A Prefix-based Hybrid Solution for Main Memory Indexing

A Thesis

submitted to the designated
by the General Assembly
of the Department of Computer Science and Engineering
Examination Committee

by

George Christodoulou

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WITH SPECIALIZATION
IN SOFTWARE

University of Ioannina

February 2020

Examining Committee:

- **Nikolaos Mamoulis**, Professor, Computer Science and Engineering Department, University of Ioannina (Supervisor)
- **Panagiotis Vasiliadis**, Associate Professor, Computer Science and Engineering Department, University of Ioannina
- **Apostolos Zarras**, Associate Professor, Computer Science and Engineering Department, University of Ioannina

Dedication

I would like to dedicate this thesis to my family.

Acknowledgements

First and foremost, I would like to express my graditude to my advisor, Nikos Mamoulis, for giving me this opportunity, the support, the guidance, and the patience he offered me through all the time we have worked together. I am deeply grateful to my family for their unconditional support on any level that a human being can imagine. Their encouragement and faith in me were eternal through all these years. I would like to thank my friends who supported and believed in me even when it was not rational and for often giving me good reasons not to work. Last but definitely not least, I would like to thank Evangelos Papapetrou for all the influential discussions, knowledge and advice he gave me through the years about Computer Science and life as well.

Table of Contents

List of Figures

List of Algorithms

Abstract

George Christodoulou, M.Sc. in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece, February 2020.
A Prefix-based Hybrid Solution for Main Memory Indexing.
Advisor: Nikolaos Mamoulis, Professor.

Efficient retrieval in main memory is becoming increasingly important in modern applications that manage huge amounts of data (e.g. IoT and social network data). This is especially relevant for key-value stores which handle numerous key-value pairs which should be accessed in nanoseconds.

In this thesis, we introduce POT (Performance Optimized Tree), a novel hybrid tree, combining bucketing with data prefixes. We design a tree, which supports fast search operations. The tree consists of levels depending on segments of the data representation called prefixes and the leaf nodes point to ranges of the indexed data values, called buckets. We construct the tree so that we can hop through its levels without comparisons and then use binary search for the last mile accuracy. Our tree can easily adapt to any type of data and data distribution as we consider the binary representation of the indexed values. The layout of each node is carefully constructed for compactness and fast search. POT supports point and range queries.

We evaluate the performance of the tree on two real and one synthetic dataset and compare the results with other popular index structures. Our results indicate that our approach can perform much better than existing techniques. We also study the problem of tuning the index, depending on the data size and distribution.

Εκτεταμενη Περιληψη

Γεώργιος Χριστοδούλου, Μ.Δ.Ε. στην Πληροφορική, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Φεβρουάριος 2020.
Υβριδικό Ευρετήριο Κύριας Μνήμης με βάση τα Προθέματα.
Επιβλέπων: Νικόλαος Μαμουλής, Καθηγητής.

Η αποδοτική ανάκτηση στην κύρια μνήμη καθίσταται όλο και πιο σημαντική στις σύγχρονες εφαρμογές που διαχειρίζονται τεράστιους όγκους δεδομένων (π.χ. δεδομένα IoT και κοινωνικών δικτύων). Αυτό είναι ιδιαίτερα σημαντικό για αποθήκες δεδομένων κλειδιών-τιμών που χειρίζονται πολλά ζεύγη κλειδιών-τιμών τα οποία θα πρέπει να προσφέρουν πρόσβαση σε νανοδευτερόλεπτα.

Σε αυτή τη διατριβή, παρουσιάζουμε το POT (Performance Optimized Tree), ένα νέο υβριδικό δέντρο, που συνδυάζει το bucketing με τα προθέματα δεδομένων. Σχεδιάζουμε ένα δέντρο το οποίο υποστηρίζει λειτουργίες γρήγορης αναζήτησης. Το δέντρο αποτελείται από επίπεδα που εξαρτώνται από τα τμήματα της αναπαράστασης δεδομένων τα οποία ονομάζονται προθέματα και οι κόμβοι φύλλων υποδεικνύουν εύρη τιμών των ευρεθέντων δεδομένων, που ονομάζονται "κάδοι" δεδομένων. Κατασκευάζουμε το δέντρο έτσι ώστε να μπορούμε να προσπελάσουμε τα επίπεδα του δέντρου χωρίς συγκρίσεις και στη συνέχεια να χρησιμοποιήσουμε δυαδική αναζήτηση για να καταλήξουμε με ακρίβεια στην τιμή που αναζητάμε. Το δέντρο μας μπορεί εύκολα να προσαρμοστεί σε οποιοδήποτε τύπο δεδομένων και κατανομή δεδομένων, καθώς χρησιμοποιεί τη δυαδική αναπαράσταση των κλειδιών αναζήτησης. Οι κόμβοι του δέντρου είναι προσεκτικά κατασκευασμένοι ώστε να είναι συμπαγείς και να υποστηρίζουν γρήγορη αναζήτηση. Το POT υποστηρίζει ερωτήματα σημείων και εύρους.

Αξιολογούμε την απόδοση του δέντρου σε ένα συνθετικό και δύο πραγματικά σύνολα δεδομένων και τη συγκρίνουμε με τις αποδόσεις άλλων δημοφιλών δομών

ευρετηρίου. Τα αποτελέσματά μας δείχνουν ότι η προσέγγισή μας μπορεί να αποδώσει πολύ καλύτερα από τις υπάρχουσες τεχνικές. Επίσης, μελετάμε το πρόβλημα της προσαρμογής της δομής ευρετηρίου, ανάλογα με το μέγεθος και τη κατανομή των δεδομένων.

Chapter 1

Introduction

Databases are constructed to store information. When a user approaches the database in search of data, he provides a search key which corresponds to his target. Obviously, a user does not want to search the whole database in order to retrieve the query result. Therefore, databases are enhanced with indexes to speed up the retrieval of interesting information. From the user's point of view, it is desirable that the index supports search keys of different data types (e.g. numbers, strings) and that the search time is as low as possible. At the same time, it is desirable that the index will serve the user's needs for high speed search while it is economical in space, easily maintainable and can be modified to support additions and deletions from the database. The performance of the tree comes at the cost of space consumption. Many existing indexing solutions are focusing on the right balance between space requirements and look up performance.

The accessing and processing of data, in a fast and efficient way (i.e. the storing and querying of any type of data) is especially important for many popular applications (e.g. information retrieval and social networks), mainly in the fields of Big Data or IoT, which require the handling of very large and complex amount of data (index, search etc.). Memories are becoming larger and cheaper, hence, the focus has turned recently to the design of efficient main-memory indexes.

There are many algorithms and data structures used to index and search, including tries. Tries are multi-way tree structures, designed for organizing data in the main memory. The word trie emerges from the word "re-trie- val", because the trie can retrieve a word in a dictionary using a prefix of the word. Tries are efficient data retrieval structures. Using a trie, the search time of a stored value can be minimized by partitioning the corresponding key. Trie-based data structures have been successful in multiple applications such as text compression and dictionary management. Some specific examples are:

- Insert, delete and search for a word in a dictionary.
- Find out if a string is a prefix of another string.
- Find out how many strings have a common prefix.
- Suggestion of contact names in our phones depending on the prefix we enter.

In terms of implementation, tries -which are basically trees- are represented as a set of linked nodes, starting from a root node. Usually, the number of nodes at every level of the tree depends on the total number of possible prefixes up to a length corresponding to the level. For example, if we want to represent Greek words in a trie with one letter as the substring length used per level, there will be 24 children nodes at every level because there are 24 different letters (available options) in the Greek alphabet. So, the size of a trie depends completely on the data that it contains. Tries are used to index data, so the nodes at the lowest level, called leaves, end up pointing to values. Thus, every node contains a reference to children nodes that can be null and a value that can be null as well. The time to do a search is typically O(length of key) if the length of the prefix used is 1, like in our example. In general, the search time is O(length of key / prefix size). Insertions have the same complexity. The main drawback of tries is their large memory requirements for storing the key strings. For each level a large number of node pointers (equal to the number of possible words or characters, used to decide the next level).

