



UNIVERSITÄT PADERBORN

Die Universität der Informationsgesellschaft

Faculty for Computer Science, Electrical Engineering and Mathematics

Department of Computer Science

Research Group Data Science

Master's Thesis

Submitted to the Data Science Research Group

in Partial Fulfilment of the Requirements for the Degree of

Master of Science

Space Reduction of Tentriss Hypertrie with Path Compression

by

BURHAN OTOUR

Thesis Supervisors:

Prof. Dr. Axel-Cyrille Ngonga Ngomo

Prof. Dr. Stefan Böttcher

Paderborn, August 22, 2020

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Ort, Datum

Unterschrift

Abstract. In my thesis, I investigate and implement a space reduction approach for the in-memory indexing data structure Hypertrie.

Contents

1	Introduction	1
2	Preliminaries and Foundations	3
2.1	Pointer Tagging	3
2.1.1	Data Structure Alignment	3
2.1.2	Tagged Pointers	4
3	Related Work	5
4	Space Reduction Approach	7
4.1	Motivation	8
4.2	Basic Concept	9
4.3	Compressed Hypertrie Nodes	11
4.3.1	Features	11
4.3.2	Internal Node Representations	11
4.4	Algorithms	12
4.5	Storage Discussion	14
5	Evaluation & Benchmarking	15
6	Conclusion	17
	Bibliography	18

1

Introduction

Preliminaries and Foundations

2.1 Pointer Tagging

Pointer Tagging is a low-level programming technique that uses the spare low bits in a pointer to encode additional information. Using the Pointer tagging technique, pointer value (initially a memory address before tagging) can hold extra information about the point-to heap object or can be used as a meta-data to further describe the usage of the pointer data. Pointer tagging is mainly enabled because of the way heap objects are situated and accessed on modern computer architectures.

2.1.1 Data Structure Alignment

Data alignment (also referred to as data structure padding) is a way in which heap objects are arranged and accessed by the CPU. CPUs in modern computer architecture (say 64-bit architecture) read data from and write data to memory more efficiently when data is aligned.

On an abstract level, computer memory can be seen as an array of words or bytes, each with its own address. Unlike bytes, the term word has ambiguous meaning. In the context of this work, we are targeting the generic term in the context of CPU architecture. That is, a "processor word" refers to the size of a processor register or memory address register. The term word also refers to the size of CPU instruction, or the size of a pointer depending on the exact CPU architecture. For example, in a 64-bit architecture, the word size (also pointer size) is 64 bits = 8 bytes.

Generally, when a source program is executed, it is loaded into memory and put into a process `p` for execution. All data objects in the program are mapped at certain point in time (during compilation or execution) to a physical memory address [ref: operating system concept]. Let us suppose we have the following snippet written in C language:

```
long *x = new long(123.4); // x = x21DE int* a = new int(123); // a = x21E6 char* c = new char('A'); // c = x21EE
```

According to C language specification, the size of integer value in memory is 4 bytes and size of char value is 1 byte [C spec]. When we execute the previously mentioned statements, however, the compiler (or linker) books 8 bytes of memory to hold the integer value and not 4 bytes as expected. The reason is that, the compiler adds paddings to the heap objects in order to align

them in memory. The same applies to the character value, as shown in figure 1 (on my notebook).

Why data alignment? The CPU can access the memory only in word-sized chunks. So if our data always starts at a word it can be fetched efficiently. If it were to start somewhere in the middle of a word, the CPU will need to wait two or more memory cycles to fetch data from or write data to memory causing an increase in the CPU stall period which results in a significant performance overhead.

many modern compilers implementations handle data alignment in memory automatically, example includes C, C++, Rust, C compilers.

2.1.2 Tagged Pointers

The different cases I used the tagged pointer for. (Tagged)

Use cases list:

Integer Tagged Pointer Illustration Example Figures

Type Tagged Pointer Figure

Keep it for the
being

Related Work

asdasdasd

Space Reduction Approach

This part of the thesis discusses the approach to substantially mitigate the space inefficiency characteristic of Hypertrie. The technique relies mainly on compressing a Hypertrie path with specific characteristics. Worth mentioning that the approach does not neglect the other attempts already realized to minimize Hypertrie memory footprint. In contrast, it can be considered an added feature that further contributes to the space reduction of the overall Hypertrie data structure.

In this chapter, I deliver a motivation to the approach. Afterward, I discuss the new Hypertrie internal nodes' design needed to realize the path compression feature. Finally, algorithms defining the behaviors of the newly designed Hypertrie are also presented.

about where
programming
resides?

ter structure
sit"

4.1 Motivation

Despite its operational efficiency, Hypertrie performance comes not without a trade-off. Since Hypertrie is a special kind of a Trie data structure, it inherits some of the fundamental problems of Tries. One of these problems is the excessive space utilization in a worst-case scenario.

