Dissertation

Fast and Space-Efficient Indexing For Main-Memory Database Systems on Modern Hardware

Robert Binna

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements
for the degree of "Doctor of Philosophy (PhD)"

Advisor: Univ.-Prof. Dr. Günther Specht

Innsbruck, 2020

Abstract

The rapid progress in computer hardware and the exponential growth of main memory capacities over the last decades calls for the redesign of index structures to adapt them to today's hardware. While for traditional disk-based database systems, the ubiquitous B-tree index is predominant, it has been shown that for multi-core hardware architectures, featuring vast amounts of main memory, trie-based index structures are a viable alternative. Their major advantage is their constant time intra-node search performance. In addition to their advantages, however, tries also have a serious downside. Since they always consider a fixed range of a key's bits, tries are susceptible to non-uniformly distributed keys. In this case, trie nodes are usually sparsely filled, which leads to large tree heights and low access performance.

In this thesis, we introduce the Height Optimized Trie (HOT) that achieves consistently low tree heights by selecting the considered bits of each node individually to achieve higher average node fanouts. Therefore, we propose novel insertion and deletion algorithms that incrementally minimizes the height of the trie structure while guaranteeing a maximum fanout constraint per node. HOT structures created using this method fulfill three properties: (I) Each HOT is of minimum height, (II) its structure is deterministically defined regardless of the insertion order, and (III) each subtree itself is a deterministically defined HOT of minimum height. In this thesis, we prove that these properties hold for every HOT structure. Based on this theoretical foundation, we present seven different node layouts. While hierarchical node layouts use sophisticated encoding techniques to achieve the highest memory efficiency, linearized node layouts provide the highest access performance by exploiting the SIMD and bit manipulation instruction sets of modern hardware architectures. The results of our extensive evaluation, collected for a variety of workloads and datasets, show that our fastest node layout, the Adaptive Linearized Node Layout, outperforms other state-of-the-art index structures for string keys in both search performance and memory efficiency, while being competitive for integer keys. Our results also show that the Leaf Optimized Node Layout provides the highest memory efficiency with less than two bytes per key for all evaluated datasets. Overall, our results show that HOT is highly suitable as a general-purpose index structure and that by using different node layouts it can be tailored to a wide range of usage scenarios.

Zusammenfassung

Der rasante Fortschritt im Bereich der Hardwareentwicklung und das exponentielle Wachstum des Arbeitsspeichers im Laufe der vergangenen Jahrzehnte hat dazu geführt das Design von Indexstrukturen grundlegend zu überdenken. Während bei traditionellen plattenorientierten Datenbanksystemen B-Bäume vorherrschen, stellen Trie-basierte Indexstrukturen bei modernen Multiprozessorarchitekturen, die über enorme Hauptspeicherkapazitäten verfügen, eine geeignete Alternative dar. Der Hauptvorteil von Trie Strukturen ist ihre konstante Zugriffszeit innerhalb eines Knotens. Neben ihren Vorteilen haben existierende Tries den Nachteil, dass sie pro Knoten nur Teilschlüssel einer fixen vordefinierten Länge berücksichtigen. Dies führt dazu, dass bei ungleich verteilten Schlüsselsätzen, Knoten spärlich gefüllt sind und dadurch Baumhöhen und Zugriffszeiten ansteigen.

In dieser Arbeit stellen wir den Height Optimized Trie (HOT) vor, der die betrachteten Bits pro Knoten dynamisch wählt um einen möglichst hohen Verzweigungsgrad und damit geringe Baumhöhen und Zugriffszeiten zu erzielen. Zu diesem Zwecke stellen wir Einfüge- und Löschalgorithmem vor, die die Höhe der Trie-Struktur, unter Einhaltung eines maximalen Verzweigungsgrades pro Knoten, schrittweise minimieren. Der resultierende HOT besitzt drei Eigenschaften: (I) die Höhe jedes HOT ist minimal, (II) für den selben Datensatz besitzt jeder HOT die selbe eindeutige Struktur und (III) jeder Teilbaum eines HOT ist selbst ein HOT. Wir beweisen jede dieser drei Eigenschaften im Zuge dieser Arbeit. Aufbauend auf diesem theoretischen Fundament führen wir sieben unterschiedliche Knoten-Layouts für HOT Strukturen ein. Während hierarchische Knoten-Layouts durch ausgefeilte Kodierungstechniken eine hohe Speichereffizienz erzielen, nützen linearisierte Knoten-Layouts die SIMD-Anweisungen moderner Prozessoren um geringe Zugriffszeiten zu realisieren. Anhand der Ergebnisse einer ausführlichen Evaluierung, unter Berücksichtigung verschiedener Datensätze und Anwendungszenarien, zeigen wir, dass das Adaptive Linearised Node Layout die höchste Zugriffsleistung erreicht und andere moderne Indexstrukturen für Zeichenketten in Bezug auf Suchleistung und Speicherbedarf übertrifft, während es für ganzzahlige Schlüssel konkurrenzfähig ist. Die Ergebnisse zeigen weiters, dass das Leaf Optimized Node Layout den geringsten Speicherverbrauch aller evaluierten Indexe, mit weniger als zwei Byte pro Schlüssel, erzielt. Zusammenfassend zeigen unsere Ergebnisse, dass sich HOT als Allzweck-Indexstruktur sehr gut eignet und durch entsprechende Knoten-Layouts für unterschiedlichste Anwendungszenarien angepasst werden kann.

Acknowledgements

Writing a PhD thesis is challenging and cannot be endeavored alone. It requires the help and support of many. Therefore, I would like to thank everyone who has helped and supported me over the last twelve years. I would also like to apologize to everyone for having to reschedule or turn down activities or trips because I was busy many evenings and weekends trying to complete this work.

Wolfgang, Eva and Sarah; thanks for your friendship and for always accompanying me on this trip. Not only did we study together, but we also went on amazing trips, had long and entertaining discussions in the Treibhaus, participated in many USI courses and shared more experiences than there is room for in this chapter. You are true friends, and each of you has had a strong influence on this work and the underlying publication. Wolfgang, You were the best flatmate one could wish for. I enjoy our deep technical discussions that inspire new approaches and ideas. Several of these discussions and your tendency to always present the counterargument, however controversial, have provoked ideas that eventually paved the way for the algorithms and implementations which form the foundation of this thesis. Sarah, you always have an open ear and have always listened to my crazy informal theoretical ideas. Without your help, I would not have been able to prove the theoretical foundations of this work. Eva, almost 18 years ago, we started together at the University of Innsbruck, and since then, your friendship and support made this experience pleasanter by making the challenges easier to overcome. I don't know how many countless ways you have assisted me through our studies and research. Never have you turned down any of my countless questions and requests for help. You have helped me to organize, write and formulate my ideas in an understandable form. You have taught me that the way an idea is communicated to readers is as important as the idea itself. Without your assistance and feedback, the papers would have struggled to be published and this thesis might have not existed either.

Since research is never a solo effort, I would like to thank everyone who has worked with me during this time. Wolfi, Eva, and Dominic, I would like to thank four for starting the Spiderstore project with me. In the end and through some unforeseen twists and turns, the project led to the final HOT project. Thanks, Dominic for letting me participate in your research on spatially aware graph stores and for working with me on the DCB tree. Our late night research sessions and theoretical discussions have always been fun; producing more ideas than one could ever pursue. Martin, thank you for our countless weekend research sessions and particularly for helping last-minute with your R-skills

before the HOT paper submission. I really enjoyed our weekend research sessions in the 3S01, but also the breaks from works and our after-work runs.

I am grateful to Viktor for generously offering to help and collaborate on the HOT paper without hesitation when asked for feedback and advice. It has been a real pleasure to work with you and learn from your advice, knowledge and guidance. I also thank you for your help in the final phase of this work and for always giving valuable suggestions.

A special thanks to my advisor, Günther Specht for giving me the opportunity to work as a researcher in the DBIS research group in Innsbruck and for allowing me to pursue any research topic that came to my mind. I thank you for providing me with the resources and support to continue our research after my contract expired.

Thanks to all the other members of the DBIS research group for making my work there feel more than just a job, but also a fun, exciting and unforgettable time of my life. Thanks to Michi, Lukas, Seppi, Günter, Gabi, Peter, Rainer, Matthias, Benni, Michael, Hansi, Niko, Doris and Sylvia.

Thanks to all my friends who remained encouraging, considerate of my lack of time, listened to my rants and acted as positive distractions; despite not being directly involved with my dissertation. Many thanks to Anna, Ramona, Judith, Martin, Maggo, Herbert, Ingä, Zugi, Jakob and Steffi. A special thanks goes to my friend, trainer and sport comrade Hubert. Not only have you supported and guided me through all the sports adventures, but you were also the one who told me to refocus on the dissertation and finally bring it to an end.

My family is a very important part of my life and I would like to express my gratitude to my parents, Maria and Arno, and my sister Brigitte. They have always been there for me, no matter what happens. I am foremost grateful for their steadfast support and encouragement. When others believed that something was too difficult for me or that I wouldn't make it, they gave me confidence and showed me what I was really capable of.

Finally, I would like to thank my partner, Julia. You are the love of my life! Thank you for your continuous support, having my back and always being understanding when much of my free time was spent researching, creating prototypes, and finally writing this thesis. Thanks for motivating me in times of extreme frustration and your continued belief that I would eventually hand in this work.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht. Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

Datum	Robert Binna

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Guiding Research Questions . . . 3
- 1.3 Methodology . . . 4
- 1.4 Contributions and Published Work . . . 5
- 1.5 Thesis Outline . . . 7

2 Background and Related Work **9**
- 2.1 Formal Definition and Terminology . . . 10
- 2.2 Order-Preserving Index Structures . . . 11
  - 2.2.1 Comparison-Based Search Trees . . . 11
  - 2.2.2 Trie Structures . . . 13
  - 2.2.3 Partitioning of Tree Structures . . . 21
- 2.3 Main Memory Index Structures . . . 22
- 2.4 Synchronization Methods for Index Structures . . . 25

3 Optimizing the Height of Trie Structures **29**
- 3.1 The Height Optimized Trie . . . 29
- 3.2 Combining Binary Trie Nodes into k-Constrained Tries . . . 31
- 3.3 Structure Adaptation . . . 32
  - 3.3.1 Insertion Algorithm . . . 33
  - 3.3.2 Deletion Operation . . . 37

4 Properties of Height Optimized Tries **41**
- 4.1 Simplified HOT (sHOT) . . . 42
- 4.2 Static Minimump Height Partitioning (SMHP) . . . 45

4.3 Properties of SMHP . . . 46
4.3.1 Intrapartition Weight Function . . . 47
4.3.2 Maximum Fanout Constraint . . . 48
4.3.3 Determinism of SMHP . . . 48
4.3.4 Recursive Structure of SMHP . . . 48
4.4 Equivalence of SMHP and Simplified HOT . . . 49
4.5 Minimum Cardinality Constraint of Individual BiNodes . . . 54
4.6 Equivalence of sHOT and HOT . . . 55

5 Designing Node Structures for Height Optimized Tries 61
5.1 Overview . . . 61
5.2 Hierarchical Node Layouts . . . 63
5.2.1 Direct Node Layout . . . 65
5.2.2 Indirect Node Layout . . . 66
5.2.3 Variable Length Indirect Node Layout . . . 70
5.2.4 Node Layouts for Compacted Dense Regions . . . 74
5.3 Linearized Node Layouts . . . 84
5.3.1 Linearizing Binary Patricia Tries . . . 85
5.3.2 Fixed-sized Linearized Node Layout . . . 88
5.3.3 Adaptive Linearized Node Layout . . . 90
5.4 Size-Adaptive Nodes . . . 94

6 Synchronizing Height Optimized Tries 97
6.1 Synchronization Protocol . . . 98
6.2 Memory Reclamation . . . 100

7 Evaluation and Results 103
7.1 Experimental Setup . . . 104
7.2 Height Reduction . . . 105
7.3 Node Layouts . . . 108
7.3.1 Hierarchical Node Layouts . . . 110
7.3.2 Linearized Node Layouts . . . 113
7.3.3 Overall Node Layout Comparison . . . 116
7.4 YCSB - Single-Threaded Workloads . . . 121
7.5 Scalability . . . 125

8 Conclusion 129

Bibliography 133

CHAPTER 1

Introduction

1.1 Motivation

In the middle of the 1970s, the first generation of commercial database systems became available. These systems were based on IBM's research on System R [10]. Four decades have passed since then, the amount of digital data available has grown exponentially and the hardware architecture of nowadays computers is fundamentally different from those 40 years ago. However, the core architecture of database systems is still the same.

The design decisions that have been taken in terms of storage, locking, or logging four decades ago, lead to severe performance bottlenecks in nowadays systems. [64, 56]. However specialized systems designed for contemporary hardware may outperform the prevailing one-size-fits-all-approach of traditional database systems by up to an order of magnitude [104]. This insight inspired database researches to revise and rewrite the architecture of database systems as a whole [105].

For instance, the buffer manager, which enables databases to support larger than memory workloads, is responsible for about 35% of the executed CPU

instructions in traditional database systems [56]. As even the largest enterprise datasets nowadays can entirely be kept in RAM [88, 105], a major design decision for new database systems is to completely remove the buffer manager. As in such systems, the entire working dataset is kept in main memory and secondary storage is only used to ensure durability, these systems are subsumed as main memory databases [45, 38].

This shift to main memory architectures impacts the performance and characteristics of database systems in many aspects. For many workloads, the overall performance of main memory database systems depends on fast index structures. Therefore our focus in this thesis is on the aspect of index structures in main memory databases. Specifically, we are interested in order-preserving index structures, which support not only lookup operations but also range scans and related operations.

While, for disk-based database systems, the primary goal for index structures is to reduce the number of disk accesses, for main memory database systems the primary goal is to reduce the number of cache misses [52]. For disk-based database systems, this problem was solved by the prevailing family of B-tree index structures [30]. Their balanced height in combinations with their page-aligned node structure provides satisfying and predictable access performance as well as space consumption. However, simply adapting the size of nodes to match the size of cache lines is not sufficient as the space required to store the key information on a per-node basis limits the number of keys that can be compared in a single memory access. At the same time, applying expensive compression techniques on the key information might even decrease performance, if decompressing the keys takes longer than fetching further cache lines [116].

Therefore, modern in-memory systems (e.g., Silo [110] or HyPer [66]) use trie structures (e.g., Masstree [82] or ART [75]) for indexing. One of the reasons that well-engineered tries often outperform comparison-based structures like B-trees [116, 75, 24, 8] is that trie structures only consider a small range of the stored keys' bits for a single node, which results in two beneficial properties. First, storing only a limited amount of information per key reduces the total amount of memory fetched for each node. Thus, by fetching fewer cache lines per node, higher cache efficiency for intra node search is achieved. Second, implementations using a fixed number of bits per key allow to directly address the matching entry. Hence, by avoiding expensive key comparisons potential branch mispredictions are omitted [75].

Nevertheless, even recent trie proposals have weaknesses that preclude optimal performance and space consumption. For example, while ART [75] can achieve a high fanout and therefore, high performance on integers, its average fanout

is much lower when indexing strings. This lower fanout usually occurs at lower levels of the tree and is caused by sparse key distributions that are prevalent in string keys. Besides that non-uniformly sparsely distributed keys negatively impact access performance, they also negatively impact the space efficiency of trie based index structures. While having space-efficient index structures is likewise relevant for disk-based index structures it is even more important for main memory database systems as a large fraction of the total amount of main memory available is often occupied by indexes [116].

In this work, we therefore aim for a solution for main memory indexing that fosters the beneficial aspects of existing trie structures, while alleviating the weaknesses of common main memory trie structures regarding non-uniformly sparsely distributed keys. While existing solutions, statically define the span that represents the number of key bits used per node on a global level, our proposed solution called the Height Optimized Trie (HOT) [22] dynamically adapts to different datasets and arbitrary key distributions. To achieve this goal, the key idea of HOT is to combine multiple nodes of a binary trie into a single node until a previously defined maximum fanout is reached. Thus, instead of increasing the span on a global level to implicitly reduce the overall tree height, HOT explicitly defines a maximum fanout and rearranges the underlying binary nodes accordingly.

1.2 Guiding Research Questions

To address the challenge of creating a trie based index structure providing consistently high performance and low memory consumption regardless of the actual distribution of the stored keys, we tackle the following research questions in this thesis.

RQ1 How can trie based index structures be designed to adapt to different key distributions while maintaining a low overall tree height?

RQ2 How can such a trie-based index structure be implemented to meet the requirements of fast access performance and low memory consumption?

RQ3 How can such a trie based index structure be synchronized to provide high scalability in a multithreaded environment.

To answer *RQ1* we search for an alternative method to increase a node's fanout on a global level to reduce the overall tree height and thereby improve the overall access performance. More specifically in *RQ1*, we aim for

a theoretical framework of how such an index structure looks like, what its properties are, and how and its basic operations search, lookup and deletion work. In *RQ2* the question arises whether an efficient implementation in terms of performance and memory consumption for the proposed theoretical framework exists. *RQ3* builts upon the foundations of *RQ1* and *RQ2* by analyzing whether and how the chosen approach can be extended to provide high scalability. Its goal, therefore, serves two different purposes. First, we are interested in whether a synchronization protocol can be designed which is based on the logical structure developed for *RQ1*. Second, we are interested in, how the designed synchronization protocol can be applied to the result of *RQ2* such that high scalability as well as good performance can be provided.

1.3 Methodology

The research throughout this thesis is conducted by using an iterative approach, consisting of the following individual steps *Literature Review*, *Data Structure and Algorithm Design*, *Prototype Implementation*, and *Evaluation*. The iterative nature of this approach stems from the fact that results and insights of later steps are fed back to enhance the developed algorithms and improve the overall research results. In the following, we describe each of the involved individual steps in more detail.

Literature Review: The initial step applied is a literature review. By collecting and reviewing prior scientific publications it is used to gather the current state-of-the-art regarding each of the individual research questions. The thereby collected knowledge has a twofold purpose. First, it acts as the basis and foundation of the algorithms and structure developed throughout this thesis. Second, it is required to put the results of this thesis in the context of prior research and to select meaningful approaches as evaluation candidates to benchmark the systems developed in this thesis.

Data Structure and Algorithm Design: Based on the previously conducted literature review and aligned with the research questions *RQ1* and *RQ2* posed in Section 1.2, algorithms and index structures are designed. Proofs are used to verify the height-reducing effect of the designed algorithms and structures required by *RQ1*. In addition, for each of the designed algorithms and index structures, prototypes are developed to evaluate the practical characteristics of the algorithm. The results of the evaluation are fed back into the algorithm design to refine the algorithms with respect to the original research questions.

Prototype Implementation: In our research, prototype implementation serves two different purposes. On the one hand, as stated previously, it is applied to validate and refine the developed algorithms and structures. On the other hand, finding efficient implementations is an independent research objective on its own, which in this thesis is used to address research questions *RQ2* and *RQ3*. Additionally, prototypical implementations support the discovery of shortcomings in the initially designed structures and algorithms and lead to further refinements of the chosen approaches.

Evaluation: To evaluate the effectiveness of the designed algorithm and the efficiency of the developed prototypes, various experiments are conducted. To evaluate the effectivity of the data structures and algorithms regarding their postulated properties, the structures are intrinsically evaluated. To evaluate the efficiency of the chosen implementations addressing *RQ2* and *RQ3*, experiments are conducted and the results are compared against different implementation strategies as well as against other state-of-the-art index structures. In accordance with the iterative nature of our research, the results of the evaluations are directly used to improve the algorithms and their corresponding implementations.

1.4 Contributions and Published Work

Although the core concept of this PhD thesis, the Height Optimized Trie [22] is covered by a single publication, several peer-reviewed publications in the field of graph stores and data structures have finally let to the research questions posed in Section 1.2 and to the findings presented in this thesis. In the following, we list all contributions, which have been published in the context of my PhD studies.

In the field of graph stores, we published two papers [19, 20] that introduce the concept of a native main memory graph store called *Spiderstore*.

- R. Binna, W. Gassler, E. Zangerle, D. Pacher, and G. Specht. SpiderStore: Exploiting Main Memory for Efficient RDF Graph Representation and Fast Querying. In *Proc. Workshop on Semantic Data Management (SemData) at VLDB*, 2010.

- R. Binna, W. Gassler, E. Zangerle, D. Pacher, and G. Specht. Spiderstore: A native main memory approach for graph storage. In *Proceedings*

of the 23rd GI-Workshop "Grundlagen von Datenbanken 2011", Obergurgl, Austria, May 31 - June 03, 2011, pages 91–96, 2011.

We further conducted research on providing data locality in graph stores. This research led to three publications [89, 90, 91] and the concept of a *spatially aware graph store*, which solves the problem of data locality in graph stores by organizing graph information in a highly parallel spatial computer.

- D. Pacher, R. Binna, and G. Specht. Data locality in graph databases through n-body simulation. In *Proceedings of the 23rd GI-Workshop "Grundlagen von Datenbanken 2011", Obergurgl, Austria, May 31 - June 03, 2011*, pages 85–90, 2011.

- D. Pacher, R. Binna, and G. Specht. Graph stores based on spatial computers. In *Proceedings of the 7th Spatial Computing Workshop (SCW 2014)*, pages 1–6, 2014.

- D. Pacher, R. Binna, and G. Specht. Optimizing large knowledge networks in spatial computers. *Knowledge Engineering Review*, 31(4):367–390, 2016.

Our research on main memory graph stores led us to the field of index structures. Our initial attempt to find an efficient index for the set of triples resulted in the DCB-tree [21], a highly customized index structure for storing short fixed-sized integer keys. The initial experience gained with the *DCB-tree* in the field of main memory index structures finally resulted in the general-purpose *Height Optimized Trie (HOT)* index. HOT is the main focus of this PhD thesis and represents a novel approach to improve trie structures as index structures for main memory database systems.

- R. Binna, D. Pacher, T. Meindl, and G. Specht. The DCB-tree: A space-efficient delta coded cache conscious B-tree. In *Memory Data Management and Analysis - First and Second International Workshops, IMDM 2013, Riva del Garda, Italy, August 26, 2013, IMDM 2014, Hangzhou, China, September 1, 2014, Revised Selected Papers*, pages 126–138, 2014.

- 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 ACM International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA*. ACM, 2018.

In addition to the research done in the fields of graph storage and index structures, a contribution was made within the context of bioinformatics [70], which is outside the scope of this thesis.

- A. Kloss-Brandstätter, D. Pacher, S. Schönherr, H. Weissensteiner, R. Binna, G. Specht, and F. Kronenberg. Haplogrep: a fast and reliable algorithm for automatic classification of mitochondrial dna haplogroups. *Human Mutation*, 32(1):25–32, 2011.

1.5 Thesis Outline

The remainder of this PhD thesis is structured as follows. Chapter 2 covers related work and background in the fields of main memory index structures and binary tree partitioning. In Chapter 3 we introduce the Height Optimized Trie *HOT* and explain its underlying algorithms in detail. In Chapter 4 we prove three theoretical properties of HOT: *Determinism*, *Minimum Height* and *Recursive Structure*. Chapter 5 introduces seven different node layouts for HOT and describes their structures and algorithms. Chapter 6 introduces a new ROWEX-based synchronization protocol for HOT and other copy-on-write based index structures and discusses its implementation details for HOT with respect to memory reclamation. In Chapter 7, HOT is extensively evaluated against three other state-of-the-art index structures on four different data sets in terms of height reduction, access performance, memory consumption, and scalability. Chapter 8 concludes this PhD thesis and gives an outlook on possible future research directions.

CHAPTER 2

Background and Related Work

Providing fast and flexible access to data is one of the key characteristics of any database management system. Depending on the amount of data stored, organizing the data in an arbitrary order and scanning the whole database on each access is therefore not feasible. To reduce the number of expensive scan operations and to speed up lookup operations, auxiliary data structures, so-called index structures, are used. The cost to achieve this improvement is the space required to store the index structure as well as the additional maintenance overhead to keep the index structure in sync whenever the underlying data is updated.

The characteristics of the used index structures, heavily influence the performance and space consumption of a database system. Consequently, several approaches have been developed over the last five decades to satisfy this demand. These approaches can be categorized by the following aspects. On the one hand, these approaches differ in their behavior, and therefore, in the type of operations, they support and accelerate. On the other hand, different implementation strategies have been developed to improve properties like performance, maintenance cost, or the space required to store these structures. As these properties mutually depend on each other, improving a single one might not always improve the other one. For instance, applying compression

algorithms on an index structure might trade improved memory consumptions against lower access performance. Therefore, specialized index structures optimized and tailored to specific usage scenarios as well as general-purpose index structures achieving acceptable performance regardless of the actual data distribution, workload, or usage scenario have been developed.

The remainder of this chapter is structured as follows. We first give a more formal definition of index structures. Then we focus on order-preserving index structures. In the domain of order-preserving index structures, we first give a short background on comparison-based indices and will then discuss trie-based index structures and high-level optimization techniques to make trie index structures more predictable in terms of access performance and space consumption for general-purpose workloads. Additionally, we discuss work in the field of tree partitioning as the specific trie optimization presented in this thesis is closely related to the partitioning of tree structures. After having discussed order-preserving index structures in general, we focus on the field of main memory index structures. We discuss what led to the development of main memory index structures, how they differ from disk-based index structures, and which optimizations are used in this domain. Finally, we conclude this chapter with an overview of synchronization methods for tree-based index structures.

2.1 Formal Definition and Terminology

An index structure is a collection of index entries, where each index entry consists of a key and an associated value. In the context of relational databases, these collections are used to improve the access to individual relations and therefore, each index entry corresponds to a single tuple of a single or joined relation. Thereby, the key corresponds to one or multiple fields of this tuple, and the tuple itself is either directly embedded in the index entry or referenced via a database-internal reference. These database-internal references are called tuple identifiers (TID) and provide fast means to access the actual tuple.

Depending on the supported operations of the underlying collections, we distinguish two main categories of index structures: *order-preserving* index structures and *non-order-preserving* index structures. While order-preserving index structures allow to look up a range of keys and iterate those entries in the order of the keys, non-order-preserving index structures are restricted to point queries for exact matches on keys. Typical representatives of order-preserving index structures are tree structures having a time complexity of $\mathcal{O}(log(n))$. Typical representatives for non-order-preserving index structures are hash ta-

bles having time complexity $\mathcal{O}(1)$. Due to the different access characteristics, choosing one index structure over the other highly depends on the anticipated query workload. Therefore, for each combination of fields to index, the type of index structure to use is chosen separately. In this work, we are specifically interested in the category of order-preserving index structures.

As previously explained, an indexed tuple can either be embedded right in the index entry or referenced via a tuple identifier. If the tuples are embedded in the index entries and are physically stored close to each other and ordered by the indexed field(s), the index is called a *clustered* index. Therefore, only one single clustered index can be defined per single relation. In contrast, *non-clustered* indexes use TIDs to identify the tuples in the index entries. Additionally, non-clustered indexes are denoted as *secondary* indexes, as they can be interpreted as a secondary means to access a relation's tuples.

2.2 Order-Preserving Index Structures

As stated in Section 2.1, in the course of this thesis we are interested in order-preserving index structures. More specifically, our main focus is on order-preserving index structures for main memory. Therefore, in this section, we will first give an overview of previous work on order-preserving index structures in general and will then provide an extensive discussion on order-preserving main memory index structures in Section 2.3. For simplicity's sake, in the remainder of this thesis, we will use the terms index structures and order-preserving index structures synonymously.

Generally, two major approaches exist to create order-preserving index structures. The first approach is based on a pair-wise comparison of the stored entries' keys. Various binary- and multiway search trees fall into this category. The second approach facilitates the binary representation of the stored keys and index structures using this "digital search" principle belong to the family of trie-based index structures. In the following, we discuss each of these categories separately.

2.2.1 Comparison-Based Search Trees

The most basic approach for **comparison-based search trees** is the family of binary search trees. They have been developed independently by several people during the 1950s [36]. While providing good access performance for data inserted in random order, binary search trees degrade into linked lists

for data inserted in key order [33]. Balanced trees—limiting the height difference between left and right subtrees—solve this problem. To incrementally create and maintain such balanced tree structures, different index structures have been invented, which apply compensation operations, whenever the balance conditions are violated. Among these approaches are the AVL-trees by Adelson-Velsky and Landis [2], the 2-3 tree by Hopcroft [12], or the Symmetric Binary B-trees trees by Bayer et al. [12], which are commonly known as red-black trees [53].

While balanced binary trees provide logarithmic insertion as well as access performance, they are not practically useful for block-oriented storage devices, which have high latencies and transfer data in larger chunks. Therefore, these structures are not used in traditional database systems. The reason is that traversing a binary tree might issue one additional read access to the storage device for each intermediate node. To reduce the number of these expensive storage operations, block-oriented multiway tree structures, with more than one child per node were proposed. The initial approaches by Landauer et al. [72] and Muntz et. al [85] have been superseded by the B-tree by Bayer and McCreight [13]. B-trees are balanced, multiway search trees that execute the primitive index operations in logarithmic time. The fanout of B-tree nodes is optimized for nodes to match the block sizes of the respective storage device and hence, to reduce the overall tree height and thereby to reduce overall access latency. Therefore, for disk-based database systems, variations of B-tree-based index structures are preferred [12, 30].

Its most well-known variation is the B^+-tree [36, 113, 30]. In contrast to the original B-tree, the B^+-tree stores the actual entries at the leaf level only. Additionally, to improve scan performance the B^+-tree interlinks nodes at the leaf level thereby separating the indexing part from the actual sequence of entries. Besides its use case as an order-preserving index structure for database systems [86, 44, 29, 108, 109], B^+-trees are used as a foundation for modern journaling file systems [97, 107, 93, 96]. Parallel to the B^+-tree, several more variations of the B-tree have been developed [30]. The B^* tree [36] uses local key redistribution to ensure that storage utilization is at least 66%. The Prefix B-tree [15] extends the B^+ tree by applying prefix and rear compression on keys in the index part to improve space utilization and increase the fanout of intermediate nodes.

Based on these basic variants, several optimization techniques have been proposed which optimize the intra-node architecture (e.g. [7, 6]) and intra-node search (e.g. [80, 106, 40]) to achieve better space utilization and higher performance. To further improve B-trees for modern hardware, optimizations have been proposed which increase cache efficiency, memory consumption, and access performance. As the majority of these optimizations are not specific to

comparison-based structures, we will discuss these optimizations separately in Section 2.3, which covers main memory index structures in general.

2.2.2 Trie Structures

In contrast to comparison-based tree structures where each node divides the search space by the natural order of the stored keys, tries are tree structures where all descendants of a node share a common prefix and the children of a node are searched using the remaining characters. Tries were originally developed by De La Briandais [34] in 1959. The term trie was coined by Fredkin [43] and derived from "information retrieval" in 1960. As the binary alphabet is the simplest alphabet, the most basic trie structure is a binary trie (also called 1-bit trie), in which each node corresponds to a bit position. This bit position is equal to the distance from the root node and in the course of search operations, the bit at the corresponding position in the search key determines whether to proceed in the left or right subtree. A major advantage of tries in contrast to comparison-based search trees is that by exploiting the binary structure of the stored keys, trie structures have a deterministic layout, which is independent of the insertion order of the stored keys.

However, rooted in the late 1950s, the main use of tries has primarily been in the domain of text retrieval for a long period in time. Another specialized domain where tries are preferable is the domain of address lookup for IP routing, where fast prefix matching is required [98]. However, the developments in hardware over the last decade have renewed the interest of database researches on trie-based index structures. More specifically, the growth of main memory capacities has led to the development of index structures that are optimized for in-memory workloads (e.g., [95, 28, 118, 100, 67, 78, 115]). Tries, in particular, have proven to be highly efficient for arbitrary main memory workloads [24, 8, 69, 82, 75, 116].

While binary tries are conceptually simple, they do not perform well due to their large tree heights (e.g., a height of 32 for 4-byte integers). Hence, substantial research has been done in this area to mitigate the problem of large trie heights. In the following, we discuss different optimizations of trie-based index structures. The most relevant approaches are illustrated with a common example storing 13 keys, all of which are 9 bits long. In Figures 2.1-2.7 compound nodes are surrounded by solid lines and are colored according to their level in the respective tree structure. Dots in theses figures represent either leaf values or vertices corresponding to bit positions, which are used to distinguish between different keys. The simple binary trie is depicted in Figure 2.1. It clearly shows the main drawback of unoptimized binary trie structures.

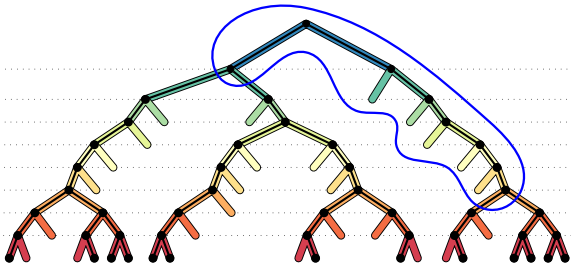


Figure 2.1: Binary trie (height=9, #nodes=37).

Even though only 13 keys are stored, the binary trie structure requires 9 levels (in correspondence with the length of the stored keys) and therefore is only sparsely filled. In contrast, a densely packed tree structure would require less intermediate nodes and would therefore provide faster access times and reduced memory consumption. The non-branching intermediate nodes, which are framed blue in Figure 2.1, illustrate the problem of sparsely populated trees, which impact search performance and memory consumption.

The Patricia trie [84] mitigates these large tree heights and wasted space by omitting all non-branching intermediate nodes with only one child[1]. As the Patricia optimization shortens paths by eliminating these non-branching nodes, the Patricia optimization is also referred to as path compression. The binary Patricia trie for our common example is shown in Figure 2.2. An example of omitted non-branching nodes is highlighted in blue and corresponds to the highlighted nodes in Figure 2.1. The resulting structure of a Patricia trie resembles a full binary tree, where each node either is a leaf node or has exactly two children. This structure of a full binary tree has the major advantage that the number of branching nodes can directly be inferred from the number of keys stored. Hence, also the amount of memory required can directly be determined from the number of entries stored. While the Patricia optimization often reduces the tree height (e.g., from nine to five in our example), the small fanout of two still yields large tree heights.

To increase the fanout and thereby reduce the overall height of the tree structure, several techniques have been developed. All these techniques have in common that they consider more than one key bit per node. This is analogous to using a different alphabet for the underlying trie structure. The special case of alphanumeric characters was already proposed in the original trie paper

[1] As a result of the Patricia optimization, keys are not necessarily stored fully in the trie and every key must therefore be available at its corresponding leaf node. For main memory database systems, this is usually the case because the leaf node will store a reference to the full tuple (including the key).

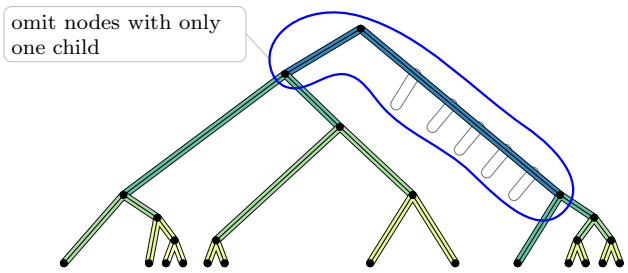


Figure 2.2: Patricia trie, 1 bit span (height=5, #nodes=12).

of Fredkin et al. [43]. Depending on the field of research, the number of bits considered per node is either called stride (e.g. [103, 55]) or span (e.g. [75, 22]). For simplicity reasons, we will only use the term *span* in the remainder of this thesis. In the following, we divide the approaches for increasing the span and optimizations based on this in two categories. The first category includes fixed span approaches that determine the span on a global level independent of the actual dataset stored. The second category includes variable span approaches that adapt the span to the actual data and their distribution.

Fixed Span Approaches The simplest form of a fixed span is to consider a fixed number s of bits at each node regardless of the actual data stored. This reduces the tree height by a factor of s in comparison to a binary trie. For a given span s, this is generally implemented using an array of 2^s pointers in each node. An example containing our 9 sample keys and a span of 3 bits is illustrated in Figure 2.3. Examples of approaches using a fixed span are for instance the initially developed trie structures for storing ASCII strings [34, 43] or lately the Generalized Prefix Tree [24] using a span of 4 bits. Although considering multiple bits per nodes effectively reduces the tree height by a constant factor, the downside of a larger span is wasted space for sparsely distributed keys (e.g., long strings) as most of the pointers in the nodes will be empty and the actual fanout is typically much smaller than the optimum (2^s). The resulting tree structure, therefore, remains vulnerable to large tree heights and wasted space. Therefore, choosing a fixed span is similar to trading speed for memory consumption. Depending on the actual data distribution, small spans may lead to efficient memory consumption at the cost of unsatisfying access performance, whereas large spans may lead to faster access performance at the cost of sparse node utilization and therefore, wasted space.

The problem of a non-suitable span can be observed in Figure 2.3, which depicts a trie with a span of 3 bits.

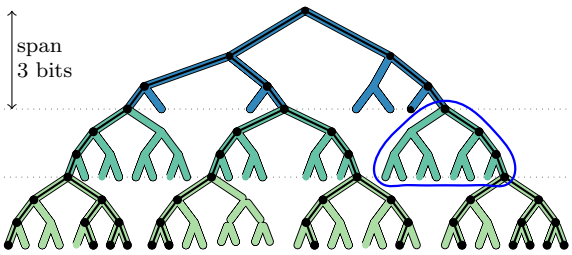


Figure 2.3: fixed span trie, 3 bit span (height=3, #nodes=8).

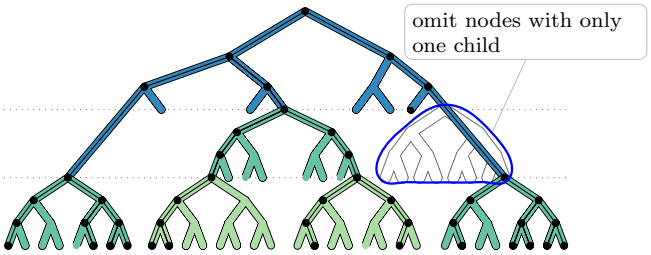


Figure 2.4: Patricia trie, 3 bit span (height=3, #nodes=6).

Although a fixed span reduces the height of the tree in comparison to the binary trie, its average fanout is still only 2.5, which is considerably smaller than the optimum fanout of eight. Also note that, while the Patricia optimization can be applied to tries with a larger span, it becomes less effective (though may still be worthwhile). As Figure 2.4 shows, when applied to the trie depicted in Figure 2.3 with a span of three bits, the Patricia optimization saves only two nodes and does not reduce the maximum tree height. Interestingly, the fact that none of Patricia's space guarantees holds in the case of a non-binary alphabet was already discussed in the original PATRICIA paper [84]. Nevertheless, Patricia's path compression—which omits all non-branching nodes—is also used (e.g. [5, 75]) in combination with spans larger than two.