Tries have some properties which makes them interesting:

- The height -and therefore also the complexity- of tries depends generaly on the length of the keys and not on the number of elements in the tree.

- Tries require no rebalancing operations and the tree construction is independent from the insertion order.
- The path to a leaf node represents the key of that leaf. Therefore, keys are stored implicitly and can be reconstructed from paths.
- Tries offer O(1) complexity for searching every level without the need of comparisons.

While memory consumption is important, main memory becomes larger and cheaper and the performance of main-memory databases is crucial, thus we need fast index structures. Therefore, we focus on optimizing the performance of the index and not on minimizing its space consumption.

Recently, Kraska et al. [1] introduced the interesting approach of replacing (at least partially) the principled and structured mechanisms of B-trees (and other classic indexes) by machine learning models. In a nutshell, a database index can be considered as a model which, given a search key value, predicts the position of the record that holds this value. However, finding an accurate single model for the entire domain of the key values might be hard. Motivated by this, our first steps of research have generated some thoughts for new indexing approaches based on a hybrid approach. The partitioning that the current implementations of trees define is based on the old (and obsolete) tree structure which demands that each node occupies the space of a disk block, and the (obsolete) objective is to minimize the I/O cost. Currently, however, we typically use main memory storage and main memory indexing. Hence, it is no longer a requirement that all partitions occupy the same space and we can seek for optimization in a different direction. Alternative partitioning and search approaches could be defined. In this thesis, we will investigate the design and use of a data-adaptive tree for main memory indexing. Originally, trees have been used with restrictions in node sizes and available span. On the other hand, the learned-index approach inserts multiple types of overhead in the solution. Specifically, an approximation of value distributions can be used to determine (very fast) the number of keys that are contained in a particular value range. In our context, an approximation of the data distribution could be used to determine node represenations and span. In conclusion, our problem is to find the position of a given key value as fast as possible and therefore we aim at designing a performance optimized tree.

In our implementation, we use a combination of trie logic and bucketing. We create a trie with multiple prefix lengths. The available prefix lengths are 4, 8 and 16. We consider the binary representation of each entry before insertion. So, the keys that we actually store are considered to be binary strings. Every level contains pieces of binary strings with length of 4, 8 or 16. The main difference with other trie-like indexes is that the keys in our approach are not stored until their full length. In that way, our trie avoids to expand in height and therefore minimizes the overhead in search that comes with it. In order to achieve that, we use buckets with a certain size as leaves, which contain multiple keys with a common prefix. Thus, when a search is executed, the trie is traversed until the corresponding bucket is found. Then, inside the bucket, we use binary search for the last mile accuracy.

1.1 Contributions

In summary, this thesis makes the following contributions:

- We propose the novel concept of a hybrid trie with data buckets. To our knowledge, this is the first work that defines this combination as a data structure for indexing.
- We propose an overall efficient indexing solution targeting low look-up times.
- We evaluate our approach using three real datasets, which have different distributions and sizes, and demonstrate that it operates well and fast.

1.2 Outline

This dissertation consists of 5 chapters. Chapter 2 provides related work by reviewing existing indexing solutions. Chapter 3 describes the structure of our proposed index.

It describes in detail every part in the tree and the algorithms used for tree construction and searching. Chapter 4 contains an experimental evaluation of our solution, where it is compared to other indexing solutions. Chapter 5 contains our conclusions and provides directions for future work.

Chapter 2

Related Work

The Adaptive Radix Tree (ART) is a trie-based data structure for main-memory indexing. The main idea of ART [4] is that each inner node adapts to reduce space consumption. Two key practices in ART are: path compression and lazy expansion. In lazy expansion the inner nodes are created only to distinguish two or more leaf nodes. In path compression, nodes with a single child are removed. ART aims at using a large span for performance enhancement, without the space inefficiency as a trade off. So the key idea is to use different nodes with a different span. Each inner node maps partial keys to child pointers. There are nodes with a span of size 4, 16, 48 and 256. Also, the leaves can be: single valued, multi valued and combined pointer-value. However, the use of string data forces ART to a lower fanout because of the sparse distribution.

Wormhole [3] is an ordered index structure with $O(logL)$ worst case time for look up of keys with L length. Thus, it is fully depended on the length of the keys. It is a main memory index which uses a trie for its non-leaf part and then a hash table. It does not support range queries, or inserts of new keys in the leaflist. The supported search continues until the searched key is smaller than the key in the list which is compared to, if the key is not in the list. Each item in wormhole has a bitmap for the existence of children. The space efficiency of the index depends on the similarity of the keys. Also, concurrency is supported with locks.

Mass tree [5] is a storage system specialized for key-value data (all in-memory) which must persist across server restarts. It uses variable-length keys and supports

range queries. It uses a wide fanout to reduce depth, prefetches nodes from DRAM and data is stored for cache awareness. Logs are kept in batches for cache recovery and are periodically checkpointed. Mass tree comprises one or more layers of B+-trees and each layer is indexed with a 8-byte slice of key. The tree has the same query complexity as B+-trees. For range queries, it has higher worst-case complexity than the B+-tree. It also bounds the number of none-node memory references to find a key in at-most one look-up. For concurrent accesses, the system uses writer-writer and writer-reader coordination. Furthermore, it syncs cores with ports to support fast operations through the network.

HOT [6] (Height optimized tree) aims at having high average fanout. The bits that are considered at each node are not fixed but chosen depending on the data distribution. This helps with the sparsity and the consistent high fanout. For a given span s there are 2^s pointers at each node. The downside is the increased space consumption for sparse data where the actual fanout will be much smaller than 2^s. HOT combines multiple nodes of binary Patricia trees with maximum node fanout of predefined value k. Each node of the tree uses custom span. In order to consume less memory adaptive node sizes are used. Each patricia trie has exactly n-1 nodes for n keys so a HOT node needs to store at most k-1 binary inner nodes (k maximum fanout). The maximum fanout consists of 32 bits for fast SIMD operations. There are available representations of 8, 16, 32 bits. Furthermore, instructions from BMI2 instruction set are used for key extraction speed up. The node layout has two dimensions of adaptivity:the size of the partial keys and the representation of the bit position (single or multi mask). The physical node layout consists of: (1) node header, (2) bit positions, (3) partial keys, (4) values. Also, any given set of keys results to the same structure regardless of the insertion order. Finally, there is a sync protocol, which locks the affected nodes by the action performed until the action is performed with safety.

HAT trie [7] structure is based on the reduction of the number of the trie nodes, which are reducted by selectively collapsing chains of nodes in buckets. HAT trie begins as an empty bucket which is populated until full. Although it is not cache conscious, it is efficient in a setting where all memory accesses are of equal cost. HAT uses linked list representation, although, traversing a linked list the address of a child cannot be known until the parent is processed. It is a combination of trie index with buckets which are cache-conscious hash tables. There are two main approaches: Pure and hybrid. The difference between them is in how buckets are maintained and split.

In the first case, buckets that contain strings sharing only one prefix are removed and in the second case buckets that share a single prefix are not removed (more than one parent pointer).

Fast Architecture Sensitive Tree (FAST) [8] is a binary tree which is designed, based on architecture features like page size, cache line size and SIMD width. FAST aims to optimize those architecture features in order to eliminate memory latency and takes full advantage of parallelism on thread and data level. As the tree grows, the CPU search can be bound from memory bandwidth. Therefore, a key proposal of FAST are compression techniques, which handles the length of string and integer keys. Finally, Chankyu Kim et al. propose four search techniques: FAST, FAST with compression, buffered scheme and sort-based scheme.

