



**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 19, 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

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Preliminaries and Foundations</b>	<b>3</b>
2.1	Pointer Tagging . . . . .	3
2.1.1	Data Structure Alignment . . . . .	3
2.1.2	Tagged Pointers . . . . .	4
<b>3</b>	<b>Related Work</b>	<b>5</b>
<b>4</b>	<b>Space Reduction Approach</b>	<b>7</b>
4.1	Motivation . . . . .	7
4.2	Compressed Hypertrie Nodes . . . . .	8
4.2.1	Features . . . . .	8
4.2.2	Internal Node Representations . . . . .	8
4.2.3	Node Expansion . . . . .	9
4.3	Algorithms . . . . .	9
4.4	Storage Discussion . . . . .	9
<b>5</b>	<b>Evaluation and Benchmarking</b>	<b>11</b>
<b>6</b>	<b>Conclusion</b>	<b>13</b>
	<b>Bibliography</b>	<b>14</b>





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**

#### **Integer Tagged Pointer**

#### **Object Tagged Pointer**

## 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.

### 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 store 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.

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 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.

define key suffix

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.

Talk about the hypertrie node compressed path container

### 4.2.1 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 key paths suffixes stored at the root node in array-based containers will increase the probability that those paths resides within cache.

In worst case, where no utilization of path compression occurred (all triples keys overlap)

### 4.2.2 Internal Node Representations

Now we come to the part where the internal structure of Hypertrie nodes are discussed. In my approach, it is a requirement to realize the container concept for each node<sup>1</sup>. 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.

discuss the existence of Hypertrie structure

Since Hypertrie has two depths for its internal nodes, we can distinguish two variants of internal node representations:

ALEX: is this sentence scientific accurate?

**Depth 3 Node** There exists a single root node (depth 3) in Hypertrie.

ALEX: Why use tagged pointer in internal nodes space

**Depth 2 Node**

<sup>1</sup>Leaf nodes are not considered.



## CHAPTER 4. SPACE REDUCTION APPROACH

### 4.2.3 Node Expansion

## 4.3 Algorithms

Key Insertion

Key Retrieval

Slicing

Virtual Nodes

Diagonal

## 4.4 Storage Discussion

]



## Evaluation and Benchmarking

asdasdasd



## Conclusion

asdasdasd



# Bibliography



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