Alternative Node Structures One effective method to address the shortcomings of larger spans in terms of memory consumption is to use other data structures to represent nodes instead of plain arrays. Examples of such alternative node structures are linked lists in the original works of De La Briandais [34] or binary search trees in the case of the ternary search trie [17]. By heavily making use of pointers, both approaches sacrifice locality and trade memory consumptions against access performance. Additionally, the 64-bit pointer-size on contemporary hardware architectures diminishes the space savings of these node types. A more recent approach, which uses an alternative node

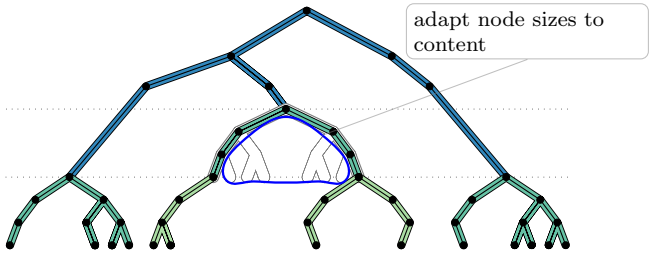


Figure 2.5: patricia trie, 3 bit span, adaptive nodes (height=3, #nodes=6).

type to support even larger spans is Masstree [82], which uses a fixed span of 64 bits and B-trees as internal node structure. In comparison to linked list or binary search trees as internal node structures, Masstree ensures better locality by exploiting the block-oriented nature of B-trees as internal node structures. However, all three approaches solve the sparsity problem at the cost of relying more heavily on comparison-based search, which due to branch mispredictions is often slower than the bitwise trie search.

Adaptive Node Types An alternative to using the same data structure independently of the actual fanout for each node is to dynamically adapt the node structure to the actual fanout of the respective node. Acharaya et al. [1] describe an approach where they use four completely different data structures (partitioned-array, B-tree, hashtable, and vector) for individual nodes depending on cache characteristics and the actual fanout. A more recent approach is the Adaptive Radix Tree (ART) [75]. It uses a fixed span of 8 bits but avoids wasting space by dynamically choosing more compact node representations (instead of always using an array of 256 pointers). ART's optimized node representations avoid branch mispredictions and exploit SIMD to achieve search performance for individual nodes which are comparable to array-based trie nodes. Hence, adaptive nodes reduce memory consumption and enable the use of a larger span, which increases performance through better cache efficiency. However, even with a large span of 8 bits, sparsely distributed keys result in many nodes with a small fanout at lower levels of the tree. The concept of adaptive nodes is depicted in Figure 2.5, which adds adaptive nodes to the trie of Figure 2.4. It clearly shows that using adaptive nodes is able to decrease the memory consumption in case of sparsely distributed data. However, it also shows that, in our example, adaptive nodes do not have an impact on the effective node fanout and the overall tree height.

Compressed Nodes An alternative way to solve the problem of sparsely distributed nodes is to only use a single node type but to apply compression methods on the nodes' content. For instance, the Lulea scheme by Degermark et al. [35] as well as the Tree Bitmap by Eatherton et al. [37] combine a fixed span with compressed nodes. Both algorithms are specifically designed for IP-routing and use bitmaps to encode the nodes' internal tree structure.

Different Spans per Level Instead of using different node types or compression on the node's content, it can be useful to statically define different spans for different levels of the tree. Depending on the actual domain, root nodes tend to be more densely filled than leaf nodes and a larger initial span can be beneficial. Such an initial larger span is often used in tries for IP-routing (e.g. [35, 37]). The KISS-Tree by Kissinger et al. [69], which is fixed to three levels uses a different span for each level (Level 1: 16bit, Level 2: 10bit and Level 3: 6bit).

Having surveyed the different fixed span approaches to reduce the overall tree height of trie-based index structure, we conclude that all optimizations discussed so far combine multiple nodes of a binary trie into a compound node structure by a fixed factor, such that the height of the resulting structure is reduced and the average node fanout is increased. Moreover, these approaches choose the criteria to combine multiple binary trie nodes independently of the data stored, namely the span representing the bits considered per node. Therefore, the resulting fanout, memory consumption, and access performance heavily depend on the data actually stored.

Variable Span Approaches While a predefined fixed span may optimize the tree height, space consumption, and access performance for a certain dataset, the same configuration may yield insufficient memory consumption or access performance or both for another dataset. Variable span approaches try to tackle this challenge by adapting the trie configuration to the actual data stored.

Static Variable Span The simplest form of a variable span approach is to statically choose a global configuration for the entire trie structure depending on the dataset stored. One such approach is the Expanded Tries by Srinivasan et al. [103]. In their Expanded Tries, they use dynamic programming to calculate the optimal spans for a predefined number of levels l and a given dataset. They further suggest to extend their approach by optimizing each subtree individually, such that the depth for each leaf node remains l, but

nodes on the same level may use different spans. However, they do not elaborate on the details of the latter algorithm. Although defining the maximum number of levels upfront allows for optimizing the span on a per-node-level, it is not sufficient for a generic approach, which is able to deal with arbitrary dataset sizes.

Hybrid Tries Another family of variable span approaches, which can either be interpreted as hybrid trie structures or as "lightweight variations" of dynamic spans are Burst-Tries [58], B-Tries [9], and HAT-Tries [8]. These structures are two-level data structures consisting of an upper part and a lower part. While the upper part is a fixed span trie, the lower part is based on non-trie data structures. Initially, all three structures start with a single element container. Only when a certain fill level is reached, the lower level container is burst into several smaller sized containers, all of which share a common prefix and a new upper-level trie node, which addresses each of these lower-level containers. While Burst-Tries use rather cache-inefficient linked lists as lower-level containers and B-Tries are optimized using disks as their storage layer, HAT-Tries use a cache-conscious structure to ensure high performance on modern hardware. Although this two-level technique achieves high performance for string workloads, it is limited in terms of memory consumption and access performance in case of integer- or densely distributed keys, as it only optimizes the lower levels of the underlying trie structure.

Level Compressed Tries A more fine-grained variable span trie is the Level Compressed Trie (LC-trie) by Andersson et al. [5], which uses different spans for each node. Its level compression combines dense areas of a binary trie structure by traversing the tree in breadth-first order and merging fully occupied nodes with their descendants until a level containing a single non-branching node s is reached. An example of such a level compressed trie is shown in Figure 2.6. In this figure, the dense areas, forming compound nodes are enclosed by a blue rectangle.

To combine descendant vertices with a common ancestor, level compressed tries require that all descending vertices at the same level must have two children. Hence, a single non-branching node in an otherwise densely populated area is sufficient that vertices at this level and all their descendants are not merged into a common compound node. To mitigate this drawback of level compressed tries Nilsson et al. [86] propose the LPC-trie, which extends LC-tries with a more lenient threshold defining whether level compression can be applied or not. Even though LPC tries are able to reduce the overall height in comparison to a binary Patricia trie, the LPC trie is not able to effec-

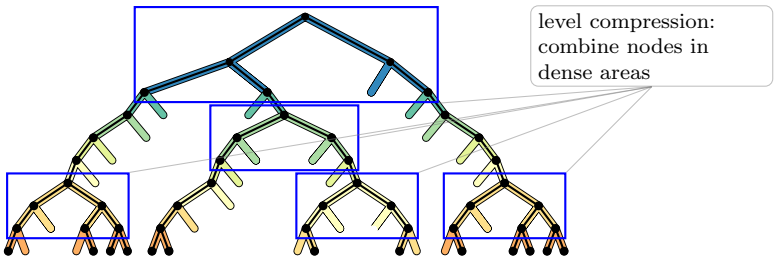


Figure 2.6: LC-Trie (height=7, #nodes=27).

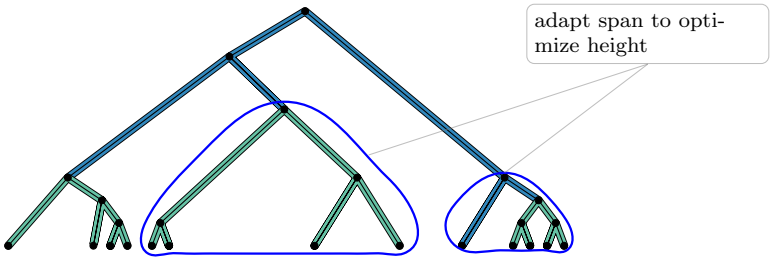


Figure 2.7: HOT with maximum fanout k=4 (height=2, #nodes=4).

tively reduce the height of sparse areas in the underlying binary Patricia trie. Furthermore, according to Nilsson et al. [87] it is not clear how worst-case cascading updates can entirely be prevented in an efficient update strategy.

Static Partitioning In contrast, the Shape Shifting Trie (SST) [55] uses a more holistic approach for trie optimization. The SST combines individual nodes of an underlying binary trie such that all compound nodes conform to a predefined maximum node fanout and the induced tree structure defined over the compound nodes is of minimum height. However, the algorithm used by SST is only applicable to static datasets and is not able to apply incremental updates without deteriorating the minimum height property. Therefore, the SST must periodically be rebuilt to retain minimum height. As the SST does not apply path compression it is limited to short keys, like in the domain of IP route lookup, which is the intended use case of SST [55].

Therefore, to the best of our knowledge, our proposed solution HOT is the first trie-based index structure, which supports a variable span with a minimum height guarantee in combination with path compression and a maximum node fanout, while supporting inserts and deletion in $\mathcal{O}(log(n))$.

2.2.3 Partitioning of Tree Structures

The key idea of the Height Optimized Trie is to combine multiples nodes of a binary Patricia trie into a compound node such that the overall height is optimized and the maximum fanout per compound node is restricted to a factor k . This problem is equivalent to finding a partitioning of minimum height on the same binary Patricia trie, such that each partition has no more than k entries. Besides our goal to reduce the height of index structures for databases, this problem is relevant in several different domains. Other applications for minimum height partitioning, besides our intended use as an index structure, are the optimization of planning and scheduling processes [111], efficient access to network routing tables [55], or delay reduction for FPGA mapping [31]. Depending on the respective problem domains, different solutions for minimum height partitioning have been proposed.

One of the initial attempts to partition tree structures such that the partitioning is of minimum height was the shifting algorithm of Becker and Perl [16]. While it is able to create a partitioning of minimal height, it requires to define the number of partitions upfront. For an arbitrary number of partitions and adhering to a size constraint k, Kovács and Kis [71] define a static algorithm that for weighted tree structures produces a minimal height partitioning in the runtime of $\mathcal{O}(n)$. This algorithm traverses a weighted tree structure in post-order, labels each vertex with a level and the weight of the maximal subtree rooted a the respective vertex, and assigns connected nodes with the same level to the same partition. In the same publication, they also present an algorithm that produces an optimal height partitioning of minimal cardinality. Another method to statically find a minimum height partitioning is Breadth-First Pruning as proposed by Song et al. [55] on their work on the shape shifting tries, which is used for IP route lookup. Similar to the approach by Kovács and Kis [71], Breath-First Pruning is a static algorithm that operates on binary trees. The algorithm first labels all vertexes v with the number of their respective descendants $s(v)$. It then traverses the labeled tree in breath-first order. Whenever it encounters a node that is labeled with $s(v) < k$, it prunes the subtree rooted at v from the overall structure and forms a new compound node consisting of v and its descendants. Next, it updates the labels of the ancestor nodes of v. The algorithm repeats the tree traversals until all vertices are pruned. The overall runtime of Breath-First Pruning is $\mathcal{O}(n^2)$ for a single static dataset. In the domain of FPGA mapping, the problem of finding a mapping of minimum delay from a (statically defined) boolean network onto k-constrained lookup tables (K-LUTs) is similar to finding a partitioning of minimum height [55]. However, as boolean networks represent arbitrary directed acyclic graphs finding a mapping of minimum depth solves a more general problem, than finding a partitioning of minimum height

on tree structures. Nevertheless, it was shown by the FlowMap algorithm of Cong et al. [31] that this problem can optimally be solved in polynomial time. Their suggested approach uses an efficient algorithm to find minimum height K-feasible cuts in a network. However, several approaches exist which partition static tree structures such that the overall height is minimized, to the best of our knowledge no algorithm exists which is able to incrementally maintain a partitioning of minimum height while inserting or deleting elements in the tree.

2.3 Main Memory Index Structures

With the increasing amount of main memory available, even large enterprise datasets can entirely be kept in RAM [92]. Previously, databases were optimized to reduce the number of hard disk accesses. As accessing RAM is several orders of magnitudes faster than accessing a hard disk, storing the whole database in RAM instead of disk significantly increases the overall system performance [104]. However, database architectures originally designed for disk-oriented usage, are a hindrance to achieving even higher performance. Therefore, it is necessary to overthink database design as a whole [105]. One of the key components affected by this paradigm shift are index structures. The B-tree variations, which are the prevailing index structures [30] for disk-based database systems, are designed to reduce the number of expensive disk seeks.

Initial approaches of main memory index structures like the T-Tree [74] heavily rely on the assumption that all main memory accesses have a uniform cost. However, this is not the case as the rate of improvement of main memory performance has not kept pace with the increasing CPU performance. If this disparity keeps growing, at some point in the future the system performance will completely be dominated by main memory access performance. To compensate for the speed disparity and delay this specific moment in time when the so-called "Memory Wall" [114] is hit, multi-layer cache hierarchies are used [101]. Whenever the CPU requests data from the main memory that is not already resident in the CPU's caches, the data is transferred block-wise from main memory into these caches. These blocks are called cache-lines and have a typical size of 64 bytes on modern hardware. While fetching data that is already resident in the first level cache takes about 1ns, fetching an uncached reference directly from DRAM costs about 100ns. Hence, designing main memory index structures such that the raw number of CPU instructions are optimized—as it was the case for early main memory index structures like the T-Tree—is not enough. Therefore, to utilize these caches a cache-conscious design of data structures, which exploit temporal and spatial

memory locality is required [4]. From a bird's eye view, overcoming the disparity between CPU speed and memory access latency is similar to bridging the gap between main memory and disk access latencies [50]. It is therefore obvious to apply techniques similar to the techniques used to reduce the amount of disk seeks for disk-based database systems. Index structures like the CSS-tree [94] or the cache-conscious B-tree (CSB^+) [95] are, therefore, variations of the B-tree optimized for main memory workloads. A survey on different high-level techniques to make B-tree structures more cache-conscious was conducted by Graefe et al. [50]. However, the majority of these techniques are also applicable for other types of index structures and therefore, also on tries. These techniques include using cache lines as atomic building blocks for nodes instead of disk pages, aligning these nodes on cache line boundaries, packing more information into a single cache line, and reorganizing the node structure. Almost every index structure dedicated to main memory workloads applies the first two optimization techniques. Even normal B^+-trees that use cache-aligned nodes, spanning one or multiple cache lines are therefore able to provide better performance than the T-Tree or binary search trees [94].

To pack more key information into a single cache line, it is beneficial to separate key information and pointers into two separate arrays (e.g., [75, 18, 82]). Structures like the Cache Sensitive Search Tree (CSS) [94], the Cache Sensitive B^+-tree (CSB^+-tree) [95], FAST [67], or the KISS-tree [69] omit pointers by serializing parts or the whole tree structure. To compress the pointers themselves is another approach. The KISS-tree [69] uses so-called compact pointers, which requires 32 instead of 64 bits per pointer. The DCB-tree [21] uses a custom memory allocator and is, therefore, able to use node IDs instead of pointers. Additionally, these smaller node IDs are adaptively encoded such that the amount of space required for these node IDs is minimized. Another way to save memory is to reduce the amount of wasted space per node. Zhang et al. [116] apply such optimizations on B-trees, tries, and Masstree. They show that a combination of compact versions of these structures for cold data and their normal variations for hot data is able to improve memory consumption and access-performance. By using succinct data structures [63], which have space requirements close to the information-theoretical optimum, the amount of space required can be improved even further. Recently, the static read-only index structure SuRF [117] has shown that succinct representations can outperform pointer-based structures also in terms of access performance. Besides using succinct data structures or compacting wasted space, another approach to increase the amount of key information per cache line is to compress the actual key information used to distinguish individual entries of a node. While a page fault results in an access latency of several milliseconds, a cache miss takes about 100 nanoseconds. Hence, while more expensive compression methods can improve access performance for disk-based index structures, the same compression methods can be too expensive for main

memory index structures [116, 46, 62]. Therefore, main memory databases primarily use light-weight compression techniques, like dictionary compression, prefix compression, tail compression, or delta compression [50, 23, 21]. Besides improving the node fanout, compression methods can be exploited to improve the number of elements that can be processed by a single SIMD instruction. For instance, FAST [67] extends prefix compression to generate 32-bit wide partial keys, which contain only those bits that are required to distinguish the keys stored within the same node. Thus, for intra-node operations, SIMD operations on a vector of 32-bit elements can be applied. This is similar to the partial keys in our HOT-index structure. The major drawbacks of partial keys as used by FAST are that groups of four bits instead of individual bits are used to distinguish individual entries. In the worst case, for each group, three out of four bits are identical for all keys within the same node. Due to the partial key length of 32 bits, only 8 bits are used to distinguish a node's entries in this scenario. This can lead to partial keys, which are not able to uniquely distinguish all entries within the same node. Such non-unique partial keys may then result in potential false-positive matches when traversing the tree. In contrast, for HOT we use a fine-grained scheme to generate partial keys, which is based on discriminative bits and prevents false positives while creating shorter partial keys.

To further improve the performance and space consumption of main memory index structures it is often necessary to completely update the node layout or to use a different search strategy. Restructuring the node layout allows for the optimization of intra-node search performance by reducing the two major sources of CPU stalls on contemporary hardware, namely branch mispredictions and cache misses. Node structures spanning several cache lines and using binary search are prone to both types of CPU stalls [118]. For typical binary search access patterns, the CPU is unable to predict whether to proceed search left or right and will, therefore, in 50% of all cases trigger a cache miss [118]. Thus, the CPU is also not able to predict which cache line will be accessed next and will therefore yield a cache miss whenever a new cache line is touched. Depending on the node structure and the applied search strategy, only one or both types of CPU stalls can be prevented. For instance, probing result candidates in parallel SIMD-based search strategies [100, 118] can drastically reduce branch-mispredictions. FAST [67] improves on SIMD-based search by using a strategy called hierarchical blocking, which splits a cache line into multiple chunks (each the size of a single SIMD register), such that the search result of the first chunk determines which chunk within the same cache line to process next. FAST further suggests combining prefetching with software pipelining to prevent cache misses when fetching the next cache line and fully utilize the available CPU cycles. Another method to improve intra-node performance is to directly calculate the potential location of the entry corresponding to the search key instead of using comparison-based search approaches. Exam-

ples of such strategies are interpolation search and searches in array-based trie nodes [47, 50]. While for linearly distributed data, interpolation search is able to improve performance, for non linearly distributed data, an increased number of false-positive matches can lead to performance worse than binary search [94]. In contrast, searching in a typical array-based trie node eliminates false positives, as either the chosen result candidate matches or no matching result is available. This efficient search strategy is the primary reason why trie-based index structures, despite their worst-case tree height and worst-case space consumption, have shown to achieve superior performance for main memory workloads [24]. By using different node types depending on the actual filling degree as shown by the Adaptive Radix Tree (ART) [75], the number of cache misses can be reduced even further. To maintain the advantage of trie-based search, ART uses SIMD as well as sophisticated encoding strategies for the node types that have a lower fanout. However, the performance of this sophisticated intra-node search is limited by the overall tree height in case of sparse non-uniformly distributed data. As each tree level may trigger both a branch misprediction as well as a cache miss, the access performance degrades with large tree heights. In contrast, our approach HOT prevents this performance degradation, by using a minimal height partitioning algorithm to create compound nodes with a high fanout and a sophisticated node layout. This sophisticated node layout combines a compact linearized binary Patricia trie with a SIMD-based search.

2.4 Synchronization Methods for Index Structures

Besides making use of multi-layer cache hierarchies, modern CPUs contain multiple cores to increase the computational power as CPU clock-frequency has stagnated [3]. Hence, featuring a cache-conscious data layout is not enough. To exploit the computational power provided by today's multi-core CPUs, an efficient synchronization protocol is required. Similar to main memory optimizations, most synchronization strategies for tree-based index structures can be applied to comparison-based as well as trie-based index structures. To distinguish low-level locking primitives—guarding physical structures against conflicting accesses by multiple threads—from logical locks—guarding individual database items against conflicting access by multiple users—we use the well-established term *latch* [51, 48] in the remainder for low-level locking primitives. One well-known synchronization strategy for hierarchical index structures is *lock-coupling* or more precisely *latch-coupling* [99, 102]. Depending on the type of operation, it starts with a shared or exclusive latch on the root node. In the course of the traversal of the tree structure, it latches all descendants. For read operations, it releases the latch of the parent node, whenever the current node has been latched.

For write operations whenever a safe node—a node, which is guaranteed to not be modified—is reached, the latches on the ancestor nodes are released. Bayer and Schkolnick improved [14] on basic latch coupling by introducing the more fine-grained intention latches IS, IX, and SIX and an optimistic two-pass approach, which is optimized for the common case where the leaf node affected by the insertion operation is safe. However, in the pessimistic case, all nodes, which are updated must be latched at the same time. Therefore, several approaches have been developed, which split update operations into multiple shorter operations to support more fine-grained latching strategies. The family of top-down algorithms [102] makes use of so-called preparatory updates, which split or merge nodes whenever they encounter an unsafe node. Another alternative for fine-grained updates is the B-link tree [73], which latches only one node at a time. To lock only a single node at a time, the B-Link tree introduces a link pointer between neighboring nodes on the same level and executes a node split in two so-called half splits. First, the original node is split by adding its right half entries into a new node and linked to the original node. Secondly, a link to the newly created node is added to the original node's parent.

All these previously mentioned fine-grained latching techniques have in common that they are designed for disk-based systems. For disk-based database systems, the cost of acquiring a latch is negligible in comparison to the IO-cost of a single disk access. In contrast, in the context of main memory database systems, the cost of acquiring a latch is expensive. Whenever multiple threads access the same latch, it results in contention, cache invalidation, and cache misses and therefore limits concurrency, Hence, synchronization protocols using fine-grained latching like (lock coupling) do not scale [27].

To reduce the synchronization overhead of fine-grained latching strategies, several approaches have been developed that reduce the amount of latching required. The most extreme approach in this regard is to use latch-free data structures. These data structures use atomic operations without any latching to update the structure. To support concurrent accesses, latch-free data structures use sophisticated protocols to ensure that the data structure is always in a consistent state and that it is guaranteed that concurrent threads always make progress. Examples of latch-free index structures are the BW-tree [78, 112] or latch-free skip lists [41]. While latch-free skip lists are easy to implement, they suffer from bad access locality due to the heavy use of pointer chasing. On the other side, the BW-tree is a B-tree variant and hence achieves good memory locality. It combines the use of append-only delta updates, a mapping table to indirect pointers, and fine-grained structure modification operations (SMO) to iteratively update the structure such that each intermediate step results in a well-formed structure.

However, latch-based approaches, which sparingly use fine-grained latches for write operations, achieve scalability on the same level as latch-free structures [77, 27, 25, 82]. The concurrent AVL-Tree [25], Masstree [82], OLFIT [27], and Optimistic Lock Coupling [77] use optimistic version checks instead of latches for read operations. These optimistic version-checks verify whether the version numbers of accessed nodes have changed during the current read operation. In case the version number has changed, an intermediate modification is detected and the operation restarts. An improved variation of using optimistic read operations is used by the FOEDUS system [68]. For each node, it checks whether the actual key range of a single node corresponds to the metadata of this node. If the key range does not match, it requires a local restart at the inconsistent node. A method entirely overcoming the need to restart read operations, is the Read-Optimized Write EXclusion (ROWEX) [77]. It uses latching only for write operations and ensures that updates always keep the index structure in a consistent state. Hence, readers can operate entirely latch-free and never need to restart. In contrast to Optimistic Lock Coupling, which is a generic synchronization protocol and therefore applicable to many different hierarchical index structures, ROWEX is a paradigm with a specific complex implementation for ART [75, 77], which requires changes to the underlying structure.

From a theoretical point of view, latch-free index structures are superior to index structures using latches as they are guaranteed to always make progress and are invulnerable to starvation. In practice, this is actually not an issue. Faleiro et al. [39] show, that carefully crafted index structures making sparingly use of fine-grained latches only for write operations, typically outperform their lock-free counterparts. The reason is that both paradigms (latch-based and latch-free algorithms) require access to shared memory locations and are therefore prone to cache invalidation and contention. Moreover, the overhead of complex synchronization protocols to realize latch-free structures leads to typical worse performance than their latch-free counterparts. An exception is when the number of OS-threads exceeds the number of CPU cores. However, in the same work, it has been shown, that by limiting the number of worker threads to the number of CPU cores, the problem of preemption, which normally is one of the biggest issues of latch-based index structures, can be mitigated. It can be argued that both latch-free and read-optimized latch-based synchronization protocols cannot be designed independently of the garbage collection method. Moreover, to achieve a completely non-blocking index structure—which normally is the purpose of latch-free structures—a non-blocking memory reclamation strategy is required. However, non-blocking memory reclamation techniques like Hazard Pointers [83] are prone to cache-invalidation and therefore provide inferior performance [57] compared to blocking memory reclamation strategies like epoch-based memory reclamation [42]. Hence,

epoch-based memory reclamation is currently the most widely used memory reclamation strategy for main memory index structures [78, 112, 77, 25, 82].

Recently, synchronization strategies using hardware transactional memory (HTM) instead of latches have been investigated [65, 26, 77, 81]. Although the idea of HTM was introduced in the late 1970s [79] and a proposal of how such a mechanism could be implemented with a CPU's cache coherence protocol was presented by Herlihy [59] in the mid-1990s, it was introduced in commodity hardware only recently. With the introduction of the Haswell CPU architecture [54], Intel added the TSX extension for the support of hardware transactional memory. Several approaches apply HTM to convert single-threaded index structures into scalable synchronized index structures, which are able to outperform latch-coupled index structures [65, 76]. However, the currently available HTM implementations only support transactions of limited size. This limitation might lead to frequent transaction restarts and hence, limits the scalability of the chosen approach. For instance, read optimized latching strategies outperform these initial approaches to synchronize index structures by using HTM [26, 77]. Yet, it was shown by Makreshanski et al. [81] that using HTM to build synchronization primitives like a multi-word compare-and-swap operation, simplifies the complex design of latch-free index structures while providing decent performance.

To conclude, synchronization strategies based on the paradigm of Read-Optimized Write EXclusion currently offer the highest performance. However, using such an approach requires to create a custom structure-specific synchronization protocol [77]. Therefore, we introduce the "Copy on Write ROWEX" synchronization protocol in Chapter 6 as a generic synchronization protocol following the ROWEX [77] paradigm. In contrast to existing solutions, it is applicable to any hierarchical index structure as long as this structure uses a copy-on-write approach which modifies only a single pointer per update operation. It can therefore be interpreted as the counterpart to optimistic lock coupling for copy-on-write based structures. While optimistic lock coupling is applicable to tree-based index structures using in-place updates, "Copy on Write ROWEX" is applicable to tree-based index structures using copy-on-write for updates.

CHAPTER 3

Optimizing the Height of Trie Structures

3.1 The Height Optimized Trie

A common approach to improve the access performance of index structures is to reduce the number of random memory accesses to retrieve a single entry. In the case of tree-based index structures, this corresponds to a reduction in the overall height and thus, the number of nodes that need to be accessed when traversing the tree structure.

For comparison-based search trees, the family of B-Tree index structures represents the typical approach to reduce the overall tree height. For this purpose, B-Trees apply the following two optimization techniques. First, they combine multiple binary tree nodes into tree nodes of higher arity. Secondly, B-Trees apply a rebalancing algorithm that ensures that all leaf entries are on the same level.

Even though, B-Trees are the prevailing index structures for disk-based database systems, trie structures—due to their fast intra-node search strategies (cf. Section 2.2.2)—represent promising alternatives for main memory database systems. Similar to B-Tree structures, trie variations that support higher node fanouts like the ART [75] or the GPT [24] combine multiple binary nodes into larger compound nodes to reduce the overall tree height.

However, in contrast to B-Trees, trie structures do not apply remediation strategies to prevent unbalanced trees in the case of non-uniformly distributed keysets. As a result, access performance and memory consumption vary depending on the currently stored dataset. For instance, consider the example of a trie structure with a fixed-span of 8 bits that stores one million 64-bit integers. If the tree contains monotonically increasing integers ranging from 1 to 1,000,000, almost all nodes are full and the average fanout is close to the maximum of 256. Hence, performance, as well as space consumption, is optimal. In contrast, if the dataset contains random integers from the full 64-bit domain, many nodes at lower levels of the tree are only sparsely filled. String-based datasets are also generally sparsely distributed, with genome data representing nucleic acids using a single-byte character (A, C, G, T) being an extreme case. Using a fixed-span trie to index such sparsely distributed datasets results in a low average fill factor, which has a negative impact on access performance and memory consumption.

We conclude that analogously to balancing algorithms for comparison based search trees, optimization strategies for trie based index structures are required to ensure a consistently low overall tree height, regardless of the stored keys' distribution.

We therefore propose the *Height Optimized Trie* (HOT). HOT defines a global maximum node fanout k and adjusts the span for each node according to the stored data, instead of choosing a fixed span regardless of the actual key distribution. This enables a consistently high fanout and avoids the sparsity problem that plagues other trie variants. As a result, the overall height of the trie structure is reduced. We schematically depict the height-reducing effect of HOT in Figure 3.1.

To satisfy the maximum node fanout constraint k, HOT is designed as an overlay over a binary Patricia trie that combines multiple binary nodes into compound nodes of higher arity. While the general idea is simple, the crucial aspect of HOT is to determine which binary nodes form a compound node. To solve this challenging problem, HOT introduces a novel insertion and deletion algorithm that dynamically adapts the tree structure such that the height of the resulting trie structure is minimal under the maximum fanout constraint k. While the general idea of dynamically adapting the tree structure is similar

(a) Adaptive Radix Tree

(b) Height Optimized Trie

Figure 3.1: Illustration of adaptive radix tree and height-optimized trie storing sparsely distributed keys. Whereas the average node fanout for the adaptive radix tree decreases at lower levels of the tree, HOT retains a consistently high fanout and therefore has a smaller overall height.

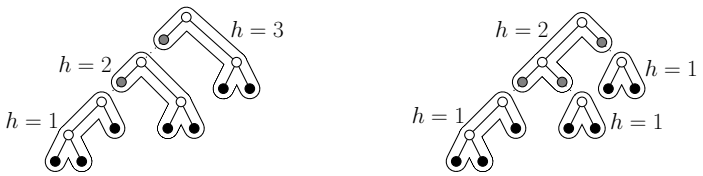
to the rebalancing mechanism of B-Trees, the designed algorithms differ fundamentally, because HOT's reorganization algorithm has to preserve the trie property that all nodes in a common subtree have a common prefix.

In the remainder of this chapter, we first discuss the challenges of combining multiple binary trie nodes such that the maximum fanout constraint k is satisfied and the overall tree height is minimized. Next, we explain our insertion and deletion algorithms in detail and how these algorithms evolve the HOT structure over time.

3.2 Combining Binary Trie Nodes into k-Constrained Tries

A crucial property of HOT is that every *compound node* represents a binary Patricia trie with a fanout of up to k. As can be observed in Figure 2.2, a binary Patricia trie storing n keys has exactly $n - 1$ inner nodes. A HOT compound node therefore only needs to store at most $k-1$ binary inner nodes (plus up to k pointers/leaves).

For a given parameter k, there are multiple ways of combining binary nodes into compound nodes. Figure 3.2 shows two trees with a maximum fanout of $k = 3$ that store the same data. While the tree shown in Figure 3.2a reduces the total number of compound nodes, the tree shown in Figure 3.2b is usually preferable, as it minimizes the overall tree height. In the figure and in our implementation every compound node n is associated with a height $h(n)$, such that $h(n)$ is the maximum height of its compound child nodes + 1. Based



(a) Optimized for # of nodes

(b) Optimized for height

Figure 3.2: Different ways of combining binary nodes into compound nodes, which are annotated with their height h.

on this definition, the overall tree height is the height of the root node. More formally, assuming a node n has $n.m$ child nodes, $h(n)$ can be defined as

$$h(n) = \begin{cases} 1, & n.m = 0 \\ \max_{i=1}^{n.m}(h(n.child[i])) + 1, & \text{else.} \end{cases}$$

Creating k-constrained nodes in a way that minimizes the overall tree height is analogous to partitioning a full binary tree into disjoint subtrees, such that the maximum number of partitions along a path from the root node to any leaf node is minimized. For static trees, Kovács and Kis [71] solved the problem of partitioning trees such that the overall height and cardinality are optimized. We present a dynamic algorithm, which is able to preserve the height optimized partitioning while new data is inserted.

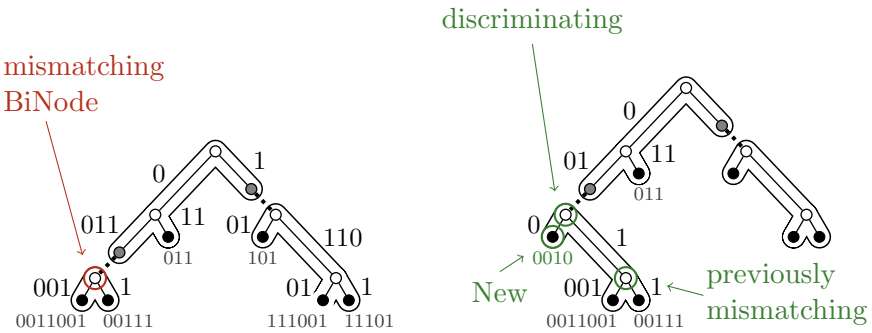
To avoid confusion between binary nodes and compound nodes, we use the following terminology: whenever we denote a node in a binary Patricia trie we use the term *BiNode*. In all other cases, the term *node* stands for a compound node. In this terminology, a node contains up to $k - 1$ BiNodes and up to k leaf entries.

3.3 Structure Adaptation

Similar to B-trees, the insertion and deletion algorithms of HOT have a normal code path that affects only a single node and other cases that perform structural modifications to the tree. In the remainder of this section, we therefore discuss the operations necessary to maintain an optimized height while inserting and deleting entries.

3.3.1 Insertion Algorithm

Besides the normal code path, HOT's insertion algorithm distinguishes three different types of structural modification. In the following, we describe these different cases by successively inserting four keys into a HOT structure with a maximum node fanout of $k = 3$. The initial tree is shown in Figure 3.3a. Insertion always begins by traversing the tree until the first BiNode is reached that does not correspond to a prefix of the key to be inserted. We call this BiNode the mismatching BiNode. In our running example, the *mismatching BiNode* for the first key to be inserted, 0010, is highlighted in Figure 3.3a.



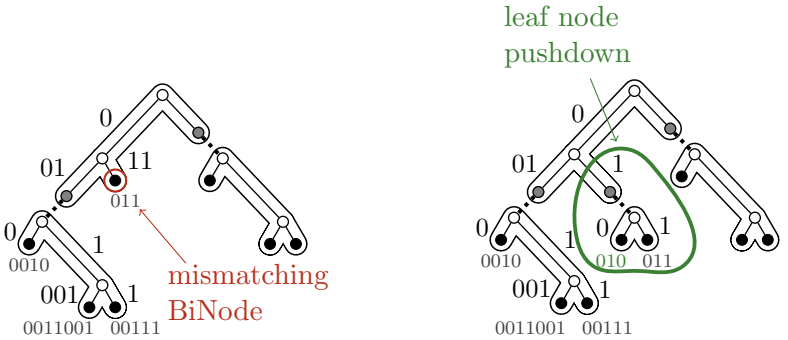
(a) Initial HOT before inserting 0010.

(b) *Normal* insert of 0010 before inserting 010.

Figure 3.3: Step-by-step example of a normal insert of the key 1111 into a HOT with a maximum fanout of $k = 3$.

In the *normal* case, insertion is performed by locally modifying the BiNode structure of the affected node. More precisely, and as shown in Figure 3.3b, a new *discriminating BiNode*, which discriminates the new key from the keys contained in the subtree of the mismatching BiNode, is created and inserted into the affected node. The normal case is analogous to inserting into a Patricia tree. However, because nodes are k-constrained, the normal case is only applicable if the affected node has less than k entries.

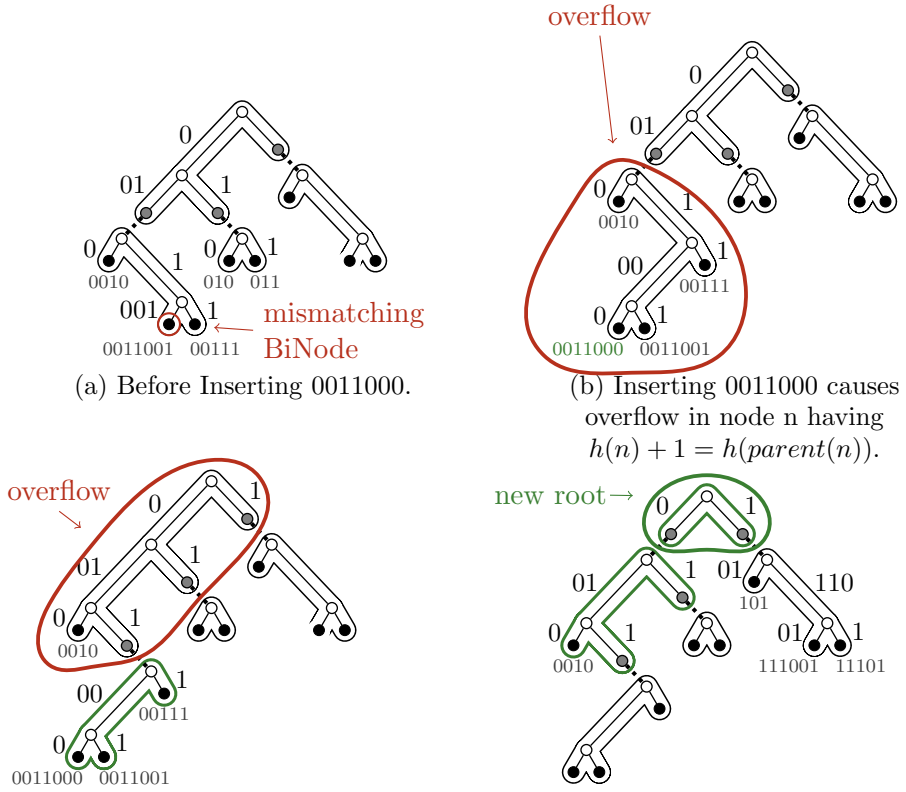
The second case is called *leaf-node pushdown* and involves creating a new node instead of adding a new BiNode to an existing node. If the mismatching BiNode is a leaf *and* the affected node is an inner node ($h(n) > 1$), we replace the leaf with a new node. The new node consists of a single BiNode that distinguishes between the new key and the previously existing leaf. In our running example, this case is triggered when the key 010 is inserted into the tree shown in Figure 3.4a. Leaf-node pushdown does not affect the maximum tree height as can be observed in Figure 3.4b: Even after leaf-node pushdown, the height of the root node (and thus the tree) is still 2.



(a) HOT before the insertion of 010.