Another practice on indexing includes machine learning. Machine learrning provides various solutions for understanding the distribution of data. Kraska et al. [1] suggested learned index structures which use machine learning to predict the location of the queried keys. The basic idea is that if a machine learning model can learn the CDF of the underlying data it can directly predict the value which is connected with the queried key. However, ML models predictions are susceptible to errors and this is a challenge that occurs. The learned index structure must calculate and overcome this error with local search. Thus, if the prediction error is small enough, a learned model can be competitive in look-up time. A little different practice is suggested by Ali Hadian on Interpolation-friendly B-tress (IFB-trees) [9] . In this case, the algorithmic data structure is not fully replaced by a learned model. So, the learned index ideas can be embedded in a data structure (e.g. B-trees). IFB-trees keep the b- tree structure and aim to optimize the search performance by using machine learning. ML is used in order to minimize intra-node searching which is the main performance drawback while keeping the theoretical guarantees of B-trees.

Another approach is to fit the index on the data distribution without machine learning. FITing Tree's implementation [2] fits an index to a dataset and workload with a balance between look-up performance and space consumption. A cost model is used to determine lookup latency requirement and storage budget. Also, it uses piecewise linear functions to quickly approximate the position of an element. The contributions of [2] are (1) an index structure (2) an efficient segmentation algorithm with tunable error parameters (3) a cost model to tune the appropriate error threshold.

FIT stores only the starting key of each linear segment and the slope of the linear function in order to compute a key's approximate position using linear interpolation. Inner nodes are the same as in a B+ tree (lookup and insert operations) until the leaf level where there is need for a local search.

Another index structure can be used for the inner nodes. It also supports non clustered indexes. Adds a layer "key pages" with the sorted version of the index. In order to execute an in-place insert the trie takes into account the error threshold, shifts the values of the segment involved and re-approximates the segmentation. Finally, each segment has a buffer for extra inserts, which is taken into account when calculating the error.

Chapter 3

Pot Implementation

This chapter presents the Performance Optimized Tree (POT). We begin with describing the POT construction procedure and the algorithms for search and insertion. Next, we analyze the construction and search costs and discuss on the advantages of our approach over other indexes.

The starting point of this thesis is to solve the indexing problem of an array with sorted data in the main memory. In order to solve this problem, we design POT, a hybrid trie which supports point and range queries.

Due to room for improvement of existing structures, we outline our so-called performance optimized trie (POT) as a new in-memory data structure. It is a hydrid solution based on the idea of prefix trees (tries) for indexing arbitrary data types of fixed and variable length in the form of byte sequences. The novel characteristic of our trie-based structure is the assembly of known and new techniques. In detail, we use (1) a prefix size of variable length, (2) prefix-based trie expansion, (3) grouping of values with common prefix.

Our trie consists of two types of nodes: inner nodes, which map partial keys to

nodes and leaf nodes that point to data that we index. The representation of an inner node is an array of 2^p pointers that can either point to a leaf node or point to a child node, where p is the prefix length. We use the array of 2^p size, so that every hop in the path of finding a value will cost O(1) without comparisons. The main idea is that we can use every part of the key as an indepedent number, matching the position of the node in the array. This leads to 2^p options, thus we need an array of 2^p size in every node that is not a leaf. Each leaf node contains two pointers on the sorted array of the input data circumscribing the group of data with common prefix for which it is responsible. We define these groups of data as buckets, in which a binary search gives the key-value pair.

During tree traversal, a part of the key with determined length is used as the array position in the children array of the node and determines the next child node in the path without any comparisons. The length of the partial key used at every level, which we call prefix length, is critical for the performance of our trie, because it determines the height of the tree for a given key length. Our trie stores bit keys with length k so a path can have at maximum k/p levels of nodes. The constraint of the bucket size is directly connected to the tree height and the binary search time inside the bucket, thus is crucial for the performance of the trie.

An issue that we want to investigate further is that the data that we want to index may have many common segments in their binary representations. This leads to reserved space which is not actually used, but still gives us the advantage of hops through nodes with O(1) cost. Although we aim for performance, we cannot afford large memory space misuse. In our future work, we will deal with this problem by changing our approach for prefix handling.

3.1 POT Construction

Given an input of data as a sorted array, we construct our index tree on top of it. The indexed values are considered binary strings with size of 32 bits (e.g. 4-byte numbers). Longer elements can also be supported by our index. Then, we compute the common prefix of the whole input and exclude from indexing the bit string up to that point. For example, if the max common prefix of the whole input is the first 4-bit part, we index the 28-bit part of the values. After that, we construct the first level

of the tree which consists of the root node pointing to its children nodes, an array of 2^p size. The prefix length is tunable and predefined. Then, we iterate the input array while we compare the first segment of the data bit strings. We accumulate the group of search keys that have the same (current) prefix and then we check the size of the group. If the group of strings is smaller than the size constraint s (which is a predefined parameter), the node becomes a leaf node and points on the starting position and the ending position of the group in the data array. The node is then stored in the children array of the root node. The position of the node in the children array is calculated by converting the segment of the bitstring used as prefix to an integer. If the group of strings is bigger than s the node expands to children nodes which are represented as a next level array. In the corresponding position we store a pointer to the children array. This procedure continues recursively until the group of strings in every leaf node is smaller than s. As we can see in Figure 3.1, for the construction procedure we need the input array. We initialize the root node along with its children array. Then we group the data entries with common segment of key. The segment of key is defined as the first part of the key bits with size p. In the next level, the segment of the key is defined as the second part of the key bits with size of p, and so on. If the groups are smaller than s we create a node and we place it in the children array of the root node. Otherwise, we use the node expand function, wherein we recursively apply the same procedure for the next level of nodes.

An example of POT is illustrated in Figure 3.1. In this example, the data that we want to index are the numbers $4, 5, 7, 8, 9, 11$. The bucket size constraint s is 4. The prefix size is 2, so the size of the children array of the root node is $2^p = 4$. As described before, all the values $(4, 5, 7)$ with prefix 01 are assigned to the node in the second position $(children[1])$ of the children array. All the values $(8, 9, 11)$ with prefix 10 are assigned to the node in the third position $(children[2])$ of the children array. So, the two nodes are leaf nodes and point to the start and the end of their indexed group of values.

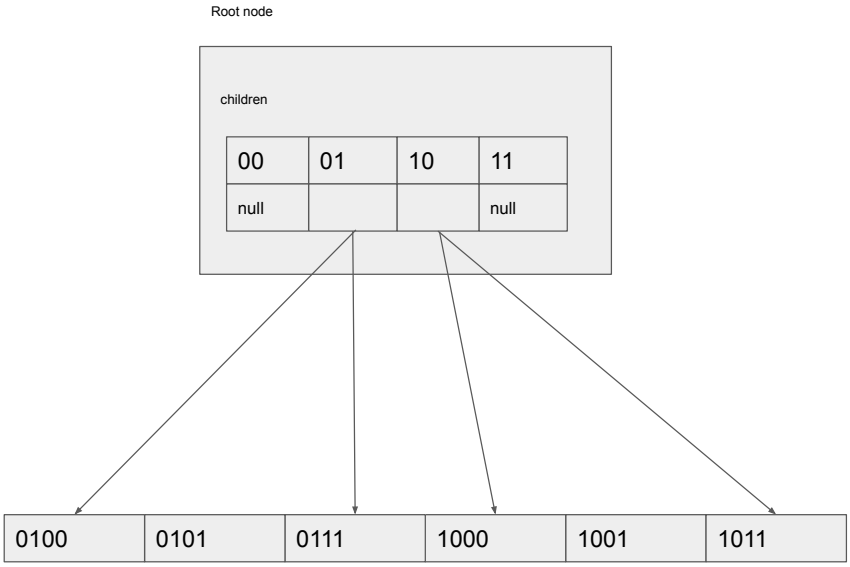

Figure 3.1: Example of a POT instance

In Figure 3.2 we observe the same example data as before, but in this case we set the constraint of the bucket size 2. This means that the nodes, which both have three entries, have to expand, in order to have a maximum of 2 entries. So, in the positions where the leaf nodes were before, now we have pointers to the next nodes which use the next segment of the data bit string to index the values. As we can see, each bucket now contains one value because of the bucket size constraint and the bit string size of the data.

