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 Fullfilment of the Requirements for the Degree of

Master of Science

Space Reduction of Tentris Hypertrie with Path Compression

by
Burhan Otour

Thesis Supervisors:
Prof. Dr. Axel-Cyrille Ngonga Ngomo
Prof. Dr. Stefan Böttcher

Paderborn, August 25, 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

2 Preliminaries and Foundations

In this part of the work, I setup the foundations for my approach.

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 ambiguate 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.

Figure need to
added

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:

lng *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 padding 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

Some high-level programming languages, for example C++, offer developers a tool set to work with memory. Using such tool set, developers have access to low level memory abstraction. The main building block that enables memory management is the **pointer data type** and its ecosystem. A variable of type pointer holds a memory address of an object stored in the heap. Due to data alignment (cf. 2.1.1), the memory address of any object in the heap memory is always $\alpha \cdot w$ where $w = 8$ (the word size). This implies all addresses held as a pointer value are multiple of 8. A pointer thus can be 8, 16, 24, 109144, etc. But it can not be 7 or 13.

Speaking in binary, example of pointer values are 0b1000 (=8), 0b10000 (=16), 0b11000 (=24), 0b11010101001011000 (=109144). The lower three bits, also called the least significant bits (LSBs), are always zero. So those three bits are basically free to use. We can use them to store a **tag**, which is an integer between 0 (0b000) and 7 (0b111).

Pointer tagging technique allows a dynamic representation of the value based on the tag. Thus, the actual bits payload of the pointer could represents a memory address for some time during the process execution but can later express the binary representation of a `char` value for example, depending on the execution context and after a change in the tag value during run-time.

An approach to implement a tagged pointer is to develop a wrapper object around a pointer type variable. The wrapper can be equipped with adequate behaviours that govern the tag/payload manipulation and retrieval. An example of pointer tagging implementation can be seen in listing ()

the listing

Pointer tagging can be applied in many use cases. In my work, there are a couple of use cases where pointer tagging served perfectly the purpose. Namely: storing integers in pointers and Dynamic de-referencing of void pointers (`void*`).

to check this

Integer Tagged Pointer

Type Tagged Pointer Figure

it for the time
g

3 Related Work

asdasdasd

4

Space Reduction Approach

This part of the thesis discusses the approach to substantially mitigates the space inefficiency characteristic of Hypertrie. The technique relies mainly on compressing a Hypertire 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.

about where
rogramming
resides?

ter structure
sit"

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 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 $id : RT \to N$ 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.

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

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.