(b) Insertion of 010 using *leaf node pushdown*.

Figure 3.4: Step-by-step example of a *leaf node pushdown* triggered by the insertion of key 0010 into a HOT with a maximum fanout of $k = 3$.



(a) Before Inserting 0011000.

(b) Inserting 0011000 causes overflow in node n having $h(n) + 1 = h(parent(n))$.

(c) Handling overflow by *parent pull up* (propagates overflow to parent node).

(d) Overflow resolution by *parent pull up* resulting in *new root node creation*. Before inserting 1111.

Figure 3.5: Step-by-step example of a parent pull up triggered by inserting 0011000 into a HOT with a maximum fanout of $k = 3$.

An overflow occurs if neither leaf-node pushdown nor normal insert is applicable. As Figure 3.5b shows, such an invalid intermediate state occurs after inserting 0011000.

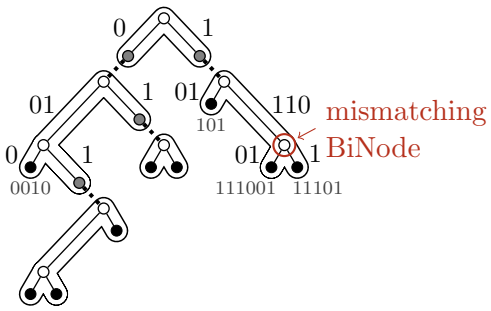
There are two different ways of resolving an overflow. Which method is applicable depends on the height of the overflowed node in relation to its parent node. As Figure 3.5 illustrates, one way to resolve an overflow is to perform *parent pull up*, i.e., to move the root BiNode of the overflowed node into its parent node. This approach is taken when growing the tree "downwards" would increase the tree height and thereby would yield unbalanced trees. It is therefore favorable to keep the tree structure balanced by growing the tree "upwards". More formally, parent pull up is triggered when the height of the overflowed node n is "almost" the height of its parent: $h(n)+1 = h(parent(n))$. By moving the root BiNode, the originally overflowed node becomes k-constrained again, but its parent node may now overflow—this indeed happens in the example shown in Figure 3.5c. Overflow handling, therefore, needs to be recursively applied to the affected parent node. In our example, because the root node is also full, overflow is eventually resolved by creating a new root, which is the only case where the overall height of the tree is increased. Thus, similar to a B-tree, the overall height of HOT only increases when a new root node is created.

The second way to handle an overflow is *intermediate node creation*. Instead of moving the root BiNode of the overflowed node into its parent, the root BiNode is moved into a newly created intermediate node. Intermediate node creation is only applicable if adding an additional intermediate node does not increase the overall tree height, which is the case if: $h(n) + 1 < h(parent(n))$. In our example, this case is triggered when the key 1111 is inserted into the tree shown in Figure 3.6a. As can be seen in Figure 3.6b, the overflowed node n has a height of 1 and its parent has a height of 3. Thus, there is "room" above the overflowed node and creating an intermediate node does not affect the overall height, as can be observed in the tree shown in Figure 3.6c.

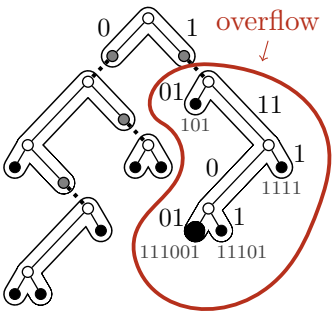
To summarize, there are four cases that can happen during an insert operation. A normal insert only modifies an existing node, whereas leaf-node pushdown creates a new node. Overflows are either handled using parent pull up or intermediate node creation. These four cases are also visible in Listing 3.1, which shows the full insertion algorithm.

```
1  insert(hot, key):
2     n = traverse hot for key
3     m = traverse n until mismatch
4     if (isLeafEntry(m) and h(n) > 1):
5        # leaf node pushdown
6        l = createNode(m, key)
7        n̂ = replaceNode(n, m, l)
8     else:
9        d = createBiNode(m, key)
10       n̂ = replaceBiNode(n, m, d)
11       handleOverflow(n̂)

13 handleOverflow(n):
14    if (not isFull(n))
15       # normal path
16       return
17    n̂ = split(n)
18    p = parentNode(n)
19    if (height(n̂) == height(p)):
20       # parent pull up
21       e = createBiNode(n̂[0], n̂[1])
22       p̂ = replaceBiNode(p, n, e)
23       handleOverflow(p̂)
24    else
25       # intermediate node creation
26       p̂ = replaceNode(p, n, n̂)
```

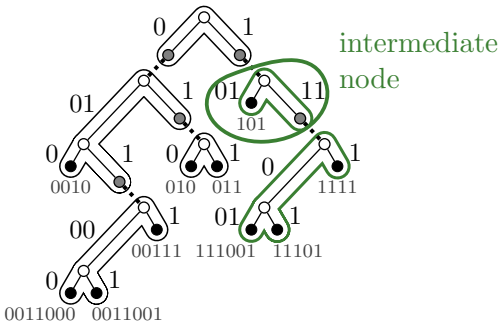
Listing 3.1: Structure-adapting insertion algorithm.



(a) Before Inserting 1111.



(b) Inserting 1111 causes overflow in node n having $h(n) + 1 < h(parent(n))$.



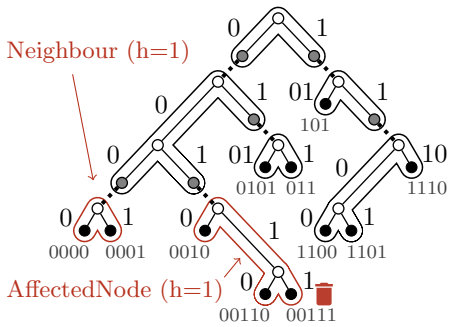
(c) Overflow resolution by *intermediate node creation*.

Figure 3.6: Step-by-step example of an *intermediate node creation* triggered by the insertion of the key 1111 into a HOT with a maximum fanout of $k = 3$.

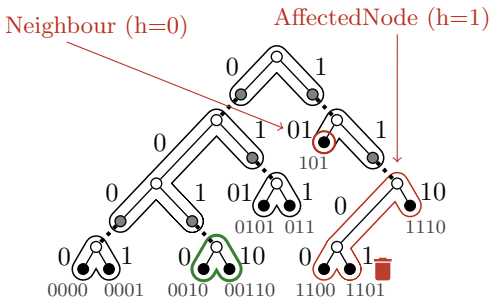
3.3.2 Deletion Operation

To maintain an optimized overall tree hight, it is crucial to not only incrementally maintain an optimized height in case of insertion operations but also in the case of deletion operations. In contrast to the insertion algorithm, the deletion algorithm only requires two kinds of structural modifications.

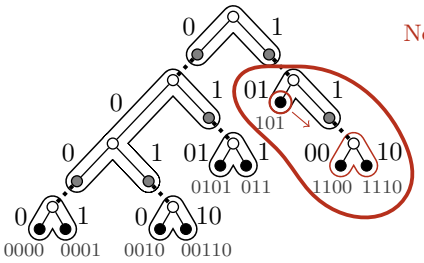
We describe these different cases by sequentially removing three keys (00111, 1101 and 00110) from an existing HOT structure with a maximum node fanout of $k = 3$. The initial tree is depicted in Figure 3.7a. Each deletion operation starts by traversing the tree until the entry to delete is found. We denote the node containing the entry to delete as the *affected node*. Depending on its height and the height and the number of entries in its neighbor node we distinguish three different cases.



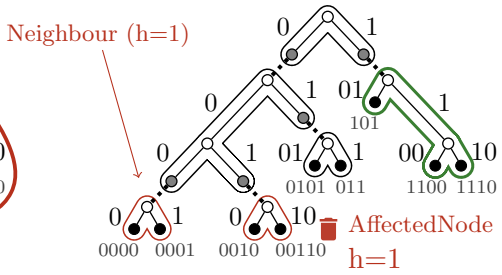
(a) Initial HOT before deleting 00111.



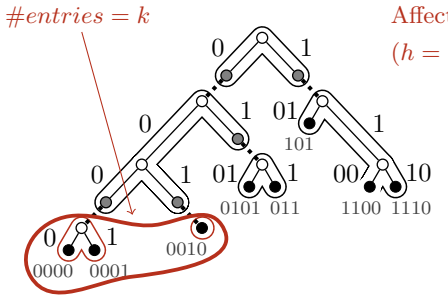
(b) *Normal* deletion of 00111 before deleting 1101.



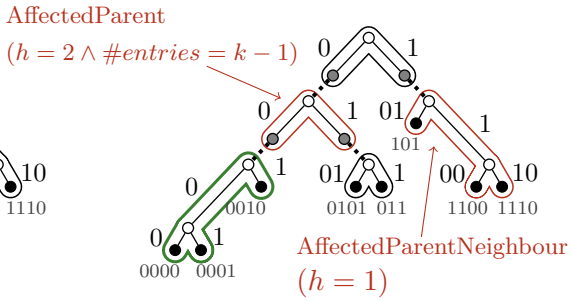
(c) After deletion of 1101.
Before *simple BiNode pull down*



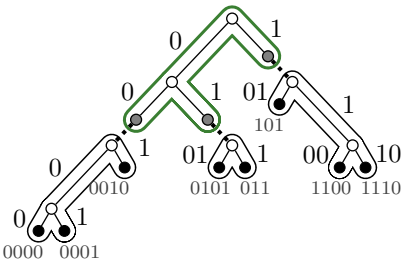
(d) After *simple BiNode pull down.*
Before deletion of 00110.



(e) After deletion of 00110.
Before *node merge.*



(f) After *node merge.*
Before *simple BiNode pull down*



(g) Final tree after *simple BiNode pull down*, which in this case reduces the overall tree height by 1.

Figure 3.7: Step-by-step example deleting the keys 00111, 1101 and 00110 from HOT with a maximum fanout of $k = 3$.

Before we describe the three different kinds of HOT's deletion operations, we introduce how we define an underflow in HOT's structure, which can occur by removing an entry from a node. An underflow occurs whenever the total number of entries of two sibling nodes is less or equal than the maximum node fanout k. An underflow cannot occur when an *affected node's* sibling is a BiNode.

A *normal* deletion is executed, whenever removing the entry to delete with its associated *discriminating* BiNode, does not yield an underflow. Removing the key 00111 from the initial tree shown in Figure 3.7a is an example of a normal deletion resulting in the tree depicted in Figure 3.7b. A normal deletion on a node, which contains only two entries, represents an edge case as the affected hot node is transformed into a leaf node. Thus, it is the inverse of the leaf node pushdown operation.

To resolve an underflow, we distinguish two scenarios. First, the affected node's neighbor has a smaller height. Second, the affected node's neighbor has the same height.

In the first case, we pull the parent BiNode down into the affected node. We therefore call this *simple BiNode pull down*. Such a simple BiNode pull down is applied when deleting the entry with the key 1101 from the tree in Figure 3.7b. This operation is depicted in a two-step process. First, the entry to delete is removed, yielding the tree of Figure 3.7c. The two nodes affected by the resulting underflow are highlighted in this Figure. Next, we execute *simple parent pull down*, which results in the tree shown in Figure 3.7d.

In case the nodes affected by the underflow both have the same height a *node merge* is applied. This node merge is the inverse of the parent pull up operation known from the insertion operation and combines the three parts, parent BiNode, left sibling, and right sibling into a single node. A node merge is illustrated in Figure 3.7f, which combines the remaining part of the affected node and its neighbor after deleting 00110 from the tree depicted in Figure 3.7d.

In contrast to insertion operations, where only parent pull up may recurse up the tree, both underflow resolution strategies simple BiNode pull down as well as node merge may affect their parent node. Therefore both operations may potentially recurse up the tree until the overall tree height is reduced. The reason is that both operations reduce the number of entries in the affected node's parent node. Thus, a potential underflow operations may occur in the parent node which again must be resolved by one of the possible underflow resolution strategies. This is shown in Figure 3.7g, which shows the result of executing a simple BiNode pull down after the node merge shown in Figure 3.7f.

```
1  delete(hot, key):
2     n = traverse hot for key
3     m = traverse n until leaf entry
4     if (getKey(m) == key):
5        n̂ = delete(n, key)
6        handleUnderflow(n̂,m,s)

8  handleUndeflow(n,m,s):
9     s = determineSiblingNode(n)
10    if(height(n) < height(s) or (count(n) + count(s)) > max_fanout):
11       # no underflow handling necessary
12       return

14    p = parentNode(n)
15    pBi = determineParentBiNode(p, key)
16    if (height(n) > height(m)):
17       # simple BiNode pulldown
18       n̂ = pulldownBiNode(n, pBi)
19    else: # (height(n) == height(m))
20       # node merge
21       n̂ = mergeNodes(n, s, pBi)
22    p̂ = removeBiNode(p, pBi)
23    handleUnderflow(p̂)
```

Listing 3.2: Structure-adapting deletion algorithm.

In summary, HOT's deletion algorithm supports three different cases. The normal deletion is used when no underflow occurs and only affects a single node. In the case of an underflow and depending on the height of the affected node's sibling, either a simple BiNode pulldown or a node merge is used. To illustrate under what conditions each of these three cases is used, we show the complete deletion algorithm in Listing 3.2.

CHAPTER 4

Properties of Height Optimized Tries

In this chapter, we discuss some interesting theoretical properties of HOT. We will first give a high-level description of the claimed properties and will then provide proofs showing their validity.

For HOT our experiments hint at the following three properties:

(I) Determinism: for the same set of keys regardless of their insertion order the resulting HOT will have the same structure and the same set of compound nodes.

(II) Minimum height: for a given set of keys and a maximum fanout k no set of compound nodes can be found, such that that the height of the resulting tree is lower than the height of a HOT with the same maximum fanout k and the same set of keys.

(III) Recursive structure: each subtree of a HOT structure is itself a HOT structure and thus of minimal height.

In the remainder of this chapter, we are going to prove that all three properties hold for HOT. All proofs are based on the simple idea that the implicit partitioning created by HOT's compound nodes over the underlying binary Patricia trie is equivalent to a minimum height partitioning as produced by the algorithm of Kovács and Kis [71]. We therefore first introduce a simplified variant of HOT called *sHOT*. We then use this simplified representation to prove that the partitioning created by sHOT is equivalent to the partitioning created by Kovács and Kis [71]. Finally, we prove the equivalence of the partitionings created by sHOT and HOT.

4.1 Simplified HOT (sHOT)

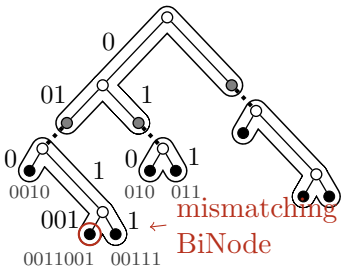
For all subsequent proofs, we introduce a simplified version of HOT called *sHOT*, which stores leaf entries in leaf nodes only. This restriction has implications on the insertion algorithm and therefore sHOT's insertion algorithm differs from the insertion algorithm of HOT that is presented in Section 3.3.1. Both algorithms feature a "normal" and three "special cases". As sHOT does not contain any leaf entries in non-leaf nodes, sHOT does not support the special case of leaf node pushdown. Instead, it introduces *intermediate insert*, which is applicable whenever the mismatching BiNode is contained in an intermediate node. Besides the new intermediate insert operation sHOT uses modified versions of the two overflow handling strategies parent pull and intermediate node creation that ensure that intermediate nodes will never contain leaf entries. In the following, we will discuss each of these three special cases separately. To illustrate the effects of sHOT's insertion algorithm we use separate step by step examples. Each example inserts a single key into an sHOT structure with a maximum node fanout of $k = 3$.

(a) Initial HOT before inserting 1011.

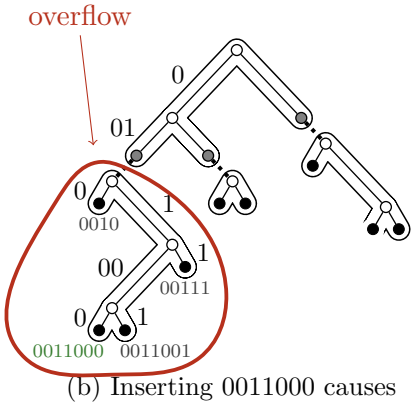
(b) *Intermediate insert* of 1011.

Figure 4.1: Example of an intermediate insert of the key 1011 into an sHOT with a maximum fanout of $k = 3$.

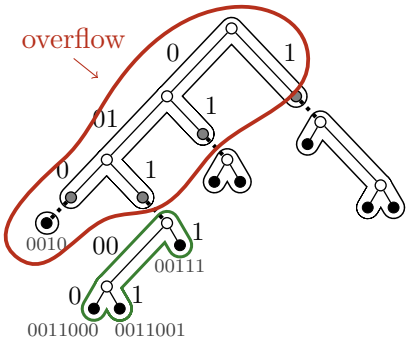
The special case of *intermediate insert* is triggered, whenever the affected node is an intermediate node. As sHOT inhibits intermediate nodes to contain any leaf entries intermediate insert consists of two parts. The first part creates a leaf node containing only the newly inserted entry. The second part inserts the discriminating BiNode into the affected node and uses a boundary entry to connect it with the new leaf node. We show an example of intermediate insert which inserts Key 1011 into the sHOT depicted in Figure 4.1. It can clearly be seen that the resulting sHOT shown in Figure 4.1b uses a separate node for the new key 1011, although the originally affected node was not full.



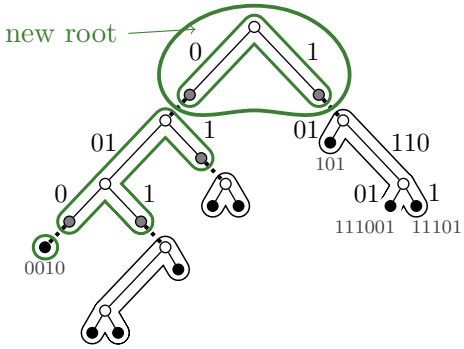
(a) Before Inserting 0011000.



(b) Inserting 0011000 causes overflow in node n having $h(n) + 1 = h(parent(n))$.



(c) Handling overflow by *parent pull up* (propagates overflow to parent node).

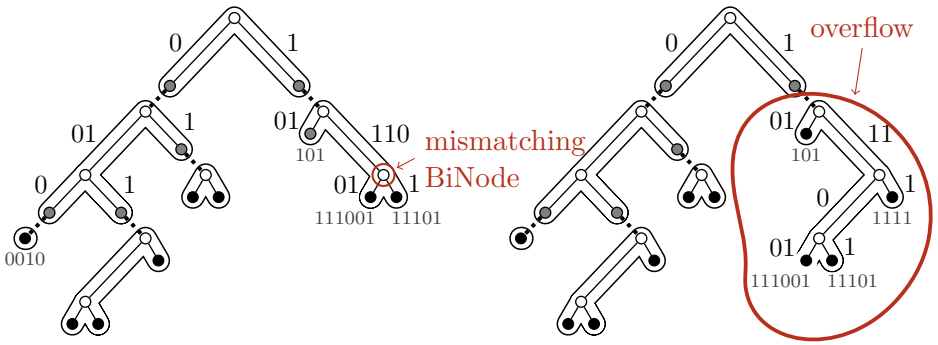


(d) Overflow resolution by *parent pull up* resulting in *new root node creation*.

Figure 4.2: Step-by-step example of a parent pull up triggered by inserting 0011000 into a HOT with a maximum fanout of $k = 3$.

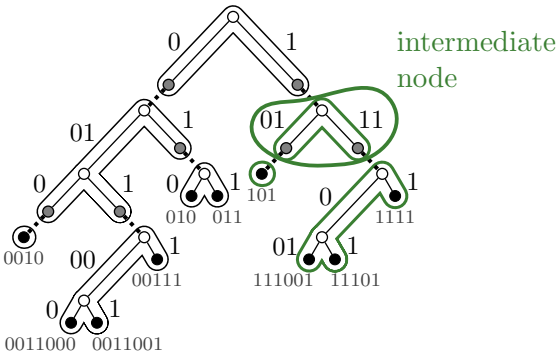
Besides intermediate node insert, also both overflow handling strategies intermediate node creation and parent pull up are affected by the restriction

that no intermediate node is allowed to contain any leaf entries. This restriction comes into play whenever the affected node is a leaf and one of its subtrees contains only a single entry. HOT normally hoists this single entry into the parent node in the case of parent pull up and into the newly created intermediate node in the case of intermediate node creation. In contrast, for both overflow handling strategies sHOT plainly splits overflown nodes without hoisting single entry nodes into their parent nodes. Thus, single entry nodes are created whenever one of the overflown node's left or right subtree contains only a single entry. The effect of sHOT's plain split is shown in Figure 4.2 when inserting 0011000 causes a parent pull up and in Figure 4.3 when inserting 1111 triggers an intermediate node creation. Both examples are identical to their HOT counterparts shown in Figure 3.5 and Figure 3.6. It can clearly bee seen in the respective subfigures Figure 4.2c and Figure 4.3c that sHOT executes a plain split, without any optimization.



(a) Before Inserting 1111

(b) Inserting 1111 causes overflow in node n having $h(n) + 1 < h(parent(n))$.



(c) Overflow resolution by *intermediate node creation*.

Figure 4.3: Step-by-step example of an *intermediate node creation* triggered by the insertion of the key 1111 into a HOT with a maximum fanout of $k = 3$.

4.2 Static Minimum Height Partitioning

The algorithm by Kovács and Kis [71] partitions a weighted tree structure, such that for each partition the intrapartition weight of the vertexes contained in a single partition satisfies a predefined so-called "knapsack" constraint W and the induced tree defined over the resulting partitioning is of minimum height (by Lemma 1 from [71]). The algorithm of Kovács and Kis [71] assigns each vertex v a level $l(v)$ and an intrapartition weight $rw(v)$, such that connected subgraphs consisting of nodes with the same level form individual partitions. Initially, each leaf vertex v_l is labeled with level $l(v_l) = 0$ and its intrapartition weight $rw(v_l)$ is equal to its weight $w(v_l)$. For the definition of the labeling functions $l(v)$ and $rw(v)$ the algorithm defines the helper function $cl_{max}(v) = max_{u \in child(v)} l(u)$, which for a given vertex v returns the maximum level of any of its child nodes. Additionally, we define the helper function $maxc(v) = \{v \in child(v) | l(v) = cl_{max}(v)\}$ which returns the set of all child nodes of a vertex v having the same level as $cl_{max}(v)$. The level $l(v)$ of a non-leaf vertex v depends on the intrapartition weight and levels of its children and is defined as follows:

$$l(v) = \begin{cases} cl_{max}(v) & \text{if } w((v) + \sum_{u \in cmax(v)} rw(u) \leq W \\ cl_{max}(v) + 1 & \text{otherwise} \end{cases}$$

For all non-leaf vertices v, the intrapartition weight $rw(v)$ is the summed up weight of all its descendant vertices within the same partition. More formally the intrapartition weight $rw(v)$ of a single non-leaf vertex v is defined as follows:

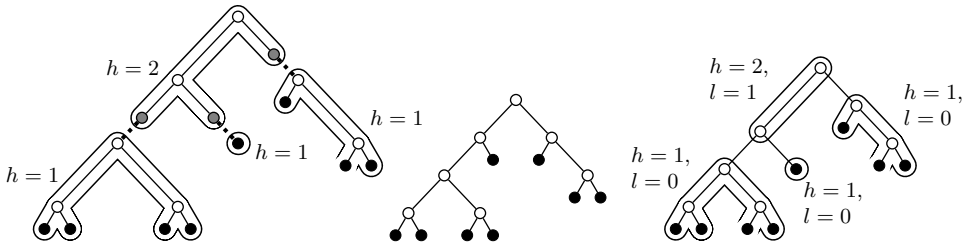
$$rw(v) = w(v) + \sum_{u \in cmax(v)} rw(u)$$

Our weight function $w(v)$ assigns the weight zero to all leaf entries and one to each BiNode. This weight function implies that only the weights of BiNodes are taken into account.

$$w(v) = \begin{cases} 0 & \text{if } v \text{ is a leaf} \\ 1 & \text{otherwise} \end{cases}$$

We denote the constraint, which is called the knapsack constraint in the original paper by Kovács and Kis [71] as the *maximum intrapartition weight* constraint. We set this constraint to equal $k - 1$, where k is the maximum node fanout of an equivalent sHOT. In the remainder of this chapter, we call the combination of the minimum height partitioning algorithm by Kovács et. al and our custom weight function w and the maximum intrapartition weight constraint $k - 1$ as *static minimum height partitioning* (SMHP). Although, sHOT as well as SMHP partition binary Patricia tries, both algorithms use different representations for the actual partitions. In the context of sHOT in-

dividual compound nodes represent individual partitions. While both sHOT as well as SMHP assign each vertex to a partition, sHOT adds boundary entries to link individual partitions together. Additionally, Kovács and Kis [71] originally defined the height of individual partitions to be equal to their level l and therefore the height of leaf partitions is equal to 0. However, the height of sHOT leaf nodes is 1. We therefore use a redefined definition of the height of individual partitions, which is comparable to the height of sHOT nodes. By omitting the boundary entries of sHOT partitions and using an alternative height definition $height(p) = l(p) + 1$ for partitions created by SMHP, we are finally able to compare whether a partition created by SMHP is equivalent to a partition created by sHOT. Figures 4.4a-4.4c illustrate the difference between these two representations.



(a) sHOT (b) Binary Patricia trie (c) SMHP

Figure 4.4: Two equivalent representations of the same underlying binary Patricia trie. Both representations sHOT as well as SMHP apply a maximum fanout constraint. The sHOT representation restricts the maximum fanout to three, whereas the SMHP restricts the maximum intrapartition weight of its vertices to two. In both representations the partitions/compound nodes are annotated with the height of the corresponding subtree. Additionally, each partition generated by SMHP is labelled with the corresponding level of the partition. The grey boundary entries which link individual sHOT nodes together are missing in the case of SMHP.

4.3 Properties of Static Minimum Height Partitionings

Before we actually prove the properties of sHOT we first show that the function $rw(v)$ calculating the intrapartition weight of individual vertices is equal to counting the number of descendant BiNodes contained in the same partition. We then use this finding to show that each partition generated by SMHP satisfies the same maximum fanout k as nodes in sHOT. Afterward, we show

that SMHP deterministically generates the same partitioning for the same structure.

4.3.1 Intrapartition Weight Function

Static minimum height partitioning restricts the number of vertices besides leaves contained in a single partition p by the intrapartition weight $rw(r)$ of the partition's root vertex r to be $rw(r) \leq k - 1$.

Lemma 1. *Given a partition p, its root vertex r, and the number of BiNodes n contained in p, the intrapartition weight of the root vertex is equal to the number of BiNodes:* $rw(r) = n$.

Proof. The proof is by induction on the number of BiNodes contained in a partition p with root vertex r.

Base Case: The smallest possible partition created by SMHP, with regards to the number of BiNodes, is a partition consisting only of a single leaf entry. Hence, the weight of the root vertex in this specific case is trivially zero, which is equivalent to the number of BiNodes contained in such a partition.

Induction step: Next, we consider partitions containing $n + 1$ BiNodes. Such partitions can incrementally be constructed from two subtrees the left subtree ST_{left} with root vertex v_{left} and the right subtree ST_{right} with root vertex v_{right}. Without loss of generality, we assume that the left subtree contains o BiNodes and the right subtree contains k BiNodes with $o + k = n$. Both subtrees contain at most n entries and according to the induction hypothesis the intrapartition weight of these vertices is $rw(v_{left}) = o$ and $rw(v_{right}) = k$. Therefore, substituting $rw(v_{left})$ and $rw(v_{right})$ by o and k in the intrapartition weight function of the root BiNode r yields $rw(r) = o + k + 1 = n + 1$, which is exactly the number of BiNodes in the considered partition.

□

For simplicity's sake, in the remainder, we abbreviate the intrapartition weight of the root node $rw(r)$ of a partition p with $rw(p)$ and call it the intrapartition weight of partition p. As all vertices within the same partition have the same level, we define the level of a partition $l(p)$ accordingly.

4.3.2 Maximum Fanout Constraint

The maximum fanout constraint of sHOT ensures that each node contains at most k entries. These entries are either leaf entries or boundary entries. As partitions generated by SMHP do not contain boundary entries, a different comparable representation for the maximum fanout constraint is required which is applicable for sHOT as well as SMHP. Due to the fact that a binary Patricia trie is a full binary tree, each sHOT node is a full binary tree containing a certain number of n leaves. This implies that each sHOT node contains at most $n - 1$ BiNodes. Thus, we can reformulate the maximum fanout constraint in the following way: each sHOT node with a maximum node fanout of k contains at most $k - 1$ BiNodes. According to our proof in the previous Section 4.3.1, the intrapartition weight is equal to the number of BiNodes contained in the respective partition. As the maximum value of the intrapartition weight function for a single partition generated by SMHP is restricted by $k - 1$, the number of BiNodes contained in a single partition is at most $k - 1$. Hence, the same maximum fanout constraint is applicable for partitions generated by SMHP and nodes generated by sHOT.

4.3.3 Determinism of Static Minimum Height Partitioning

Lemma 2. *For a given set of keys, the SMHP over a binary Patricia trie containing those keys is deterministic and independent of the actual insertion order.*

Proof. As the algorithm of SMHP is deterministic and the structure of a binary Patricia trie for a given set of keys is independent of the actual insertion order, SMHP over a binary Patricia trie and a given set of keys is also independent of the actual insertion order and therefore deterministically defined. □

4.3.4 Recursive Structure of Static Minimum Height Partitioning

Lemma 3. *Given a tree structure T, a partitioning P created by SMHP over T and an induced tree $T(P)$ defined over the partitioning P. Then, each subtree $ST_i(P)$ of the induced tree is equal to an SMHP over the underlying vertices.*

Proof. As the intrapartition weights and levels of vertices only depend on the weights of its descendant, the level and weights of all vertices contained in

the subtree $ST_i(T)$ corresponding to the subtree $ST_i(P)$ depend on vertices in $ST_i(T)$ only. Hence, the levels and intrapartition weights of these vertices are identical to an SMHP applied on the corresponding subtree only. Therefore each subtree of the induced tree over a partitioning created by SMHP is identical to an SMHP over the underlying vertices and therefore of minimum height. □

4.4 Equivalence of Static Minimum Height Partitioned Patricia Tries and Simplified HOT

In this section, we will prove that both algorithms sHOT and SMHP applied on the same set of keys result in an equivalent structure. To that end, we first define the equivalence between sHOT compound nodes and partitions generated by SMHP. A compound node generated by sHOT is equivalent to a partition generated by SMHP if and only if all vertices of the original binary Patricia trie contained in the respective partition are also contained in the corresponding compound node. More formally, a compound node n and a partition p are equivalent if and only if $vertices(n) \setminus boundaryEntries(n) = vertices(p)$ holds. Based on this definition we define the equivalence of a static minimum height partitioning P and a corresponding sHOT H, such that for all partitions $p \in P$ a compound node $n \in H$ exists which is equivalent to p and for all nodes $n \in H$ a partition $p \in P$ exists which is equivalent to n. This implies that whenever a partitioning P and an sHOT H are equivalent, the height of the induced subtrees over the partition/nodes is also the same. In Figure 4.4 we depict this notion of equivalence by showing a binary Patricia trie in Figure 4.4b and the corresponding HOT representation in Figure 4.4a and the static minimum height partitioned tree in Figure 4.4c.

Lemma 4. *An sHOT for a set of keys K is equivalent to an SMHP structure defined over a binary Patricia trie for the same set of keys K.*

Proof. The proof is by induction on the number of keys n.

Base Case: For trees having up to k entries both SMHP as well as sHOT construct a single entity. SMHP denotes this entity as partition whereas sHOT calls it a compound node. However, in both cases, this entity contains all vertices of the tree itself and therefore both representations are equivalent.

Induction Step: An sHOT having $n + 1$ entries with $n \geq k$ results from an sHOT of n entries, for which the induction hypothesis holds. Hence,

sHOT before the insertion operation is equivalent to an SMHP defined over a binary Patricia trie for the same set of keys.

We consider each of the three possible insert operations that are supported by sHOT individually: normal insert, intermediate node creation, and parent pull up. In the following, we extend the nomenclature introduced in Section 3.3.1 to describe the structures in the initial SMHP which correspond to the affected structures in an equivalent sHOT. For any type of insertion operation, we denote the partition which corresponds to the affected node (n_{aff}) containing the mismatching BiNode (bi_{mis}) as the *affected partition* (p_{aff}). Additionally, we denote the partition which corresponds to the affected node's parent node as the *parent partition* (p_{par}).

We exploit the fact that the structure of a binary Patricia trie for the same set of keys is always the same regardless of the insertion order and that inserting a new key into an existing binary Patricia trie is only a *local modification* inserting two new vertices. These two vertices are the *discriminating BiNode* (bi_{dis}) and the vertex corresponding to the new key (v_{new}). (see Section 3.3.1). The local modification according to the definition of a binary Patricia trie inserts the discriminating BiNode directly above the mismatching BiNode such that the new leaf entry becomes the direct sibling of the mismatching BiNode. To determine whether sHOT's insertion algorithm results in the same partitioning as inserting the same key x into the original underlying binary Patricia trie and creating an SMHP we exploit the properties of the local modification in the following way. First, for each of sHOT's insertion operation types, we determine the affected partition of an SMHP, which is equivalent to the corresponding sHOT before the insertion of x. Next, we apply the effects of the local modification to determine the SMHP, which is equal to a binary Patricia trie after the insertion operation. Finally, we check whether the resulting partitioning is equivalent to the sHOT structure after the insertion of x.

Δ Before we consider each of sHOT's insertion operation types individually, we define properties of v_{new} which are independent of the actual type of insertion operation. According to SMHP, the intrapartition weight and level of leaf vertices are always zero. Therefore leaf vertices are always contained in leaf partitions. Whenever the level of the mismatching BiNode is larger than zero, the level of the discriminating BiNode is also larger than zero. Thus, in this particular case v_{new} is placed in a new single-element partition. In all other cases v_{new} is placed in the same leaf partition as the mismatching BiNode.

- **Normal Insert**: Normal insert for sHOT is applicable whenever the affected node is a leaf node and contains strictly less than k entries (cf. Section 3.3). According to the induction hypothesis, the affected partition is a leaf partition with an intrapartition weight of $rw(p_{aff}) < k-1$. Hence the maximum weight of any vertex v in the affected partition is $rw(v) < k-1$.

  We first consider the effect of the local modification on the affected partition. As, the maximum weight of any vertex v in the affected partition is $rw(v) < k-1$, the highest feasible value for $rw(bi_{dis}) = k-1$. Therefore $rw(bi_{dis}) \leq k-1$ holds and the discriminating BiNode is labeled with the same level as the mismatching BiNode. On the other hand, the insertion of the discriminating BiNode may increase the intrapartition weight of all other vertices previously contained in the affected partition by one. As their initial intrapartition weights before the local modification were below $k-1$, the local modification does not affect the partition assignment of these vertices. Due to the fact that the affected partition is a leaf partition, the new vertex corresponding to the newly inserted key is also placed in the affected partition (cf. Δ). Furthermore, as the weight of v_{new} is zero it does not affect any of the other vertices.

  Hence, we observe that the transformation of an SMHP for the keys K into an SMHP for the keys $K' = K \cup \{x\}$ has the same behavior as a "normal insert" operation, which inserts the key x into an equivalent sHOT containing the keys K.

- **Intermediate Insert**: Normal insert for sHOT is applicable whenever the affected node is a non-leaf node and contains strictly less than k entries (cf. Section 3.3). According to the induction hypothesis, the level of the affected partition is therefore $l(p_{aff}) > 0$ and the intrapartition weight of the affected partition is $rw(p_{aff}) < k-1$. Hence the maximum weight of any vertex v in the affected partition is $rw(v) < k-1$. We argue along the lines of the equivalence proof for "normal insert" that to compensate the local modification the discriminating BiNode will be added to the affected partition while all other vertices previously contained in the affected partition will remain in the same partition. Additionally v_{new} will be placed into a new single entry partition (cf. Δ).

  We again observe that the operations required to transform the initial SMHP into an SMHP for the binary Patricia trie after the insertion of x, are identical to the modifications issued by an "in-

termediate insert" operation inserting the key x into an equivalent sHOT.

- **Intermediate Node Creation**: An intermediate node creation for sHOT (cf. Section 4.1) is applicable whenever the affected node contains k entries, and the height of its parent node is $h(n_{parent}) \geq h(n_{aff})+1$. According to the induction hypothesis, we infer that the intrapartition weight of the affected partition is $rw(p_{aff}) = k - 1$ and that the level of the affected partition's parent is $l(p_{par}) > l(p_{aff})+1$. We further infer that the level of the affected partition's sibling is $level(p_{sib}) = level(p_{par}) - 1 > l(p_{aff})$. In the following, we examine how the local modification of the underlying binary Patricia trie effects the SMHP.

  In the case that the mismatching BiNode is the root of the affected partition its intrapartition weight is $rw(bi_{mis}) = k - 1$. Hence, all vertices previously contained in the affected partition become descendants of the discriminating BiNode. Therefore neither their assigned levels nor their intrapartition weights are affected by the local modification. Due to the maximum intrapartition weight constraint and the intrapartition of the mismatching BiNode, the discriminating BiNode is assigned level $l(bi_{dis}) = l(bi_{mis}) + 1 = l(p_{aff}) + 1$ and intrapartition weight $rw(bi_{dis}) = 1$. As the level of the parent partition is strictly larger than the level of the discriminating BiNode $l(p_{par}) > l(bi_{dis})$, no vertices higher up in the tree structure are affected by the local modification. In total two new partitions are created. One partition contains the discriminating BiNode and the other contains the vertex corresponding to the new key x (cf. Δ).

  If the mismatching BiNode is not the root of the affected partition, inserting the discriminating BiNode increases the intrapartition weight of the mismatching BiNode's ancestors within the same partition by one. Thereby the intrapartition weight of the affected partition's former root BiNode r violates the intrapartition weight constraint. Therefore the level of r becomes $l(r) = l(bi_{mis}) + 1$ and its intrapartition weight $rw(r) = 1$. Again, the parent partition has a strictly larger level than vertex r and the ancestor vertices of r are not affected by the local modification. However, the assignment of level $l(r) = l(bi_{mis}) + 1$ to r implicitly splits the previous partition into three partitions. One partition contains the vertices in the left subtree. Another partition contains the vertices in the right subtree and another partition only contains r. Depending on

the level of the affected partition, v_{new} either forms a new partition or is integrated into the same partition as the mismatching BiNode (cf. Δ). As the intrapartition weight of the mismatching BiNode is $rw(bi_{mis}) < k-1$ the intrapartition weight of the discriminating BiNode is $rw(bi_{dis}) \leq k-1$. Thus, it will be placed in the same partition as the mismatching BiNode.

We observe that the compensation operations required to transform an SMHP for keys K into an SMHP containing the additional key x are identical to using intermediate node creation to insert the key x into an sHOT for keys K.