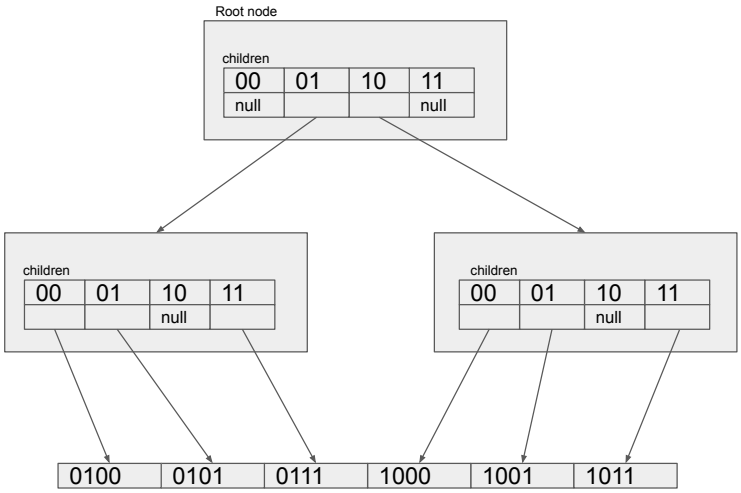

Figure 3.2: Example of a POT instance

Algorithm 3.1 Construction algorithm

Require: Input array $Input$

1: Initialize root node
2: Define children array of root node with 2^p size
3: **for** each entry $e \in Input$ **do**
4: segmentOfKey = get part of e bit string
5: **if** segmentOfKey $\neq$ segmentOfPreviousKey **then**
6: create new node n with group of entries with common segmentOfKey
7: root $\rightarrow$ children[segmentOfKey] = n
8: **else**
9: add e in the group of entries with common segmentOfKey
10: **end if**
11: **end for**
12: **for** each child f of root node **do**
13: **if** $f \rightarrow entriesgroup > s$ **then**
14: $f \rightarrow expand()$
15: **end if**
16: **end for**

3.2 POT Search

The index is designed such that search costs O(1) per level until we find the leaf which containts the searched value. The tree is traversed by using successive parts of the key as an index position until a leaf node or a null pointer is encountered. The next child node at every level is found by casting the key part used on that level to an integer, which is the position of the child node in the children index. While the searched value is not found and the bitstring has not come to an end, iteratively we select a part of the key and search the corresponding position in the array of the node at each level. If a null pointer is encountered, the searched key is not in the data index. If a leaf node is encountered, we use binary search on the data array between the two index positions which are pointed by the leaf node. The search procedure is defined in Section 3.1.

Algorithm 3.2 Search algorithm

Require: Key to search key

1: Define starting node as root
2: **while** next part of key is not last and we have not reached a leaf node **do**
3: segmentOfKey = get next part of key bit string to check
4: **if** node is leaf **then**
5: $value$ = Binary Search(node $\rightarrow$ startingPoint , node $\rightarrow$ endingPoint , key)
6: **return** $value$
7: **else if** node is null **then**
8: **return** key is not in the index
9: **end if**
10: **end while**

In Figure 3.3 we show an example of search using POT. The searched value is the number 11 and its binary representation is 1011. As the search is applied to the tree shown in Figure 3.2, the prefix size used is 2. So, we use the first two bits of the binary representation to determine the first branch to follow along the path that leads to the searched value. The first two bits of the binary representation (10) correspond to the number 2. Thus, the first node of the search path must be in the position 2 at the children array of the root node. The connection between the prefix and the children array positions gives us the opportunity to iterate through the tree without comparisons. The same procedure takes place in the next node which is a leaf node.

In the example, the bucket on which the leaf node points is of size 1, so there is no need for binary search.

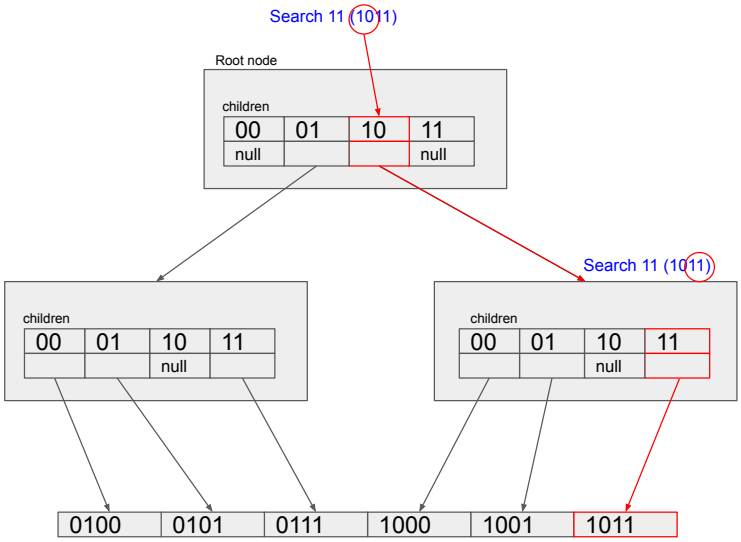

Figure 3.3: Example of a POT search

3.3 POT Construction and search complexity analysis

In this section we analyze the complexity of POT construction and search. For the construction of our trie, at first we iterate through the whole array that is given as input, which costs $O(n)$ where n is number of the entries. Then groups of entries are made, which depending on the size will make the nodes expand. In the worst case, we will need to expand nodes for all the entries until the last segment of the bit strings is checked. The maximum number of levels is $((N-p)/p)$. This leads to a complexity of $O(n * ((N-p)/p)$ where N is the length of the bit strings.

For the search of a value in our trie, the worst case is to search a value that is inserted in a bucket which is at the $(N-p)/p$-th level. Thus, $(N-p)/p)$ hops will be needed to access the bucket and then we use binary search with complexity of $O(log(s))$ where s is the maximum number of entries in a bucket. In conclusion, the complexity of a search in the worst case is $O((N-p)/p) + O(log(s))$

3.4 POT Comparative advantages

In the next chapter, we experimentally compare our approach with the Adaptive Radix Tree (ART) and the B-Tree. In comparison to the B-Tree, our approach has the advantage of a fast iteration through the levels of the index without comparisons. In other words, we can hop through our trie with a cost of $O(1)$ per hop before we find a leaf. For the same reason POT has an advantage in comparison to a simple binary search because with this practice we replace multiple comparisions in a binary search by a single hop with cost $O(1)$.

ART is not optimal in datasets of strings due to skewness and data sparsity. There are examples like a case where multiple entries with common prefixes have long paths until the leaf nodes. In such cases, our approach can perform better by adding all these entries in a bucket, minimizing the height of the path and use binary search inside the bucket.

Chapter 4

Experimental Evaluation

In this section, we experimentally evaluate POT and compare its performance to other trees. The evaluation has three parts: First, we check the performance of POT while tuning our parameters. In the second part we compare the performance of our approach to other indexing solutions and finally we evaluate the space consumption of our data structure.

We used a system with an Intel Core i5 7200U CPU which has 2 cores, 4 threads, 2.5 GHz clock rate and 3.1 GHz turbo frequency. The system has 8 GB LPDDR3 1066 RAM. We used Linux Ubuntu 18.04 in 64 bit mode as operating system and GCC 7.4 as compiler.

