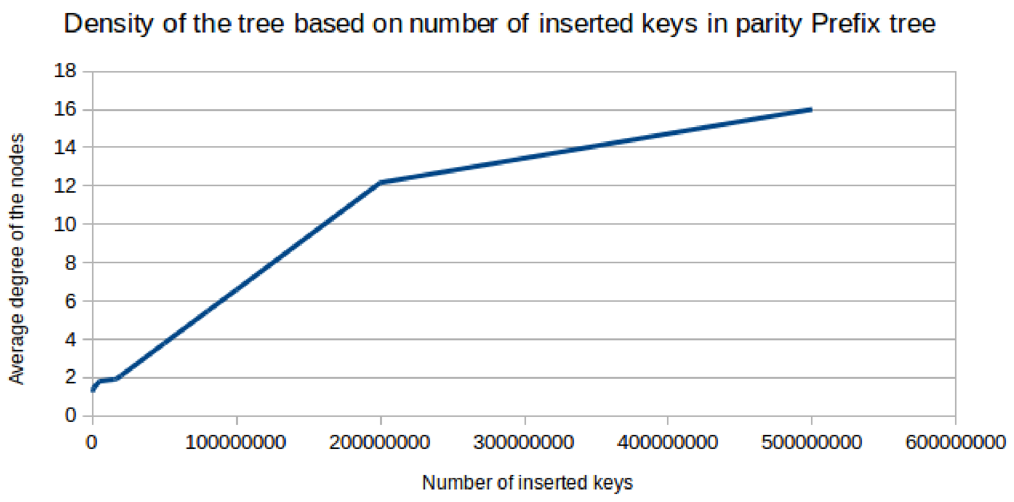


Figure 5.4: Average degree of nodes based on the number of inserted keys in the tree with parity bits and high memory access

solution is totally acceptable, since number of nodes increases with increasing the number of keys, this will cause increase in the memory usage until the time that tree becomes full.

In the second solution, whenever a node is created in a level (except of leaf level) or if a flag bit is set or if a key-value pair is set, since mmap has been called to allocate memory for the whole level, physical memory for the size of the level is allocated. As we can see in the graph, until insertion of 17,000,000 keys, memory usage is not high, because levels' sizes are small. If the level's size is big, then this can cause a lot of memory consumption. Memory consumption after insertion of 17000000 keys shows this fact, since with insertion of this number of keys, we touch the seventh level, which consumes a lot of memory.

Based on the given information the memory usage is shown in tables 5.5 and 5.6.

In the fourth set of pictures, bit flip detection in both of the approaches is shown. In the first solution with using parity bits, all the bit flips are put in the leaf level and since the number of bit flips in each pointer is 3, all of them could be detected. Result is shown in table 5.7.

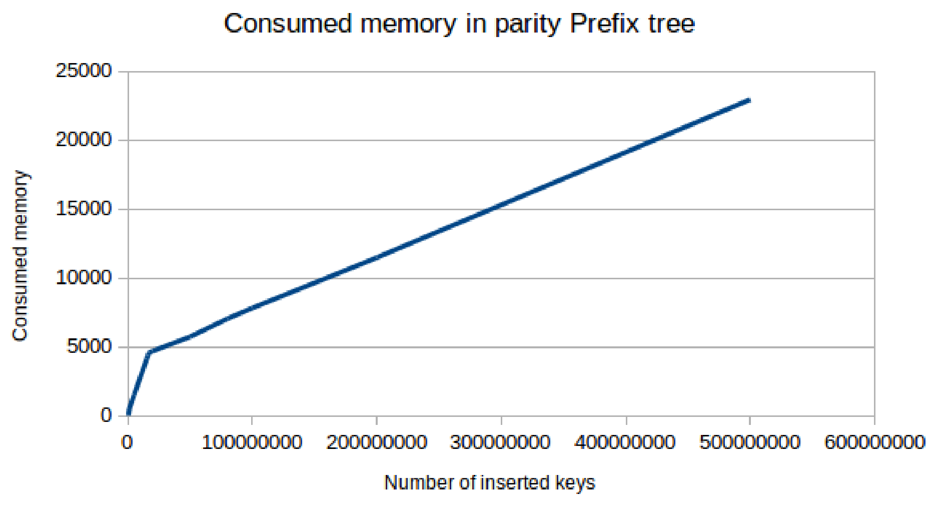


Figure 5.5: Memory consumed based on the number of inserted keys in the tree with parity bits and high memory access

1000	10,000	100,000	1,000,000	5,000,000	15,000,000	17,000,000	20,000,000	80,000,000	100,000,000
3.4	10.412	65.32	494.54	1536.28	4114.4	4581.26	4698.49	7042.280	7823.532

Table 5.5: Memory consumed based on the number of inserted keys in the tree with parity bits and high memory access