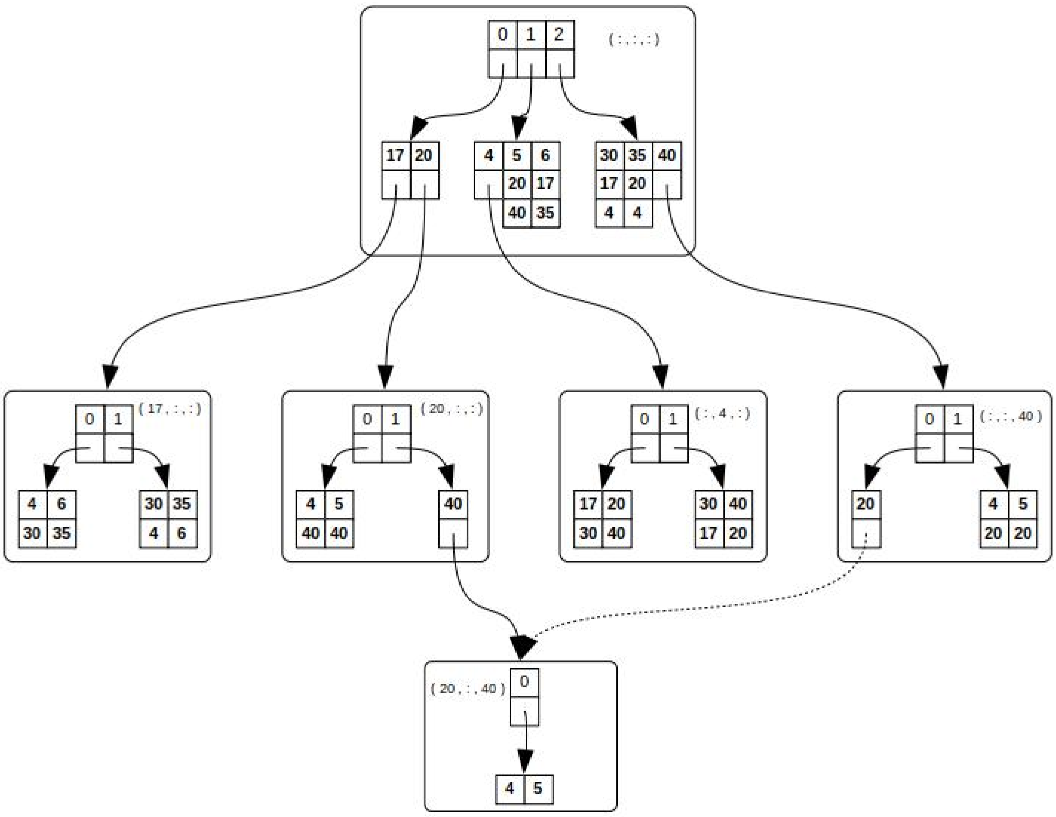

Figure 4.1: Storing RDF in space-efficient Hypertrie

4.3 Compressed Hypertrie Nodes

e key suffix

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 Hyptertrie nodes to **static containerss**.

about the Hy-
ie node com-
ed path con-
r

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 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 node[1] . 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.

ler sentences
less pronoun

ss the exis-
of Hypertire
ture

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

X: is this sen-
scientifically
ate?

Depth 3 Node (root node)

X: Why using
ed pointer for
nal nodes saves
(hash tables
dy have ini-
pace values?
er: Yes)

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 $CommHT_p$ **compressed edges**. Clearly, a key part at a particular position p can either represents a compressed or non-compressed edge at a time, so it exists in either HT_p or $CommHT_p$. The remaining key path $k_S =< k_1, k_2 >$ 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 $< k_1, k_2 >$ that evaluates to 1.

[1]Leaf nodes are not considered.

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 $d = 2$ $Node_2$ 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, pointers to children that corresponds to edges k_j in $Node_2$ are denoted by $ptr_j = (value, tag)$. Such pointers carry two pieces of information. (1) A *value* is the actual payload of bits, which can be viewed as either a memory address or a raw integer value depending on the tag value. (2) A *tag* is the value held in the least significant bits (LSB) indicating whether the associated edge k_j is compressed or non-compressed. Figure 4.3 visualizes the structure.

4.3.2 Node Expansion

Node Expansion text. Node Expansion text.Node Expansion text.Node Expansion text.Node Expansion text.

4.3.3 Virtual Nodes

Due to the fact that some paths in Hypertrie are collapsed to a single node of depth d, we still need to maintain the concept that collapsed path still represents a tensor of the order $d-1$ with a single entry that evaluates to 1. Maintaining the concept becomes very important when we implement Hypertrie programmatically. In particular, when we want to perform slicing in the position of the collapsed path.

A straight forward solution is to return the container representation of the collapsed path (array, int value) from the slicing call. However, that contradicts with slicing function contract defined in the Hypertrie interface which states that slicing always return a child node d. Also, following this approach will add extra code complexity on handling the different types of returned values from slicing. It will also force us to define a method for each returned type from slicing resulting in substantial amount of code redundancy.

A clean solution is to wrap the defined key path containers in a special type of Hypertrie nodes. The wrapper realizes all the methods defined in the Hypertrie interface. As a result, we treat slicing operations uniformally. I call these wrappers **virtual nodes**.

Example of virtual nodes in depth = 3

The same goes with depth = 2 nodes. Where slicing returns a virtual node around the bound key part value or an actual pointer to a leaf node.