Bitcoin dataset: Bitcoin is a cryptocurrency, invented in 2009 by Satoshi Nakamoto [11]. His goal was to create a completely decentralized electronic cash system, which is not controlled by any central server or authority. Currency owners exchange money using anonymized (public) addresses. We extracted the history of Bitcoin payments since 2014. More specifically, we obtain the information from February 2014 to November 2014 [1].

Simply speaking, each bitcoin transaction includes a timestamp with, one or more *inputs*, and one or more *outputs*. When a user sends money to another user, he specifies in an output the address of the recipient and the amount of money to be sent. For the needs of experimental analysis, we used as data the binary representations of the timestamps. The dataset consists of 45588785 entries.

[1] https://bitcoin.org/en/

Taxi dataset: We processed trips of yellow taxis in NYC in February 2018.[2]. Each record includes the pick-up and drop-off taxi zones (regions) the date/time of the pick-up and drop-off, and the number of passengers inside the taxi. The data used, were collected and provided to the NYC Taxi and Limousine Commission (TLC) by technology providers authorized under the Taxicab and Livery Passenger Enhancement Programs (TPEP/LPEP). For the needs of our work, we used the binary representations of the timestamps. The dataset consists of 8491370 entries.

Network dataset: The CTU-13 is a dataset of botnet traffic [10] that was captured in the CTU University, Czech Republic, in 2011. The goal of the dataset was to have a large capture of real botnet traffic mixed with normal traffic and background traffic. The CTU-13 dataset consists in thirteen captures (called scenarios) of different botnet samples. On each scenario a specific malware was executed, which used several protocols and performed different actions. Each scenario was captured in a pcap file that contains all the packets of the three types of traffic. These pcap files were processed to obtain other type of information, such as NetFlows, WebLogs, etc. The first analysis of the CTU-13 dataset used unidirectional NetFlows to represent the traffic and to assign the labels. For the needs of our work, we kept the DateTime data of the dataset which we converted to binary representations. The dataset consists of 2824637 entries.

Figure 4.1 shows the data distributions of the datasets that we used to evaluate our method, as histograms, where for each range of values, the total number of key values in the range are shown.

[2]obtained from http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml

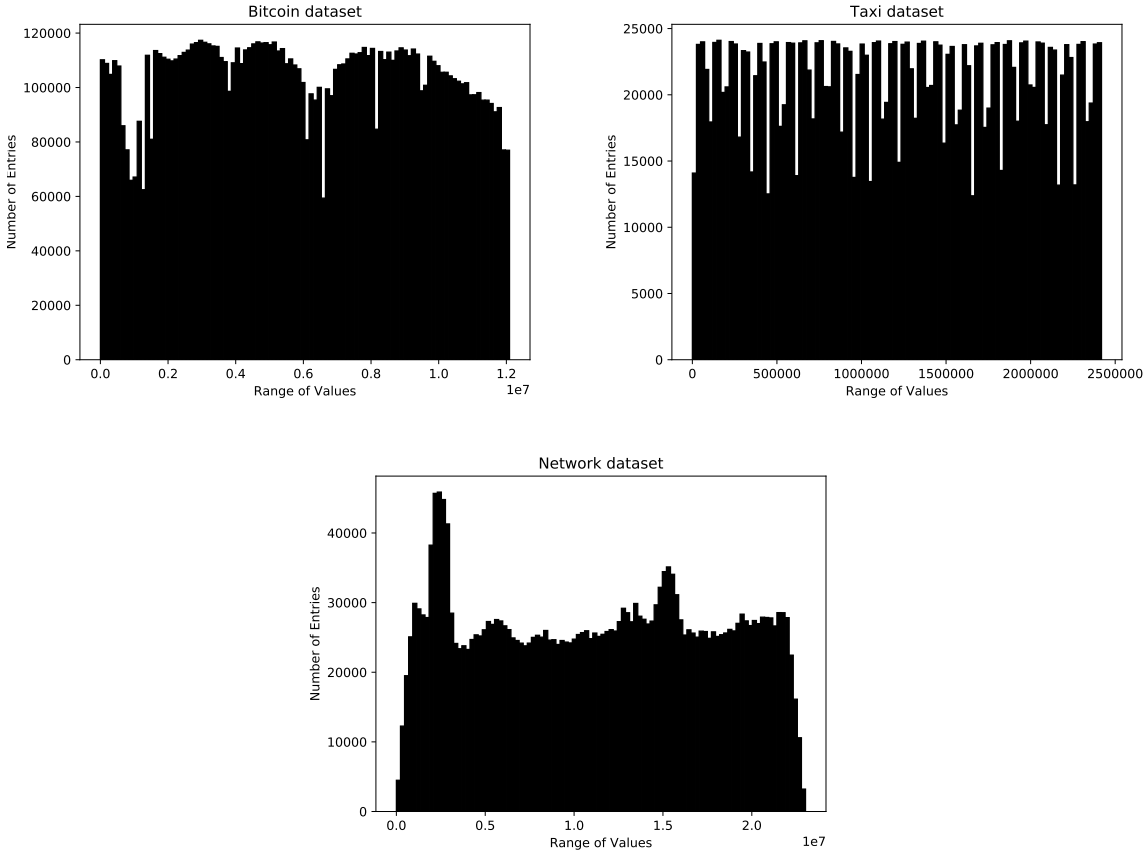

Figure 4.1: Our dataset distributions

In order to evaluate the performance of our approach we selected random samples of each dataset which were used as query values. We also created for every dataset a set of query values by adding subsets of the datasets along with query values that are not indexed. In that way we checked the behaviour of POT even with values that did not exist. Every query value set contains 10000 query values.

In Figure 4.2 we compare mean search times for a query value of different versions of POT with different bucket sizes. We use as prefix length 4 bits in this experiment. In every dataset that we used, the best search time was accomplished with bucket size 100, so we conclude that the best search time is granted with a bucket size of 100 data entries.

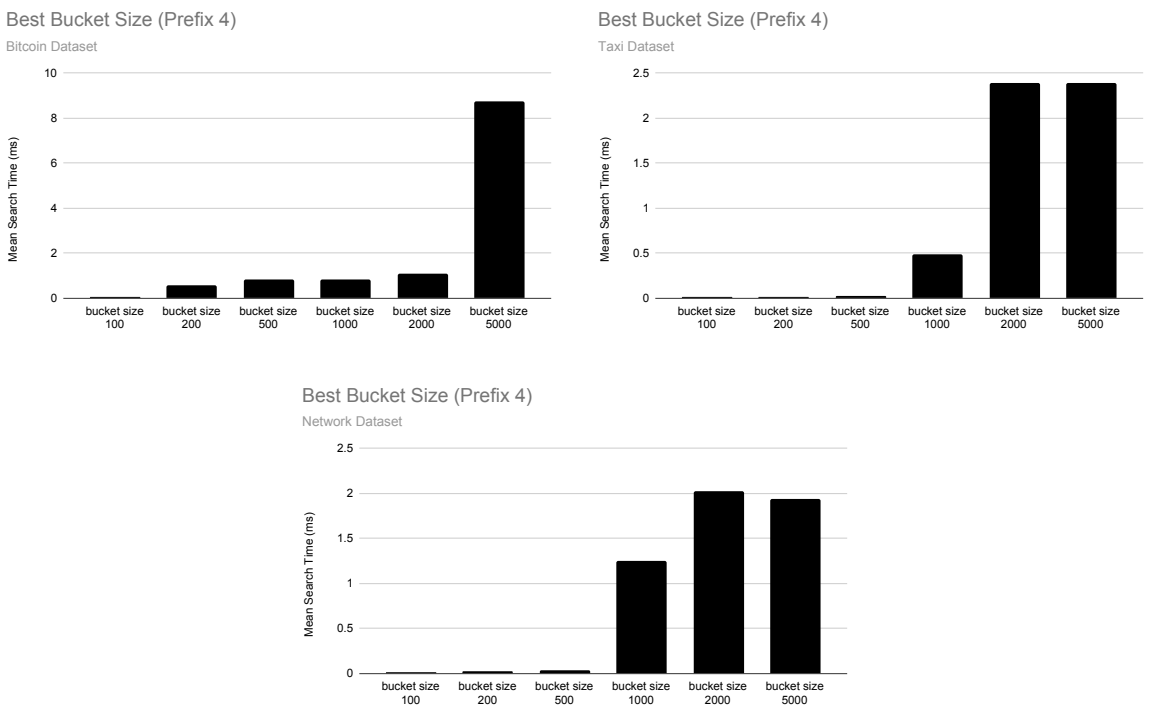

Figure 4.2: Best bucket size for Prefix 4

In Figure 4.3 we repeat the experiment using as prefix length 8 bits. We can see that in the bitcoin dataset the best bucketsize is equal to 100 entries. Then for the taxi dataset, the best performance is accomplished with a bucket size of 200, but other bucket sizes have similar performance. In conclusion, in the network dataset our method performs good with any bucket size, except the case of 2000 entries.