1000	10,000	100,000	1,000,000	5,000,000	15,000,000	17,000,000	20,000,000	80,000,000	100,000,000
1.55	3.75	18.98	28.46	321.776	400.108	20397.78	20712.532	22290.580	22512.364

Table 5.6: Memory consumed based on the number of inserted keys in the new structure of tree with less memory access

In the second solution based on the details which were given in the implementation chapter, the rate of bit flip detection depends on the method

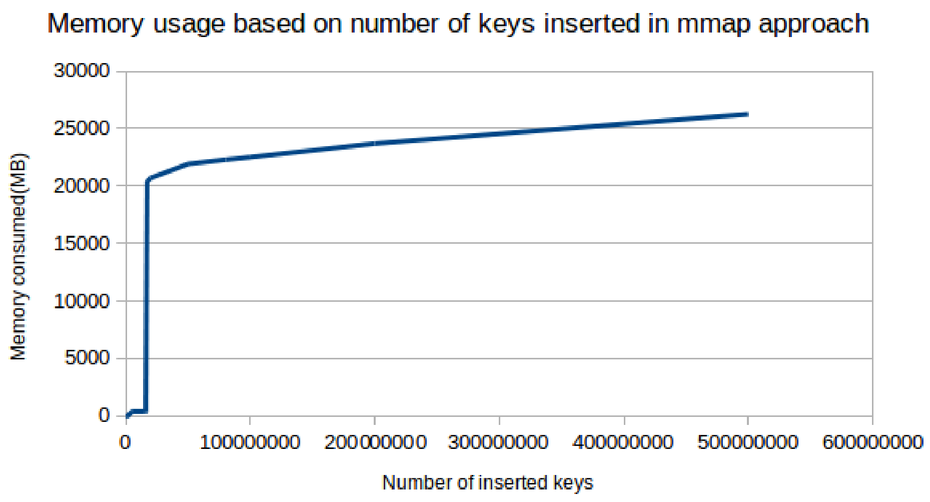
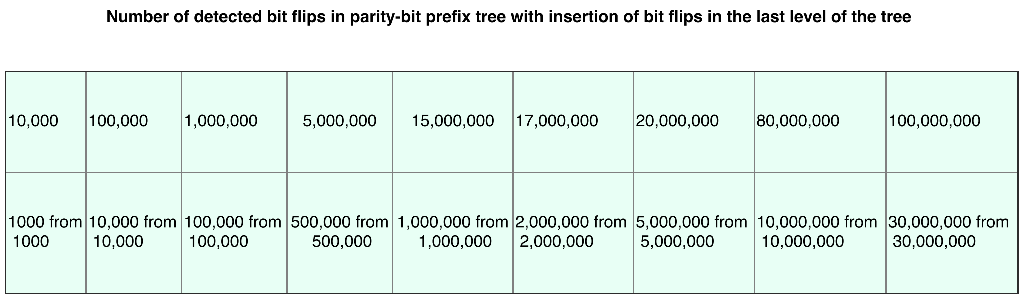


Figure 5.6: Memory consumed based on the number of inserted keys in new structure of tree with less memory access



Number of detected bit flips in parity-bit prefix tree with insertion of bit flips in the last level of the tree

10,000	100,000	1,000,000	5,000,000	15,000,000	17,000,000	20,000,000	80,000,000	100,000,000
1000 from 1000	10,000 from 10,000	100,000 from 100,000	500,000 from 500,000	1,000,000 from 1,000,000	2,000,000 from 2,000,000	5,000,000 from 5,000,000	10,000,000 from 10,000,000	30,000,000 from 30,000,000

Table 5.7: Number of detected bit flips in tree with parity bits and high memory access with insertion of bit flips in the leaf level

for bit flip insertion. If bit flips are inserted in places that we are not sure are set, it is possible that some of the bit flips are not detected because that part of memory might not get accessed at that time. The result is shown in table 5.8.

In order to prove that the solution which was given for bit flip detection works, we used another bit flip insertion method which inserts bit flips in the flag bytes, key-value pairs and pointers which are already set and then we started detecting bit flips. In this way we proved that all of the bit flips could be detected by this method. The result is shown in table 5.9.

1000	10,000	100,000	1,000,000	5,000,000	15,000,000	17,000,000	20,000,000	80,000,000	100,000,000
15 from 15	193 from 204	193 from 204	1127 from 1203	1127 from 1203	5781 from 6202	5781 from 6202	5781 from 6202	5781 from 6202	5781 from 6202

Table 5.8: Number of detected bit flips in the new approach while inserting bit flips level by level into the tree based on the number of nodes

40,000	50,000	100,000	200,000	500,000	1,000,000	5,000,000	15,000,000	20,000,000
4,096 from 4,096	4,096 from 4,096	32,415 from 32,415	73,728 from 73,728	170,271 from 170,271	401,407 from 401,407	401,407 from 401,407	401,407 from 401,407	401,407 from 401,407