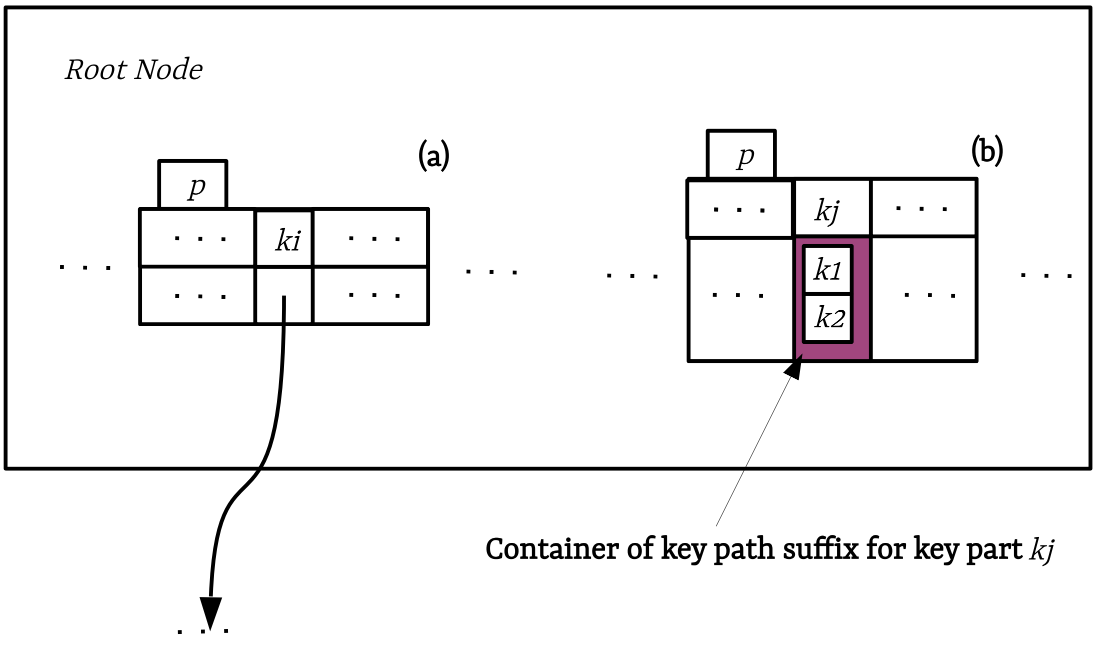

Figure 4.2: Depth 3 Node

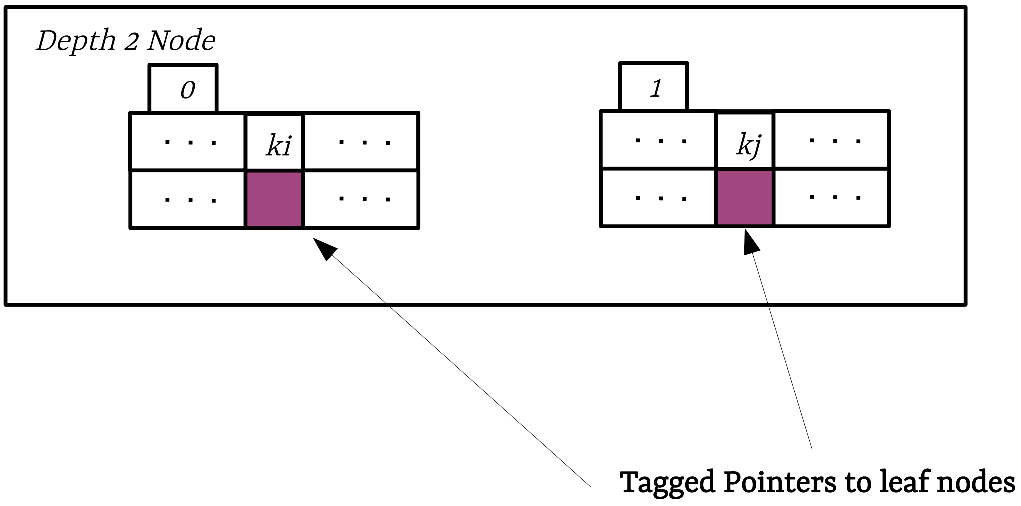

Figure 4.3: Depth 2 Node

4.4 Algorithms

The change in the internal nodes representations in the compressed version of Hypertrie leads to change in how we define its behavior. Hence, the new realization add the logic necessary to consider the compressed paths containers. At the same time, the logic has to have the same contract defined in the original Hypertrie interface. Following that approach will allow us to integrate the new Hypertrie implementation into the Tentris system efficiently.