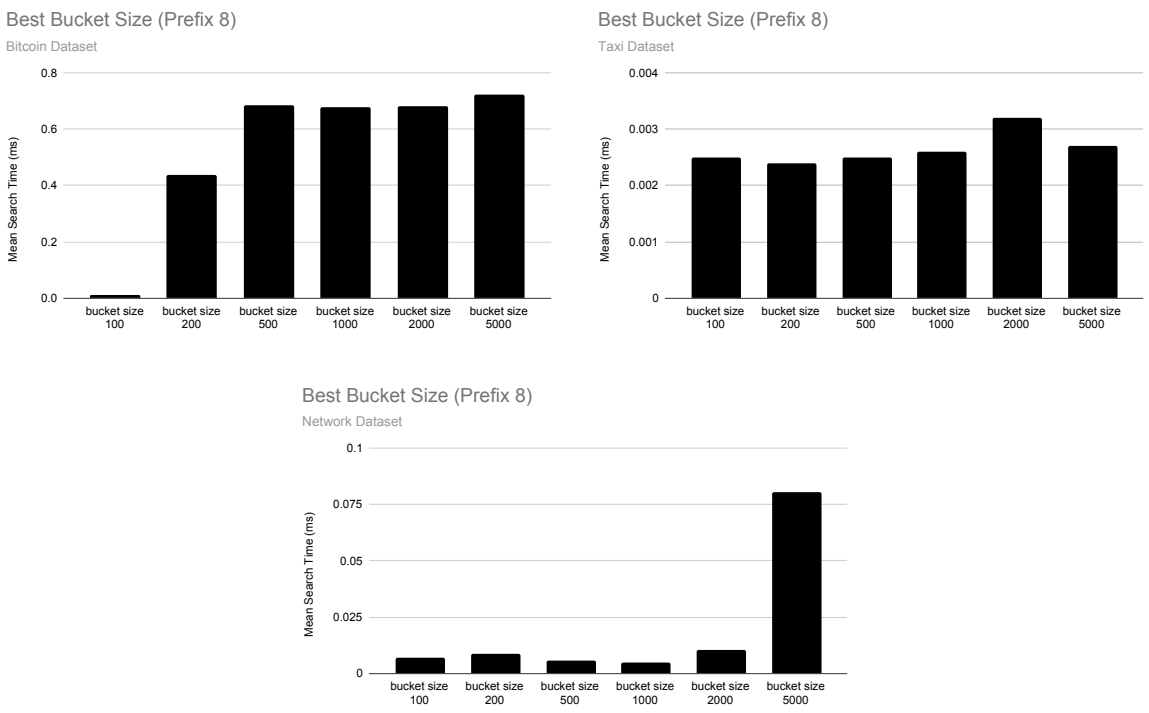

Figure 4.3: Best bucket size for Prefix 8

In Figure 4.4 we repeat the experiment using a prefix length of 16 bits. In the bitcoin dataset we see again that the best bucket is size of 100 entries, in the taxi dataset we see the same behaviour between bucket sizes of 100, 200 and 500. The same also holds for the taxi dataset. In conclusion, the best bucket size in all cases is around 100-200.

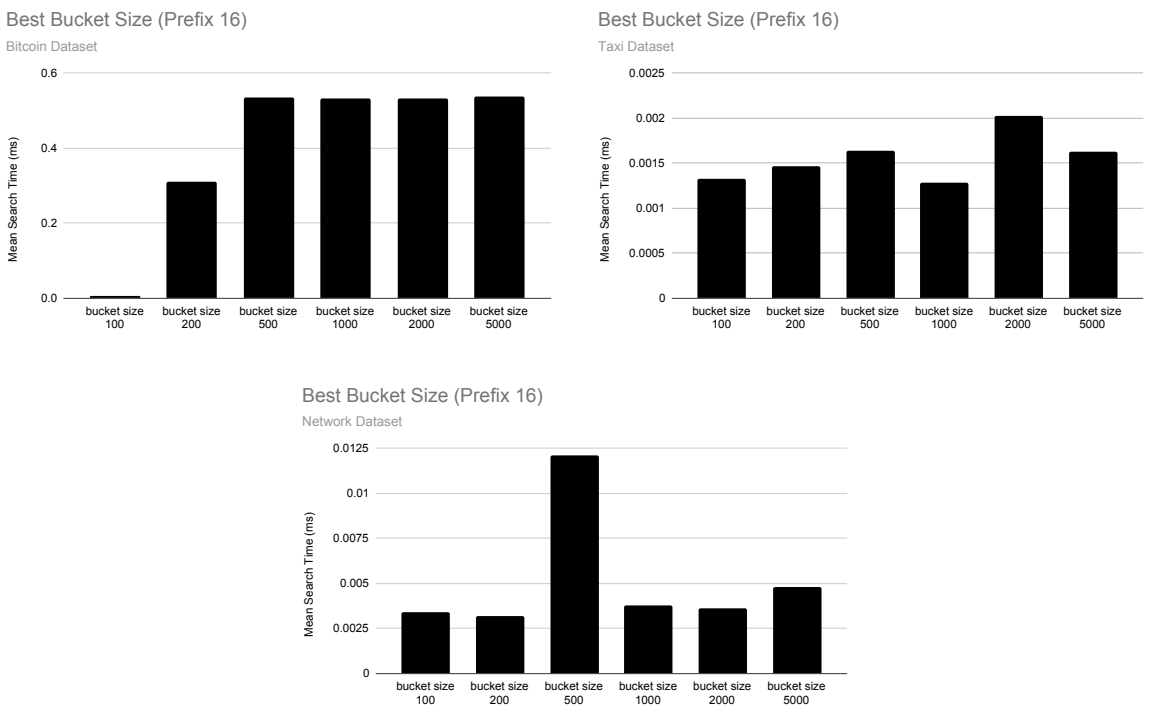

Figure 4.4: Best bucket size for Prefix 16

We can conclude that there are bucket sizes (e.g., 100) that can be applied in a general approach for every dataset. In the rest of the experiments, we set as the bucket size in each case the best possible depending on the prefix length and the dataset.

In Figure 4.5 we compare our different prefix sizes in order to conclude which one is the best in terms of search performance. As we can see, in every dataset the best search time is accomplished with a prefix size of 16. This is an outcome that we could forsee because with prefixes of 16 bits we get a tree with very few levels (usually one level). This means that we need only one hop in order to get to the indexed data. In constrast, with smaller prefix sizes we get bigger trees in height so we need more hops to get to the indexed data. On the other hand, a large prefix length leads to a larger index, as we show in the experiment that we illustrate in Figure 4.8.

