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, September 7, 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

This chapter is logically divided into two main parts. The first part presents the tooling and techniques I utilized to implement aspects of my space-reduction solution. I take into consideration the pointer tagging technique (Sec. 2.2) and the utilization of template-meta programming in modern C++ (Sec. 2.3). In addition, the first part introduces common notations and terminologies that I use through out the work (Sec. 2.1).

In the second part, I present **Tentris** in section 2.4. An RDF triple store based on Tensor arithmetic. I describe its main principle and implementation in a nutshell. Primarily, I focus on its indexing data structure Hypertrie.

need to men-
here I build my
ion on top of
ris?

2.1 Notation and Convention

For a function f, the domain and co-domain of f are denoted by $\text{dom}(f)$ and $\text{codom}(f)$, respectively. The set of natural numbers $\mathbb{N}$ includes zero in this work. Further $\mathbb{N}_n$ with $n \in \mathbb{N}$ is equal to $\{0, 1, ..., n-1\}$. We use angle brackets $< ... >$ to define a tuple t which represents a sequence with fixed order for its elements. The entries $< t_0, t_1, ..., t_{n-1} >$ of a tuple t with length n can be accessed using the square bracket notation (subscript) after the tuple symbol. For example, $t[i] = t_i$ is the tuple t entry at position i. Entries of a tuple t are zero-indexed. The domain of a tuple t, denoted by $\text{dom}(t) = < 0, 1, ..., n-1 >$, is a tuple of t entries' positions.

2.2 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.

2.2.1 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 archi-

tecture) 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.

2.2.2 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. **??**), 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

2.3 Template Meta-programming

In this section I discuss the C++ meta programmin concept.

2.4 Triple Stores and Tentris

Triple stores are special kind of data management systems designed to store RDF triple data. An RDF triple is composed of three elements in order: *a subject*, *a predicate* and *an object*. Each of these elements is called an *RDF Term* (RT). In turn, each RDF term value can either be *an Internationalized Resource Identifier* (IRI), *a blank node*, or *a literal* [**?**]. For example, `<http://example.com/Bob http://example.com/name "Bob">` is an RDF triple instance.

Generally, triple stores provide a standard interface to enable performing queries and other semantic operations on the stored RDF triples through a query language such as SPARQL [**?**]. Triple stores are considered central elements in the "Storage and Querying" phase of the linked data life cycle [**?**].

As there is no standard design guideline for triple stores, different implementations of triple stores co-exist. Each subgroup of these implementations utilizes a category of underlying data structures as well as corresponding algorithms that govern the behaviour. In production, triple stores are used to store up to billions of RDF triples. To that extent, quality factors like efficiency and scalability are considered first-class citizens during the construction of triple stores. And the selection of internal data structures and the behaviour definitions greatly influence the overall system efficiency.

Fuseki, Blazegraph, Virtuoso and RDF-3X are popular implementations of Triple stores. One of the common design characteristics is that they all utilize B+ trees for storing the indices. Other categories of triple stores use 3D Boolean tensors to store and process RDF data. In such systems, each tensor dimension is mapped to a triple data aspect, i.e. subject, predicate or object. Examples of tensor-based triple stores include systems like TensorRDF and BitMat.

Mention the `.nt` file.

2.4.1 Mapping RDF to Tensor

2.4.2 Hypertrie

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.

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.

talk about the i
of preserving th
space saving tec
nique like the o
of pointing to a
ready existed n
only once