- **Parent Pull Up**: Parent pull up for sHOT is applied whenever the affected node contains k entries (cf. Section 3.3), and the height of the parent node is $h(n_{parent}) = h(n_{aff}) + 1$. According to the induction hypothesis, we infer that the intrapartition weight of the affected partition is $rw(p_{aff}) = k-1$ and the level of its parent partition is $l(p_{par}) = l(p_{aff}) + 1$.

  We apply a similar proof technique as for intermediate node creation. The first part is to determine the effects on the levels and intrapartition weights of the vertices contained in the affected partition. Identically to intermediate node creation either the discriminating BiNode or the previous root BiNode of the affected Partition gets assigned $l(bi_{mis})+1$ as level and one as its intrapartition weight. As the level of this BiNode is equal to the level of the parent partition it is propagated into its parent partition. We therefore denote this BiNode as the *hoisted BiNode*. Next, we determine the effects of the hoisted BiNode on the parent partition's vertices. Again this adds one to the intrapartition weights of the ancestor BiNodes within the parent partition. In the case that the number of BiNodes in the affected partition's parent partition is below $k-1$ before the insertion, the intrapartition weight of its root BiNode is strictly below $k-1$. Hence, after the insertion, the largest possible intrapartition weight of any vertex in the parent partition is at most $k-1$, and therefore, all vertices in this partition satisfy the intrapartition weight constraint. In case the number of BiNodes in the affected partition's parent partition is $k-1$, the intrapartition weight of its root BiNode becomes k. Hence, it violates the intrapartition weight constraint $k-1$. Depending on the level and intrapartition weight of the grandparent partition either the previous root of the parent partition is hoisted into the grandparent or a new partition containing only the hoisted node is created. If the

vertex is hoisted, the algorithm may recurse up the tree structure until it eventually will create a new root partition.

Again the behavior to compensate the local modification is identical to the behavior of an sHOT structure inserting a new key by using "parent pull up".

For all four cases, the operations required to transform an SMHP containing the set of keys K into an SMHP containing the keys $K \cup x$ are identical to inserting x into an equivalent sHOT for the keys K. Thus, we have shown, that all sHOT structures created by sHOTs insertion algorithm are equivalent to SMHPs created over Patricia tries for the same set of keys.

□

As we have shown that for the same keys sHOT is always equivalent to SMHP, we have shown that the same properties proven for SMHP hold for sHOT as well:

- By Lemma 1 from [71], the tree is of *minimum height*.
- By Lemma 2, the algorithm produces a *deterministic* tree structure *independent of the insertion order* of the keys.
- By Lemma 3, the generated node structure is *recursively* defined, such that the height of each node is minimal.

4.5 Minimum Cardinality Constraint of Individual BiNodes

To be able to prove that the same properties, which hold for sHOT, also hold for HOT, we introduce the minimum cardinality constraint that ensures that each BiNode bi_i with $h(b_i) > 1$ has enough descending BiNodes such that its descendants cannot be merged without violating the maximum fanout constraint.

Lemma 5. *Each BiNode contained in a node n_i of height $h(n_i) > 1$ has at least $k - 1$ descending BiNodes contained in nodes of height $h - 1$.*

To show that Lemma 5 holds for sHOT as well as for HOT we provide separate proofs for sHOT and HOT.

Proof. For sHOT there are three ways how BiNodes in nodes n_x of height $h(n_x) > 1$ are generated. The first two are the overflow handling strategies parent pull up and intermediate node creation. According to the definition of sHOT in Section 4.1, an overflow occurs in a node n_o of height $h(n_o)$ whenever the node contains $k + 1$ entries. Hence, it contains at least k BiNodes. To resolve the overflow the node is split into two nodes n_{left} and n_{right}, while the root BiNode bi_r is hoisted into a node of height $h(n_o) + 1$. As n_{left} and n_{right} become the direct descendants of bi_r the number of descending BiNodes of bi_r is $k - 1$. The third case to create intermediate nodes is an intermediate insert. It is applicable whenever the mismatching BiNode bi_{mis} is contained in a node n_i of height $h(n_i) > 1$. It will then insert the discriminating BiNode as a parent of the mismatching BiNode. As the mismatching BiNode has at least $k - 1$ descendants of height $h(n_i) - 1$, the newly created discriminating BiNode has also at least $k - 1$ descendants of height $h(n_i) - 1$. □

Proof. For HOT there exist only two cases how BiNodes in nodes n_x of height $h(n_x) > 1$ are generated. These cases are the overflow handling strategies of parent pull up and intermediate node creation. As the overflow handling strategies of sHOT and HOT only differ in the aspect of hoisting single entry leaf nodes, we consider this special case in the following. As we only consider BiNodes for the minimum cardinality constraint it doesn't matter whether a leaf entry is hoisted or not and therefore the number of BiNodes contain in descendant nodes is not affected by hoisting single entry leaf nodes. Hence, the minimum cardinality constraint holds for BiNodes generated by HOT as well. □

From this constraint, we can infer that each BiNode contained in a node of height h for each $hd < h$ has $k - 1$ descending BiNodes contained in nodes of height hd.

4.6 Equivalence of sHOT and HOT

In Section 4.4 we have shown that sHOT produces an equivalent partitioning as the minimum height partitioning algorithm by Kovács et al. [71]. In this section we show that all properties which hold for sHOT hold for HOT as well. Therefore, we first show that a deterministic transformation between sHOT and HOT exists which preserves the properties previously shown for sHOT.

Secondly, we show that directly inserting a value into HOT is equivalent to first inserting the same value into an equivalent sHOT and then transforming the resulting sHOT into HOT.

(a) HOT stores leaf entries in the same node as its parent discriminating BiNode

(b) sHOT stores leaf entries in leaf nodes only

Figure 4.5: A HOT and an sHOT representation for the same binary Patricia trie and a maximum fanout of $k = 3$. Differences between both representations are highlighted in green.

To transform an sHOT structure into a HOT structure, it is sufficient to hoist all single entry leaf nodes into their parent nodes. An example of this transformation is depicted in Figure 4.5. To hoist a single element leaf entry into its parent node, its corresponding boundary entry in the parent node is replaced with the leaf entry itself.

Lemma 6. *Hoisting all single element leaf entries of an sHOT structure generates an equivalent HOT structure of the same height.*

Proof. By hoisting single element leaf entries, the fanout of the leaf entries' previous parent node is not changed. Additionally, hoisting single element leaf entries does not influence the assignment of BiNodes to their corresponding compound nodes. Therefore, the minimum cardinality constraint (see Section 4.5), which ensures that the descendant BiNodes can not be merged, still holds. As neither the maximum fanout constraint nor the minimum cardinality constraint of individual BiNodes is affected, no subsequent node merges or node splits, which could affect the overall tree height are possible. Hence, the tree height of the resulting HOT is equal to the height of the original sHOT in the first place. □

Lemma 7. *Inserting a new key into HOT is identical to first inserting the key into an equivalent sHOT and then transforming it into a HOT by hoisting all single element leaf nodes into their parent nodes.*

Proof. The proof is by induction on the number of keys n.

Base Case: Up to k keys, both structures consist of only a single node and therefore are trivially equal.

Step Case: In the following, we will show the equivalence of sHOT and HOT for each type of insertion operation separately. A HOT containing $n+1$ entries results from inserting a new entry into a HOT containing n entries, for which the induction hypothesis holds. Hence, we assume that the corresponding HOT and sHOT structures before each insertion operation are equivalent.

- **Normal Insert:** By definition of HOT, normal insertion only modifies the affected node n_{aff}. If n_{aff} is a leaf node the updated affected node after the insertion operation is the same for both structures. If n_{aff} is a non-leaf node, HOT's normal insert creates a new node n'_{aff} by adding the discriminating BiNode and the new leaf entry to n_{aff}, whereas sHOT's normal insert creates two new nodes n_{new} and $\hat{n}'_{aff}$. The new leaf node n_{new} consists only of the leaf entry corresponding to the newly inserted value. The modified node $\hat{n}'_{aff}$ is constructed by adding the discriminating BiNode and a boundary entry linked to n_{new} to sHOT's affected node $\hat{n}_{aff}$. According to our induction hypothesis, the node $\hat{n}_{aff}$ can be transformed into the node n_{aff} by hoisting its single element child nodes. Therefore by hoisting all single element child entries of $\hat{n}'_{aff}$ including n_{new} we obtain n'_{aff}. Hence, normal insert on two equivalent HOT and sHOT structures yields two equivalent structures.

- **Leaf Node Pushdown:** Leaf node pushdown of HOT is only applicable whenever the mismatching vertex v_{mis} is a leaf in a non-leaf node. By design sHOT does not allow leaf vertices in non-leaf nodes. Thus, in equivalent sHOT structures such vertices are represented by single entry leaf nodes. According to the induction hypothesis, we consider two equivalent sHOT and HOT structures where inserting the value x triggers a leaf node pushdown in the HOT structure. For HOT, the leaf node pushdown creates a new node containing v_{mis}, the discriminating BiNode and the vertex corresponding to x. For the equivalent sHOT, a normal insert is applied to the single entry leaf node containing v_{mis}. Thereby, it creates a new node, which is identical to the node created by leaf node pushdown containing the leaf vertices v_{mis} and x and the discriminating BiNode. By definition of sHOT the newly created node—containing two leaf entries after the insertion—will no longer be considered for hoisting. However, this is not necessary, as the two nodes containing v_{mis}, x and the discriminating BiNode are identical. As no other single entry leaf node is modified and due to the induction hypothesis, applying our transformation algorithm will therefore create an identical HOT. Thus, leaf node pushdown on

HOT is identical to first inserting the same value into an equivalent sHOT structure and applying the transformation algorithm from sHOT to HOT.

- **Overflow Handling:** Whenever the number of entries contained in the affected node is k and leaf node pushdown is not applicable, inserting a new value will violate the maximum fanout constraint and the resulting overflow must be handled. For simplicity's sake, we split overflow handling into two separate logical parts. First, we apply a normal insertion. In doing so we violate the maximum fanout constraint. Thus, secondly, we apply the respective overflow handling strategy, which is either an intermediate node creation or a parent pull up. We have already shown that for two equivalent sHOT and HOT structures the resulting structures after a normal insertion are again equivalent. The actual overflow handling strategies consist of two parts: splitting the affected node at its root BiNode and placing the root BiNode into its destination node. For simplicity's sake, we assume that an intermediate node containing the root BiNode is always created. In case of parent pull up this intermediate node is then hoisted into its parent. For the split operation, we distinguish two cases. If each subtree contains at least a single non-leaf entry both sHOT and HOT create identical nodes for the intermediate node as well as for each subtree. Hence, equality trivially holds. If one subtree consists of exactly one leaf entry, sHOT creates a new single entry leaf node for this subtree, whereas HOT integrates this leaf entry into the intermediate node. However, by applying our transformation operation, we hoist the single entry into the intermediate node, which again yields two equivalent structures. Thus, intermediate node creation for two equivalent HOT as well as sHOT structures yields equivalent structures.

  Next, we focus on parent pull up, where the intermediate node is hoisted into its parent node. We first assume that hoisting the root BiNode into its parent node does not trigger an additional overflow. As we have previously shown that the results of intermediate node creation are two equivalent structures, we can infer that pulling the intermediate node up may only influence child nodes of the intermediate node. In particular, this holds for single element leaf entries. Normally in the case of sHOT this single element entry would be hoisted into its parent node. After the parent pull up, the leaf node is hoisted into the original parent of the intermediate node. Nonetheless, this does not affect the equivalence of the resulting structures. As HOT for parent pull up, also moves the leaf entry—

which was previously part of the intermediate node—into the parent node, equivalence again holds.

We finally consider the recursive case, which occurs when a parent pull up causes another overflow. As we are able to split the recursive case in a "normal" parent pull up and an overflow handling step, we can reuse our previous proof, which has first shown the equivalence of sHOT and HOT for split, intermediate node creation and finally parent pull up. Thus, for all types of overflow handling, we have shown that for the same set of keys HOT and sHOT produce equivalent structures

□

By proving the equivalence of HOT and sHOT we have also proven the equivalence of HOT and SMHP. Thus, we have proven the three core properties of HOT:

(I) **Determinism** by Lemma 2 : For the same set of keys, regardless of the insertion order, the compound node structure generated by HOT is always the same.

(II) **Minimum Height** by Lemma 1 from [71]: No partitioning of a binary Patricia trie under a maximum fanout constraint k can be found, such that the height of the induced tree structure is lower.

(III) **Recursive Structure** by Lemma 3 : Each subtree of HOT is identical to a HOT structure that only contains those keys, which are contained in the respective subtree.

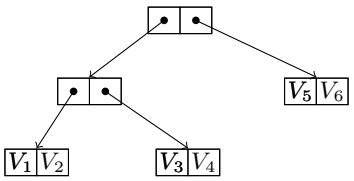
CHAPTER 5

Designing Node Structures for Height Optimized Tries

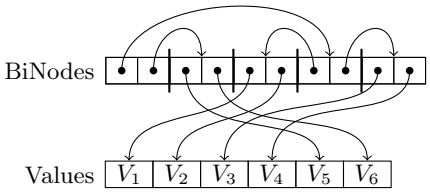
5.1 Overview

The Height Optimized Trie introduced in Chapter 3 is a trie-based index structure that uses a novel algorithm to minimize the overall tree height of binary Patricia trie structures by combining multiple BiNodes into k-constrained compound nodes. In Chapter 4, we showed that the achieved tree height is optimal with regards to a previously chosen constraint k. However, we have not discussed the actual physical layout of these compound nodes in memory. To benefit from a Height Optimizing Trie, the use of a fast and space-efficient compound node layout is crucial. In this chapter, we therefore introduce different memory layouts for k-constraint tries.

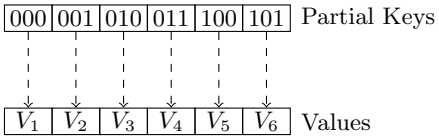
In this thesis, we particularly focus on node layouts geared towards the use in main memory database systems. However, we note that designs optimized for other storage systems are also possible.



(a) Sample Binary Tree Structure



(b) Hierarchical Node Layout



(c) Linearized Node Layout

Figure 5.1: An example of a hierarchical and a linearized node layout for the same binary tree structure.

An important aspect when designing structures for main memory systems is to improve cache locality and to restrict the number of cache misses. A well-known technique that improves cache locality is to separate keys and pointers in nodes [50, 75, 18]. We apply this technique to all our memory layouts by dividing a node's memory into two contiguous chunks of memory. The first part contains the actual structure of the k-constrained trie used for searching and the second part consists of pointers to child nodes or leaf entries in key order. We denote the first part of each node as the *structural area* and the latter as the *data area*. Accordingly, the search operation for all node types consists of two stages. First, the index of the entry in the data area is determined by searching the structural area. Secondly, the corresponding entry in the data area is fetched.

As this overall scheme applies to all our node designs, we focus on the different structural representations. We classify our node designs according to their search operations into two categories. Nodes in the first category are searched by traversing a hierarchical structure. Nodes in the second category are searched using linear search or equivalent data-parallel SIMD operations. Hence, we denote node layouts falling into the first category as *hierarchical node layouts* and node layouts falling into the second category as *linearized node layouts*. Figure 5.1 illustrates the difference between a hierarchical and a linearized node layout for a sample binary tree structure.

Although the various hierarchical and linear node layouts use different approaches to represent a k-constrained binary Patricia trie, they share two

design decisions. All our node layouts are designed to be as space-efficient as possible. Hence, instead of reserving spare memory for in-place updates, we use a copy-on-write approach for updates. Besides saving space, applying copy-on-write techniques allows for concurrent operations (cf. Chapter 6). As we target contemporary 64-bit hardware architectures, all node layouts store 64-bit values. These values are either pointers to other nodes or tuple identifiers. We use the most significant bit to distinguish between a pointer and tuple identifiers. In the case that stored values require less than 64-bits, we embed the value directly in the tuple identifier.

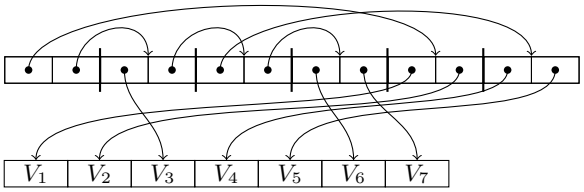
In the remainder of this chapter, we first introduce the hierarchical node layout and different variations thereof. Secondly, we introduce the linearized node layout and its SIMD based optimizations. Finally, we present an alternative node splitting algorithm and its effects on linearized node layouts.

We will conduct an extensive evaluation of the individual node layouts with regards to access performance and memory consumption in Chapter 7.

5.2 Hierarchical Node Layouts

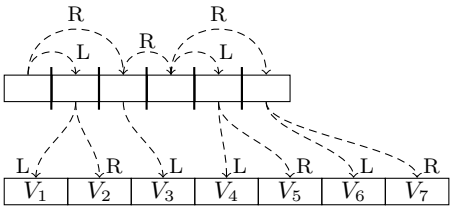
On a logical level, each compound node of a Height Optimizing Trie is itself a binary Patricia trie structure with at most k entries. Typically, a binary Patricia trie is represented by a linked structure over its BiNodes in memory. In such a structure, each BiNode is a triple consisting of the discriminative bit position and a pointer to the left and right child, respectively. This simplicity is the major advantage of binary Patricia tries. However, as BiNodes can be scattered arbitrarily in memory and child pointers make up for a large fraction of a BiNodes memory, they suffer from bad cache locality and insufficient memory consumption.

In this section, we present five different approaches for hierarchical node layouts for k-constrained tries which embrace the simplicity of a hierarchical layout, while improving cache locality and memory efficiency. Figure 5.2 shows an overview of these approaches that we discuss in detail in the following subsections. First, we introduce the direct node layout that uses node-relative pointers and places all its BiNode contiguously in memory. Next, we introduce the indirect node layout that stores BiNodes in pre-order and exploits the inherent properties of pre-order stored full binary trees to omit redundant information. Finally, we present the leaf optimized node layout and the generalized indirect node layouts. Both extend the concept of the indirect node layout by exploiting densely populated areas in a node's tree structure to reduce the overall memory footprint.



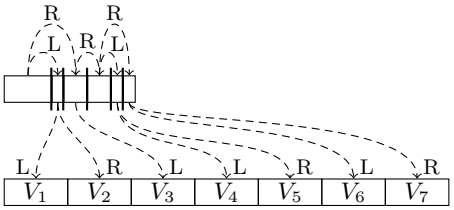
(a) Direct Node Layout.

- Fixed-sized inner nodes
- Explicit tree structure



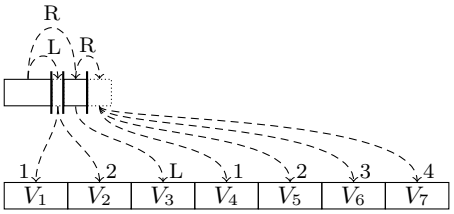
(b) Indirect Node Layout.

- Smaller inner nodes
- Implicit tree structure



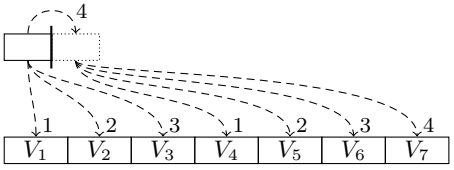
(c) Variable Length Indirect Node Layout.

- Variable sized inner nodes



(d) Leaf Optimized Node Layout.

- Higher arity inner nodes for leaf dense areas



(e) Generalized Indirect Node Layout.

- Higher arity inner nodes also for inner dense areas

Figure 5.2: Overview of the proposed hierarchical node layouts and how they build upon each other. The first row of each node layout shows the structural area containing the inner Nodes. The second line corresponds to the node's data area with values V_1 to V_n. Solid arrows represent actual pointer, while dashed lines represent inferred pointers. Dotted rectangles represent virtual inner nodes. Labels L and R denote left and right child pointers of inner BiNodes, while labels 1-n denote child pointers of higher arity inner nodes.

5.2.1 Direct Node Layout

The *Direct node Layout (DL)* is our first approach for a hierarchical node layout. In contrast to a normal binary Patricia trie, the direct node layout places all its BiNodes in the structural area of the node. Hence, we can use relative addresses for node internal pointers. As a binary Patricia trie with n leaf entries contains exactly n-1 BiNodes, we use addresses 0 to n-2 to reference a node's BiNodes and addresses n-1 to $2n$-2 to reference a node's leaf entries.

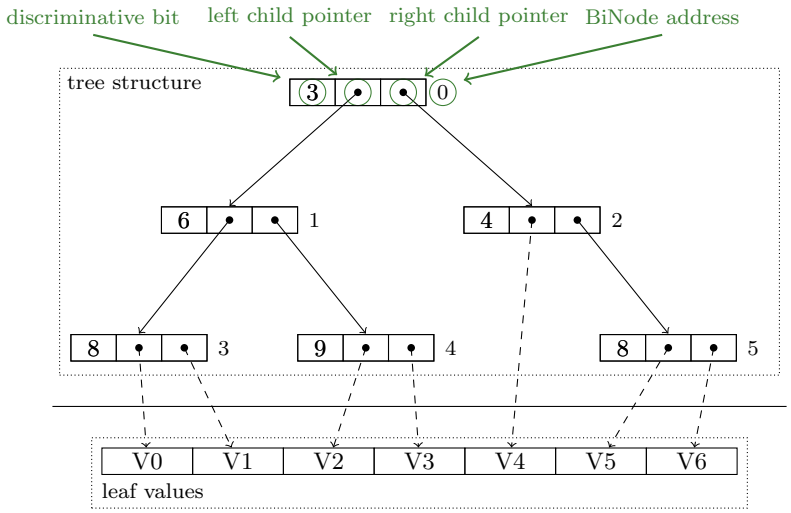


Figure 5.3: Traditional binary Patricia trie containing seven entries. BiNodes are represented as a triple of discriminative bit, left and right child pointer. Next to each BiNode we specify the address, where it is placed in memory. Entries $V0$-$V6$ represent the actual leaf values.

We show an example of a DL in Figure 5.4. The DL shown corresponds to the binary Patricia trie shown in Figure 5.3. As the range of the child addresses used to link BiNodes and leaf entries inside a node is bound by the node's maximum fanout k, optimized data types can be used for internal pointers instead of general-purpose 64-bit pointers. For instance, 8-bit integers are sufficient to store the internal pointers of nodes with a maximum fanout of up to 128. Besides reducing the memory footprint, this also reduces cache misses as fewer cache lines are required to store the same number of BiNodes.

With relative addressing being the only difference to a traditional binary Patricia trie, it retains the simplicity of its access operations. The actual search algorithm is shown in Listing 5.1.

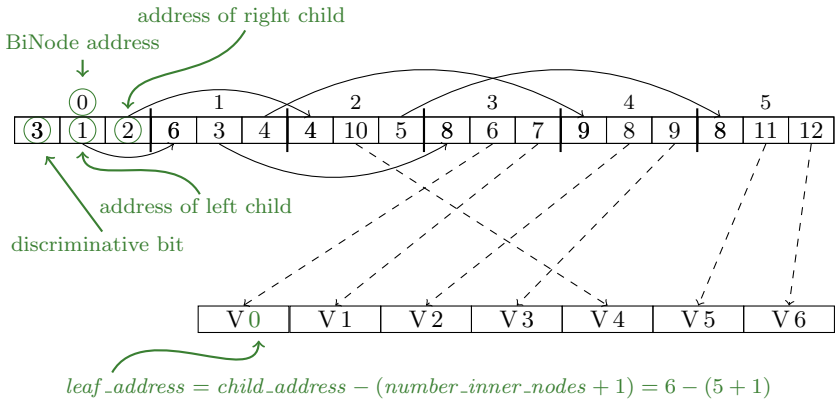


Figure 5.4: Direct node layout of the binary Patricia trie containing seven leaf entries illustrated in Figure 5.3. Child pointers with addresses 0-5 reference BiNodes. Child pointers with addresses 6-12 reference leaf entries.

```
1 Leaf* search(uint8_t const * key) const {
2   uint currentAddr = root;
3   while(currentAddr < numberBiNodes) {
4     BiNode* current = biNodes[currentAddr];
5     uint discrBit = extractBit(key, current->discrBitPos);
6     currentAddr = current->children[discrBit];
7   }
8   return leafs[currentAddr - numberBiNodes];
9 }
```

Listing 5.1: Lookup in naïve pointer based k-constrained tries.

5.2.2 Indirect Node Layout

While already an improvement to traditional binary Patricia tries, the direct node layout presented in the previous Section 5.2.1 requires storing the triple of discriminative bit, left- and right- child pointer for each BiNode. In this section, we introduce the *Indirect node Layout (IL)*, which reduces the memory footprint in comparison to a DL, by using an indirect representation of the tree structure in memory. To create such an indirect representation we exploit the following inherent properties of a full binary tree, stored in pre-order.

When storing a binary tree in pre-order, the left child of an inner node is the inner node's direct successor in memory and its right child is located at the address directly succeeding all inner nodes and leaf entries of the left subtree. Accordingly, a pre-order arranged direct node layout stores each right child leaf entry at the address in the data area, which is subsequent to all leaf entries contained in its parent's left subtree.

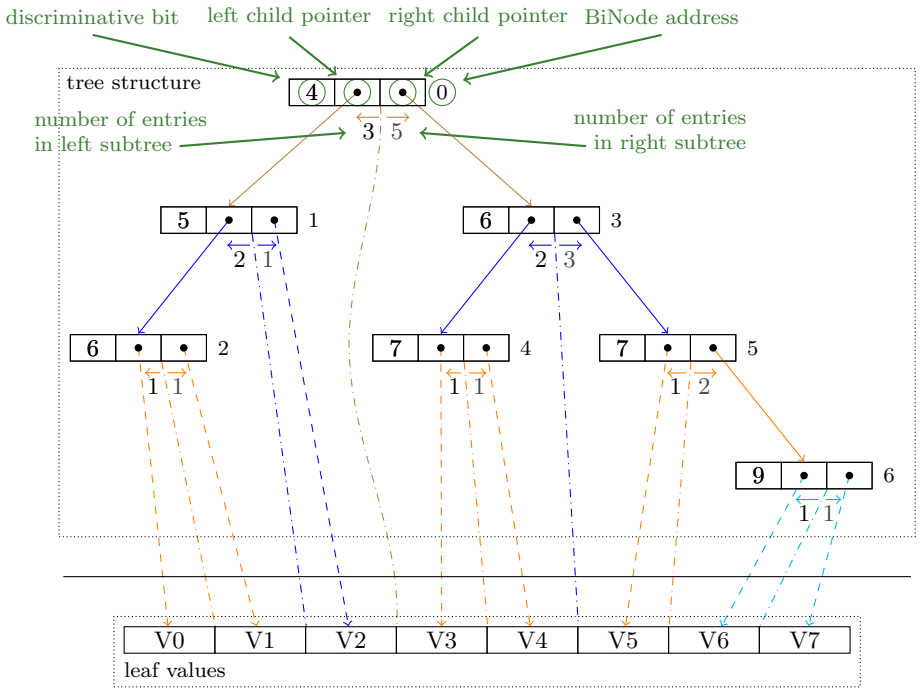


Figure 5.5: A binary Patricia trie stored in pre-order. We annotate each BiNode with the number of leaf entries in its left and right subtree.

Figure 5.5 illustrates such a pre-order arranged binary Patricia trie. In this example, address 0 stores the root inner node. The subsequent address 1 stores the root's left inner node, while address 3—the next address after all inner nodes in the root's left subtree—holds the root's right inner node. Furthermore, the figure highlights another trivial property of pre-order arranged node layouts. Namely that leaf entries contained in an inner node's right subtree are placed subsequent to all leaf entries contained in its left subtree. For instance, the left subtree of the inner node located at address 3 contains values V3 and V4 and the right subtree values V5, V6, and V7. Hence, the shown layout places the values V5-V6 right after values V3 and V4.

Another interesting property of each full binary tree, and hence of every Patria trie, is that a full binary tree containing n leaf entries contains exactly $n - 1$ inner nodes. In combination with a pre-order arranged storage layout, the correlation between the number of leaf entries and the number of inner nodes allows us to infer the address of right child inner nodes solely from its parent's address and the number of leaf entries in its parent's left subtree. More so, for an arbitrary inner node, we can infer the address of the right subtree's smallest leaf entry, from the first leaf entry of the inner nodes subtree and the number of entries in the left subtree. For example, the left subtree of the binary Patricia

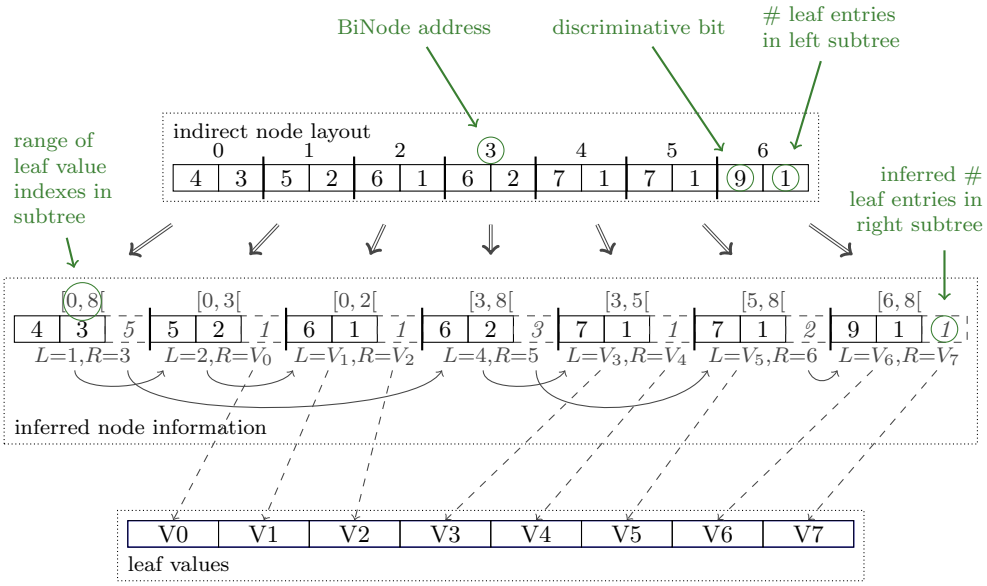


Figure 5.6: An indirect node layout containing eight entries. The box in the second line illustrates how the actual tree structure can be inferred from the information stored in the indirect node layout. The letters L and R, which denote the inferred pointers to the left and right child of the respective BiNodes, represent this tree structure.

trie shown in Figure 5.5 contains three leaf entries and hence exactly two inner nodes. As the address of the root inner node is 0 in the structural area, we infer the address of the root's right child inner node to be 3 in the structural area. Further, from the address of the root's smallest leaf, which is trivially 0, and the three entries contained in the root's left subtree we infer address 3 to be the first leaf entry's address in the root's right. Based on the fact, that the exemplary tree structure contains seven leaf entries, and the left subtree contains three leaf entries, we are able to infer the number of leaf entries in the right subtree to be four. Thus, for the subtree rooted at the inner node at address 3 in the structural area, we infer its number of leaf entries and the address of the subtree's first leaf. If for this inner node only the additional information of the number of entries contained in its left subtree is provided, we are recursively able to infer the properties of its descending subtrees.

Thus, based on the properties that for any full binary tree stored in pre-order, the overall number of entries together with the number of entries in each inner node's left subtree is sufficient to infer the tree's structure, we introduce the *indirect node layout (IL)*. In comparison to a direct node layout, the indirect node layout always stores its inner nodes in pre-order. For each inner node,

```
1  Leaf* search(uint8_t const * key) const {
2    uint currentAddr = root;
3    // entries in the current subtree
4    uint remaining = totalNumberEntries;
5    // index of first leaf entries in current subtree
6    uint indexOfFirstLeaf = 0
7    while(remaining > 1) {
8      BiNode* current = biNodes[currentAddr];
9      if(extractBit(key, current->discrBitPos) == 0) {
10         currentAddr = currentAddr + 1;
11         remaining = current->entriesInLeftSubtree;
12     } else {
13         currentAddr = currentAddr + current->entriesInLeftSubtree;
14         indexOfFirstLeaf = indexOfFirstLeaf + current->entriesInLeftSubtree;
15         remaining = remaining - current->entriesInLeftSubtree;
16     }
17   }
18   return leafs[indexOfFirstLeaf];
19 }
```

Listing 5.2: Lookup in indirect node layout.

the indirect node layout only stores the number of entries contained in its left subtree instead of direct pointers to its child pointers. Figure 5.6 shows such an indirect node layout for the binary Patricia trie illustrated in Figure 5.5. It illustrates all the properties of an inner node that can be inferred from an indirect layout. These are the number of entries in the right subtree, the value range of the corresponding leaf entries, and the address of the left and right subtree. By inferring these properties, only $log_2(k)$ bits are required for a given k for each inner node in an indirect node layout. In contrast, at least $2 * log_2((2 * k) - 1)$ bits are required to represent an inner node in an equivalent direct node layout. To search an indirect node layout, we traverse the node layout starting from the root by recursively inferring these properties for each inner node along the path until the number of entries in the currently considered subtree is 1 and we reach the leaf level. The search algorithm is shown in Listing 5.2.

We conclude that the indirect node layout improves over the direct node layout by retaining the simplicity of the DL's search operation, while at the same time reducing the amount of memory required to store an equivalent k-constrained binary Patria trie by more than a half.

5.2.3 Variable Length Indirect Node Layout

Even though the indirect node layout introduced in Section 5.2.2 improves the direct node layout introduced in Section 5.2.1, it requires the same amount of memory, regardless of the dataset stored. For instance, in case the tree structure resembles a complete binary tree structure, we could omit to store the size of the left subtree at each inner node, as in this specific case all inner nodes are perfectly balanced and the number of entries in the left subtree and right subtree is always equal. Alternatively, in case the parent discriminative bit b_p of any node n with discriminative bit b_n is exactly $b_p = b_n - 1$, storing the discriminative bit for all inner nodes except for the root node is also redundant, because the discriminative bit can always be inferred from the parent inner node. If the structure satisfies both of these special cases, storing only the discriminative bit of the root node would be sufficient. Please note that this special case only occurs if a dense range of keys is stored (e.g., all natural numbers up to a certain value x). However, even in the case of storing uniformly distributed random keys such densely populated balanced regions occur. Specifically, the upper-level inner nodes of a binary Patricia trie containing uniformly distributed random data tend to be (almost) balanced.

In the remainder of this section, we therefore introduce a novel node layout that is able to exploit such regularities in tree structures. First, we present the *Delta encoded Indirect node Layout (DIL)* that by using an alternative encoding scheme lends itself as a basis for compressed node layouts. Secondly, we introduce the *Variable length Indirect node Layout (VIL)* which enhances the delta encoded indirect node layout with an efficient variable length encoding scheme reducing the memory consumption of nodes.

To create an indirect node layout that encodes balanced nodes more efficiently than nodes with a skewed distribution, we have to change the encoding for the number of entries contained in an inner node's left subtree. Specifically, such an encoding has to use larger numbers for the cardinality of a skewed inner node's left subtree and smaller numbers for the cardinality of a balanced inner node's left subtree. We exploit the fact that in a completely balanced inner node, half of its descending entries are stored in its left subtree and the other half in its right subtree. Hence, to create an encoding that has the desired properties we only store the delta between the actual number of entries in the left subtree and half of the number of entries in the whole subtree.

To further optimize an inner node's storage layout, we can apply a similar delta encoding technique for storing the discriminative bit information. To optimize the encoding of discriminative bits, we leverage that the discriminative bit of a child inner node is always larger than the discriminative bit of its parent

inner node. Hence, we only need to store the delta between the smallest possible discriminative bit—determined by the successor of its parent inner node's discriminative bit—and the actual discriminative bit. Thus, if a child's inner node's discriminative bit is the successor of its parent's inner node, the delta encoded discriminative bit is zero.

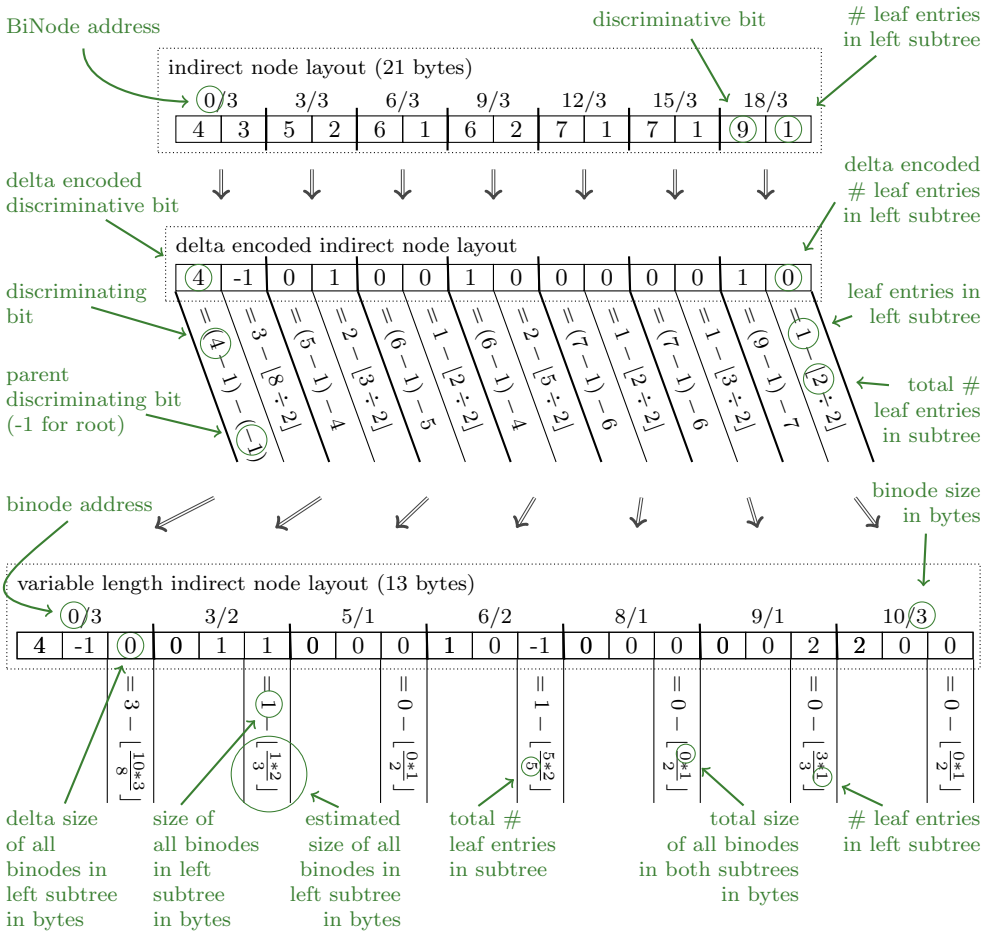


Figure 5.7: The transition from an indirect node layout of the binary Patricia trie shown in Figure 5.5 to a variable length indirect node layout.