Next, I list the main algorithms that realize the core behaviors of Hypertrie when the space reduction approach is applied. The great proportion of my work was on expanding the code base of Hypertrie project to adapt the key path compression approach including the implementation of the accompanying algorithms[2].

4.4.1 Key Search

the redefini- of d as I did earlier

a definition

A basic operation in Hypertrie is to check if a given $key \in N^d$ represents a path from a $Node_d$ to a leaf node where d is the node depth. In other words, if key is a non-zero entry in the tensor $T = h^{-1}(Node_d)$. Due to the presence of key path containers in internal nodes, the logic should be expanded to consider the children paths of both the compressed and non-compressed edges in each node.

Some text after.

4.4.2 Key Insertion

4.4.3 Slicing

4.4.4 Diagonal

4.5 Storage Discussion

X: Whether kip that to the ation

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

]

[2]In this section, I list only the main algorithms

```
Input: $node3$: a depth 3 node, $depth$: current node depth, $key$: array of key parts
       $k_i \in K_i$ of size 3
Output: a boolean if a key represents a path in Hypertrie
node3_key_retrieval($node3, depth, key$)
begin
    $p_{min} \leftarrow$ minCardPos($node3$)
    $k_i \leftarrow key[p_{min}]$
    $arr_i \leftarrow HTComm_{p_{min}}[k_i]$
    if $arr_i\ != NULL$ then
        $l \leftarrow 0$
        $c \leftarrow 0$
        while $l < depth$ do
            if $l\ ==\ p_{min}$ then
                $l = l + 1$
            else
                if $arr_i[c]\ !=\ key[l]$ then
                    return $false$
                end
                $l = l + 1$
                $c = c + 1$
            end
        end
        return $true$
    end
    $child_i \leftarrow HT_{p_{min}}[k_i]$
    if $child_i\ !=\ NULL$ then
        $subkey \leftarrow < 0, 0 >$
        $l \leftarrow 0$
        $c \leftarrow 0$
        while $l < depth$ do
            if $l\ ==\ p_{min}$ then
                $l = l + 1$
            else
                $subkey[c] \leftarrow key[l]$
                $l = l + 1$
                $c = c + 1$
            end
        end
        return node2_key_retrieval($child_i, 2, subkey$)
    else
        return $false$
    end
end
```

Algorithm 1: Key Retrieval in the root node

Input: *node*2: a depth 2 node, *depth*: current node depth, *key*: array of key parts $k_i \in K_i$ of size 2 representing a sub key
Output: a boolean if a key represents a path in Hypertrie

```
node2_key_retrieval(node2, depth, key)
begin
    $p_{min} \leftarrow$ minCardPos($node3$)
    $k_i \leftarrow key[p_{min}]$
    $ptr_i \leftarrow HTp_{min}[k_i]$
    if $ptr_i$ == $NULL$ then return false
    $(value_i, tag_i) \leftarrow ptr_i$
    $next\_pos \leftarrow (p_{min} + 1)$ % 2
    if $tag_i == INT\_TAG$ then
        return $value_i == key[next\_pos]$
    else
        $subkey \leftarrow< key[next\_pos] >$
        $child_i \leftarrow$ getPointer($value_i$)
        return node1_key_retrieval($child_i, 1, subkey$)
    end
end
```

Algorithm 2: Key Retrieval in depth 2 Nodes

5 Evaluation & Benchmarking

6 Conclusion

asdasdasd

Bibliography

[OPHS16] Tobias Oetiker, Hubert Partl, Irene Hyna, and Elisabeth Schlegl. The Not So Short Introduction To $\LaTeX\,2_\varepsilon$, 2016. Checked 2017-12-18.

[The17] The CTAN Team. CTAN Comprehensive $\TeX$ Archive Network, 2017. Checked 2017-12-18.