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

define key suffix

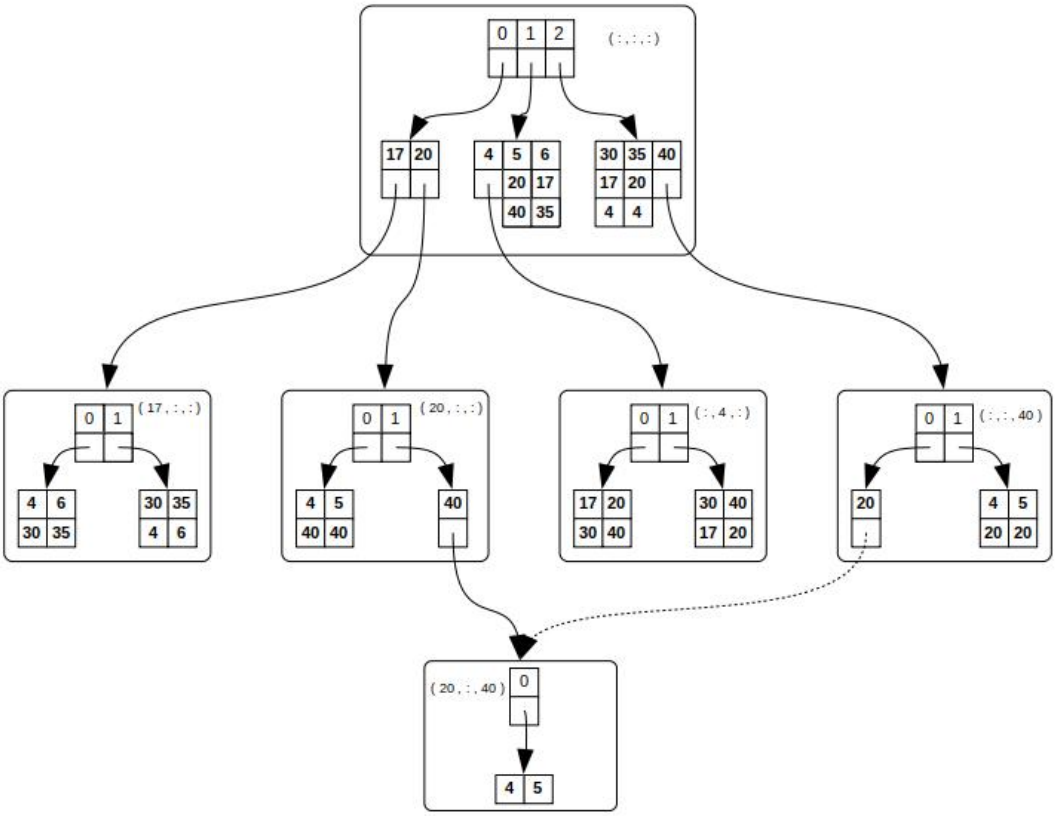


Figure 4.1: Storing RDF in space-efficient Hypertrie

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.

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 pertrie node co pressed path co tainer

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.

smaller sentenc with less pronou

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

discuss the exis tence of Hypert structure

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 $CommHT_p$ **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.

ALEX: is this s tence scientifica accurate?

ALEX: Why us tagged pointer internal nodes s space (hash tab already have ini tial space value Answer: Yes)

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. 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.

[1] Leaf nodes are not considered.

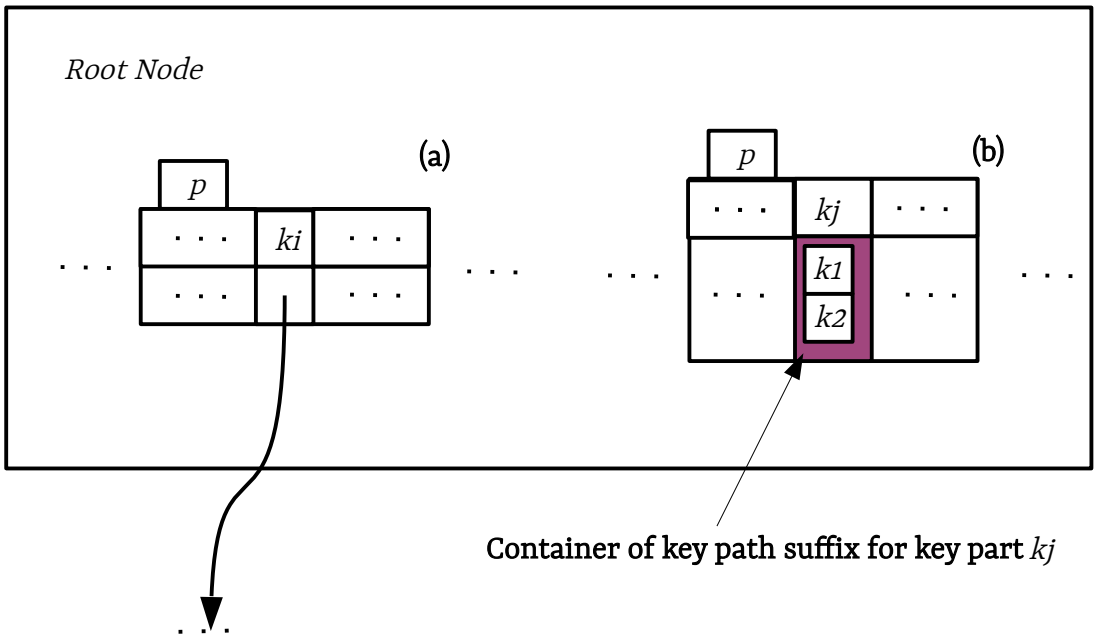


Figure 4.2: Depth 3 Node

Depth 2 Node

about hash capacity

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.

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. 2.2).

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.