An example of a delta encoded indirect node layout for the tree structure depicted in Figure 5.5 is shown in Figure 5.7. It shows for each inner node how the information stored in an indirect node layout is transformed into a delta encoded indirect node layout. For each discriminative bit, it shows that the delta encoded discriminative bit can be inferred from the indirect node layout by applying Equation 5.1.

$$delta\_disc\_bit = (disc\_bit - 1) - (parent\_disc\_bit) \tag{5.1}$$

Additionally, the figure shows for each inner node that Equation 5.2 determines the delta between the actual number of entries in the left subtree and the expected number of entries in a balanced subtree with the same number of entries.

$$delta\_no\_left\_entries = actual\_no\_left\_entries - \lfloor \frac{total\_no\_entries}{2} \rfloor \quad (5.2)$$

Although a delta encoded indirect node layout encodes balanced dense trees more efficiently, due to the fixed-sized inner node layout it still has the same memory requirement as an indirect node layout. Hence, to benefit from the delta encoding we have to adapt the inner node layout to the content of the encoded inner nodes. However, naively using dynamically sized inner nodes prevents our addressing scheme from deriving the location of child inner nodes from the number of leaf entries in the left subtree. As we store the tree structure in pre-order, a potential solution is to store the offset of the right child inner node for each inner node. In the worst case, this offset can be as large as the total size of the structural area. This contradicts our aim to reduce the memory footprint of the overall node layout by using variable sized inner nodes. To still achieve our goal of reducing memory consumption by using variable sized inner nodes, we exploit the following assumption. When using variable sized nodes, we assume that the ratio between the memory requirement of inner nodes in the left subtree and the memory requirement of inner nodes in the right subtree correlates with the ratio between the number of leaf entries in the left subtree and the number of entries in the right subtree in the common case. We denote the memory requirement of a subtree's inner nodes as the *size* of the subtree's structure.

We leverage the relationship between the sizes of an inner node's subtrees and the number of entries contained in the corresponding subtrees to estimate the size of the left subtree by the following equation:

$$left\_subtree\_size\_estimation = \frac{total\_subtree\_size}{\#entries\_total} * \#entries\_left\_subtree \quad (5.3)$$

Based on the assumption that the estimated size of a subtree's structure is close to its actual size, we propose the *Variable length Indirect node Layout (VIL)* layout. Instead of storing the actual size of its left subtree structure

```
1  Leaf* search(uint8_t const * key) const {
2    // the address of the current BiNode
3    uint currentAddr = root;
4    // the position of the previously considered BiNode discriminative bit
5    uint discrBitPos = -1;
6    // entries in the current subtree
7    uint remaining = totalNumberEntries;
8    // index of first leaf entries in current subtree
9    uint indexOfFirstLeaf = 0
10   // total size of biNodes in the current subtree in bytes
11   uint subtreeSize = sizeOfStructuralArea;
12   while(subtreeSize > 0) {
13     // readBiNode is able to deal with variable sized biNodes
14     BiNode* current = readBiNode(currentAddr);
15     discrBitPos += 1 + current->deltaDiscrBitPos;
16     subtreeSize -= current->biNodeSize;
17     uint noLeftEntries = remaining/2 + current->leftSubtreeEntriesDelta;
18     uint estimatedLeftSize = (subtreeSize * noLeftEntries)/remaining;
19     uint leftSize = estimatedLeftSize + current->leftSubtreeSizeDelta;
20     if(extractBit(key, discrBitPos) == 0) {
21       currentAddr = current->biodesSize;
22       remaining = noLeftEntries;
23       subtreeSize = leftSize;
24     } else {
25       currentAddr += leftSize;
26       indexOfFirstLeaf += noLeftEntries;
27       remaining -= current->entriesInLeftSubtree;
28       subtreeSize -= leftSize;
29     }
30   }
31   return leafs[indexOfFirstLeaf];
32 }
```

Listing 5.3: Lookup in variable length indirect node layout.

for each inner node, the VIL only stores the delta between the left subtree's actual structure size and its estimation.

We show an example of a variable length indirect node layout in Figure 5.7. The figure depicts how a VIL is derived from the IL and the DIL of the binary Patricia trie structure shown in Figure 5.5. It can clearly be seen that even though the structural area of the example's VIL requires 13 bytes, the delta size of the left subtree's structure of any inner node in the example is at most two and even zero for more than half of the examples' inner nodes.

Although the VIL is able to exploit regularities in the data to reduce the memory consumption in comparison to a plain IL, it only adds minimal overhead to the search operation in comparison. To emphasize this simplicity, we show the VIL's search algorithm in Listing 5.3. Please note that the actual runtime

performance depends on the precise memory layout of a single inner node and the encoding techniques used to compress a single inner node.

5.2.4 Node Layouts for Compacted Dense Regions

The memory efficient node layouts introduced in Section 5.2.2 and Section 5.2.3 base on the assumption that all inner nodes have to be represented explicitly. However, we show in this section that depending on the stored dataset it is possible to completely omit certain inner nodes. Specifically, we focus on approaches exploiting so-called dense regions to design node layouts with increased memory efficiency.

We define dense regions in a dataset to be a set of 2^k successive keys that share a common prefix and cover all possible 2^k combinations of the k bits following the common prefix. For instance, the four binary keys 101**00**110, 101**01**110, 101**10**110 and 101**11**110 represent a dense region that shares the common prefix 101 and covers all possible combinations of the successive bits 3-4. In the following, we call these successive bits the region's *distinctive bits*. In a binary Patricia trie structure, dense regions are subtrees of binary Patricia tries that have the shape of perfect binary trees, and therefore, all leaf entries have the same depth. Except for the region's root, the discriminative bit $b_n = b_{parent} + 1$ of each inner node n directly succeeds its parent's discriminative bit b_{parent}.

We distinguish two kinds of dense regions: *leaf dense regions* and *inner dense regions*. While nodes in inner dense regions may have descendant nodes that are not part of the same dense region, nodes in leaf dense regions only have descendant nodes that are part of the same dense region. Hence, leaf dense regions of height h always contain 2^h leaf nodes. In the Patricia trie depicted in Figure 5.8, we identify three dense regions, one of height three and two of height two. The inner dense regions in this figure are outlined in orange and leaf inner regions are outlined in blue.

In the remainder of this section, we exploit the property of dense regions, that for a given key contained in a subtree of the dense region, we can identify which subtree of the region the given key is contained in by only considering the region's distinctive bits. For instance, in a dense region of height 2, the distinctive bits 00 address the first subtree, 01 the second, 10 the third, and 11 the last subtree. This addressability allows us to replace the subtree corresponding to the dense region by a single node with fanout 2^k.

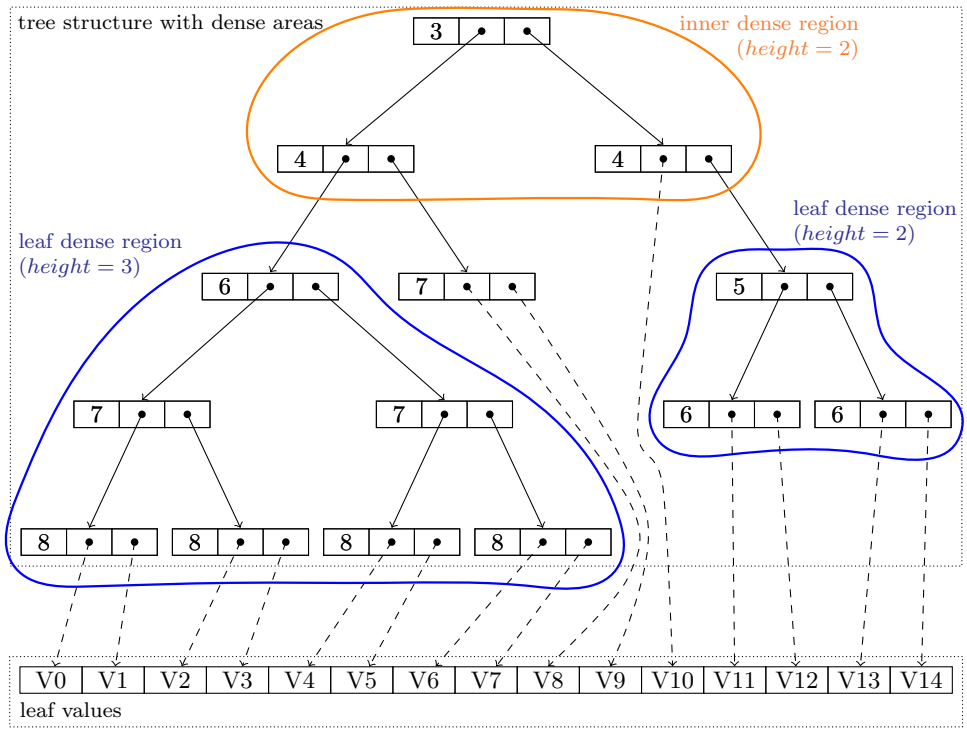


Figure 5.8: Binary Patricia trie containing three dense regions. Inner dense regions are highlighted in orange, while leaf dense regions are highlighted in blue.

In contrast to level compressed tries [5, 86] (c.f. Chapter 2.2.2) that use dense regions to determine the number of bits considered per array-based node, we leverage dense regions to optimize our internal node structures.

In the following, we present two approaches to extend our variable length indirect node layout by exploiting dense regions. Our first approach considers only leaf dense regions, whereas our second approach considers both leaf dense regions and inner dense regions. Both approaches have in common that they combine the BiNodes of the respective dense regions into higher degree nodes. We denote the transformation of a region's BiNodes into a single inner node of higher arity as *collapsing* and the resulting inner nodes as collapsed dense regions, accordingly. For the binary Patricia trie shown in Figure 5.8, we depict an equivalent structure with only its leaf dense regions collapsed in Figure 5.9 and an alternative equivalent structure with both its leaf and inner dense regions collapsed in Figure 5.11.

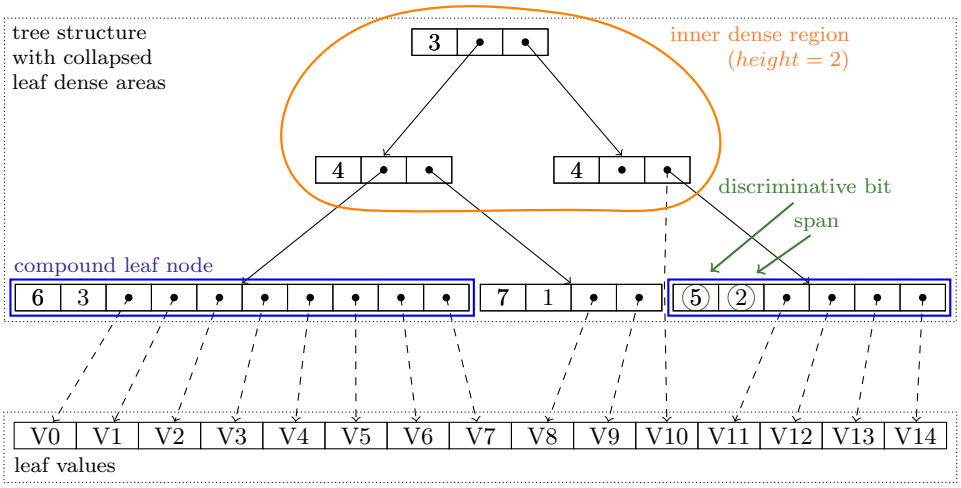


Figure 5.9: A binary Patricia trie with collapsed dense regions.

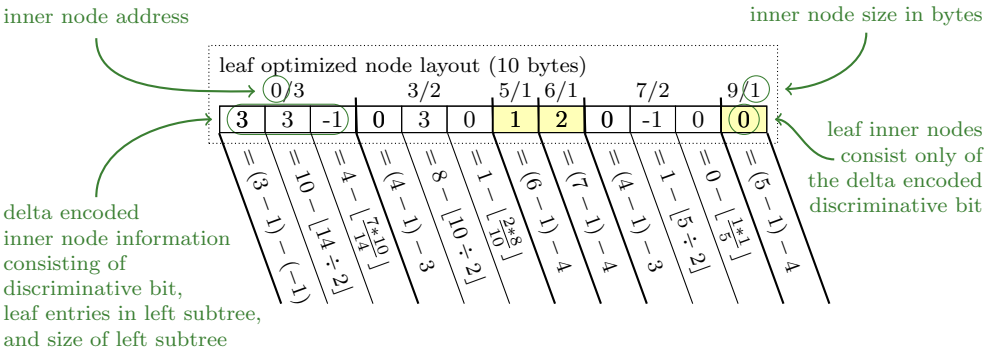


Figure 5.10: A leaf optimized node layout for the binary Patricia trie shown in Figure 5.9. A yellow background highlights the collapsed leaf nodes.

Leaf Optimized Node Layout Our first approach to create a node layout that exploits dense regions is the *Leaf Optimized node Layout (LOL)*. LOL is based on the VIL presented in Section 5.2.3. VIL uses delta encoding to implicitly store the discriminative bit for each BiNode and the ratio between the size and number of elements in its left subtree and the size and the number of elements in its right subtree. Notably, the delta encoding of these ratios all become zero for BiNodes contained in leaf dense regions. We exploit this property of variable length indirect node layouts to omit all BiNodes contained in leaf dense regions except for the region's root node. Even for the region's root BiNode, we only have to store the discriminative bit position. This observation leads us to the actual layout of LOL that we present in the following.

The leaf optimized layout extends the variable length indirect node layout by first collapsing all dense regions and secondly by storing only the first discriminative bit position for these newly created leaf collapsed regions.

```
Leaf* search(uint8_t const * key) const {
  // the address of the current bi node
  uint currentAddr = root;
  // the position of the previously considered BiNode discriminative bit
  uint discrBitPos = -1;
  // entries in the current subtree
  uint remaining = totalNumberEntries;
  // index of first leaf entries in current subtree
  uint indexOfFirstLeaf = 0
  // total size of biNodes in the current subtree in bytes
  uint subtreeSize = sizeOfStructuralArea;
  while(subtreeSize > 0) {
    // readBiNode is able to deal with variable sized biNodes
    BiNode* current = readBiNode(currentAddr);
    discrBitPos += 1 + current->deltaDiscrBitPos;
    subtreeSize -= current->biNodeSize;
    if(subtreeSize > 0) {
      uint noLeftEntries = remaining/2 + current->leftSubtreeEntriesDelta;
      uint estimatedLeftSize = (subtreeSize * noLeftEntries)/remaining;
      uint leftSize = estimatedLeftSize + current->leftSubtreeSizeDelta;
      if(extractBit(key, discrBitPos) == 0) {
        currentAddr = current->biodesSize;
        remaining = noLeftEntries;
        subtreeSize = leftSize;
      } else {
        currentAddr += leftSize;
        indexOfFirstLeaf += noLeftEntries;
        remaining -= current->entriesInLeftSubtree;
        subtreeSize -= leftSize;
      }
    } else {
      // a leaf entry is reached
      int span = log2(remaining);
      indexOfFirstLeaf += extractBits(key, discrBitPos, span);
    }
  }
  return leafs[indexOfFirstLeaf];
}
```

Listing 5.4: Lookup in leaf optimized node layout.

As leaf BiNodes can be regarded as a special case of a dense region containing only two entries, we only have to store the discriminative bit position also in these cases. In Figure 5.10, we show the resulting leaf optimized layout for the binary Patricia trie with collapsed dense regions of Figure 5.9.

In Figure 5.10, the collapsed inner nodes are highlighted in yellow. We show the search algorithm in Listing 5.4. The search algorithm extends the search algorithm of a variable length indirect node layout by deriving the span of the collapsed leaf value from the number of entries in the respective dense region. Based on the derived span, the algorithm extracts all bits corresponding to the collapsed leaf region and uses these bits of the search key for addressing the potential match candidate relative to the leaf region's start. To determine whether the current BiNode is a "normal" inner BiNode or the root of a collapsed dense region, we have to check if the size of the current subtree is equal to the current BiNode's size. Although LOL's search algorithm requires more operations per inner node than VIL's search operation, depending on the used dataset and the number of its collapsible dense regions, LOL compensates this overhead by omitting inner nodes in collapsed leaf dense regions.

Generalized Indirect Node Layout While LOL is able to collapse dense regions at the leaf level, it is not able to collapse dense regions closer to the root. We therefore present the *Generalized Indirect node Layout (GIL)*, which collapses all dense regions from bottom to top into higher degree internal nodes.

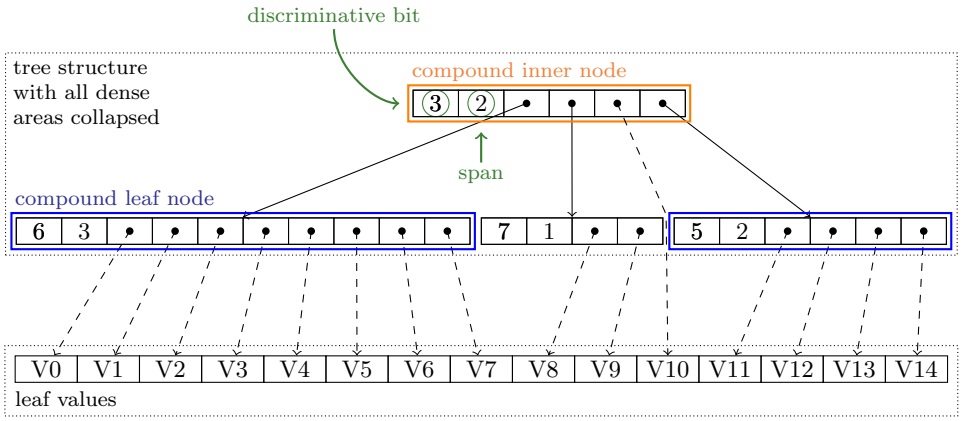


Figure 5.11: A binary Patricia trie with all dense regions collapsed.

We depict a schematic overview of a Patricia trie with all its internal nodes collapsed in Figure 5.11. We denote such a representation as fully collapsed. In this figure, we highlight the collapsed inner dense regions in orange and the leaf dense regions in blue. While for the same binary Patricia trie a representation

collapsing only leaf dense regions as shown in Figure 5.8 has three levels, its fully collapsed representation only requires two levels.

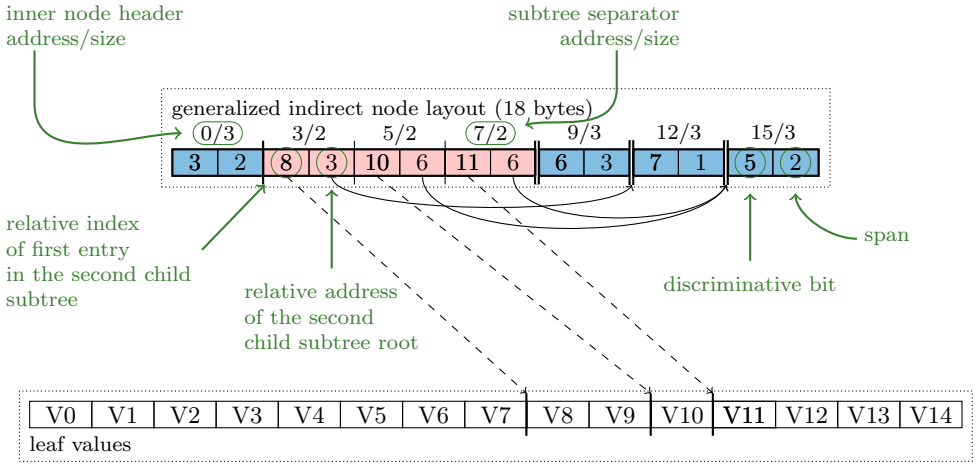


Figure 5.12: A generalized indirect node layout for the tree shown in Figure 5.11. A blue background highlights node headers, while a red background highlights subtree separators.

To encode these higher degree internal nodes, we generalize the indirect node layout to support nodes with more than two children. Therefore, at each node instead of storing only the relative offsets of entries in the right subtree of a BiNode, we also store the offsets of all subtrees except for the first subtree. We denote these offsets as *separators*, as the m^{th} offset separates leaf entries and structural information contained in subtree m from leaf entries and structural information in subtree $m+1$. In addition to these offsets, each internal node in a generalized indirect node layout contains the internal nodes span specifying the number of bits considered at this node. The resulting two-layered internal node representation consists of a header containing the span and the position of the most significant discriminative bit and a list of separators each containing the subtrees' offsets. Notably, for the special case of storing only internal nodes of arity two, this generalized representation is similar to an indirect node layout. The major difference is that each inner node in the generalized representation requires storing the span and the relative address of the child inner nodes. A GIL requires this additional information, because—in contrast to an IL—the memory required to store a subtree's inner nodes cannot be inferred from the number of elements in the respective subtree. However, having to provide the offset of child inner nodes enables us to omit the separator information at leaf level. We can see the ramifications in GIL's search algorithm provided in Listing 5.5. While INL's search algorithm returns as soon as the number of remaining entries is one, GIL's search algorithm returns as soon as the space required to store a subtree's child inner nodes becomes zero.

```
Leaf* search(uint8_t const * key) const {
  // the address of the current inner nodes header
  uint currentAddr = root;
  // the current subtree's size in bytes in the node's header
  uint remainingSize = sizeOfStructuralArea;
  // index of first leaf entries in current subtree
  uint leafIndex = 0;
  while(remainingSize > 0) {
    header = readHeader(currentAddr);
    span = header->span;
    discrBit = header->discrBit;
    childIdx = extractBits(key, discrBitPos, span);
    remainingSize -= header.size;
    currentAddr += header.size;
    numberSubtrees = pow(2, span);
    if(remainingSize == 0) {
      leafIndex += childIdx;
    } else {
      uint totalSeparatorSize = SEPARATOR_SIZE * (numberSubtrees - 1);
      uint currentSubtreeOffset = currentAddr + totalSeparatorSize;
      remainingSize -= totalSeparatorSize;

      if(childIdx > 0) {
        SubtreeSeparator* sep = readSeparator(currentAddr, childIdx - 1);
        currentSubtreeOffset += sep->subtreeOffset;
        leafIndex += sep->leafOffset;
      }

      if(childIdx < numberSubtrees - 1) {
        SubtreeSeparator* nextSep = readSeparator(currentAddr, childIdx);
        remainingSize = nextSep->subtreeOffset - currentSubtreeOffset;
      } else {
        remainingSize -= currentSubtreeOffset;
      }
      currentAddr += currentSubtreeOffset;
    }
  }
  return leafs[leafIndex];
}
```

Listing 5.5: Lookup in generalized indirect node layout.

Similar to LOL, GIL compensates the number of operations per iteration with the number of iterations of its search algorithm by reducing the tree height due to collapsed dense regions.

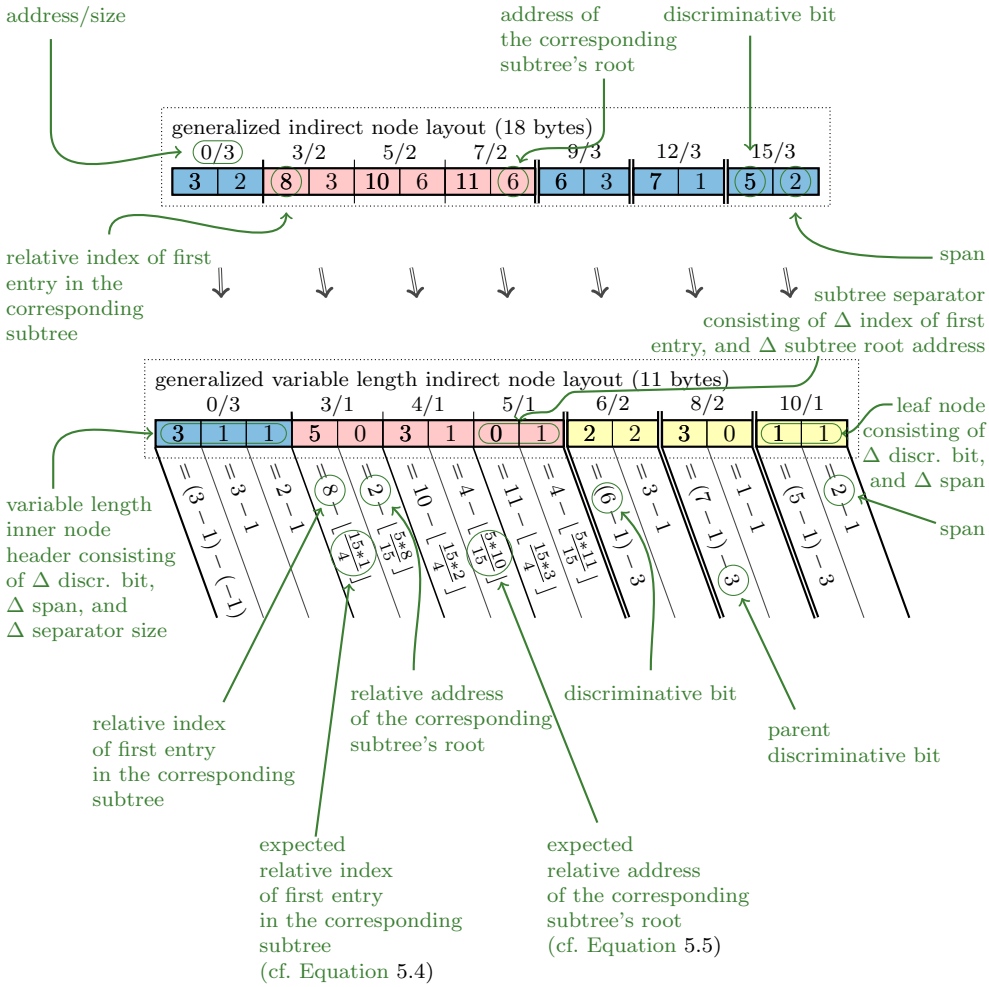


Figure 5.13: Transition from a generalized indirect node layout into a generalized variable length indirect node layout for the tree structure shown in Figure 5.12. Inner nodes have a blue background color, subtree separators have a red background color, and leaf nodes have a yellow background color.

Variable Length Generalized Indirect Node Layout Even though the generalized indirect node layout presented in Section 5.2.4 takes advantage of dense regions to reduce the height of the inner node structure, it is not able to utilize dense regions to reduce a node's memory footprint at the same time. This is due to the fact, that all inner nodes require the same amount of memory for their headers and separators regardless of regularities in the keys stored.

We have previously shown in Section 5.2.3 that variable length inner nodes as applied in the variable length indirect node layout are able to utilize regularities in the stored keysets to reduce the memory consumption of plain indirect layouts. As the generalized indirect node layout is a generalization of the indirect layout we propose the *Variable length Generalized Indirect node Layout (VGIL)* in this section that applies a similar technique to exploit similarities in the keyset to reduce the memory footprint. The key idea of the variable length indirect node layout is the assumption that binary Patricia trie structures tend to be more balanced than not. Otherwise, the mean depth of a binary Patria trie's entries would be close to the average key length in bits or even worse. For instance, for the five evaluated datasets with 50 million keys each in Section 7.2, the average depth in the worst case is 96 for the URL dataset, which consists of 440 bit long keys on average. Hence, instead of storing the inner nodes child offsets directly, the VIL stores the delta between the assumed offsets of a balanced tree instead. For the variable generalized indirect node layout, we apply the same technique of delta encoding the tree structure. More specifically, instead of encoding the delta between the offsets of a balanced BiNode, we now encode the offsets from an assumed balanced multi-way inner node. Whether the numbers representing these offsets are small or large depends on how balanced the stored keys are. To benefit from using smaller numbers to encode this information in more balanced nodes, we apply a variable length encoding scheme. As inner nodes in a generalized indirect node layout consist of two parts, headers and separators, these parts have to be encoded separately. For each inner node, we determine the smallest separator size that is able to store each of its separators. We store this size in the corresponding inner node's header and encode its separators accordingly. Figure 5.13 shows a VGIL representation of the collapsed binary Patricia trie depicted in Figure 5.11. The figure also illustrates how the equivalent GIL is converted into the final VGIL.

For each separator, the VGIL stores the delta between the expected number of entries left of the separator and the delta to the expected number of bytes required to encode all child subtrees of the current inner node left of the respective separator. The crucial part is how to estimate the number of entries in the respective subtrees and how to estimate the number of bytes required to encode these subtrees. We choose a two-step approach. We first assume that the number of entries is equally distributed between all subtrees. Based on this assumption, Equation 5.4 estimates the index of the first entry in a node's i^{th} subtree based on the total number of the nodes descending entries n

```
Leaf* search(uint8_t const * key) const {
  // the address of the current inner nodes header
  uint currentAddr = root;
  // the current subtree's size in bytes in the node's header
  uint remSize = sizeOfStructuralArea;
  // the number of leaf entries in the current subtree
  uint remEntries = totalNumberEntries;
  // index of first leaf entries in current subtree
  uint leafIndex = 0;
  while(remSize > 0) {
    header = readHeader(currentAddr);
    childIdx = extractBits(key, header->discrBit, header->span);
    remSize -= header.size;
    currentAddr += header.size;

    if(remSize == 0) {
      leafIndex += childIdx;
    } else {
      uint numberSubtrees = pow(2, header->span);
      remSize -= header->separatorSize * (numberSubtrees - 1);
      // relative address of respective child's subtree
      uint currentSubtreeOff = 0;
      uint nextSubtreeOff = remSize;
      // relative index of first entry in respective child's subtree
      uint relLeafOff = 0;
      uint nextRelLeafOff = remEntries;

      if(childIdx > 0) {
        uint separatorAddress = currentAddr + childIdx * header->
            separatorSize;
        DeltaSeparator* currentSep = readDeltaSeparator(separatorAddress);
        uint expRelLeafIdxOff = (childIdx * remEntries)/numberSubtrees;
        relLeafOff = expRelLeafIdxOff + currentSep.deltaIdxOff;
        uint expChildSubtreeOff = (remSize * relLeafIdxOff)/remEntries;
        currentSubtreeOff = expChildSubtreeOff + currentSep.deltaSubtreeOff;
      }

      if(childIdx < numberSubtrees - 1) {
        uint nextSepAddress = currentAddr + (childIdx + 1) * header->
            separatorSize;
        DeltaSeparator* nextSep = readDeltaSeparator(separatorAddress);
        uint expRelLeafIdxOff = (childIdx * remEntries)/numberSubtrees;
        relLeafOff = expLeafIdxOff + nextSep.deltaIdxOff;
        uint expChildSubtreeOff = (remSize * relLeafOff)/remEntries;
        uint nextSubtreeOff = expChildSubtreeOff + nextSep.deltaSubtreeOff;
      }

      leafIdx += relLeafOff;
      currentAddr += currentSubtreeOff; %TODO check offsets !!
      remSize = nextSubtreeOff - currentSubtreeOff;
      remEntries = nextRelLeafIdxOff - relLeafOff;
    }
  }
  return leafs[leafIndex]
}
```

Listing 5.6: Lookup in variable length generalized indirect node layout.

and the node's fanout f. This estimation is equivalent to the total number of entries in all subtrees up to index $i - 1$.

$$index\_first\_entry(i, n, f) = \frac{n * i}{f} \tag{5.4}$$

Secondly, based on the actual number of entries in each subtree, we interpolate the space required to encode each subtree's structure. Instead of directly estimating the size of each subtree separately, we estimate the accumulated size of all subtrees up to subtree $i-1$ which corresponds to the relative address of the i^{th} subtree's root. Given a node's overall subtree structure size s, the total number of its descending entries n, and the actual index idx of the i^{th} subtree's first entry, Equation 5.5 estimates the relative address of the i^{th} subtree's root.

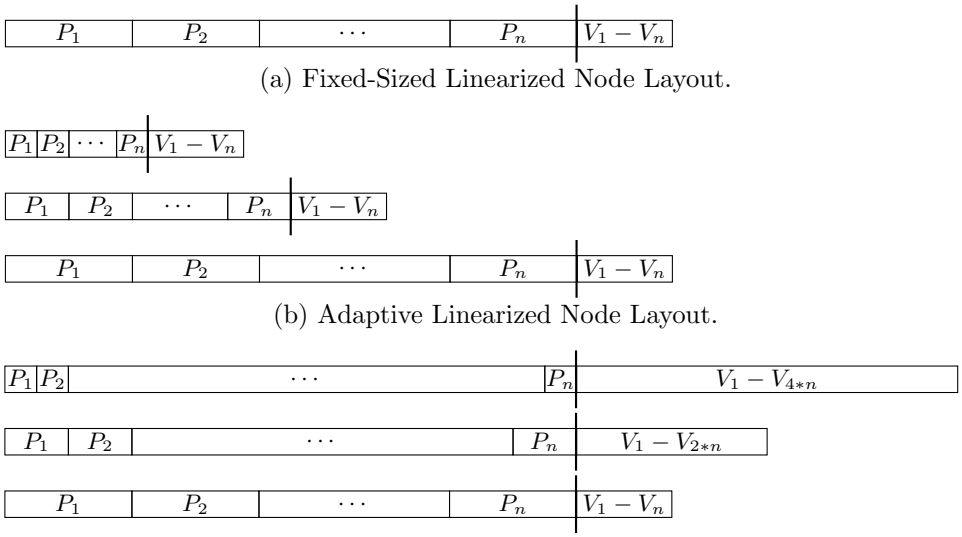
$$subtree\_root\_address(idx, n, s) = \frac{s * idx}{n} \tag{5.5}$$

We can see the reason for this two-step approach, by looking at VGIL's search algorithm depicted in Listing 5.6. During the search, at each separator, first the actual number of elements in the respective subtree has to be decoded and secondly, based on this result, the actual offset of the corresponding inner node can be derived.

Similar to the VIL, the VGIL trades the number of operations required to search a single node with its storage requirement. However, this does not automatically imply that searching a VGIL has to be slower than searching a GIL. The actual performance of a VGIL depends on how good the delta encoding compresses the structure of a node and how expensive cache misses are on the respective hardware architecture the structure is getting deployed to.

5.3 Linearized Node Layouts

After discussing different approaches for hierarchical node layouts in Section 5.2, we now turn our focus to so-called linearized node layouts. While hierarchical node layouts enable compact representations of a node's internal k-constrained trie, these hierarchical representations require iterative search operations that traverse the internal tree structure from top to bottom. As each layer of an internal trie increases the latency of a search operation, we



(a) Fixed-Sized Linearized Node Layout.

(b) Adaptive Linearized Node Layout.

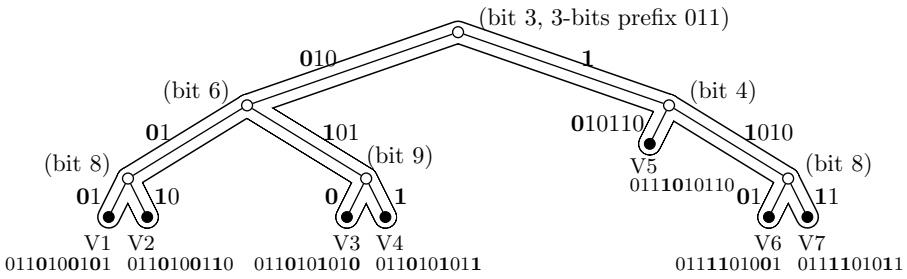
(c) Size Adaptive Linearized Node Layout.

Figure 5.14: Schematic overview of the proposed linearized node layouts. Each node consists of two parts. The structural area contains the partial keys (P_i) and is separated by a thick vertical line from the data area containing the actual values (V_i). The width of the partial key cells correspond to the width of the used data type and the combined width of the structural and data area corresponds to the node's overall memory consumption.

strive for a different approach that achieves constant intra-node access performance. To achieve constant access times we combine two ideas. First, we devise a representation of the tree structure, which allows us to use a linear scan for search operations. Secondly, we replace this linear scan with data-parallel SIMD instructions, which can be executed in constant time for each node. In the remainder of this section, we first present the linearization of a node's internal binary Patricia trie structure. Next, we present how the instruction sets of modern hardware can be leveraged to search such a linearized representation efficiently. Finally, we present three different variations of such hardware accelerated linearized node layouts with the goal to further improve performance and memory consumption. An overview of these three node layouts is shown in Figure 5.14.

5.3.1 Linearizing Binary Patricia Tries

The challenge of linearizing binary Patricia tries lies in creating an implicit representation of the trie structure which can be searched by linear scan operations or equivalent data-parallel operations. Our approach to creating such



(a) Example Binary Patricia Trie.
Discriminative bits along a path are typeset in bold.

Raw Key	Bit Positions	Partial Key (dense)	Partial Key (sparse)
0 1 1 0 1 0 0 1 0 1	{3, 6, 8}	0 1 0 0 1	0 0 0 0 0
0 1 1 0 1 0 0 1 1 0	{3, 6, 8}	0 1 0 1 0	0 0 0 1 0
0 1 1 0 1 0 1 0 1 0	{3, 6, 9}	0 1 1 1 0	0 0 1 0 0
0 1 1 0 1 0 1 0 1 1	{3, 6, 9}	0 1 1 1 1	0 0 1 0 1
0 1 1 1 0 1 0 1 1 0	{3, 4}	1 0 0 1 0	1 0 0 0 0
0 1 1 1 1 0 1 0 0 1	{3, 4, 8}	1 1 1 0 1	1 1 0 0 0
0 1 1 1 1 0 1 0 1 1	{3, 4, 8}	1 1 1 1 1	1 1 0 1 0

(b) Raw and partial keys for the trie in (a).

Figure 5.15: Illustration of linearizing a binary Patricia trie.

an implicit representation is to store the path information of each key in the original binary Patricia trie.

As each BiNode represents the position of a single discriminative bit, we can represent the path of a single key by extracting its discriminative bits along the path and storing them consecutively. To create a uniform representation of these extracted partial keys, we extract all the discriminative bits of the same binary Patricia trie structure and not only those bits along the key's path. We call this specific definition of partial keys *dense*. To interpret the extracted partial keys, we also need to store the positions of the used discriminative bits. Consider, for example, the trie shown in Figure 5.15a, which consists of seven keys and has the discriminative bit positions $\{3, 4, 6, 8, 9\}$. The five discriminative bits for each key form the *partial keys (dense)* and are shown in Table 5.15b. Searching this implicit binary Patricia trie represented by its dense partial keys is straight-forward. First, we extract the discriminative bits of the search key. Secondly, we find the partial key that is equal to the search key's partial key.

```
1  Leaf* search(uint8_t const * key) const {
2    uint densePartialKey = extractBits(discriminativeBitPostions, key);
3    uint leafCandidateIndex = 0;
4    // start at index one because the
5    // first sparse partial key is always trivially zero
6    for(uint i=1; i < numberOfKeys; ++i) {
7      // check if the sparse partial key is a potential match candidate
8        if(densePartialKey & sparsePartialKey[i] == sparsePartialKey[i]) {
9              leafCandidateIndex = i;
10       }
11   }
12   return leafs[leafCandidateIndex]
13 }
```

Listing 5.7: Lookup in a linearized binary Patricia trie.