Table 5.9: Number of detected bit flips in the new approach while inserting bit flips in the places that are definitely set (in fourth level of tree)

5.2 Future Work

One of the main issues of the second given solution was that memory was allocated for the whole tree at the beginning. In our solution, mmap was called on each level. This will lead to a lot of memory usage whenever a node is created, because an amount of memory equal to the size of mmaped area will be allocated in physical memory. For the first solution, mmap can be called in smaller areas of memory, in order to prevent allocation of a large area, whenever a node is created. This solution can cover some problems of the memory allocation, but still memory for a full tree is allocated at the beginning. Another solution for this problem is to call mmap whenever a node is going to be created. In this way, reserving memory at the start is not needed. By calling mmap, we can define the starting address of allocation. This seems to solve all the problems but in practice, the starting address of allocation is not the exact address that we enter, but it is the nearest address to the page boundary. By knowing this information, if mmap is called whenever a node is created, each node will be set at the starting address of a page, so a page will be reserved for a node.

A solution for this problem also exists. When a node is going to be created, we can check through a bitmap whether the page corresponding to that node is already reserved or not. If the page is already reserved, we don't need to call mmap again for creating that page. Otherwise, we should call mmap to reserve a page. In our case, the node size is 64 byte, so 64 nodes can reside in one page and mmap can be called once for creation of all of them.

Chapter 6

Bibliography

[1] M.Boehm,B.Schlegel,P.B Volk,U.Fischer,D.Habich , Online Bit Flip Detection for In-Memory B-Trees on Ureliable Hardware 2014,TU Dresden

[2] Efficient In-Memory Indexing with Generalized Prefix Trees 2011,Tu Dresden

[3] T J.Lehman,M J.Carey , A Study of Index Structures for Main Memory Database Management Systems 1986: University of Wisconsin

[4] G.Graefe,R.Stonecipher , Efficient Verification of B-Tree Integrity : Hewlett-Packard laboratories,Microsoft.

[5] Hard Error Definition, http://www.webopedia.com/TERM/H/hard_error.html

[6] Soft Error Definition, http://www.webopedia.com/TERM/H/soft_error.html

[7] Reinhardt, Steven K.; Mukherjee, Shubhendu S. "Transient Fault Detection via Simultaneous Multithreading".(2000) ACM SIGARCH Computer Architecture News 28 (2): 25–36

[8] Mukherjee, Shubhendu S.; Kontz, Michael; Reinhardt, Steven K. "Detailed Design and Evaluation of Redundant Multithreading Alternatives".(2002) ACM SIGARCH Computer Architecture News 30 (2): 99.

[9] Vijaykumar, T. N.; Pomeranz, Irith; Cheng, Karl . "Transient-Fault Recovery Using Simultaneous Multithreading".(2002) ACM SIGARCH Computer Architecture News 30 (2): 87.

[10] Soft Errors Detection, https://en.wikipedia.org/wiki/Soft_error#Detecting_soft_errors

[11] Immunity Aware Programming, https://en.wikipedia.org/wiki/Immunity-aware_programming

[12] Immunity Aware Programming,https://en.wikipedia.org/wiki/RAM_parity

[13] ECC Memory,https://en.wikipedia.org/wiki/ECC_memory

[14] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, Wolfgang Lehner. "KISS-Tree: Smart Latch-Free In-Memory Indexing on Modern Architectures".(2012) In: Proceedings of the Eighth International Workshop on Data Management on New Hardware, Scottsdale, AZ, USA.

[15] Mmap Manual,http://man7.org/linux/man-pages/man2/mmap.2.html

[16] C-Trie,https://en.wikipedia.org/wiki/Ctrie

[17] Hash Array Mapped Trie, https://en.wikipedia.org/wiki/Hash_array_mapped_trie

[18] UHCL 35a Graduate Database Course-Extendible Hashing, https://www.youtube.com/watch?v=TtkN2xRAgv4

[19] UHCL 36a Graduate Database Course-Linear Hashing, ,https://www.youtube.com/watch?v=Yw1ts57uL7c

[20] Aleksandar Prokopec, Phil Bagwell, Martin Odersky, "Cache-Aware Lock-Free Concurrent Hash Tries". (2011) École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.

[21] Aleksandar Prokopec, Nathan G. Bronson, Phil Bagwell, Martin Odersky, "Concurrent Tries with Efficient Non-Blocking Snapshots". (2012) In: New Orleans, Louisiana, USA.

[22] Steffen Heinz, Justin Zobel, Hugh E. Williams, "Burst Tries: A Fast, Efficient Data Structure for String Keys". (2002) In: School of Computer Science and Information Technology, RMIT University. GPO Box 2476V, Melbourne 3001, Australia

[23] D.D. Sleator and R.E. Tarjan. Self-Adjusting Binary Search Trees".(1985) In: AT&T Bell Laboratories, Murray Hill, NJ.