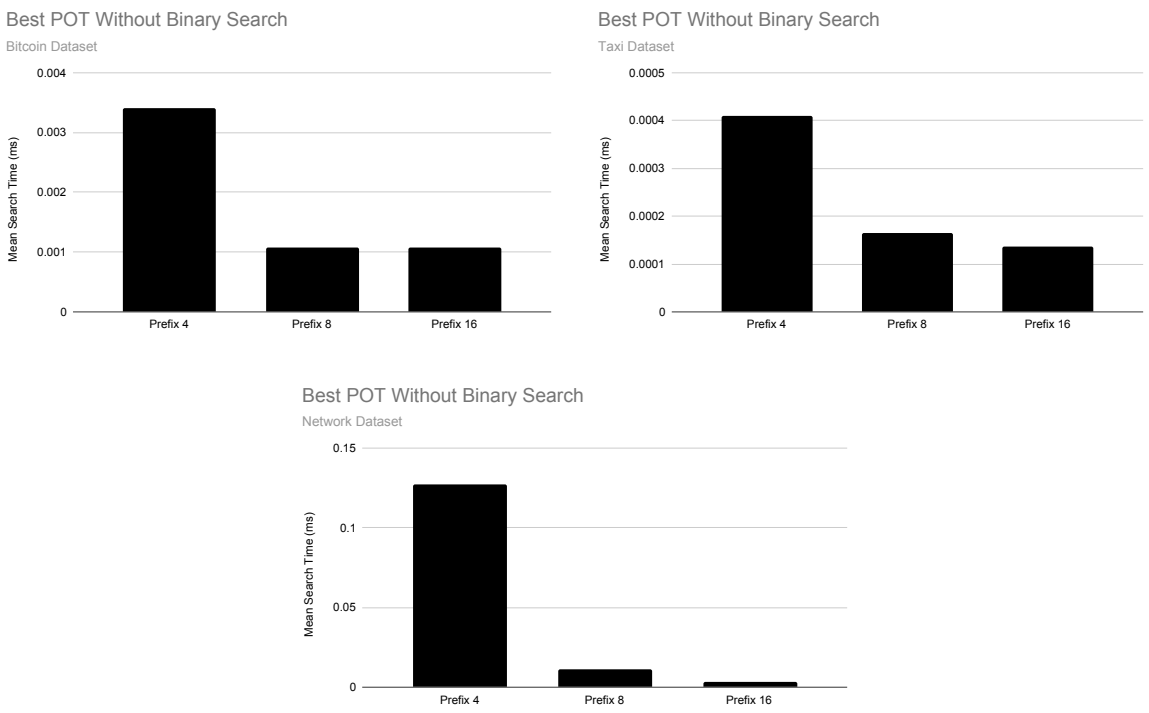

Figure 4.5: Best POT without Binary Search

The experimental results that are shown in Figures 4.6 and 4.7 compare POT to other existing solutions for indexing. Specifically, we compare our methods with the B-tree [12] and ART [4] .

Figure 4.6 compares the different implementations of POT, B-tree and ART. The queries used for testing are random thus they may contain values that are not present in the datasets. As we observe, our approach always performs better than ART and its version is always better than the B-tree. The best performance is accomplished with the POT approach without binary search which however is space-inefficient in contrast with our other approaches.

In Figure 4.7 we illustrate an experiment between our approach, B-tree and ART. The queries used for testing are randomly chosen between existing values in the datasets. As we observe, there is always at least one of our approaches that performs better than either the B-Tree and ART. As before, the best performance is accomplished with the POT approach without binary search.