However, using dense partial keys to represent a binary Patricia trie has a major caveat. While search and deletion operations on dense partial keys can be implemented efficiently, inserting a new key into a node can be slow. The reason is that a new key may yield a new discriminative bit and may, therefore, require resolving this new discriminative bit for all keys already stored in that node. For instance, inserting the key 0110101101 into the binary Patricia trie of Figure 5.15a would result in bit 7 becoming a new discriminating bit and thus, a new bit position. All existing dense partial keys would have to be extended to 6 bits length and therefore, we would have to determine bit 7 for the existing partial keys by loading the corresponding actual keys, which would slow down insertion. To overcome this shortcoming, we use a slightly modified partial key representation, which we call *sparse partial keys*. The difference to dense partial keys is that for sparse partial keys, only those discriminative bits are extracted that correspond to inner BiNodes along the path from the root BiNode and all other bits are set to 0. Thus, sparse partial key bits set to 0 are intentionally left undefined. In the case of a deletion, this allows us to remove unused discriminative bits. To illustrate the difference between dense and sparse partial keys, we show both in Table 5.15b for the trie in Figure 5.15a.

As zero bits in sparse partial keys have ambiguous semantics, we have to adapt the search algorithm. For instance, key 0111010110 shown in row 5 of Table 5.15b yielding the dense partial key 10010 does not match its corresponding sparse partial key for the binary Patricia trie depicted in Figure 5.15a. To solve this issue we developed the search algorithm shown in Listing 5.7. First, the algorithm extracts the dense partial key corresponding to the provided search key. Next, the algorithm checks whether the bits set to one in the sparse partial key also have the value one in the dense partial key. Finally, we select the largest sparse partial key where only these bits are set.

While inserting a new key into a normal binary Patricia trie structure is straightforward, inserting a new key into a linearized binary Patricia trie operating solely on the linearized representation requires a novel, more sophisticated algorithm. Hence, we describe how such an insertion operation works in the following. Before the actual insertion, a search operation is issued checking whether the key to insert is already contained. If the thereby retrieved key does not match the search key, the mismatching bit position is determined. In contrast to a traditional binary Patricia trie, explicitly determining the corresponding mismatching BiNode is impossible, as explicit representations for BiNodes do not exist in a linearized binary Patricia tries. Instead, we directly determine all leaf entries contained in the subtree of the mismatching BiNode and denote these entries as the *affected entries*. To do so, we mark all partial keys that have the same prefix up to the mismatching bit position as the initially matching false-positive partial key, as affected. Next, if the mismatching bit position is not contained in the set of the node's discriminative bit positions, all sparse partial keys have to be recoded to create partial keys containing also the mismatching bit position. To directly construct the new (sparse) partial key representation, we exploit the fact that it shares a common prefix up to the mismatching bit with the affected entries. Therefore, to obtain the new (sparse) partial key, we copy this prefix and set the mismatching bit accordingly. As the bit at the mismatching bit position discriminates the new key from the existing keys in the affected subtree, the affected partial keys' mismatching bits are set to the inverse of the new key's mismatching bit. Finally, again depending on the mismatching bit, the newly constructed partial key is inserted either directly in front or after the affected entries.

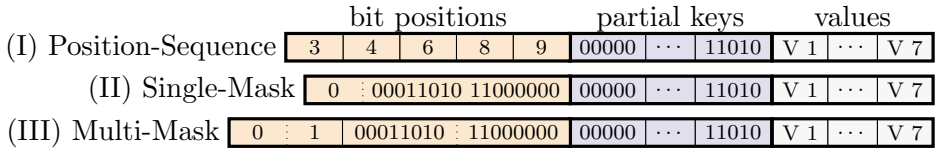
5.3.2 Fixed-sized Linearized Node Layout

So far, searching a linearized node layout requires linear search. In this section, we show that by using SIMD operations we can achieve constant inter-node search behavior. In addition, we show that also the extraction of partial keys can be improved by using bit manipulation instruction set extensions of modern hardware. As all partial keys in the resulting node layout have a fixed length, we call it *Fixed-sized Linearized node Layout (FLL)*.

The key idea to improve over linear search in the FLL is to exploit the data-parallel instructions of modern CPU architectures to search all of a node's sparse partial keys in parallel. Given an extracted dense partial key, the SIMD-based algorithm works as follows. First, the dense partial key is broadcast to all slots of a single SIMD register. Secondly, we load multiple sparse partial keys into a SIMD register. Thirdly, we determine all sparse partial keys that only match the dense partial key's discriminative bits in parallel. Next, we

```
1  int searchPartialKeys8(Node* node, uint32_t densePartialSearchKey) {
2    // loading all sparse partial keys
3    __m256i sparsePKeys = _mm256_loadu_si256(node->partialKeys8);
4    // broacasting the search key into a SIMD register
5    __m256i key = _mm256_set1_epi8(densePartialSearchKey);
6    // determine all bits of the partial search keys
7    // that are set to one in the dense partial key
8    __m256i selBits = _mm256_and_si256(sparsePKeys, key);
9    // determine all sparse partial keys that have only those bits set
10   __m256i complyKeys = _mm256_cmpeq_epi8(selBits, sparsePKeys);
11   // create a bit mask of all matching sparse partial keys
12   uint32_t complyingMask = _mm256_movemask_epi8(complyKeys);
13   // determine the index of the largest matching key
14   return bit_scan_reverse(complyingMask & node->usedKeysMask);
15 }
```

Listing 5.8: Searching multiple sparse partial keys in parallel using SIMD.



(a) Three different variations of a node representing a linearized version of the trie shown in (a). All representations consist of three parts: the bit positions, the partial keys, and the values (tuple identifiers or node pointers). We show 3 representations for bit positions: (I) stores them naively as a sequence of positions, (II) uses a single bit mask, (III) uses multiple bit masks.

store the result of the match operation in a bit mask. Each bit of the bit mask corresponds to one of the node's sparse partial keys and each set bit represents a matching partial key. Finally, determining the index of the sparse partial key is done by using a single bit scan reverse CPU instruction. The pseudo-code for searching 8-bit sparse partial keys is shown in Listing 5.8.

Until now, we have not detailed how a search key's discriminative bits are actually stored and how partial search keys are extracted. The most obvious way is to store the set of node's discriminative bits in an array, as shown in Figure 5.16a (I). The downside of this approach is that it slows down access performance: Before the actual data-parallel key comparison can be performed, we have to sequentially extract bits from the search key bit-by-bit to form the comparison key. Note that key extraction is done for every node and is therefore critical for performance.

To speed up key extraction, we propose two layouts that utilize the *PEXT* instruction of the BMI2 instruction set. The PEXT instruction extracts bits specified by a bit mask from a given integer. Thus, as shown in Figure 5.16a (II), we represent the bit positions as a bit mask (and an initial byte position). This layout can be used whenever the lowest and highest bit positions are close to each other (less than 64 bit positions difference). In case the range of the discriminative bit positions is larger, the multi-mask layout can be used, which is illustrated in Figure 5.16a (III). It breaks up the bit positions into multiple 8-bit masks, each of which is responsible for an arbitrary byte. Again, PEXT is used to efficiently extract the bits contained in multiple 8-bit key portions in parallel using this layout.

Besides speeding up the extraction of partial keys, FLL also uses the bit manipulation instruction set BMI2 to improve insert and deletion performance. While insertion operations use the PDEP instruction to recode partial keys when inserting a new discriminative bit, its counterpart, the PEXT instruction is used by deletion operation to recode partial keys when deleting a superfluous discriminative bit. For instance, to add bit position 7 to the sparse partial keys depicted in Figure 5.15a, the `_pdep_u32(existingKey, 0b111011)`[1] instruction is executed for each key.

5.3.3 Adaptive Linearized Node Layout

Although the fixed-sized linearized node layout presented in the previous section exploits the instruction sets of modern hardware to provide constant search performance, it has one downside. Namely, to represent tries with a maximum fanout of k it always uses partial keys of length $k - 1$. However, depending on the dataset stored, only a small fraction of these k bits may be required to represent all of a node's discriminative bits. In the best case, where a node's keys form a single dense region only $log_2(k)$ bits are necessary to represent the node's sparse partial keys (cf. Section 5.2.3 for the definition of dense regions). We introduce the *Adaptive Linearized node Layout (ALL)* to take advantage of keysets that can be represented by less than $k - 1$ discriminative bits. The ALL chooses the smallest data type per node capable of storing the node's sparse partial keys, instead of choosing the data type based on the worst-case partial key length. Besides reducing the memory footprint this optimization has two major advantages resulting in improved performance too. First, shorter partial keys require fewer memory accesses, and therefore, fewer cache misses. Secondly, using smaller data types increases parallelism because more data items can be processed in a single SIMD instruction.

[1] https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-pdep-u32-64

As the used hardware platform has a direct impact on the width of its SIMD registers and hence, on the number of partial keys that can be loaded in parallel, we have to decide on the available SIMD instruction sets in advance to design the actual SIMD based index structure. Without loss of generality, however, similar approaches can also be transferred to other hardware architectures. In our particular case, we focus on the widest SIMD instruction set, which is broadly available. For end-user systems, this is currently (as of January 2020) the AVX2 instruction set with a market share of approximately 75%[2].

As the AVX2 instruction set has 256-bit wide SIMD registers and is able to load multiple 8-bit, 16-bit, 32-bit, and 64-bit wide data words into a single SIMD register, we choose the maximum number of entries k per node to be 32. This decision has three reasons. First, in the case of the 8-bit data type, we are able to load and process all 32 items in a single SIMD register simultaneously. Secondly, in the worst case of requiring 32-bits per partial key only two cache lines need to be accessed, and traditionally adjacent cache lines tend to be automatically prefetched by the CPU. Thirdly, even in the worst case of using 32-bit per partial key, the amount of instructions needed to search the partial keys is quadrupled.

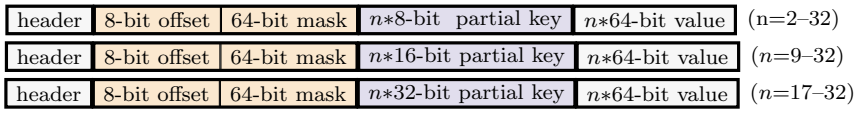
Based on the chosen maximum fanout, we choose two dimensions of adaptivity, depending on a node's discriminative bits, to adapt the node layout to the key set stored. First, depending on the range of a node's discriminative bits we choose between a (I) single-mask layout and three (II) multi-mask layouts. These three multi-mask layouts (Figures 5.17b, 5.17c, 5.17d) differ only in the number of offset/mask pairs (8, 16, or 32). The second dimension of adaptivity chooses the data types used for storing partial keys. To enable fast SIMD operations these partial keys need to be aligned. Therefore, we support partial keys to be stored in one of three representations, namely as an array of 8-bit, 16-bit, or 32-bit values.

Combining all options for both dimensions results in twelve distinct node types. However, only nine of these combinations can occur in practice. For example, the 32-entry multi-mask layout shown in Figure 5.17d implies that there are more than 16 discriminative bits and that therefore, neither 8-bit nor 16-bit partial keys would suffice. The final nine supported node layouts are shown in Figure 5.17. An optimization we apply for the ALL is to store the type of each node within the least-significant bits of each node's pointer. This allows us to overlap two time-consuming operations, namely loading a node's data, which can trigger a cache miss, and resolving the node type, which may otherwise suffer from a branch misprediction penalty.

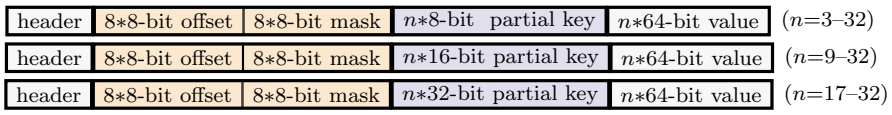
[2]https://web.archive.org/web/20200103040011/http://store.steampowered.com/hwsurvey/

```
1  TID lookup(Node* root, uint8_t* key) {
2    Node* node = root;
3    while (!isLeaf(node)) {
4      uint32_t candidates = retrieveResultCandidates(node, key);
5      node = node->value[clz(candidates)];
6    }
7    if (!isEqual(loadKey(getTid(node)), key))
8      return INVALID_TID; // key not found
9    return getTid(node);
10 }

12 uint32_t retrieveResultCandidates(Node* node, uint8_t* key) {
13   switch (getNodeType(node)) {
14     case SINGLE_MASK_PKEYS_8_BIT:
15       uint32_t partialKey = extractSingleMask(node, key);
16       return searchPartialKeys8(node, partialKey);
17     case MULTI_MASK_8_PKEYS_8_BIT:
18       uint32_t partialKey = extractMultiMask8(node, key);
19       return searchPartialKeys8(node, partialKey);
20     ...
21     case MULTI_MASK_32_PKEYS_32_BIT:
22       uint64_t partialKey = extractMultiMask32(node, key);
23       return searchPartialKeys32(node, partialKey);
24   }
25 }

27 uint32_t extractSingleMask(SMaskNode* node, uint8_t* key) {
28   uint64_t* keyPortion = (uint64_t*) (key + node->offset)
29   return _pext_u64(*keyPortion, node->mask);
30 }
31 uint32_t extractMultiMask8(MMask8Node* node, uint8_t* key) {
32   uint64_t keyParts = 0;
33   //load all 8-bit mask into a single 64-bit mask
34   for (size_t i=0; i < node->numberMasks; ++i)
35     ((uint8_t*) keyParts)[i] = key[node->offsets[i]];
36   //SIMD approach to extract multiple 8-bit masks in parallel
37   return _pext_u64(keyParts, node->mask);
38 }
39 uint32_t extractMultiMask16(MMask16Node* node, uint8_t* key)...
40 uint32_t extractMultiMask32(MMask32Node* node, uint8_t* key)...

42 int searchPartialKeys8(Node* node, uint32_t searchKey) ...
43 int searchPartialKeys16(Node* node, uint32_t partialKey) ...
44 int searchPartialKeys32(Node* node, uint32_t partialKey) ...
```

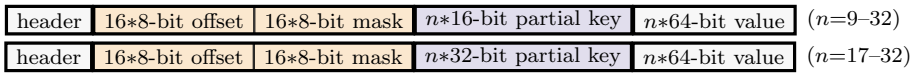
Listing 5.9: Lookup in HOT using size adaptive nodes.



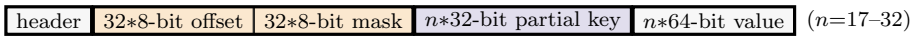
(a) Single-mask layout with a 1 64-bit mask and 8, 16, or 32-bit partial keys.



(b) Multi-mask layout with 8 8-bit masks and 8, 16, or 32-bit partial keys.



(c) Multi-mask layout with 16 8-bit masks and 16 or 32-bit partial keys.



(d) Multi-mask layout with 32 8-bit masks and 32-bit partial keys.

Figure 5.17: Physical node layouts supported by ALL.

To get an understanding of lookup operations on HOT structures using an ALL node layout we show in Listing 5.9 the (slightly simplified) code for lookup, which traverses the tree until a leaf node containing a tuple identifier is encountered.

The actual lookup operation for a single node is done by the function `retrieveResultCandidates` (lines 12–25), which, given a node and the search key, performs the actual intra-node search consisting of two steps: (1) extracting the (dense) partial key from the search key, and (2) comparing the node's (sparse) partial keys with it. The implementation of both steps depends on the node layout. For this reason, `retrieveResultCandidates` consists of a `switch` statement over all node layouts (lines 13–24). Each case then consists of calling the appropriate extraction (`extract*`) and search primitives (`searchPartialKeys*`) (e.g., lines 14–16).

To illustrate the extraction of a key's discriminative bits, we show the code for single-mask (`extractSingleMask`, lines 27–30) and multi-mask 8 extraction (`extractMultiMask8`, lines 31–38), both of which use the PEXT instruction. The implementation for an AVX2-based search for 8-bit partial keys (`searchPartialKeys8`, lines 42–49) we have already shown in Listing 5.7. The remaining search primitives are implemented analogously.

To conclude, ALL is more space-efficient than a fixed size linear node layout on the same dataset. It also reduces the number of cache misses, by adaptively

choosing the smallest fitting data type for the stored partial keys. Through carefully interleaving node type selection with loading the node itself, the overhead of the node type selecting becomes negligible.

5.4 Size-Adaptive Nodes

All our previously proposed node layouts use a fixed maximum fanout per node and combine different techniques to reduce the memory footprint while optimizing access performance. As already discussed in the previous sections of this chapter, using different datasets results in a different discriminative bit distribution. Hence, a different number of discriminative bits may result in varying sizes of a node's structural area. These varying sizes make it hard to predict how much data has to be loaded/prefetched to search a node's structural area.

Hence, we propose the *Size adaptive Linearized node Layout (SLL)* that, in addition to the maximum fanout k, also considers the maximum size m of the structural area, when determining whether a node has to be split or not. The key idea of size adaptive nodes is that depending on the dataset's keys' distribution a larger amount of short partial keys require the same amount of memory than a smaller number of longer partial keys. Hence, it allows us to support larger node fanouts if only a small number of discriminative bits are sufficient to represent the node's internal structure.

To achieve larger node fanouts in case of appropriate data distributions, the SLL extends on the ALL by allowing more than 32 entries per node as long as the size of the structural area is below 192 bytes. To support more than 32 entries, the SLL uses multiple AVX2 registers to process the stored partial keys. The results of the individual AVX2 registers are then combined into a long bit mask stored in a single AVX2 register. To identify the actual result candidate, we first set all bits in the register not corresponding to entries contained in the node to zero. Next, we select the byte corresponding to the largest matching partial key by finding all non zero bytes in the AVX register. Finally, we determine the position of the most significant bit in this byte and thereby the position of the match candidate.

While supporting larger node fanouts may result in lower tree heights and therefore in lower space consumption, its impact on lookup performance is not obvious. On the one hand, larger node fanouts may lead to lower tree heights and therefore, to a lower number of nodes to process per lookup operation. On the other hand, searching a node containing more than 32 entries requires more instructions than searching nodes containing at most 32 entries.

Therefore, whether an SLL leads to higher throughput in comparison to an ALL highly depends on the characteristics of the hardware the corresponding index structure is used on. To gain more insights into the performance characteristics of SLLs, we evaluate SLLs and ALLs in terms of throughput, memory- and cache efficiency in Chapter 7.

CHAPTER 6

Synchronizing Height Optimized Tries

Besides performance and space efficiency, scalability is another crucial feature of any index structure. A scalable synchronization protocol is therefore necessary to provide efficient concurrent index accesses.

Traditionally, index structures use fine-grained locking and lock coupling to provide concurrent accesses to index structures [48, 49]. However, it has been shown that using such fine-grained locks for reading and writing has a huge impact on the overall system performance and does not scale well [77]. Therefore, different approaches based on the concept of lock-free index structures or write-only minimal locks have been proposed [78, 81, 77, 68].

Lock-free data structures often use a single compare-and-swap (CAS) operation to atomically perform updates. Therefore, it is tempting to assume that HOT—using a copy-on-write approach and swapping a single pointer per insert operation—would be lock-free by design. However, using a single compare-and-swap (CAS) operation does not suffice to synchronize write accesses to HOT. If two insert operations are issued simultaneously, inserts may

be lost. If one insert operation replaces a node N with a new copy N', while the other insert operation replaces a child node C of N with a new copy C', it might occur that in the final tree, node C is a child of N', whereas C' is a child node of the now unreachable node N.

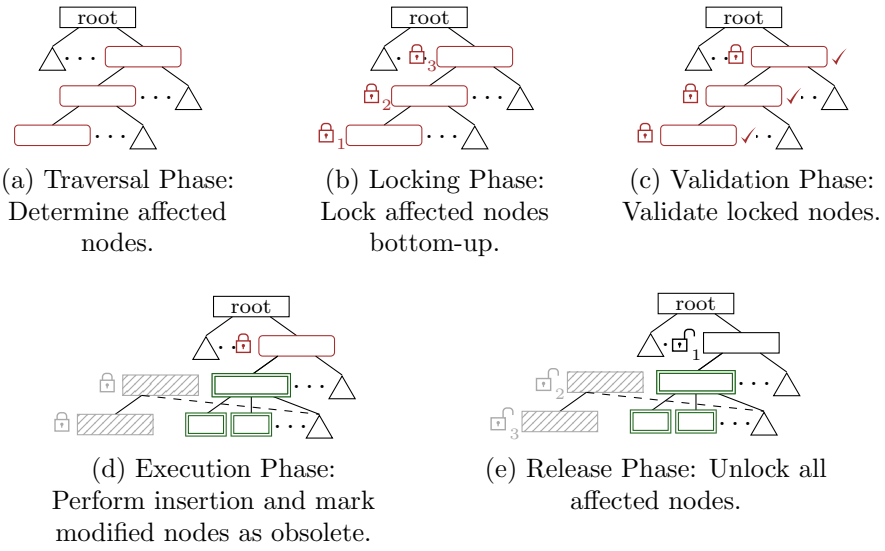
Although the combination of copy-on-write and CAS is not enough to synchronize HOT, these properties of HOT are a perfect fit for the Read-Optimized Write EXclusion (ROWEX) synchronization strategy [77]. ROWEX does not require readers to acquire any locks and hence, they can progress entirely wait-free (i.e., they never block and they never restart). Writers, on the other hand, do acquire locks, but only for those nodes that are modified during an insert operation. Writers also have to ensure that the data structure is valid at any point in time because locks are ignored by readers. As a result, the lookup algorithm remains unaffected.

In the following sections, we first present our ROWEX based synchronization protocol and finally describe how we deal with the memory reclamation in the case of obsolete nodes.

6.1 Synchronization Protocol

To synchronize HOT using a Read-Optimized Write EXclusion (ROWEX) synchronization strategy [77] we created a custom synchronization protocol. Although we designed this protocol specifically for HOT, it should in theory also apply to other tree structures that use a copy on write update strategy and exchange a single pointer per write operation. We therefore call this synchronization protocol *Copy on Write ROWEX*. The synchronization protocol consists of five phases and applies to all modification operations (e.g., insert, delete). The five phases are (a) *Traversal Phase*, (b) *Locking Phase*, (c) *Validation Phase*, (d) *Execution Phase*, and (e) *Release Phase*. An overview of these phases is shown in Figure 6.1, and we describe them in detail below.

In the initial *Traversal Phase*, we determine all nodes that will be modified—by using our copy on write updates—and place them in a stack data structure. We denote these nodes as the *affected nodes*. In the following *Locking Phase*, a lock is acquired for each of the affected nodes. To avoid deadlocks, these locks are acquired in bottom-up order. As concurrent write operations might have modified some of the affected nodes before we could acquire the lock, we apply a *Validation Phase* to check that none of the affected nodes has become obsolete in the meantime. If any of the locked nodes have indeed been marked as obsolete, we restart the operation after unlocking all previously locked nodes. If the validation succeeded we execute the actual write opera-



(a) Traversal Phase: Determine affected nodes.

(b) Locking Phase: Lock affected nodes bottom-up.

(c) Validation Phase: Validate locked nodes.

(d) Execution Phase: Perform insertion and mark modified nodes as obsolete.

(e) Release Phase: Unlock all affected nodes.

Figure 6.1: Step-by-step example of HOT's *Copy on Write ROWEX* synchronization protocol. The example shows an insertion operation resulting in a parent pull up. The three affected nodes are marked red with rounded corners, the newly created nodes are marked green with a double border and all modified and therefore obsolete nodes are marked gray and filled with a line pattern.

tion in the *Execution Phase* by replacing all of the affected nodes with their modified counterparts and mark these replaced nodes as obsolete. Finally, in the *Release Phase*, we unlock all previously locked nodes in top-down order.

While the overall design of the synchronization protocol is straightforward, the crucial and index-specific part is to determine the set of affected nodes, as a single modification operation can affect multiple nodes. Specifically for HOT, we distinguish four cases analogous to the different cases of the insert operation (see section 3.3) to determine the set of affected nodes. In the trivial case of a *normal insert*, the set of affected nodes consists of the node containing the mismatching BiNode and its parent node. For the other three cases (i) *leaf-node pushdown*, (ii) *parent pull up*, and (iii) *intermediate node creation* the set of affected nodes is determined as follows: (i) In the case of a leaf-node pushdown, the set of affected nodes solely consists of the node containing the mismatching BiNode. If an overflow occurs, all ancestor nodes of this node are traversed and added to the set of affected nodes until either (ii) in case of a parent pull up, a node with sufficient space or the root node is reached or (iii) in case of an intermediate node creation, a node n fulfilling $height(parent(n)) >= height(n)$ is reached. Finally, the direct parent of the last accessed node is added.

After describing the theoretical framework of HOT's synchronization protocol in this section, we will next focus on how our implementation deals with the crucial aspect of reclaiming the memory of obsolete nodes.

6.2 Memory Reclamation

A side effect of the ROWEX based synchronization strategy is that the memory of deleted nodes cannot be reclaimed right after the deletion operation succeeded, as concurrent read and write operations might still access these nodes.

To not impact these readers, a critical aspect of HOT's synchronization strategy is to mark deleted nodes as obsolete instead of directly reclaiming their memory. This strategy has two advantages. On the one hand, the obsolete marker allows concurrent writers to detect whether one of the currently locked nodes has been replaced in the meantime (and restart the operation). On the other hand, readers do not require any locks to deal with concurrent writes. Whenever writers modify a currently read node, the reader can finish the lookup—on the now obsolete—node.

While marking nodes as deleted allows for lock-free read access, the memory of those obsolete nodes needs to be reclaimed eventually to prevent excessive memory consumption. To determine when to free the memory of obsolete HOT nodes we use an epoch based memory reclamation strategy[57]. In the following, we describe how we integrate the epoch based memory reclamation strategy into our concurrent HOT implementation. Before a thread accesses the tree structure it joins the current global epoch and leaves this epoch after the operation has finished. In the case that a node is obsolete it is placed in a thread-local free-list associated with the current epoch. Each time a thread tries to join the current epoch, it checks if the current free-list exceeds the threshold of 64 entries. If this condition is met the thread advances the global epoch the next time it accesses the structure. This condition by design limits the total number of required epochs to three (the current, the previous, and the next epoch) and we can therefore circularly recycle the associated free-lists. Subsequently, if a thread joins a different epoch compared to the epoch used at the previously executed operation we can safely delete the free-list associated with the second to the last epoch. Due to the circular behavior, this is the same free-list as will be used for the new epoch to join.

While for search, deletion, and insertion the individual threads join the current global epoch before the actual operation and leave it again after the operation has ended, a more sophisticated approach is required for scan operations. If

scan operations–especially longer-running scans–would apply the same techniques as the other operations, memory reclamation for all threads would be blocked until the end of the entire scan. This might lead to unexpected memory consumption peaks. To prevent those peaks, we apply the same technique as ART [77] and execute scan operations in a chunk of 128 entries. The current thread only joins the global epoch when buffering the next batch of entries. For the scan to be robust against intermediate modifications, refilling the buffer starts by first looking up the entry with the smallest key larger than the previously accessed entry's key before loading the next 128 entries into the buffer. By scanning the tree structure chunkwise, we prevent stalling memory reclamation while guaranteeing high scan throughput.

CHAPTER 7

Evaluation and Results

After presenting the height optimizing trie in Chapter 3, introducing different node layouts in Chapter 5 and presenting a novel synchronization method in Chapter 6, in this chapter we experimentally evaluate HOT and compare it with other state-of-the-art main memory index structures.

The remainder of this chapter is structured as follows. First, we describe our experimental setup. Secondly, we evaluate the height-reducing effect of HOT by parameterizing it with different maximum fanouts. Next, we use read-only implementations of our proposed node layouts to evaluate their memory efficiency and access performance and use hardware counter-based metrics to interpret these results. Finally, we conclude this chapter with an in-depth analysis of HOT using a fully implemented adaptive linearized node layout and compare it against state-of-the-art main memory index structures in terms of (i) performance, (ii) memory consumption and (iii) scalability.

7.1 Experimental Setup

All experiments, except the scalability experiments, were conducted on a workstation system with a quad-core Intel i7-6700 CPU, running at 3.4 GHz with 4 GHz turbo frequency (32 KB L1, 256 KB L2, and 8 MB L3 cache). The scalability experiments were conducted on a server system with an Intel i9-7900X CPU, which has 10 cores, running at 3.3 GHz with 4.3 GHz turbo frequency (32 KB L1, 1 MB L2, and 8 MB L3 cache). Both systems are running Linux and all code was compiled with GCC 7.2.

We compare HOT against the following three state-of-the-art index structures:

- *ART:* The Adaptive Radix Tree (ART) [75], which is the default index structure of HyPer [116], features variable sized nodes and selectively uses SIMD instruction to speed up search operations.

- *Masstree:* Masstree [82] is a hybrid B-Tee/trie structure used by Silo [110].

- *STX B$^+$-Tree:* The STX B$^+$-Tree[1] represents a widely used cache-optimized B$^+$ Tree (e.g., [64]) and hence, a comparison-based approach. The default node size is 256 bytes which in the case of 16 bytes per slot (8 bytes key + 8 bytes value) amounts to a node fanout of 16.

We choose ART as the fastest fixed span tree available, Masstree, because of its high scalability and hybrid design, and the STX B$^+$-Tree as a widely used representative of the ubiquitous B-Tree. We use the publicly available implementations of ART[2], Masstree[3] and the STX B$^+$-Tree[1].

For all evaluated index structures we use 64-bit pointers as tuple identifiers to address and resolve the actually stored values and keys. In case the stored values only consist of fixed-sized keys up to 8 byte length (e.g., 64-bit integers), those keys are directly embedded in their tuple identifiers. To measure space efficiency, we add custom code to the implementations of ART and the B$^+$-Tree to compute their memory consumptions without impacting their runtime behavior. For Masstree, we use its allocation counters to measure its space consumption. To ensure a fair comparison, we do not take the mem-

[1] https://github.com/bingmann/stx-btree (last visited 2020-01-21)

[2] https://db.in.tum.de/~leis/index/ART.tgz (last visited 2020-01-21)

[3] https://github.com/kohler/masstree-beta (last visited 2020-01-21)

ory required to store Masstree's tuples into account, as the space required to represent the raw tuples is not considered for any of the evaluated data structures.

For our experiments we use the following datasets:

- *url:* The url dataset consists of a total of 72,701,109 distinct URLs, which we collected from the 2016-10 DBPedia dataset [11], where we removed all URLs that are longer than 255 characters.

- *email:* 30 byte long email addresses originating from the real-world email dataset, leaked in the Adobe hack [60]. We cleansed the dataset by removing invalid email addresses or emails solely consisting of numbers or special characters.

- *yago:* 63-bit wide triples of the Yago2 dataset [61]. The triples are compound keys, where the lowest 26 bits are used for the object id, bits 27 to 37 store predicate information, and bits 38 to 63 are used for subject information as proposed in our previous work [21].

- *integer:* uniformly distributed 63-bit random integers.

7.2 Height Reduction

To better understand the height-reducing effect of the height optimizing trie, we now present the depth distribution of leaf values, which is a measure of how balanced a tree is.

Specifically, we are interested in the effect of HOT's algorithm to combine multiple BiNodes into compound nodes in comparison to traditional trie structures. We therefore compare the depth distribution of height optimized tries with different maximum fanouts with the depth distribution of the "fixed span trie structure" ART and the depth distribution of a binary Patricia trie. Since HOT combines several BiNodes of a binary Patricia trie, we extract the binary Patricia trie's depth distribution from the underlying HOT structure.

As the depth distribution of existing trie structures highly depends on the distribution of the stored key set (cf. Chapter 2), we analyze HOT's depth distribution for five datasets (random and dense integer, yago, email, and

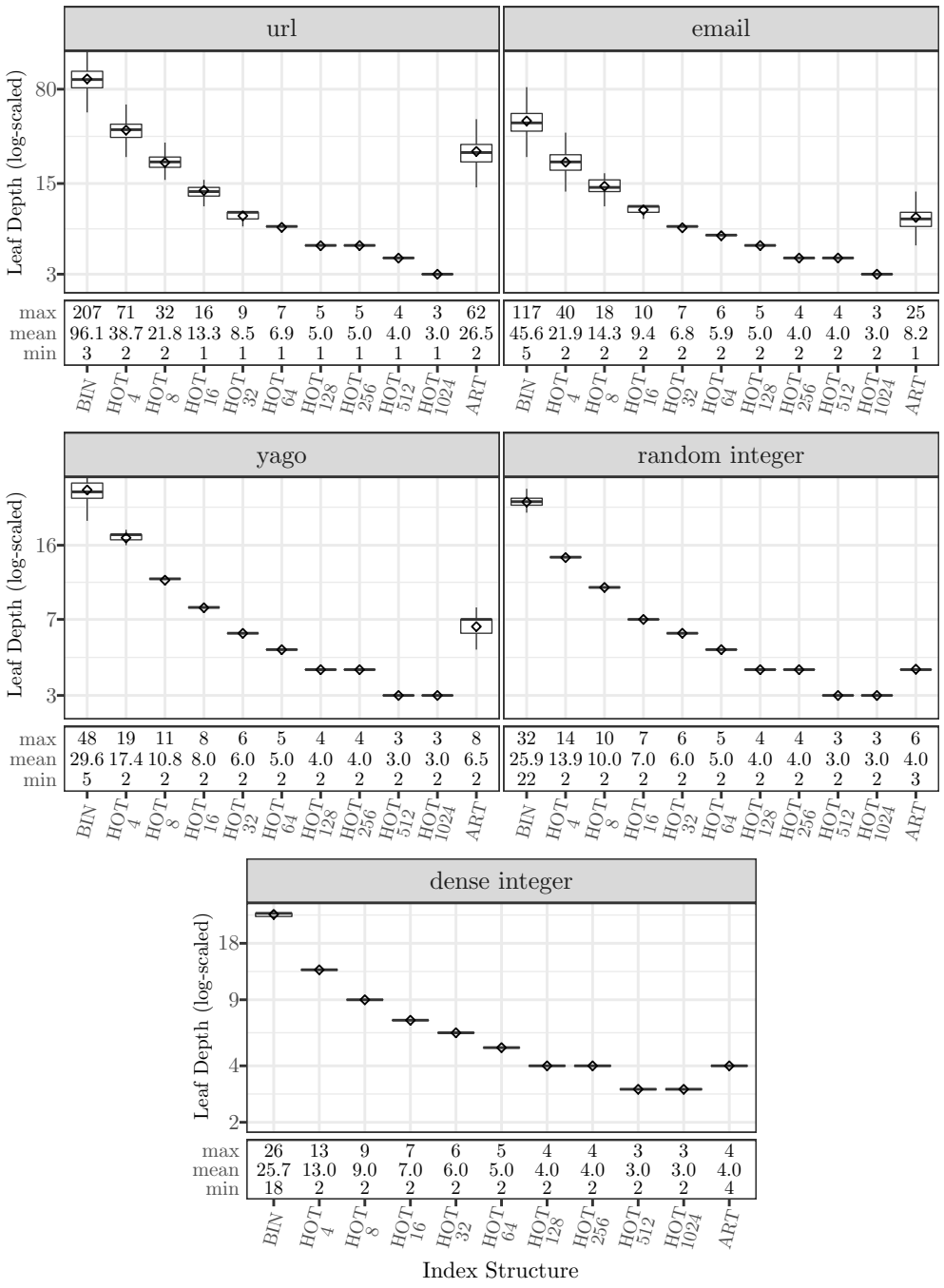


Figure 7.1: Depth distribution of leaves for HOT with maximum fanouts of 4-1024 in relation to an Adaptive Radix Tree (ART) and a binary Patricia trie (BIN) on five different datasets. For each dataset we additionally provide maximum, minimum and mean depth values. The diamonds highlight the mean depth value in the box plot.

url) and therefore, also different key distributions. The obtained results are illustrated in Figure 7.1.

The results show that regardless of the stored dataset, with increasing maximum fanout, the mean depth of a HOT index structure decreases logarithmically. In contrast, fixed span tries represented by ART achieve this logarithmic behavior only for the uniformly distributed integer datasets. HOT even for the textual datasets url and email achieves the same logarithmic behavior. While HOT achieves a 20-fold height reduction for the URL dataset with a maximum fanout of 256, ART achieves only a four-fold height reduction. This amounts to a 5 times lower height for the url dataset (HOT-256: 5.0, ART: 26.5) and a 2 times lower height for the email dataset (HOT-256: 4.0, ART: 8.2). Even for non-uniformly distributed integers, like the yago dataset, HOT reduces the height by a factor of 1.63 (HOT-256: 4, ART: 6.5). For the three non-uniformly distributed datasets (url, email, and yago), a maximum fanout of 32 is sufficient to achieve a lower average tree height than a fixed span trie with a fanout of 256. For the url dataset, even the HOT structure with a maximum fanout of 32 has a three times lower tree height than a fixed-span trie (HOT-32: 8.5, ART: 26.5) and a HOT structure with a maximum fanout of 8 has a 17% lower mean leaf depth and a 50% lower maximum leaf depth.

The charts in Figure 7.1 further illustrate that the maximum depth of leaf entries for HOT structures only slightly diverges from the mean depth level. Hence, leaf entries of HOT structures are placed on levels close to the mean depth level, which leads to predictable access performance. In contrast, the levels of their leaf entries vary significantly for binary and fixed span tries, which leads to bad worst-case performance for entries close to the maximum depth level. In our experiments, HOT structures with a fanout higher than 128, achieve a maximum leaf depth identical to the mean depth for all datasets. While the mean depth for ART in the case of the url dataset is 26.5, its maximum depth is 62.0. The effect is even more severe for binary Patricia tries. For the url dataset, the mean depth of binary Patricia tries is 96.1, whereas their maximum depth is 207.0. In contrast, a HOT with a fanout of 256 achieves a maximum and mean depth of 4 and a HOT with a fanout of 1024 even achieves a maximum depth of 3.

To conclude, we observe that HOT structures achieve a lower mean depth for non-uniformly distributed datasets in comparison to fixed span trie structures with the same maximum fanout. Even in the case of uniformly distributed datasets, HOT achieves the same maximum depth as the corresponding fixed span trie counterparts. We therefore argue that HOT fulfills the requirements of a general-purpose index structure, achieving a consistently low depth distribution of leaves regardless of the key size or distribution when compared with existing trie structures in our experiments. By maintaining a consistently

low overall tree height regardless of the distribution of the stored keys, HOT's height-reducing algorithm answers our first research question.

7.3 Node Layouts

In this section, we examine access performance and memory consumption of our proposed HOT node layouts. For this purpose, we implemented read-only variations of the node layouts described in Chapter 5. We optimized each node layout in terms of performance and cache efficiency. Hence, the maximum node fanout differs between the evaluated node layouts. In the following, we list the evaluated HOT variations and briefly describe their implementations.