The current design and implementation of Hypertrie, however, mitigates the space inefficiency characteristic in two ways. First, for each tensor dimension mapping in each node, the Hypertrie uses custom hash map data structures instead of arrays or linked lists to store the keys. By using a map, Hypertrie's nodes only stores keys that form prefixes to already existed paths. In contrast, arrays utilization in normal Tries considers the whole alphabet set in each node with many array entries store pointers that refer to null.

cite!

The adoption of maps in Hypertrie also delivers extra performance as looking up keys in a carefully designed map is nearly constant compared to linked list search where it has a linear complexity $O(n)$. The other solution realized by Hypertrie to reduce the overall space requirement is to store equal nodes (Subhypertrie) only once. In this way, Hypertrie achieves a moderate level of compression in practice.

Despite the previously mentioned attempts to minimize the size of Hypertrie, the excessive memory requirement is still a bottleneck. The case can be witnessed when the set of RDF triples needed to be indexed by Hypertrie increases in size with less overlapping between its elements. As a result, many intermediate nodes store map with a single entry for a particular dimension where the entry hosts a space for key and a pointer. This becomes a space redundancy issue when the leaf node referenced by the pointer has one key only.

The purpose of the following approach is to try to reach a more space-efficient Hypertrie.

Continue here

4.2 Basic Concept

The purpose of this section is to give a better intuition on the idea of path compression.

Assuming we want to store the set of RDF triples in listing 4.1, presented in Turtle syntax, in our space-efficient Hypertrie:

Listing 4.1: An example set of RDF triples

```
@prefix rel: <http://www.example.com/schemas/relationship/> .
@prefix ex: <http://www.example.com/schemas/entities/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:Germany rel:capital ex:Berlin .
ex:USA rel:capital ex:Washington_DC
ex:USA rel:political_city ex:Washington_DC
ex:Germany rel:population "82.79e6"^^xsd:integer
```

Tentris do not store the actual values of RDF terms (RTs). Instead, it stores their associated identifiers. For generating identifiers, a bijective function $id : RT \rightarrow N$ is used. For the example of RDF data above, a possible mapping for the terms used is given below:

RT	id
ex:Germany	17
rel:capital	4
ex:Berlin	30
ex:USA	20
ex:Washington_DC	40
rel:political_city	5
rel:population	6
82.79e6 (integer)	35

The Hypertrie will store the triple as shown in Figure 4.1, when the path compression technique is applied. It is straightforward to notice that many keys stored in the second level nodes (depth=2) do not need to be branched further to point to other nodes in the third level. As a result, the tree height is cut down, and a substantial amount of memory is saved by storing objects in-place instead of storing them on the heap.

So, the memory for the pointer to the object on the heap is saved. The same method is applied for the root node where keys for a specific dimension are branched by a *lonely path*, i.e. a key path where each element has a single child element.

about the idea
reserving the
e saving tech-
e like the one
ointing to an al-
y existed node
once