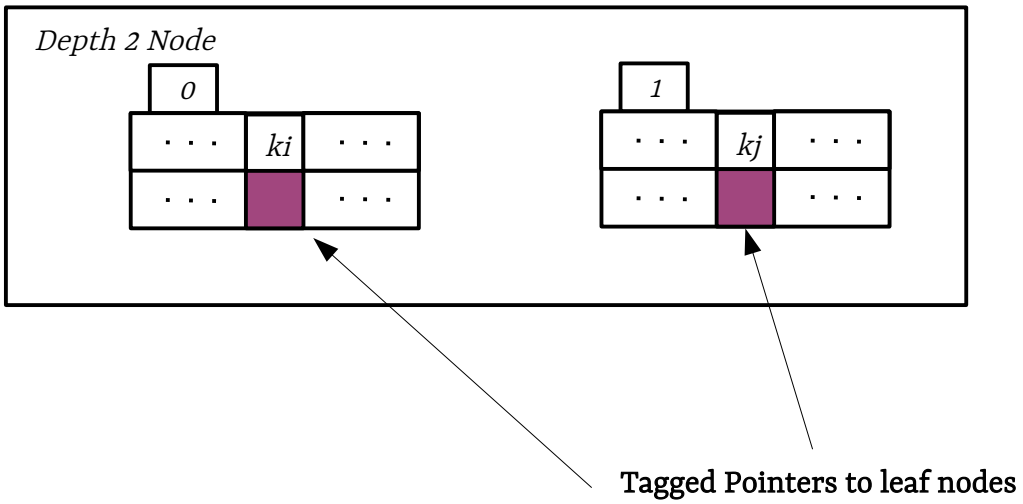


Figure 4.3: Depth 2 Node

4.3.3 Virtual Nodes

Due to the fact that some key paths in Hypertrie are collapsed into 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. Also, following this approach will add extra code complexity in the context of handling the different types of returned values from slicing. It will also force us to define a different behavior for each returned type from slicing resulting in substantial amount of code redundancy.

A clean solution is to encapsulate the defined key path containers in a special type of Hypertrie nodes. Those nodes realize all the methods defined in the Hypertrie interface. As a result, we treat slicing operations uniformally. I call these wrappers **virtual nodes** to discriminate them from the actual Hypertrie nodes which I call **concrete nodes**.

Back to the previous sub section 4.3.1, I defined two types of key chain static containers. Concretely, a static array of length 2 representing the key suffix of compressed edges in the root node. A tagged pointer that bounds the key suffix (a single key part) of compressed edges in internal depth 2 nodes. Hence, we now have two variants of nodes for each depth (except for the root node). Concrete nodes and virtual nodes.

In the original Hypertrie implementation, slicing method gives enough information about

the child node being returned from its execution as shown in listing 4.4. As we conceptually expanded the set of Hypertrie nodes to include virtual nodes, the pointer to a child node of a certain depth returned from a slicing must carry information about the nature of the node being concrete or virtual.

Listing 4.2: Slicing method signituar defined for node where d = *depth* in original Hypertrie

```
#include <array>

template<int depth>
class hypertrie_node {
        ....
        template<typename key_part_type>
        using SliceKey = std::array<std::optional<key_part_type>, depth>;
        ....
        template<int slice_depth>
        hypertrie_node<slice_count>* slice(SliceKey slice_key) {
                ....
        }
        ....
}
```

To accomplish this, we again use tagged pointers. This time, the pointer holds only a memory address of type `void *`. However, the tag here is used to distinguish between the referenced node being concrete or virtual. Accordingly, we can cast the the pointer value to the appropriate type based on the tag value. It becomes the responsibility of the slicing method to create and return that pointer properly. Listing 4.3 shows the structure of the tagged pointer I developed to reference different node types in our space-friendly Hypertrie.

Listing 4.3: Node pointer structure

```
template<typename VirtualNodePtr, typename ConcreteNodePtr, int alignedTo>
class NodePointer {
        private:
                static const intptr_t tagMask = alignedTo - 1;
            static const intptr_t pointerMask = ~tagMask;

        public:
                static constexpr int VIRTUAL = 1;

                static constexpr int CONCRETE = 0;

        protected:
                union {
                        void *asPointer;
                        uintptr_t asBits;
                }

        public:
        inline NodePointer(VirtualNodePtr ptr) {
                // Safer to clear the tag as we set a fresh tag in the setter method
                clearTag();
                asPointer = ptr;
                asBits |= VIRTUAL;
        }

        inline void clearTag() {
                asBits &= pointerMask;
        }
```

```
                inline NodePointer(ConcreteNodePtr ptr) {
                clearTag();
                asPointer = node;
                asBits |= CONCRETE;
        }

        /**
        * @param ptr it is already a tagged pointer
        */
        inline NodePointer(void *ptr) {
                asPointer = ptr;
        }

        inline NodePointer() {
                asPointer = nullptr;
        }
}
```