- Hierarchical Node Layouts:
  - *Direct node layout (DL)*: The evaluated DL has a maximum fanout of 32 and an inner node size of 48 bit. The first 16 bits of each inner node contain the corresponding discriminative bit position, while the following 16-bit wide entries address the left and right child relative to the current node.
  - *Indirect node layout (IL)*: The IL used in the evaluation has a maximum fanout of 32 and a maximum node header size of 64 bytes. Hence, each internal node is 16 bits wide. The first eleven bits out of these 16 bits store the inner node's corresponding discriminative bit position and the next five bits store the number of entries in the inner node's left subtree.
  - *Leaf optimized node layout (LOL)*: The evaluated LOL implementation uses a maximum fanout of 256. To reduce the memory footprint, we use four different inner nodes that adapt to the actual data stored. Inner node type I requires 8 bits, inner node type II requires 16 bits, inner node type III requires 24 bits and inner node type IV requires 32 bits. To distinguish the different inner nodes types, each inner node starts with a header encoding the node type. To reduce the overhead of storing the node types, we store the node type using a unary encoding to favor the shorter inner node types coded by smaller numbers. We distribute the remaining bits to store the delta discriminating bit, the delta encoded number of entries in the inner node's left subtree and the delta encoded size of the left subtree as follows: 3/2/2 (I), 6/6/2 (II), 9/7/5 (III), 11/8/10 (IV).

  - *Variable Length Generalized Indirect Node Layout (VGIL)*: The maximum fanout of the evaluated VGIL is 128. It uses four different inner node types to provide variable sized inner nodes. The first two bits of each inner node's header store the binary encoded node type. The number of bits allocated per inner node header and the corresponding separators vary depending on the node type. Inner node type I uses 8 bits and all other inner node types use 16 bits for their headers. Inner Node type I and II require 8 bits, type III requires 16 bits, and type IV requires 24 bits per separator. The next 3 bits of the inner node's header determine the inner node's span and the remaining header bits store the discriminative bit position. Each separator stores the delta-encoded number of entries in its high bits and the delta-encoded substructure size in its low bits. The exact distribution between the number of hight bits and the number of low bits per separator for the four different node types is 5/3 (I and II), 9/7 (III), and 13/11 (IV).

- Linearized Node Layouts:

  - Fixed-sized Linearized Node Layout (FLL): Each node in the FLL has a maximum fanout of 32. It uses a fixed multi-mask layout reserving 32 bytes for the extraction positions and an additional 32 bytes for the corresponding extraction masks. To store its sparse partial keys, the FLL uses 32-bit integers.

  - Adaptive Linearized Node Layout (ALL): The ALL has a maximum fanout of 32. Each ALL contains three sections: extraction information, sparse partial keys, and pointers. Depending on the stored keys, the smallest fitting node type is chosen. In contrast to the design shown in Figure 5.17, we support only 8 different node types in this particular implementation and omit the multi-mask layout with 16 extraction masks and 32-bit partial keys. Supporting only 8 different node type allows for conveniently encoding the node type in the lower three bits of each node's address. The ALL exploits this addressing scheme to overlap branch misprediction caused by node type selection with prefetching the node's first four cache lines. For alignment reasons the ALL implementation pads the partial key section to 64-bit boundaries.

  - Size Adaptive Linearized Node Layout (SLL): The SLL is an extension of the ALL. It supports a maximum fanout of 256 and a maximum node header size of 192 bytes. To support maximum fanouts of 256 it uses multiple 256-bit wide AVX registers, to com-

pare all sparse partial keys, and uses SIMD operations to identify the final result candidate.

To evaluate the access characteristics and memory footprint of the different node types, for each of the evaluated index structures, and each of the four datasets, we insert 50 million entries and measure the resulting tree height and memory consumption. To evaluate the access characteristics, we issue 100 million randomly distributed lookup operations and measure the throughput in million operations per second. Additionally, for each lookup operation we track the number of instructions, L1-cache misses, LLC-cache misses, branch misses, and the number of branch instructions per lookup operation. For hierarchical index structures, we also collect the path length per entry, which corresponds to the number of inner nodes accessed in a single lookup operation.

In the following, we first discuss the results of the hierarchical node layouts. Secondly, we discuss the corresponding results of the linearized node layouts. Finally, we compare the best linearized- and hierarchical approaches (ALL and LOL), with the state-of-the-art trie-based index structures ART and Masstree. For each comparison, we use a facetted barplot to show the results. Each facet corresponds to one of the four evaluated datasets and each row contains the data of a single metric. The last row—containing the space consumption in bytes per entry—is presented differently. As the space required to store the 8-byte tuple identifiers (tid) is the same for all index structures, we highlight the lower part of the space consumption bar that corresponds to the space required to store a single tid, with a white background.

7.3.1 Hierarchical Node Layouts

In this section, we discuss the results for the four evaluated hierarchical node layouts, namely direct node layout (DL), indirect node layout (IL), leaf optimized node layout (LOL), and variable length generalized indirect node layout (VGIL). The results of this hierarchical node layout evaluation are shown in Figure 7.2.

With regards to memory consumption, the three indirect node layouts (IL, LOL, and VGIL) require only half the amount of memory in comparison to the direct node layout (DL). The most space-efficient layout is LOL and it has an 80% smaller footprint for storing a node's structural area in comparison to the direct node layout. More precisely, LOL requires between 1.15 bytes per entry for the random integers dataset and 1.72 bytes per entry in the case of the url dataset. In comparison, the generalized indirect node layout requires roughly 40% more space and the IL between 140% (random) and

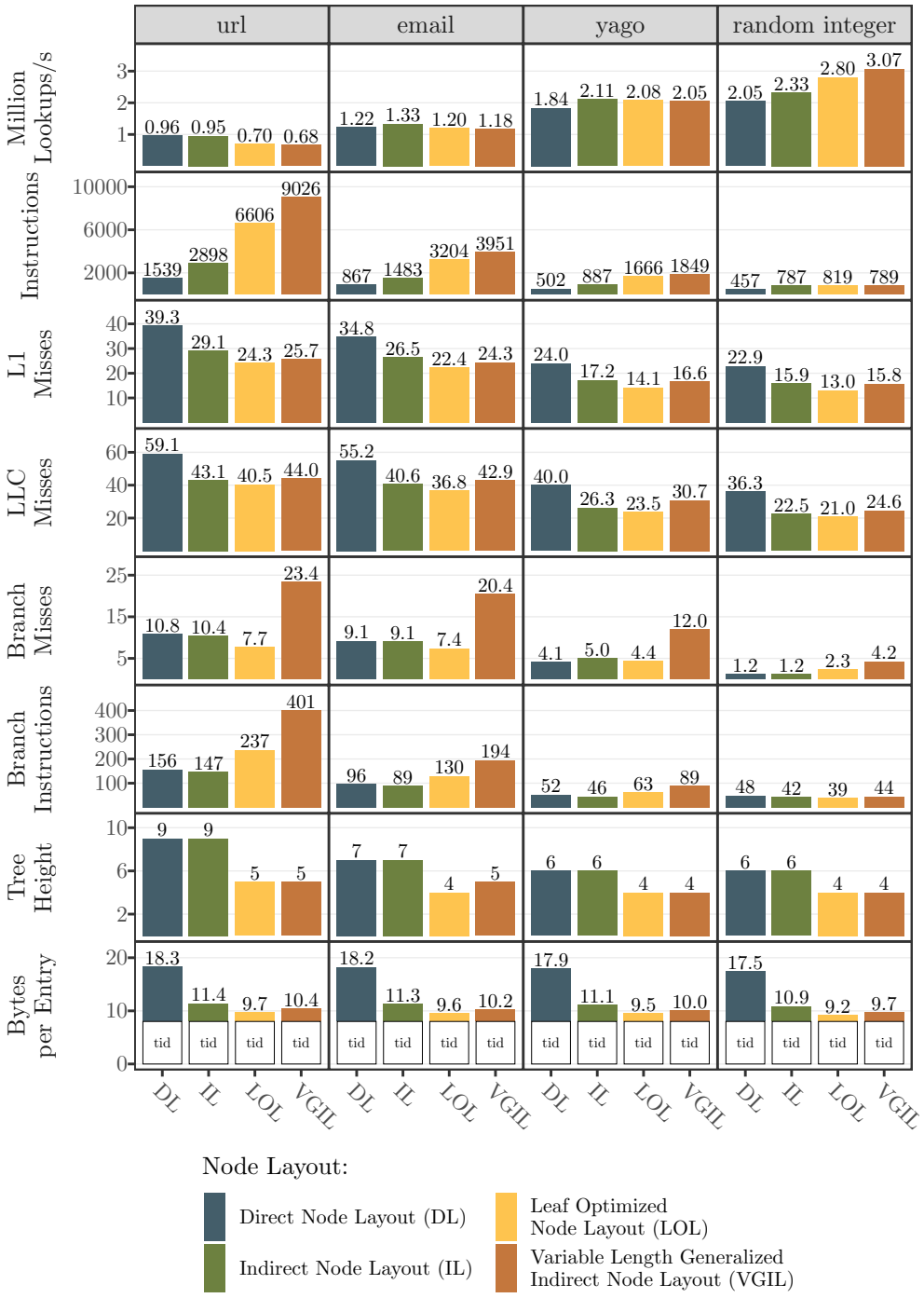


Figure 7.2: Memory consumption and access performance for the hierarchical node layouts: variable length generalized indirect node layout (VGIL), leaf optimized node layout (LOL), indirect node layout (IL) and direct node layout (DL).

90% (url) more space per entry than LOL. As both LOL and VGIL achieve their memory efficiency by collapsing multiple inner nodes, both strategies require more complex search algorithms, which have to deal with higher degree inner nodes. The complexity of supporting higher degree inner nodes manifests itself in a larger number of instructions per search operation for LOL and VGIL in contrast to the non-collapsing node layouts DL and IL. While the collapsing indirect node layouts LOL and VGIL require roughly 30% more instructions for the uniformly distributed integer dataset, for increasingly irregular datasets and more so for datasets with longer keys, this overhead increases significantly. For instance, VGIL requires 211% more instructions per search operation on the url dataset than an IL for the same dataset, with the IL itself requiring 80% more instructions than a DL. This increasing number of instructions for non-uniform datasets and even more so for the textual datasets with longer keys has a direct impact on the operations per second of the collapsing node layouts. While the collapsing node layouts achieve higher throughput (LOL 22%, VGIL 26%) for the uniform integer dataset, the throughput deteriorates for the other datasets. In the worst case, this leads to a roughly 25% lower throughput in comparison to a plain indirect node layout for the url dataset.

To get a better understanding, why the number of instructions required by collapsing node layouts explodes for more complex datasets, we evaluate the efficiency of the collapsing strategies in terms of how many BiNodes can be collapsed. To do so, for each entry we measure the number of inner nodes along its path from the root. As the number of inner nodes is equivalent for both IL and DL, we only consider the DL and omit the IL.

We show the exact numbers for the resulting path length distributions in Figure 7.3 and a boxplot that summarizes this information in Figure 7.4.

The spectra shown in Figure 7.3 allow us to identify how the node collapsing algorithms impact the actual leaf node distributions. Not only do the collapsing node strategies shorten the overall path length but they also might reshape the distribution. For instance, both node collapsing strategies convert the original distribution for the Yago dataset that contains two spikes into a gaussian distribution.

In general, it becomes apparent that both collapsing strategies are more efficient for datasets containing short uniformly distributed integers and less efficient for the textual datasets featuring longer keys. While LOL reduces the average path length for the random integer dataset by 50%, in the case of the email and url datasets it reduces the path length by only 2%. While VGIL improves on LOL by achieving a 70% reduction for the random integers

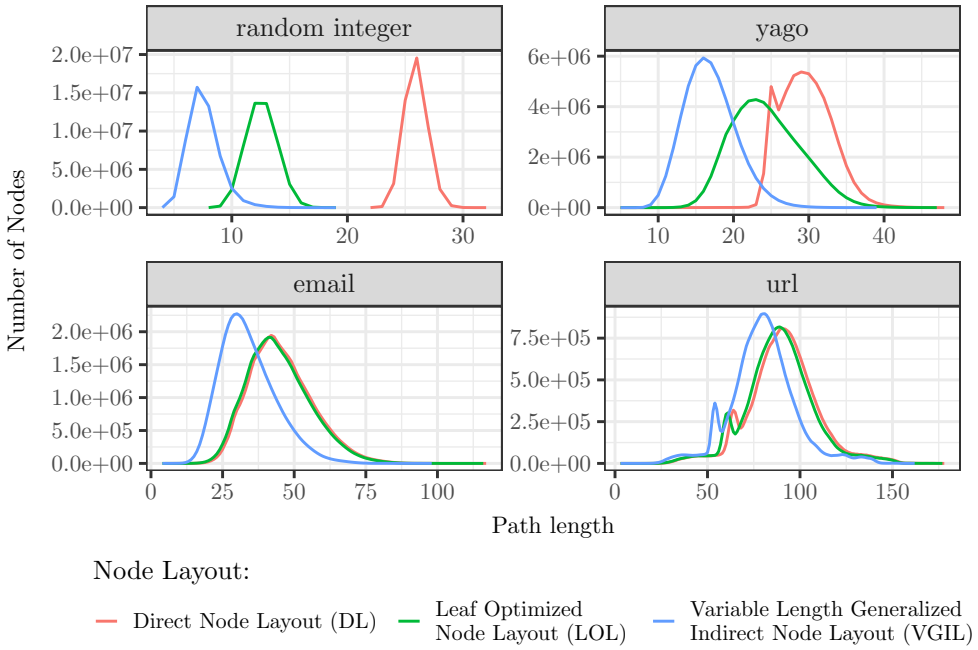


Figure 7.3: The spectrum of paths from the indexs' roots to their leaves according to their path length for the node layouts variable length generalized indirect node layout (VGIL), leaf optimized node layout (LOL) and direct node layout (DL). The path length is the number of inner nodes along the path from the root to the leaf entry.

dataset, its effect is reduced to 25% in the case of the email and to only 12.5% for the url dataset.

We conclude that indirect node layouts and especially those using variable length encoding significantly reduce memory consumption in comparison to direct node layouts. However, the collapsing node strategies that feature higher memory efficiency, achieve lower access throughput compared to a plain indirect node layout for non-uniform datasets.

7.3.2 Linearized Node Layouts

After evaluating our hierarchical node layouts, we now focus on the category of linearized node layouts. In this evaluation, we compare the fixed-sized linearized node layout (FLL), with the adaptive linearized node layout (ALL) and the size adaptive linearized node layout (SLL). The SLL is the most space-

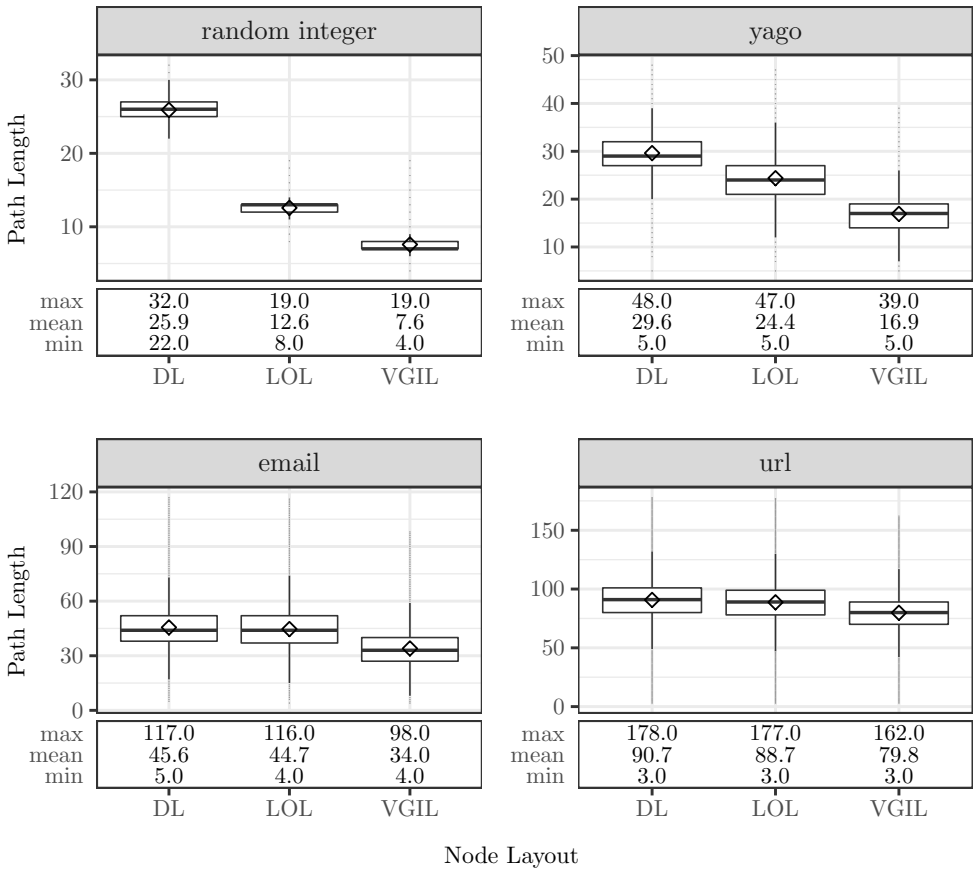


Figure 7.4: Box plot of path length distribution for the index structures variable length generalized indirect node layout (VGIL), leaf optimized node layout (LOL) and direct node layout (DL). It summarizes the path length spectrum shown in Figure 7.3. The path length is the number of inner nodes along the path from the root to the leaf entry.

efficient linearized node layout and requires between 49% and 58% less space than the plain FLL and still between 6% and 2% less space than the ALL. While SLL provides the best memory efficiency, ALL achieves the highest throughput. It executes between 25% and 125% more search operations in comparison to the FLL and still between 6% and 42% in comparison with the ALL. The reason for the higher throughput of the ALL in comparison to the SLL is the smaller number of instructions and branch misses per search operation. Although having a lower throughput, the SLL achieves slightly higher cache efficiency with less L1- and LLC misses due to smaller overall tree heights.

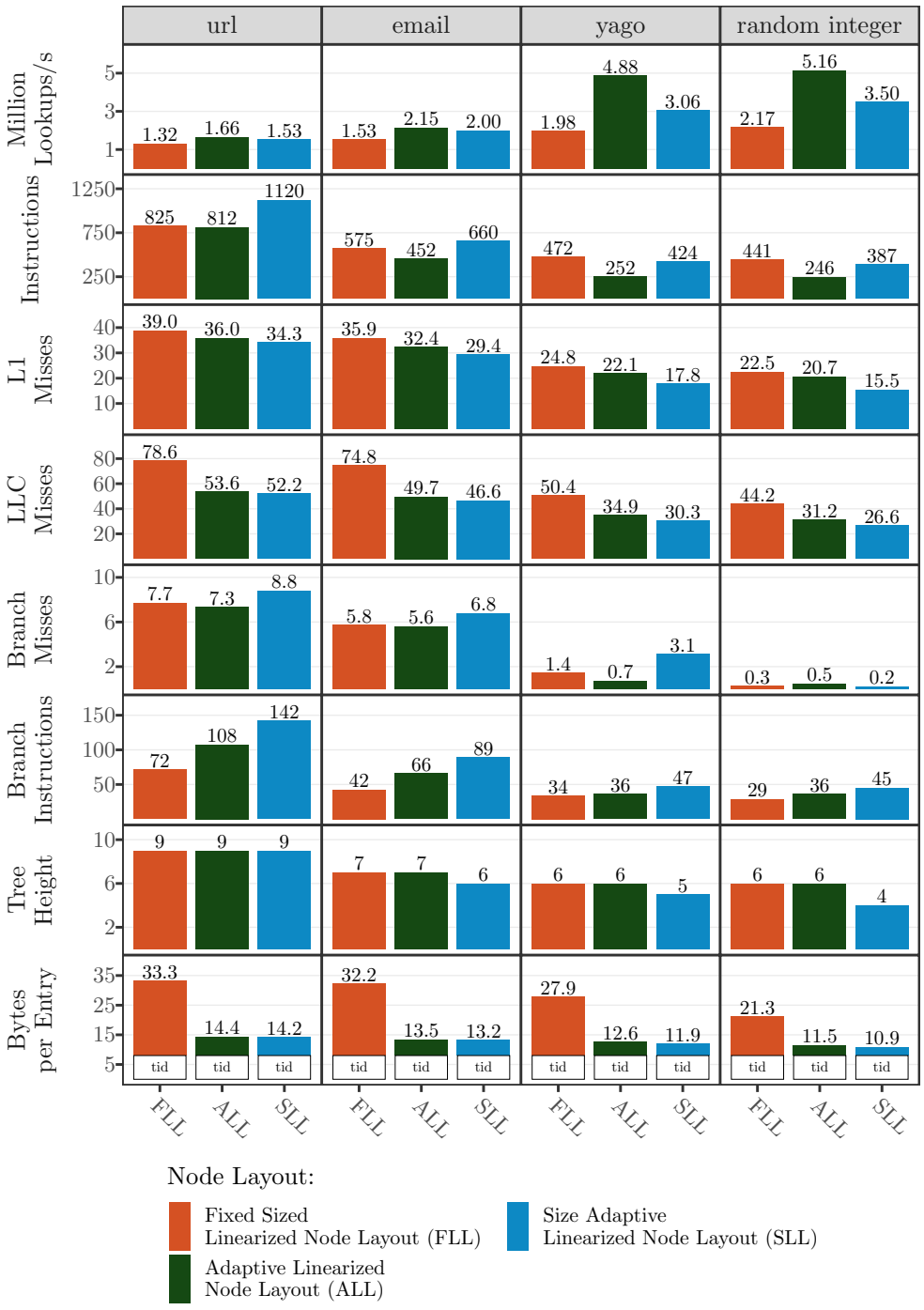


Figure 7.5: Memory consumption and access performance for the linearized node layouts: adaptive linearized node layout (ALL), size adaptive linearized node layout (SLL) and fixed-sized linearized node layout (FLL).

7.3.3 Overall Node Layout Comparison

Finally, we compare the leaf optimized node layout (LOL), which is HOT's most space-efficient node layout and the adaptive linearized node layout (ALL)—HOT's fastest node layout—with two modern trie variants ART and Masstree and a standard B$^+$-Tree in terms of lookup throughput and memory efficiency. The results of this mixed evaluation are shown in Figure 7.6. In addition to the previous comparisons, a dashed red line marks the average key length for the textual datasets in the mixed evaluation. This red line allows us to identify the index structures that require more space than the actual keyset.

With regards to memory consumption, for all datasets, both HOT variations achieve a lower memory consumption than the other evaluated index structures. The LOL layout is the most space-efficient HOT variation. Over all evaluated datasets, its spaced consumption lies within a 6% boundary, thereby it requires only between 33% (random) and 27% (url) of the ALL node layout's space to store the structural information. In absolute numbers, LOL never requires more than 2 bytes per entry to store its structural information. Even ALL's worst-case memory requirement for the structural information in the case of the URL dataset (6.45 bytes/entry) is 59% lower than the lowest measured memory requirement of the other evaluated index structures (ART:15.5 bytes/url). More remarkably, for the structural information, the memory consumption of HOT for all evaluated datasets remains within a 25% boundary, while ART's memory consumption varies up to 50% and Masstree's by as much as 215%.

Thereby, the HOT variations are the only trie based index structures, which for all datasets have a space consumption which is substantially below the space consumption of the B$^+$-Tree. Due to the B$^+$-Tree's design and the decision to use tids to resolve keys longer than 8 bytes, the amount of space required is the same (1.26 GB) for all datasets. However, for all datasets, the B$^+$-Tree requires at least 88% more space than the HOT variations' worst-case space consumption measured for the ALL structure on the url dataset. Moreover, the HOT variations are the only index structures that for both textual datasets require less space than the actual raw keys (email: 43%, url: 74%).

Although requiring more memory per entry, ALL achieves the best lookup performance of all evaluated datasets except for the random integers, where it has a slightly lower (4%) throughput than ART in this particular case. We will see later in Section 7.4 that for random integers, and for a slightly different random integer dataset as used in the YCSB benchmark ALL is slightly faster than ART. For all other datasets, ALL executes between 93% and 161%

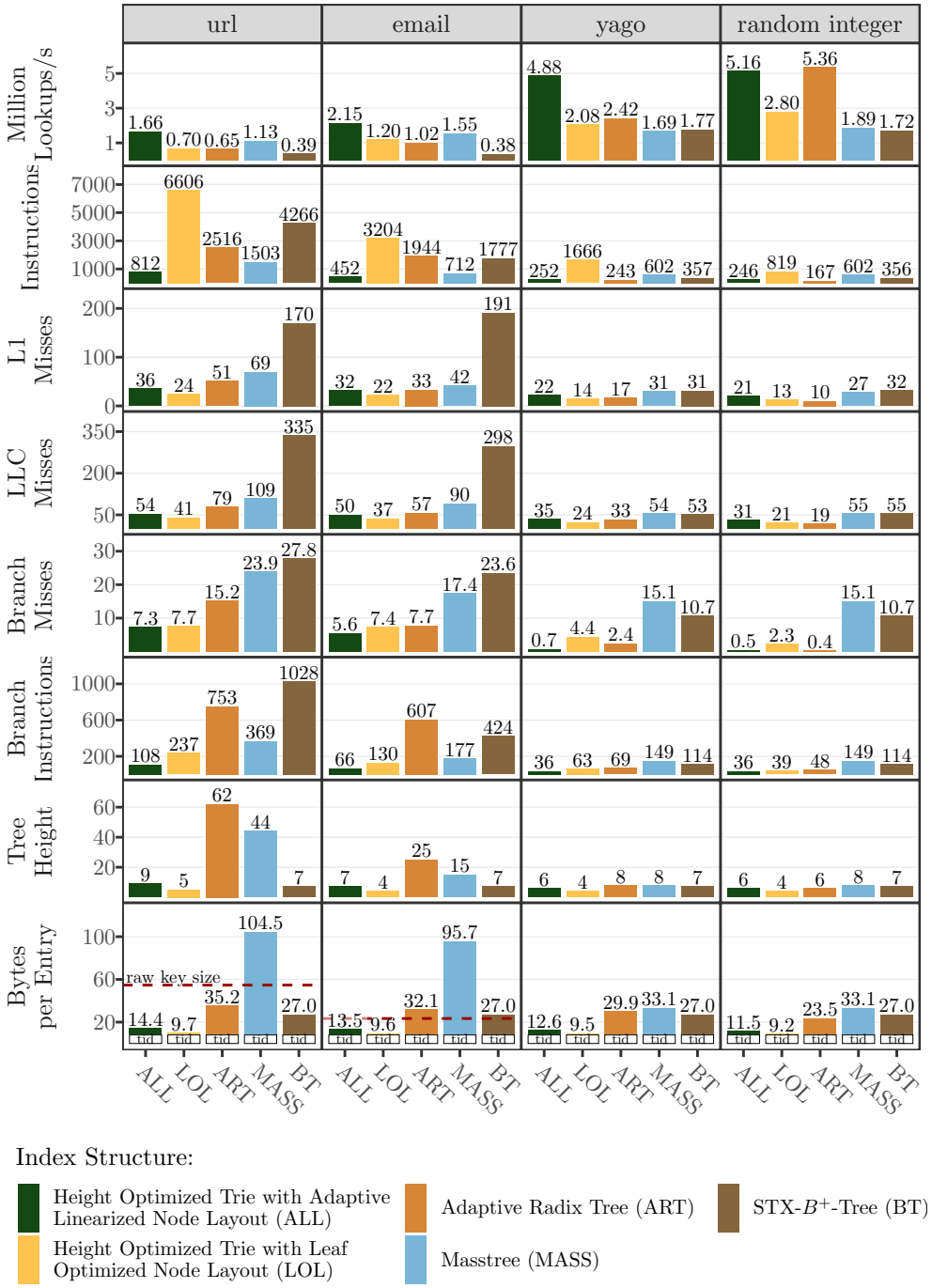


Figure 7.6: Memory consumption and access performance of the fastest (Adaptive Linearized Node Layout) and the space-efficient (Leaf Optimized Node Layout) HOT node layout in comparison to two trie based indexes (ART and Masstree) and a B$^+$-Tree. The dashed red line marks the average key length of the textual datasets (url: 55 bytes, email: 23 bytes).

more lookup operations than ART though. As expected, LOL's space-efficient memory layout results in the best cache efficiency of all evaluated data structures. Only in the case of the uniformly distributed random integer dataset—presenting the sweet spot for fixed span node layouts like ART—LOL has 5% more LLC misses and 23% more L1 misses. In the case of the URL dataset, LOL has 53% less L1 misses and 52% less LLC misses, though. However, the cost of LOL's memory and cache efficiency is the large number of instructions required to traverse its hierarchical variable length indirect node layout. For all evaluated datasets, LOL executes the most instructions per search operation. In particular, it executes between 35% and 160% more instructions than the index structure with the second most instructions per search operation for the respective dataset. This also affects LOL's throughput, which is between 42% and 58% lower than ALL's throughput. However, LOL has never the slowest lookup performance of the evaluated structures. In the case of the integer datasets, LOL executes between 34% (yago) and 64% (random) more lookup operations in comparison to Masstree and in the case of the textual datasets still between 22% (email) and 11% (url) more lookup operations as ART.

To get a better understanding of how our different HOT node layouts and the three state-of-the-art index structures ART, Masstree, and the STX-B$^+$-Tree compare to each other, we provide two faceted scatter plots that visualize the trade-offs between access performance and space consumption. In these scatter plots, the x-axis represents access latency and the y-axis represents memory consumption per entry. Each marker represents a single index structure, and the position of the marker corresponds to the access latency and memory consumption of the index structure. Figure 7.7 shows the memory consumption and access latency of the fastest (Adaptive Linearized Node Layout) and the most space-efficient (Leaf Optimized Node Layout) HOT node layout in comparison to ART, Masstree, and the STX-B$^+$-Tree, while Figure 7.8 focuses only on our HOT node layouts, which are presented in Chapter 5.

Figure 7.7 shows that regardless of the datasets evaluated, the two HOT variants are more space-efficient than the other evaluated index structures. Although the adaptive linearized node layout requires about half the memory than the other non-HOT index structures, the leaf-optimized node layout reduces the space required for the structural information by another 50%, regardless of the dataset being evaluated. Whereas the B-Tree has the worst latency for all evaluated datasets, ART provides the best access performance only in the case of the random integer dataset, and Masstree achieves competitive access performance only on textual data, the HOT index structures provide consistent access performance. While the adaptive linearized node layout provides the lowest latency on all but the random integer dataset (where its latency is only slightly higher than ART's), also the space-optimized leaf

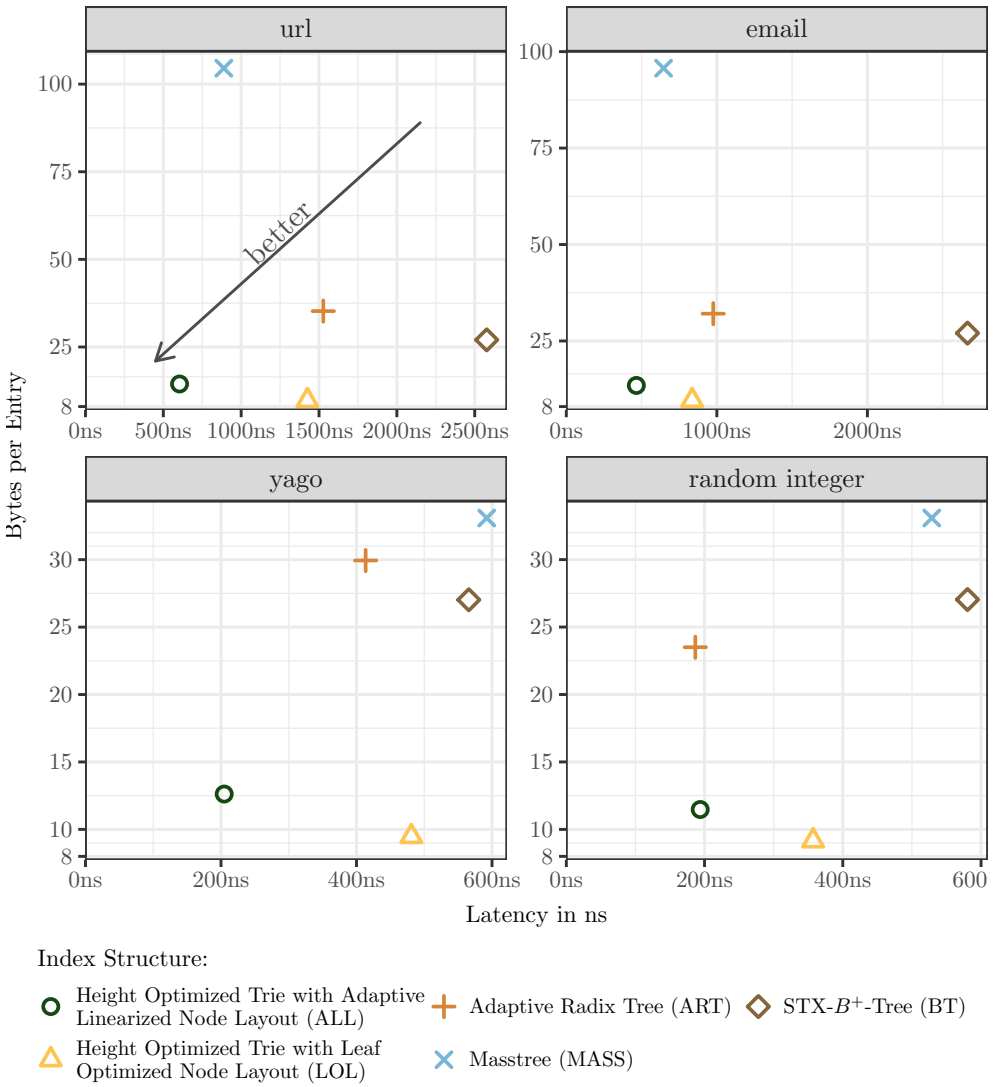


Figure 7.7: Memory consumption and access latency for the fastest (Adaptive Linearized Node Layout) and the most space-efficient (Leaf Optimized Node Layout) HOT node layout in comparison to two trie based indexes (ART and Masstree) and a B$^+$-Tree on four datasets with 50 million keys of different length (url: 55 bytes, email: 23 bytes, yago and random integers:8 bytes).

optimized node layout offers a lower access latency than two out of three non-HOT index structures across all datasets.

Concerning the remaining HOT node layouts, Figure 7.8 shows that the more sophisticated layouts require less memory compared to their base variants. Furthermore, the figure shows that linearized node layouts have lower access

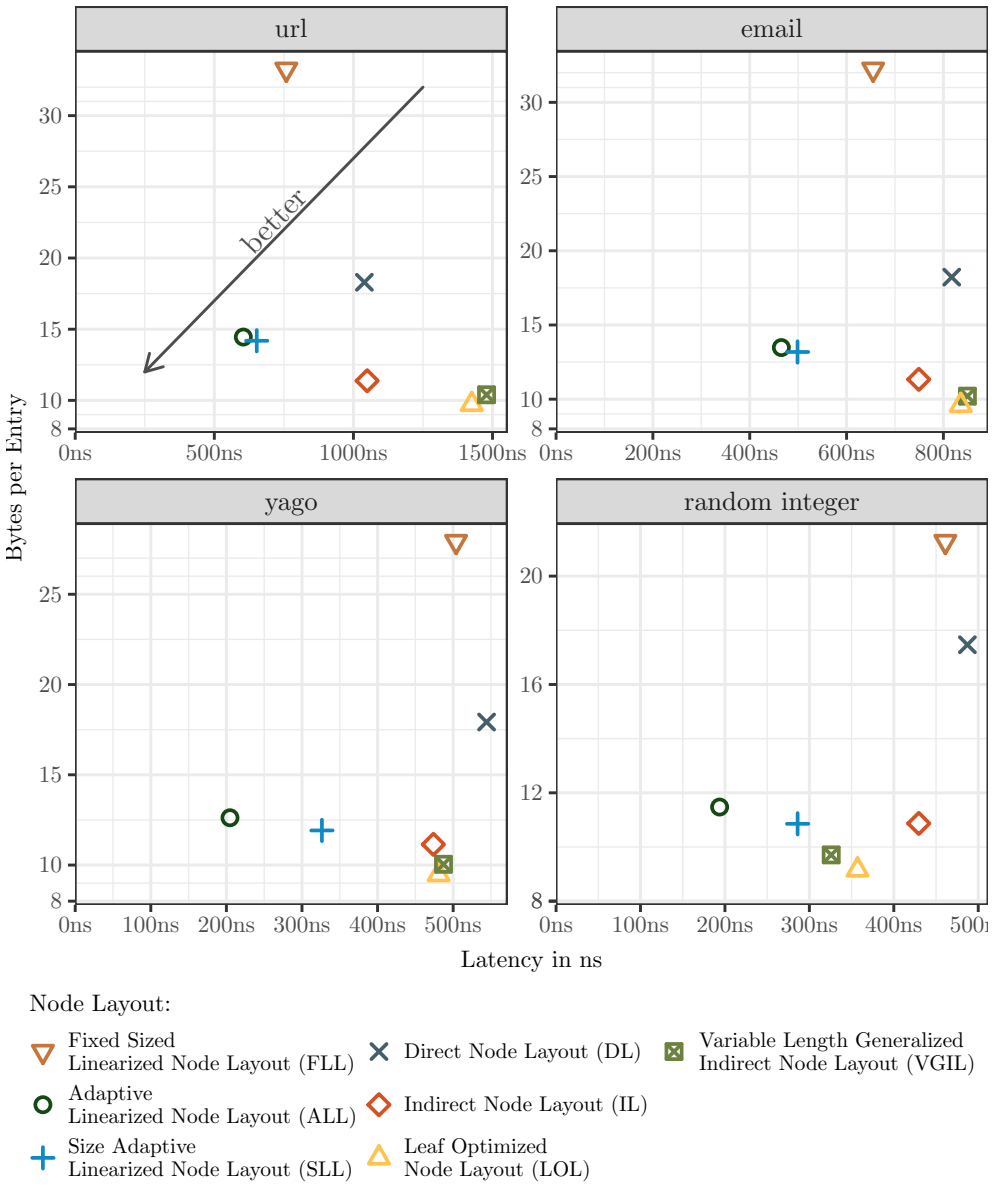


Figure 7.8: Memory consumption and access latency for all HOT node layouts on four datasets with 50 million keys of different length (url: 55 bytes, email: 23 bytes, yago and random integers:8 bytes).

latency, while hierarchical node layouts provide better memory efficiency. In the case of the linearized node layouts, the size adaptive linearized node layout offers the highest memory efficiency, while the adaptive linearized node layout has lower latency, but also slightly higher memory requirements. For hierarchical node layouts, the more sophisticated leaf optimized node layout and the generalized variable length indirect node layout have lower memory

requirements than the simpler direct and indirect node layouts. In return, these simpler node layouts achieve lower access latency. Of these two simpler hierarchical node layouts, the indirect node layout is preferable to the direct node layout because it requires less space and generally offers lower access latency. Only in the case of the url dataset, the indirect node layout has a negligible 1% higher latency than the direct node layout. In summary, of the index structures evaluated, the HOT-based index structures are the only indexes that are located in the favorable lower-left corner of the scatterplot, thus achieving the desirable characteristics of low access latency combined with high memory efficiency.

Overall, from this read-only micro-benchmark we conclude, that LOL is the best overall node layout in terms of memory consumption and cache efficiency, while ALL is the best overall node layout in terms of throughput. Even more, ALL is still superior to all non height optimizing trie structures in terms of memory consumption. Therefore, in the remainder of the evaluation, we focus on holistically evaluating ALL against three state-of-the-art index structures by applying different workloads and studying HOT's multi-threaded performance.