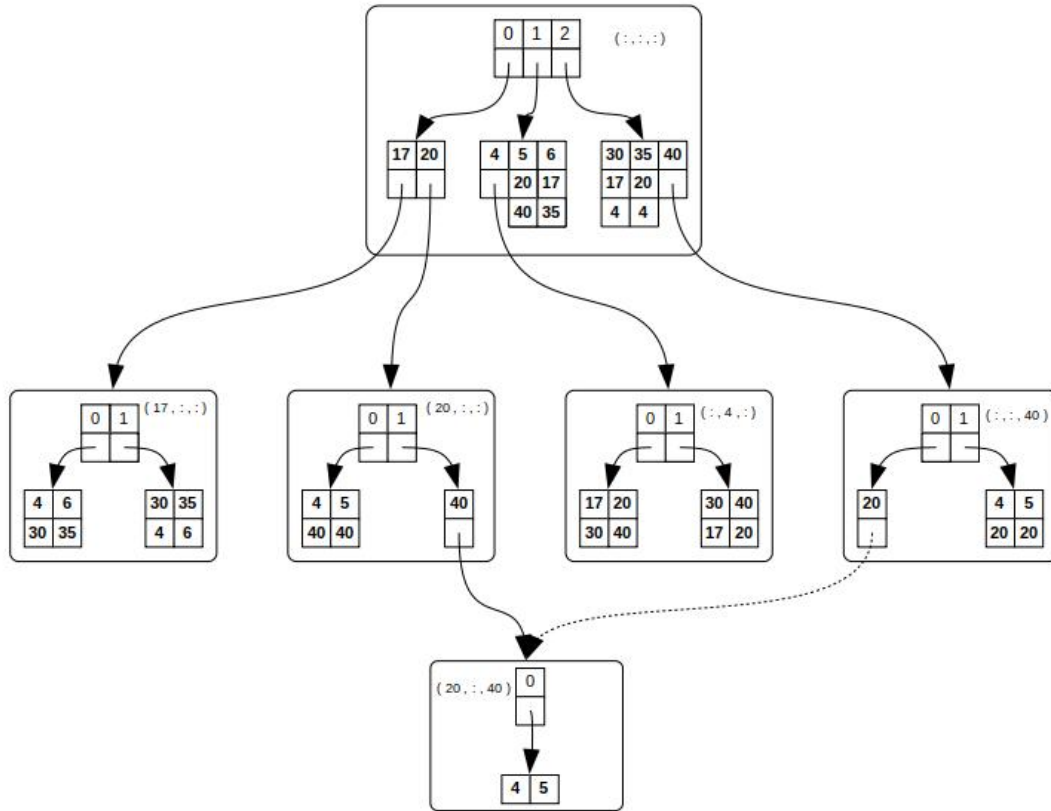


Figure 4.1: Storing RDF in space-efficient Hypertrie

4.3 Compressed Hypertrie Nodes

In order to achieve path compression in Hypertrie, fundamental design changes need to take place. By that, we can enable the node to store the entire key suffix. Concretely, each group of Hypertrie nodes in certain tree depth will have their own internal node representation.

We create **Adaptive Hypertrie nodes**.

Concretely, each node must hold internal **containers** where key suffixes can be stored.

We reduce the space requirement of Hypertrie by collapsing Hypertrie nodes to **static containers**.

From programming point of view, the redesign of Hypertrie nodes' structures is low level. Thanks to C++17 template meta-programming feature, we could separate the compressed nodes realization from the Hypertrie data structure interface. By that, we can still insure a smooth integrity of Hypertrie with other components in Tentris system.

4.3.1 Features

During the evaluation phase, I will prove that the performance of the space efficient Hypertrie is at least as much as the performance of the base (reference) Hypertrie. The compressed Hypertrie is **cache-conscious**. That is the frequently accessed compressed key paths suffixes stored at the root node in array-based containers will increase the probability that those paths resides within cache.

4.3.2 Internal Node Representations

Now I come to the part where the internal structure of space-efficient Hypertrie nodes are discussed. In my approach, it is a requirement to realize the container concept for each node¹. As a result, each inner node should still be able to expand at certain edges to sub Hypertrie nodes while maintaining a compressed key path in its bounded container for other edges.

The compressed key path container implementation varies depending on the node depth. The idea of having different internal representations comes from the fact that, based on the current structure of nodes on depth two, I found that there is no need to add an additional structure that serves as a container for the key path. Instead, I exploit the space dedicated to pointer value existed as a value in the hash table of store the compressed key path.

Since Hypertrie's internal nodes can be either a root node or depth two nodes, we can distinguish two variants of internal node representations:

Depth 3 Node (root node)

In addition to the set of edges (hash tables) $HT_{3,p}$ for each position $p \in P = \{0, 1, 2\}$, the root node also holds another array of hash tables $CommHT_p$ that maps key parts k_j to static arrays arr_j . Each array will serve as the container for the key path prefixed by the associated key part k_j at the corresponding position p as depicted in Fig. 4.2. We call the edges k_j stored in

¹Leaf nodes are not considered.

$CommHT_p$ compressed edges. Clearly, a key part at a particular position p can either represent a compressed or non-compressed edge at a time, so it exists in either HT_p or $CommHT_p$. The remaining key path $k_S = \langle k_1, k_2 \rangle$ associated with each compressed edge k_j holds the key part chain ordered by their presence in the key.

Worth to mention that the key path associated with each compressed edge still represents a $2D$ sparse tensor S that results from slicing tensor T represented by the root node at position p with key part k_j . The resultant tensor S has a single entry $\langle k_1, k_2 \rangle$ that evaluates to 1. As a result, it is important to maintain the order of the elements in the compressed path arr_j as each key $arr[i]$ represents the single edge at position i in S whose child is the other array entry.

Depth 2 Node

Internal nodes with $d = 2$ realize the static container concept associated with compressed edges differently than for the root node. Considering the number of internal nodes, it becomes unfeasible, assigning an extra set of hash tables for each node that serve as key part chains containers.

Talk about hash table capacity

To realize the container concept, we exploit the fact that key suffixes for edges in $d = 2$ node comprise a single key part. Hence, we could reuse the space already booked to store the pointers to child nodes (leafs) to hold the suffixed key part. For the pointer ptr_j associated with the edge k_j to serve the purpose of either pointing to a child node or holding an integer value, we make it a tagged pointer (cf. 2.1).

Consequently, child pointers $ptr_j = (value, tag)$ for edges in depth=2 nodes carry two pieces of information. (1) A *value* where either an address or integer value is held. (2) A *tag* that indicates whether the associated edge k_j is compressed or non-compressed. Figure 4.3 visualizes the structure.