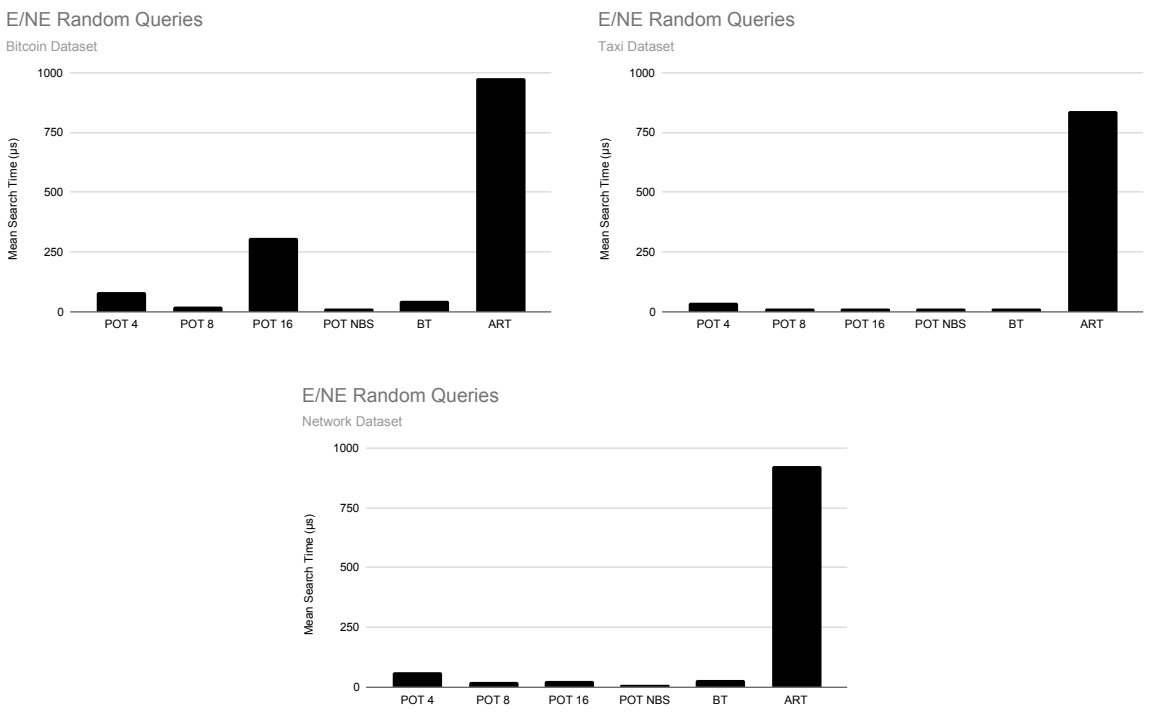

Figure 4.6: Random existing and non existing queries

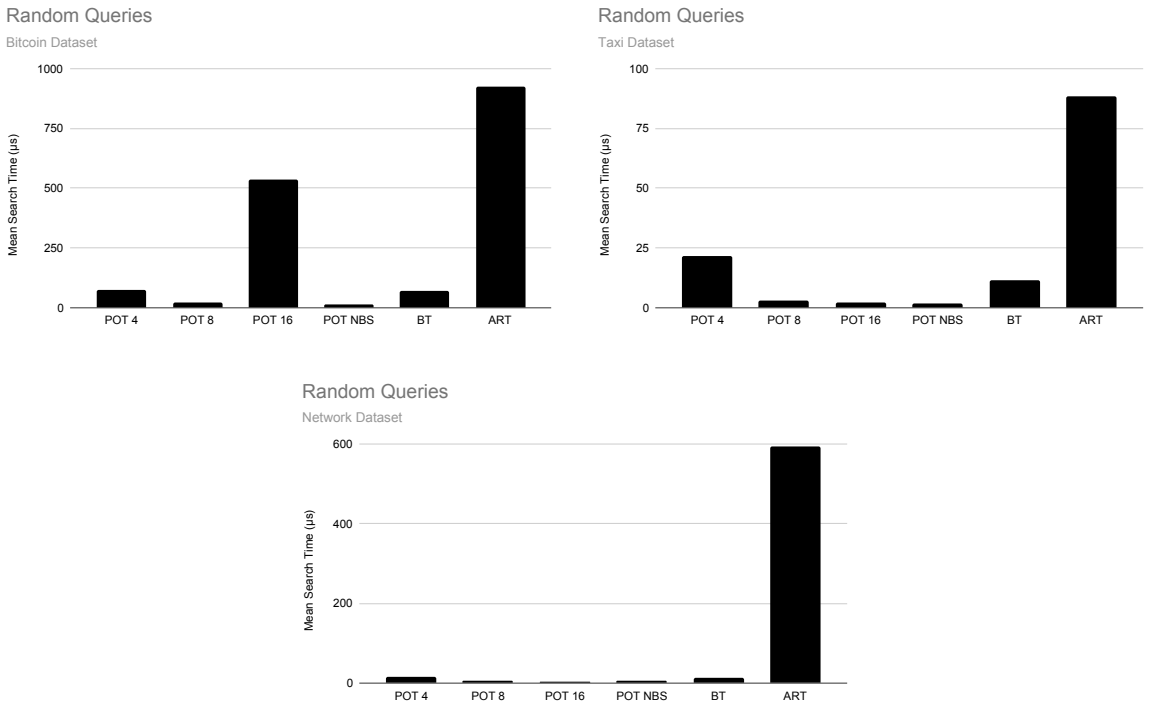

Figure 4.7: Random existing queries

In the next experiment (Figure 4.8), we compare the memory consumption of POT

with the memory consumption of B-trees. It is reasonable to expect a bigger memory consumption by POT as the prefix size rises, because the prefix size is equivalent to the size that we need to reserve in every node for children nodes (2^p) . Although the memory consumption of POT is roughly less than the B-tree with $p = 4$ and exceeds the memory consumption of the B-tree with $p = 8$. Then, with $p = 16$ the memory consumption gets reasonably bigger as expected.

In conclusion,

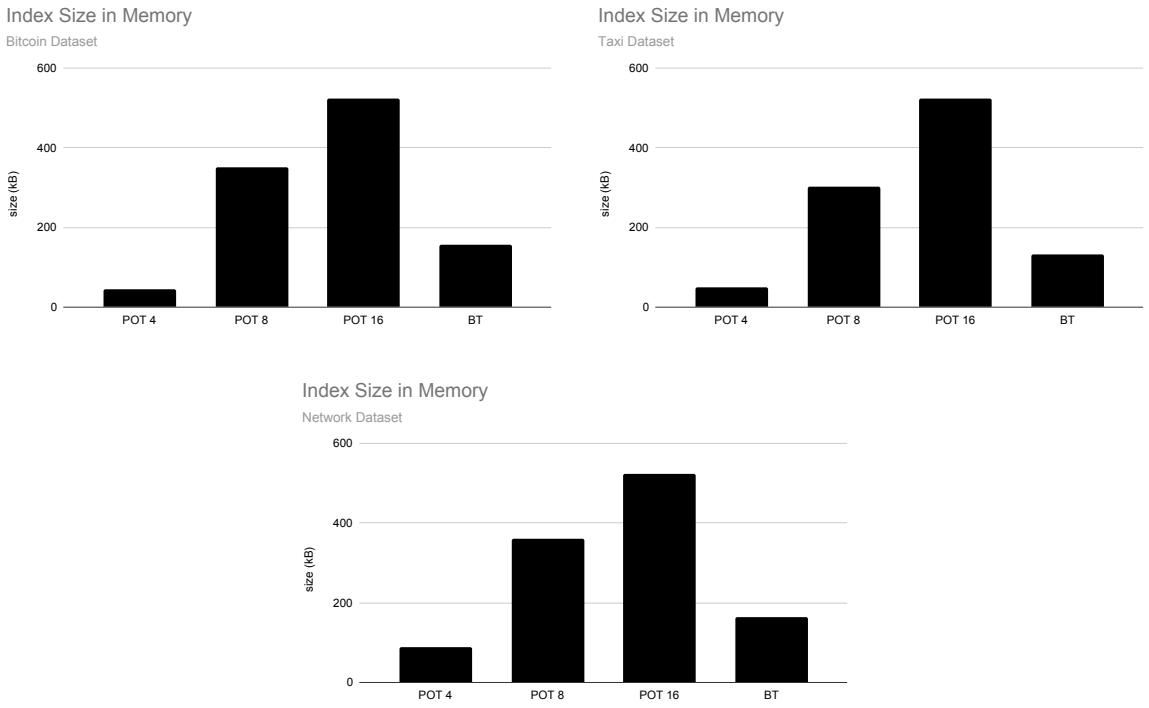

Figure 4.8: Memory size of trees

Chapter 5

Conclusions

5.1 Summary

In this thesis, we introduced POT, a novel hybrid tree, combining bucketing with data prefixes. To the best of our knowledge, we are the first to combine bucketing and data prefixes in order to create a fast in-memory index. We tuned basic parameters of the index, depending on the data size and distribution. We evaluated the performance of the tree on two real and one synthetic dataset and compared the results with other popular index structures. Our results indicated that our approach can perform much better than existing techniques.

5.2 Future Work

As future work, we plan to experiment with alternative distributions of synthetic and real networks and investigate in more depth the effect of data density and distribution in the performance of POT. That will help us tune our parameters dynamic and our tree can adapt by itself to the needs of the data that we plan to index. Also, we aim at designing a variant of POT, which supports different prefix sizes at every level. In

addition, we will work in the direction of finding an efficient way to support online inserts so that we can support dynamic data indexing. Finally, we will investigate the design of POT-like index on spatial data.

To summarize, our goals for future work is:

- We aim at supporting dynamic updates (insertions,deletions) and concurrency control.
- We plan to replace the static tuning of our parameters which are: the bucket size and the prefix length. In order to accomplish that, we need to find a rule of thumb that will apply by dynamic tuning approaches on the distributions of the datasets.
- Last but not least, we also plan to expand our work in the territory of spatial data.

Bibliography

[1] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, (New York, NY, USA), p. 489–504, Association for Computing Machinery, 2018.

[2] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "Fiting-tree: A data-aware index structure," in *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, (New York, NY, USA), p. 1189–1206, Association for Computing Machinery, 2019.

[3] X. Wu, F. Ni, and S. Jiang, "Wormhole: A fast ordered index for in-memory data management," *CoRR*, vol. abs/1805.02200, 2018.

[4] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, (USA), p. 38–49, IEEE Computer Society, 2013.

[5] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, (New York, NY, USA), p. 183–196, Association for Computing Machinery, 2012.

[6] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis, "Hot: A height optimized trie index for main-memory database systems," in *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, (New York, NY, USA), p. 521–534, Association for Computing Machinery, 2018.

[7] N. Askitis and R. Sinha, "Hat-trie: A cache-conscious trie-based data structure for strings," in *Proceedings of the Thirtieth Australasian Conference on Computer*

Science - Volume 62, ACSC '07, (AUS), p. 97–105, Australian Computer Society, Inc., 2007.

[8] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "Designing fast architecture-sensitive tree search on modern multicore/many-core processors," *ACM Trans. Database Syst.*, vol. 36, Dec. 2011.

[9] A. Hadian and T. Heinis, "Interpolation-friendly b-trees: Bridging the gap between algorithmic and learned indexes," in *EDBT*, 2019.

[10] S. García, M. Grill, J. Stiborek, and A. Zunino, "An empirical comparison of botnet detection methods," *Comput. Secur.*, vol. 45, p. 100–123, Sept. 2014.

[11] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system http://bitcoin.org/bitcoin.pdf," 2007.

[12] D. Comer, "Ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, p. 121–137, June 1979.

Short Biography

George Christodoulou was born in Kos, Greece in 1993. George received his B.Sc.degree from the CSE Department in 2017. At the same year, he became a MSc student M.Sc. student at the Department of Computer Science and Engineering (CSE) of the University of Ioannina, Greece. His research interests revolve around data management and Data Analytics (especially, time-series and network data).