7.4 YCSB - Single-Threaded Workloads

After evaluating different node layouts in terms of their memory consumption and simple read-only access characteristics, we now focus on HOT's performance for various workloads using a HOT implementation that supports all fundamental operations like inserts, lookups, updates, and scans. The evaluated HOT implementation uses our fastest node layout, namely the adaptive linearized node layout (ALL). For simplicity, in the remainder of this section, we use HOT synonymously for the evaluated HOT variation based on the ALL node layout. The evaluated HOT structure is implemented in C++14 and is optimized for single-threaded workloads. To ensure a fair comparison, we use single-threaded or lock-free versions of the other contestants, namely ART, Masstree, and the STX-B$^+$-Tree.

For our workloads, we rely on the work of Zhang et al. [116], who proposed an index micro-benchmark adapted from the YCSB framework [32]. We therefore base our work on the available implementation of their workload generator[4] and add support to jointly configure multiple workloads. To foster repro-

[4]https://github.com/huanchenz/index-microbench (last visited 2020-01-21)

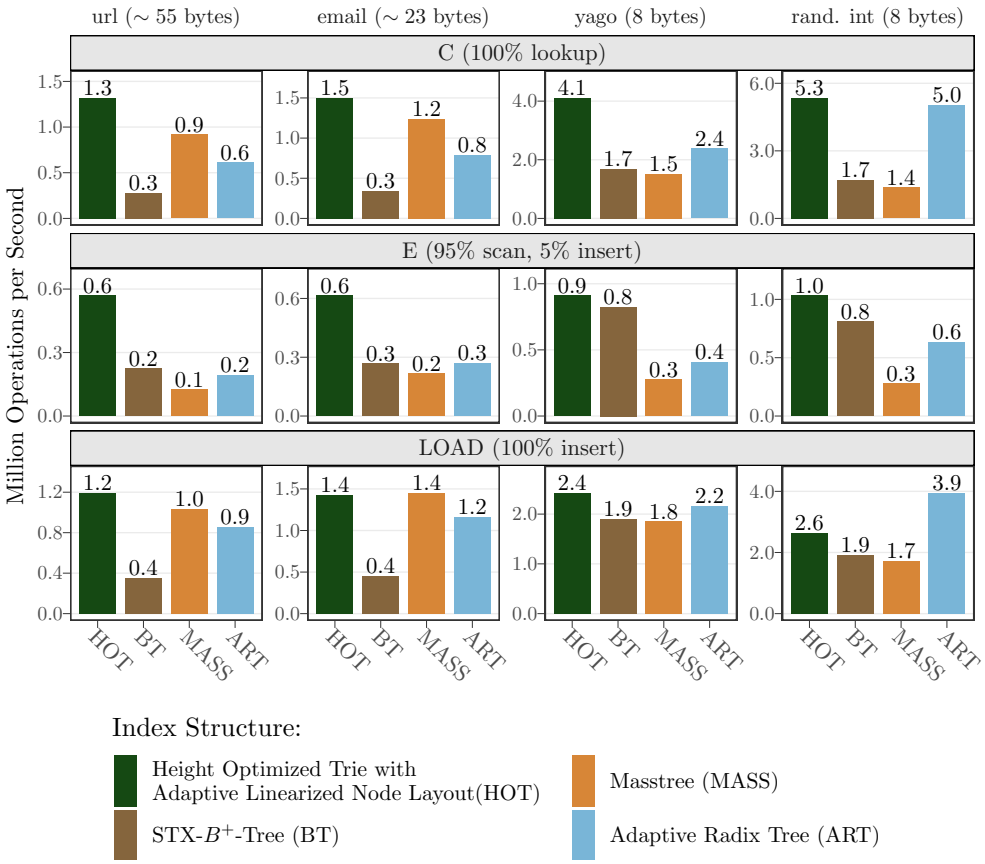


Figure 7.9: Throughput using different datasets in million operations per second for read-only workload C, scan-heavy workload E, and the insert-only load phase.

ducibility and repeatability, the extended workload generator is available online[5].

Our benchmark configurations correspond to the six YCSB core workloads: A (50% read, 50% update), B (95% read, 5% update), C (read-only), D (latest-read, 95% read, 5% insert), E (95% range-scan accessing up to 100 elements, 5% insert) and F (50% read, 50% read-modify-write) [32]. Each workload is separated into two phases: the load phase inserts 50 million keys in random order into the index structure to evaluate, and the transaction phase executes 100 million operations of the respective workload. Each benchmark configuration is created in two variants: using a Zipfian and a uniform distribution to select the records to operate on. We execute each of these benchmark config-

[5]https://github.com/speedskater/index-microbench (last visited 2020-01-21)

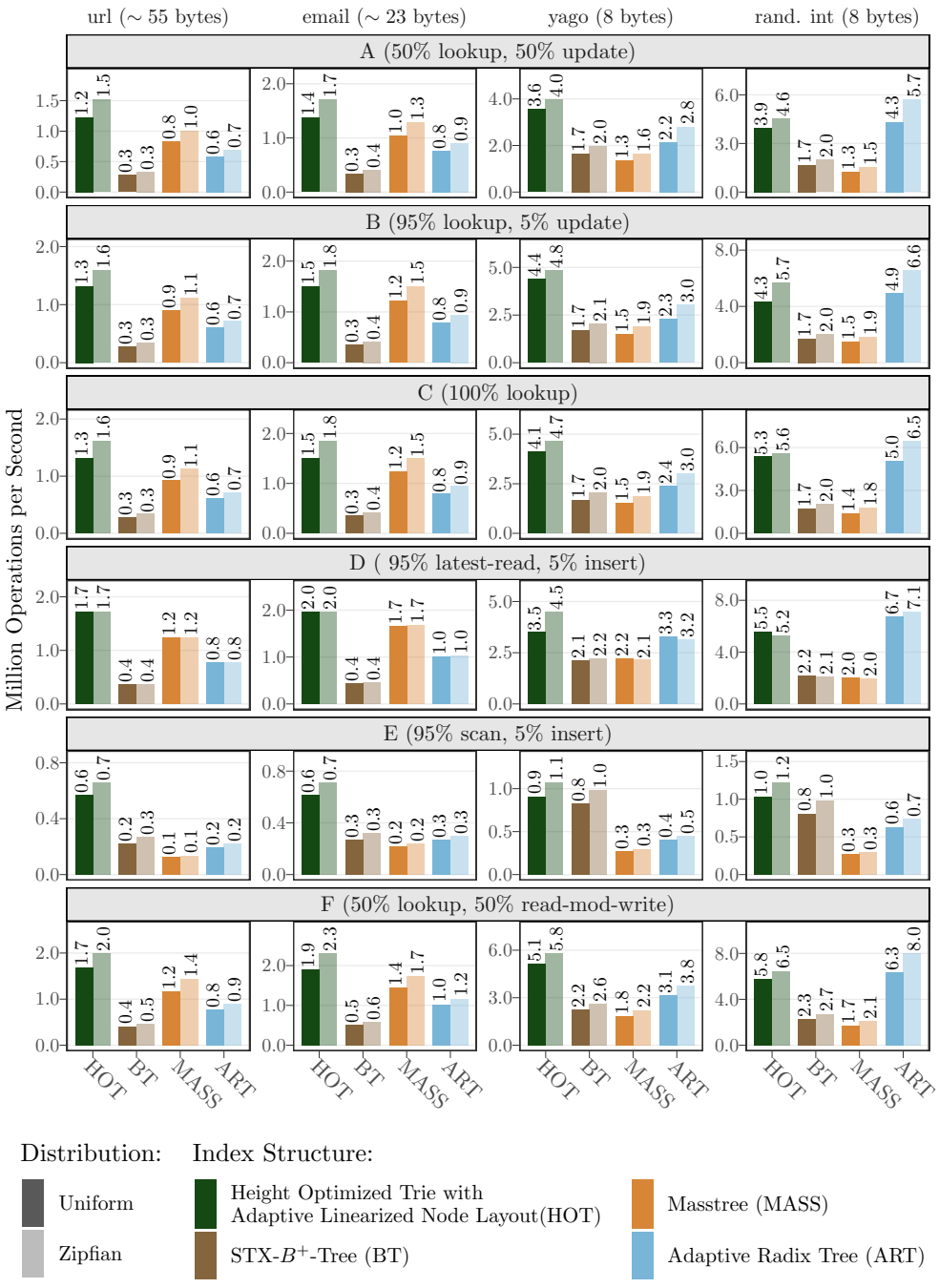


Figure 7.10: Throughput for all YCSB workloads in million operations per seconds. For each workload we provide performance numbers for uniformly distributed as well as for zipfian distributed access pattern.

urations for four different datasets, two string datasets representing long keys, and two 8 byte short key datasets.

Overall, the 6 workloads (A, B, C, D, E, F), 4 datasets (url, email, yago, integer) and 2 operation distributions (uniform, Zipf) amount to a total of 48 different benchmark configurations. For each of the 48 benchmark configurations, we collect the achieved throughput in terms of million operations per second. We do not discuss the memory consumption for the individual workloads, as these values are identical to the memory consumption collected in Section 7.3.

We mainly discuss the results of the uniformly distributed lookup-only workload C, the scan-heavy (95% scan, 5% insert) workload E, consisting of short range-scans accessing up to 100 entries and an insert-only workload that corresponds to the load phase of workload C. As the remaining workloads A, B, D, and F only represent combinations of the workloads presented in Figure 7.9 and the performance results for the Zipfian distributed operations are similar to the uniformly distributed ones, we omit those results for the remainder of this section. However, for completeness, an overview of all results are depicted in Figure 7.10.

The results of these three workloads are depicted in Figure 7.9. For each combination of workload and dataset, the figure shows a bar plot illustrating the number of million operations per second (mops) executed by each of the evaluated index structures.

For the lookup-only workload C, HOT achieves the highest throughput across all evaluated datasets compared to the other evaluated index structures. In the case of the uniformly distributed random integers, HOT has a 6% higher read-only throughput than ART. However, we conclude from the previous comparison from Section 7.3—where ART shows a 4% higher throughput for the same dataset—that HOT and ART achieve an equivalent read throughput for these types of datasets. We attribute the exact difference to a different random number sequence and different compiler optimizations chosen for the alternative code paths. For the other evaluated index structures, HOT achieves an at least 25% higher throughput across all evaluated datasets. For the scan-heavy workload E, consisting of short range-scans accessing up to 100 elements, HOT has the highest throughput for all key distributions. In the case of the URL dataset, HOT's throughput is 200% higher than for the other data structures. For the insert-only workload, the measured performance characteristics are similar to the read operations discussed in the context of workload C. An exception is the insertion throughput for the integer dataset, where ART achieves a 50% higher insertion throughput compared to HOT. The reason for ART's higher insertion rate is ART's simpler in-place insertion algorithm. For

the other datasets, however, the lower tree height of HOT and thus the better cache-efficiency outweighs the overhead of HOT's copy-on-write insertion algorithm.

Having discussed the raw runtime characteristics of the evaluated index structures, we conclude that HOT can achieve consistently high performance regardless of the evaluated workload or dataset, which makes it highly promising as a general-purpose index structure for main memory databases. While the other evaluated index structures achieve peak performance for specific workloads, they are not able to provide consistently high throughput for all workloads and datasets.

7.5 Scalability

After evaluating HOT in terms of its single-threaded performance in Section 7.4, we evaluate HOT in terms of scalability in this chapter. Specifically, we evaluate a HOT implementation using an ALL node layout and the synchronization protocol presented in Chapter 6.

For each of the datasets described in Section 7.1, we execute a workload consisting of 50 million randomly distributed insert operations, followed by 100 million uniformly distributed random lookups. Each workload is executed seven times for thread counts between one and ten, with ten representing the maximum physical core count of the server used to run the evaluation. To prevent outliers, the median throughput of the seven executed runs is considered for the comparison.

We conduct this experiment for the synchronized versions of Masstree, ART (using the ROWEX synchronization protocol), and HOT. In contrast to the previously conducted single-threaded experiments, these variations of the evaluated index structures support concurrent modifications. Due to the lack of synchronization, we omit the STX B-Tree for the scalability evaluation.

We show the results of the scalability evaluation in Figure 7.11. All evaluated index structures achieve a nearly linear speedup for insert and lookup operations for all of the evaluated datasets. However, the speedups vary slightly between the evaluated data structures. For instance, the mean speedups for all lookup operations are 9.96 for HOT, 9.91 for ART, and 10.1 for Masstree. The respective mean speedups for the insert operations are 9.00 for HOT, 9.51 for ART, and 7.87 for Masstree. Since our main focus in this experiment was to show the scalability of our synchronization protocol, we didn't specifically optimize the synchronization primitives like the spin-lock or epoch based

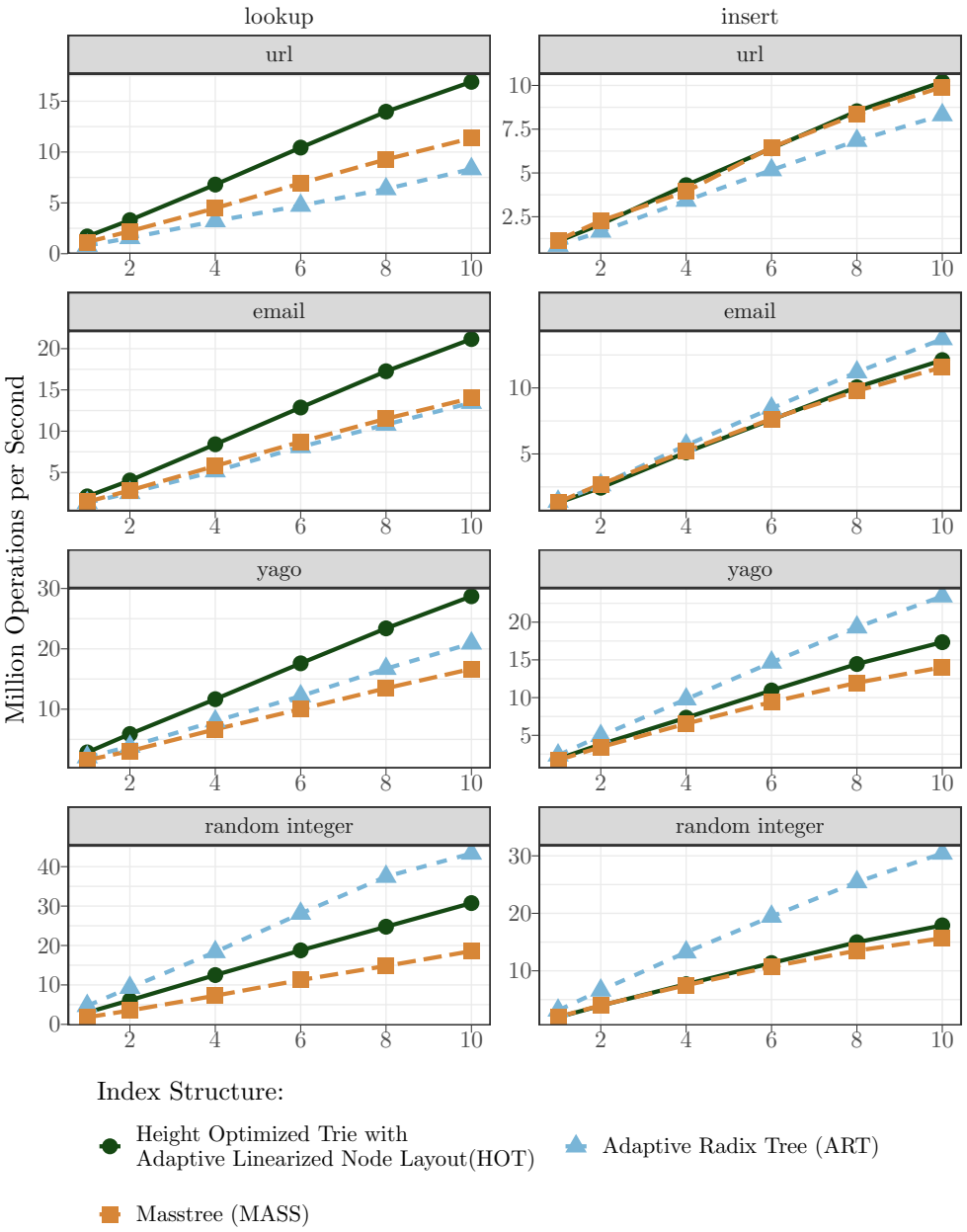


Figure 7.11: Scalability results for the url, email, yago and random integer datasets. For each index structure and dataset we provide performance numbers for 50 million concurrent inserts and 100 million concurrent random lookup operations. We perform the same experiment for thread counts between one and ten.

memory reclamation, so the absolute throughput numbers have to be interpreted from this point of view. While ART's ROWEX implementation achieves a higher ingestion rate in absolute numbers than HOT for the email, yago, and the random integers dataset, HOT achieves the highest lookup throughput for the url, email, and yago dataset.

From these experiments, we conclude that besides featuring excellent single-threaded performance, HOT's synchronization protocol achieves almost linear scalability.

CHAPTER 8

Conclusion

Although databases have been a subject of intensive research for more than four decades, the shift to modern multi-core hardware architectures with vast amounts of main memory led to revisiting well-established database design principles. In the course of this process, the design of individual core components and thus, the design of index structures as one of these core components have also been reconsidered. It turned out that trie based index structure, are particularly well suited for such modern architectures. However, in contrast to the ubiquitous B-Tree, which is predominant on disk-based database systems, the performance and memory consumption of tries is highly dependent on the stored key set. To overcome these limitations, we developed the Height Optimized Trie to answer the main research questions of this thesis:

RQ1: How can trie based index structures be designed to adapt to different key distributions while maintaining a low overall tree height?

The Height Optimized Trie (HOT) represents our approach to create a trie structure that maintains a low overall tree height by adapting to different key distributions and datasets. It is based on a novel algorithm that incrementally partitions the nodes of a binary Patricia trie into compound nodes so that each node contains at most k entries. We prove that HOT not only reduces the

overall tree height but reaches the minimum tree height under the maximum fanout constraint k. Besides, we proved that HOT satisfies the following two properties. First, HOT's incremental partitioning algorithm deterministically generates the same structure for the same set of keys regardless of their insertion order. Secondly, each subtree of HOT recursively satisfies the same three properties and is thus of minimal height. Using five datasets (random and dense integer, yago, email, and url) featuring different key distributions, we experimentally evaluate the height-reducing effect of HOT. Using a maximum fanout of $k = 256$, HOT always achieves the lowest overall tree height compared to a fixed span trie with the same maximum fanout. In the case of the URL dataset, HOT even achieves a fivefold height reduction compared to the fixed span trie. For the non-uniformly distributed datasets (yago, email, and url), a maximum fanout of $k = 32$ is sufficient for HOT to achieve a lower tree height.

RQ2: How can such a trie-based index structure be implemented to meet the requirements of fast access performance and low memory consumption?

We have shown that the key to implementing fast and space-efficient HOT-index structures is the design of efficient k-constrained tries that represent individual nodes. We presented seven different node layouts that can be divided into two categories depending on the implementation approach chosen. The category of hierarchical node layouts provides the highest memory efficiency by using sophisticated encoding techniques to represent the hierarchical trie structure directly in memory. The category of linearized-node layouts provides the highest access performance by flattening the internal trie structures and thereby exploiting the SIMD-capabilities of modern CPUs. To evaluate the performance characteristics and memory efficiency of our node layouts, we compared them with three state-of-the-art index structures (ART, Masstree, and the STX B$^+$-Tree) on four different datasets (random integer, yago, email, and url). The Leaf Optimized node Layout (LOL) is our most space-efficient layout. LOL is a hierarchical node layout that combines our indirect node layout (IL) approach with a variable length encoding and delta encoding and additionally collapses so-called dense regions. By applying these encoding techniques the LOL based HOT implementation has the smallest memory footprint of all evaluated index structures. In absolute numbers, it requires less than 2 bytes per entry for all evaluated datasets. Our fastest node layout is the adaptive linearized node layout (ALL) that extensively uses modern CPU features like SIMD and bit manipulation instruction sets. In addition, to provide cache- and memory efficiency, ALL uses eight different node types that adapt to the number of discriminative bits. Our evaluation has shown that ALL provides access performance superior to all other evaluated index structures while requiring less space than the evaluated state-of-the-art index structures. We also evaluate an ALL based implementation of HOT which

supports all basic index operations, for the six YCSB workloads. The results show that HOT generally outperforms its contestants in terms of lookup- and scan performance. Even for the insert-only workload, HOT achieves peak performance on three out of four datasets, while still providing the second-highest ingestion rate for random integers that by design represent the sweet spot of the fixed-span ART.

RQ3: How can such a trie based index structure be synchronized to provide high scalability in multithreaded environments.

To answer RQ3 we have introduced a custom synchronization-protocol for HOT that allows us to scale the access performance with the number of available CPU-cores. It is based on the ROWEX-paradigm that provides wait-free read-access while using fine-grained locks exclusively for write operations. As our designed node layouts cannot be updated in-place without affecting concurrent read-access, we combine a copy-on-write node update strategy with optimistic locking and epoch-based memory reclamation. To reduce lock contention to a minimum our synchronization protocol only locks these nodes, that are modified by the current update operation. Using an experimental evaluation, we show that our synchronization protocol for HOT achieves near linear speed-up, and thus offers scalability on the same level as the other evaluated concurrent state-of-the-art index structures ART and Masstree.

To conclude, this thesis introduces the general-purpose index structure Height Optimized Trie (HOT). It is based on a novel algorithm that incrementally minimizes the overall tree height under a given maximum node fanout constraint k. The algorithm guarantees that each HOT is a recursively defined, deterministic tree structure of minimum height. Due to these properties, HOT is the first trie-based index structure that is robust against long keys and non-uniformly distributed datasets. Using different node layouts, the general approach of HOT can be adapted to different usage scenarios. For instance, we show that the leaf optimized node layout provides exceptional memory efficiency, while the adaptive linearized node layout provides superior access performance and good memory efficiency. For multi-threaded scenarios, our ROWEX-based synchronization protocol offers near-linear speed-up and scalability. We look forward to seeing how other researchers will build on this idea and how they will extend the family of node layouts and HOT based index structures in the future.

Bibliography

[1] A. Acharya, H. Zhu, and K. Shen. Adaptive algorithms for cache-efficient trie search. In *Workshop on Algorithm Engineering and Experimentation*, pages 300–315, 1999.

[2] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for organization of information. *Doklady Akademii Nauk*, 146(2):263–266, 1962.

[3] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of 27th Annual International Symposium on Computer Architecture, ISCA '00*, pages 248–259, 2000.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th International Conference on Very Large Data Bases, VLDB '99*, pages 266–277, 1999.

[5] A. Andersson and S. Nilsson. Improved Behaviour of Tries by Adaptive Branching. *Information Processing Letters*, 46(6):295–300, 1993.

[6] G. Antoshenkov. Dictionary-Based Order-Preserving String Compression. *VLDB Journal*, 6(1):26–39, 1997.

[7] G. Antoshenkov, D. B. Lomet, and J. Murray. Order Preserving String Compression. In *Proceedings of the 12th IEEE International Conference on Data Engineering, ICDE '96*, pages 655–663, 1996.

[8] N. Askitis and R. Sinha. HAT-trie: A Cache-conscious Trie-based Data Structure for Strings. In *Proceedings of the 30th Australasian Computer Science Conference, ASC '07*, pages 97–105, 2007.

[9] N. Askitis and J. Zobel. B-tries for disk-based string management. *VLDB Journal*, 18(1):157–179, 2009.

[10] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2):97–137, 1976.

[11] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. DBpedia: A Nucleus for a Web of Open Data. In *Proceedings of the Sixth International Semantic Web Conference, ISWC '07*, pages 722–735, 2007.

[12] R. Bayer. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[13] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–189, 1972.

[14] R. Bayer and M. Schkolnick. Concurrency of Operations on B-trees. *Acta Informatica*, 9(1):1–21, 1977.

[15] R. Bayer and K. Unterauer. Prefix B-Trees. *ACM Transactions on Database Systems*, 2(1):11–26, 1977.

[16] R. I. Becker and Y. Perl. The shifting algorithm technique for the partitioning of trees. *Discrete Applied Mathematics*, 62(1):15 – 34, 1995.

[17] J. L. Bentley and R. Sedgewick. Fast Algorithms for Sorting and Searching Strings. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '97*, pages 360–369. Society for Industrial and Applied Mathematics, 1997.

[18] T. Bingmann. STX B+ tree C++ template classes, 2008. `http://panthema.net/2007/stx-btree`, last accessed on 2018-09-03.

[19] R. Binna, W. Gassler, E. Zangerle, D. Pacher, and G. Specht. SpiderStore: Exploiting Main Memory for Efficient RDF Graph Representation and Fast Querying. In *Proceedings of the International Workshop*

on Semantic Data Management at the 36th International Conference on Very Large Data Bases, SemData '10, Sept. 2010.

[20] R. Binna, W. Gassler, E. Zangerle, D. Pacher, and G. Specht. SpiderStore: A Native Main Memory Approach for Graph Storage. In *Proceedings of the 23rd GI-Workshop Grundlagen von Datenbanken*, pages 91–96, 2011.

[21] R. Binna, D. Pacher, T. Meindl, and G. Specht. The DCB-Tree: A Space-Efficient Delta Coded Cache Conscious B-Tree. In *Proceedings of the First and Second International Workshops on In Memory Data Management and Analysis, IMDM '2013 and IMDM '2014, Revised Selected Papers*, pages 126–138, 2014.

[22] 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 37th ACM SIGMOD International Conference on Management of Data, SIGMOD '18*, pages 521–534, 2018.

[23] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 28th ACM SIGMOD International Conference on Management of Data, SIGMOD '09*, page 283, 2009.

[24] M. Böhm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner. Efficient In-Memory Indexing with Generalized Prefix Trees. In *Proceedings of the 14th Conference on Database Systems for Business, Technology, and Web, BTW '14*, pages 227–246, 2011.

[25] N. G. Bronson, J. Casper, H. Chafi, K. Olukotun, N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A Practical Concurrent Binary Search Tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10*, pages 257–268, 2010.

[26] D. Cervini, D. Porobic, P. Tözün, and A. Ailamaki. Applying HTM to an OLTP System. In *Proceedings of the 11th International Workshop on Data Management on New Hardware, DaMoN '15*, pages 1–7, 2015.

[27] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *Proceedings of the 27th International Conference on Very Large Data Bases, VLDB '01*, pages 181–190, 2001.

[28] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving Index Performance Through Prefetching. In *Proceedings of the 20th ACM SIGMOD International Conference on Management of Data, SIGMOD '01*, pages 235–246, 2001.

[29] E. I. Chong, S. Das, C. Freiwald, J. Srinivasan, A. Yalamanchi, M. Jagannath, A.-T. Tran, and R. Krishnan. B+-Tree Indexes with Hybrid Row Identifiers in Oracle8i. In *Proceedings of the 17th IEEE International Conference on Data Engineering, ICDE '01*, page 341, 2001.

[30] D. Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[31] J. Cong and Y. Ding. FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(1):1–12, 1994.

[32] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the First ACM Symposium on Cloud Computing, SoCC '10*, pages 143–154, 2010.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, 2001.

[34] R. De La Briandais. File Searching Using Variable Length Keys. In *Papers Presented at the March 3-5, 1959, Western Joint Computer Conference*, volume 1, pages 295–298, 1959.

[35] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small Forwarding Tables for Fast Routing Lookups. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '97*, pages 3–14, 1997.

[36] E. K. Donald. *The Art of Computer Programming, Volume III: Sorting and Searching, 2nd Edition.* Addison-Wesley, 1998.

[37] W. Eatherton, G. Varghese, and Z. Dittia. Tree Bitmap: Hardware/-Software IP Lookups with Incremental Updates. *ACM SIGCOMM Computer Communication Review*, 34(2):97–122, 2004.

[38] F. Faerber, A. Kemper, P.-Å. Larson, J. Levandoski, T. Neumann, and A. Pavlo. Main Memory Database Systems. *Foundations and Trends in Databases*, 8(1-2):1–130, 2017.

[39] J. M. Faleiro and D. J. Abadi. Latch-free Synchronization in Database Systems: Silver Bullet or Fool's Gold? In *Proceedings of the Eighth Biennial Conference on Innovative Data Systems Research, CIDR '17*, 2017.

[40] D. E. Ferguson. Bit-Tree: A Data Structure for Fast File Processing. *Communications of the ACM*, 35(6):114–120, 1992.

[41] M. Fomitchev and E. Ruppert. Lock-Free Linked Lists and Skip Lists. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing, PODC '04*, pages 50–59, 2004.

[42] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, 2004.

[43] E. Fredkin. Trie Memory. *Communications of the ACM*, 3(9):490–499, 1960.

[44] P. Frühwirt, M. Huber, M. Mulazzani, and E. R. Weippl. InnoDB Database Forensics. In *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications, AINA '10*, pages 1028–1036, 2010.

[45] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[46] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *Proceedings of the 14th IEEE International Conference on Data Engineering, ICDE '98*, pages 370–379, 1998.

[47] G. Graefe. B-tree indexes, interpolation search, and skew. In *Proceedings of the Second International Workshop on Data Management on New Hardware, DaMoN '06*, page 5–es, 2006.

[48] G. Graefe. A Survey of B-tree Locking Techniques. *ACM Transactions on Database Systems*, 35(3):1–26, 2010.

[49] G. Graefe. Modern B-Tree Techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[50] G. Graefe and P.-A. Larson. B-tree indexes and CPU caches. In *Proceedings of the 17th IEEE International Conference on Data Engineering, ICDE '01*, pages 349–358, 2001.

[51] J. Gray. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, pages 393–481. Springer Berlin Heidelberg, 1978.

[52] J. Gray. Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King, Dec. 2006. *Presented at the Storage Guru Gong Show, Redmond, WA*, URL: `http://jimgray.azurewebsites.net/talks/Flash_is_Good.ppt`.

[53] L. J. Guibas and R. Sedgewick. A Dichromatic Framework for Balanced Trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science, SFCS '78*, pages 8–21, 1978.

[54] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro*, 34(2):6–20, 2014.

[55] Haoyu Song, J. Turner, and J. Lockwood. Shape Shifting Tries for Faster IP Route Lookup. In *Proceedings of the 13th IEEE International Conference on Network Protocols, ICNP '05*, pages 358–367, 2005.

[56] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *Proceedings of the 27th ACM SIGMOD International Conference on Management of Data, SIGMOD '08*, pages 981–992, 2008.

[57] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[58] S. Heinz, J. Zobel, and H. E. Williams. Burst Tries: A Fast, Efficient Data Structure for String Keys. *ACM Transactions on Information Systems*, 20(2):192–223, 2002.

[59] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture, ISCA '93*, pages 289–300, 1993.

[60] A. Hern. Did your Adobe password leak? Now you and 150m others can check, Nov. 2013. The Guardian, `http://www.theguardian.com/technology/2013/nov/07/adobe-password-leak-can-check`, last accessed on 2020-03-23.

[61] J. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum. YAGO2: Exploring and Querying World Knowledge in Time, Space, Context, and Many Languages. In *Proceedings of the 20th International Conference Companion on World Wide Web, WWW '11*, pages 229–232, 2011.

[62] B. R. Iyer and D. Wilhite. Data Compression Support in Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94*, pages 695–704, 1994.

[63] G. J. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, 1988.

[64] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment*, pages 1496–1499, 2008.

[65] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner. Improving in-memory database index performance with Intel Transactional Synchronization Extensions. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture, HPCA '2014*, pages 476–487, 2014.

[66] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 27th IEEE International Conference on Data Engineering, ICDE '11*, pages 195–206, 2011.

[67] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the 29th ACM SIGMOD International Conference on Management of Data, SIGMOD '10*, pages 339–350, 2010.

[68] H. Kimura. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 34th ACM SIGMOD International Conference on Management of Data, SIGMOD '15*, pages 691–706, 2015.

[69] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. KISS-Tree: Smart Latch-free In-memory Indexing on Modern Architectures. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN '12*, pages 16–23, 2012.

[70] A. Kloss-Brandstätter, D. Pacher, S. Schönherr, H. Weissensteiner, R. Binna, G. Specht, and F. Kronenberg. HaploGrep: a fast and reliable algorithm for automatic classification of mitochondrial DNA haplogroups. *Human Mutation*, 32(1):25–32, 2011.

[71] A. Kovács and T. Kis. Partitioning of trees for minimizing height and cardinality. *Information Processing Letters*, 89(4):181–185, 2004.

[72] W. I. Landauer. The Balanced Tree and Its Utilization in Information Retrieval. *IEEE Transactions on Electronic Computers*, 6(EC-12):863–871, 1963.

[73] P. L. Lehman and s. B. Yao. Efficient Locking for Concurrent Operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[74] T. J. Lehman and M. J. Careay. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the 12th International Conference on Very Large Data Bases, VLDB '86*, pages 294–303, 1986.

[75] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE '13*, pages 38–49, 2013.

[76] V. Leis, A. Kemper, and T. Neumann. Exploiting Hardware Transactional Memory in Main-Memory Databases. In *Proceedings of the 30th IEEE International Conference on Data Engineering, ICDE '14*, pages 580–591, 2014.

[77] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The ART of Practical Synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN '16*, pages 1–8, 2016.

[78] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE '13*, pages 302–313, 2013.

[79] D. B. Lomet. Process Structuring, Synchronization, and Recovery Using Atomic Actions. *ACM SIGSOFT Software Engineering Notes*, 2(2):128–137, 1977.

[80] D. B. Lomet. The Evolution of Effective B-tree: Page Organization and Techniques: A Personal Account. *ACM SIGMOD Record*, 30(3):64–69, 2001.

[81] D. Makreshanski, J. J. Levandoski, R. Stutsman, and E. Zurich. To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-Free Indexing. *Proceedings of the VLDB Endowment*, 8(11):1298–1309, 2015.

[82] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the Seventh ACM European Conference on Computer Systems, EuroSys '12*, pages 183–196, 2012.

[83] M. M. Michael. Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing, PODC '02*, pages 21–30, 2002.

[84] D. R. Morrison. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[85] R. Muntz and R. Uzgalis. Dynamic storage allocation for binary search trees in a two-level memory. In *Proceedings of Princeton Conference on Information Sciences and Systems*, volume 4, pages 345–349, 1971.

[86] S. Nilsson and M. Tikkanen. Implementing a Dynamic Compressed Trie. In *Proceedings of the Second International Workshop on Algorithm Engineering, WAE '98*, pages 25–36, 1998.

[87] S. Nilsson and M. Tikkanen. An Experimental Study of Compression Methods for Dynamic Tries. *Algorithmica*, 33(1):19–33, 2002.

[88] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The Case for RAMCloud. *Communications of the ACM*, 54(7):121–130, 2011.

[89] D. Pacher, R. Binna, and G. Specht. Data Locality in Graph Databases through N-Body Simulation. In *Proceedings of the 23rd GI-Workshop Grundlagen von Datenbanken*, pages 85–90, 2011.

[90] D. Pacher, R. Binna, and G. Specht. Graph Stores based on Spatial Computers. In *Proceedings of the Seventh Spatial Computing Workshop, SCW '14*, pages 1–6, 2014.

[91] D. Pacher, R. Binna, and G. Specht. Optimizing Large Knowledge Networks in Spatial Computers. *The Knowledge Engineering Review*, 31(4):367–390, 2016.

[92] H. Plattner and A. Zeier. *In-Memory Data Management: Technology and Applications*. Springer Science & Business Media, 2012.

[93] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference, ATEC '05*, volume 194, pages 196–215, 2005.

[94] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th International Conference on Very Large Data Bases, VLDB '99*, pages 475–486, 1999.

[95] J. Rao and K. A. Ross. Making B+-Trees Cache Conscious in Main Memory. In *Proceedings of the 19th ACM SIGMOD International Conference on Management of Data, SIGMOD '00*, pages 475–486, 2000.

[96] H. Reiser. Reiserfs whitepaper, 2004. `https://web.archive.org/web/20030212224551/http://www.namesys.com/`, last accessed on 2020-03-23.

[97] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage*, 9(3):1–32, 2013.

[98] M. Á. Ruiz-Sánchez, E. W. Biersack, and W. Dabbous. Survey and Taxonomy of IP Address Lookup Algorithms. *IEEE Network*, 15(2):8–23, 2001.

[99] B. Samadi. B-trees in a System with Multiple Users. *Information Processing Letters*, 5(4):107–112, 1976.

[100] B. Schlegel, R. Gemulla, and W. Lehner. k-Ary Search on Modern Processors. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware, DaMoN '09*, pages 52–60, 2009.

[101] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[102] V. Srinivasan and M. J. Carey. Performance of B+ Tree Concurrency Control Algorithms. *VLDB Journal*, 2(4):361–406, 1993.

[103] V. Srinivasan and G. Varghese. Faster IP lookups using controlled prefix expansion. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '98*, pages 1–10, 1998.

[104] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik. One Size Fits All? – Part 2: Benchmarking Results. In *Proceedings of the Third Biennial Conference on Innovative Data Systems Research, CIDR '07*, pages 173–184, 2007.

[105] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB '07*, pages 1150–1160, 2007.

[106] H. R. Strong, G. Markowsky, and A. K. Chandra. Search within a Page. *Journal of the ACM*, 26(3):457–482, 1979.

[107] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference, ATEC '96*, 1996.

[108] SQLite Version 3 Overview. `https://www.sqlite.org/version3.html`, last accessed on 2018-09-03.

[109] Tokyo Cabinet: a modern implementation of DBM. `http://fallabs.com/tokyocabinet/`, last accessed on 2018-09-03.

[110] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles, SOSP '13*, pages 18–32, 2013.

[111] J. Váncza, T. Kis, and A. Kovács. Aggregation - the key to integrating production planning and scheduling. *CIRP Annals*, 53(1):377–380, 2004.

[112] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 37th ACM SIGMOD International Conference on Management of Data, SIGMOD '18*, pages 473–488, 2018.

[113] H. Wedekind. On the Selection of Access Paths in a Data Base System. In *Data Base Management, Proceeding of the IFIP Working Conference*, pages 385–398, 1974.

[114] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[115] Z. Xie, Q. Cai, H. V. Jagadish, B. C. Ooi, and W. F. Wong. Parallelizing Skip Lists for In-Memory Multi-Core Database Systems. In *Proceedings of the 33rd IEEE International Conference on Data Engineering, ICDE '17*, pages 119–122, 2017.

[116] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 35th ACM SIGMOD International Conference on Management of Data, SIGMOD '16*, pages 1567–1581, 2016.

[117] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. SuRF. In *Proceedings of the 37th ACM SIGMOD International Conference on Management of Data, SIGMOD '18*, pages 323–336, 2018.

[118] J. Zhou and K. A. Ross. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the 21st ACM SIGMOD International Conference on Management of Data, SIGMOD '02*, pages 145–156, 2002.