Node Expansion

Virtual Nodes

4.4 Algorithms

This is a text for algorithms

Key Insertion

Key Retrieval

Slicing

Diagonal

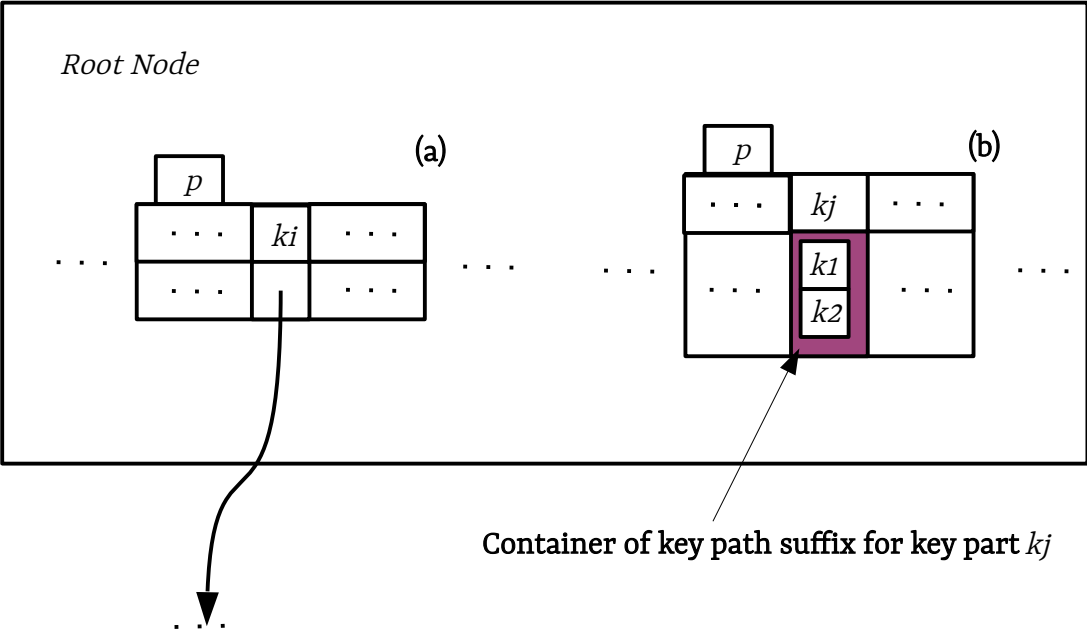


Figure 4.2: Depth 3 Node

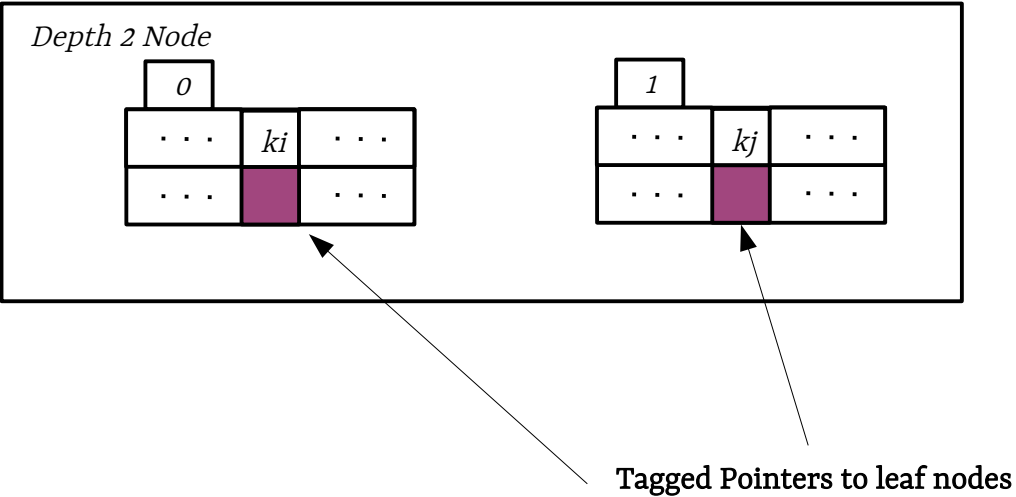


Figure 4.3: Depth 2 Node

4.5 Storage Discussion

Best case is easy. all triples are stored in the root node???

ALEX: Whether we skip that to evaluation

Evaluation & Benchmarking

Conclusion

asdasdasd

Bibliography

- [OPHS16] Tobias Oetiker, Hubert Partl, Irene Hyna, and Elisabeth Schlegl. The Not So Short Introduction To \LaTeX 2 ϵ , 2016. Checked 2017-12-18.
- [The17] The CTAN Team. CTAN Comprehensive \TeX Archive Network, 2017. Checked 2017-12-18.