Listing 4.4: Slicing method signituar defined for node where d = *depth* in the space-friendly Hypertrie

```
#include <array>

template<int depth, bool virtual>
class hypertrie_node {
  ...
  template<typename key_part_type>
  using SliceKey = std::array<std::optional<key_part_type>, depth>;
  ....
  template<int depth>
  using NodePointer =
    TaggedPointer<hypertrie_node<depth, false>, hypertrie_node<depth, true>>;
  ....
  template<int slice_depth>
  NodePointer<slice_depth> slice(SliceKey slice_key) {
        ....
  }
  ....
}
```

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].

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

4.4.1 Key Search

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

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.

]

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

```
1  node3_key_retrieval(node3, depth, key)
2  begin
3      p_min ← minCardPos(node3)
4      k_i ← key[p_min]
5      arr_i ← HTComm_{p_min}[k_i]
6      if arr_i != NULL then
7          l ← 0
8          c ← 0
9          while l < depth do
10             if l == p_min then
11                 l = l + 1
12             else
13                 if arr_i[c] != key[l] then
14                     return false
15                 end
16                 l = l + 1
17                 c = c + 1
18             end
19         end
20         return true
21     end
22     child_i ← HT_{p_min}[k_i]
23     if child_i != NULL then
24         subkey ←< 0, 0 >
25         l ← 0
26         c ← 0
27         while l < depth do
28             if l == p_min then
29                 l = l + 1
30             else
31                 subkey[c] ← key[l]
32                 l = l + 1
33                 c = c + 1
34             end
35         end
36         return node2_key_retrieval(child_i, 2, subkey)
37     else
38         return false
39     end
40 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

```
1  node2_key_retrieval(node2, depth, key)
2  begin
3      p_min ← minCardPos(node3)
4      k_i ← key[p_min]
5      ptr_i ← HTp_min[k_i]
6      if ptr_i == NULL then return false
7      (value_i, tag_i) ← ptr_i
8      next_pos ← (p_min + 1) % 2
9      if tag_i == INT_TAG then
10         return value_i == key[next_pos]
11     else
12         subkey ←< key[next_pos] >
13         child_i ← getPointer(value_i)
14         return node1_key_retrieval(child_i, 1, subkey)
15     end
16 end
```

Algorithm 2: Key Retrieval in depth 2 Nodes

Dataset	#T	#S	#P	#O	Type
SWDF	372 k	32 k	185	96 k	real-world
DBpedia	681 M	40 M	63 k	178 M	real-world
WatDiv	1 G	52 M	186	92 M	synthetic

Table 4.1: Numbers of distinct triples (T), subjects (S), predicates (P) and objects (O) of each dataset. Additionally, Type classifies the datasets as real-world or synthetic.

5 Evaluation

How much data compression does the Hypertrie achieve due to applying the space reduction approach presented earlier? Does the new data structure compromise the overall efficiency, more specifically, the query processing speed? On the one side, this chapter describes the space-friendliness of Hypertrie by presenting the compression ratio achieved w.r.t. the original Hypertrie after the bulk loading with different RDF data sets is done for both Hypertrie variants. On the other side, we have a look at how the space enhanced Tentris can still maintain its proven efficiency. For that, I ran a a series of benchmark experiments on real RDF data and queries.

As this work's effort is mainly toward enhancing the Tentris system in the context of space cost, all the experiments consider only the original Tentris' Hypertrie as the reference for comparison opting out other triple store systems. The experimental setup is described in section 5.1. In the succeeding section 5.2, the results are presented.

5.1 Experimental Setup

5.1.1 Data Sets and Queries

I used three different RDF data sets to assess the performance (load time/ size) of the compressed Hypertrie. Concretely, I used Semantic Web Dog Food SWDF (372K triples) as well as the English DBpedia 2015-10 (681 M triples) as real-world data sets. I also chose the 1 billion-triple synthetic data set from WatDiv (see Table 5.1). All the three data sets were used during the size/load speed assessment. However, only SWDF and DBpedia data set were incorporated into the benchmarks.

5.1.2 Test Environment

I used the server machine "Geiser" provisioned and maintained by the data science research group at Paderborn university to run all the experiments. The server machine has two Xeon E5-2683 v4 CPUs and 512 GB of RAM. Geiser runs Ubuntu 18.04.4 LTS 64-bit on Linux Kernel 4.15.0-112-generic with Python 3.6.9 and OpenJDK Java 11.0.8 installed[1]. The benchmark tool and the Tentris triple stores (compressed/ non-compressed) were installed locally on a single

[1]The server specification may change in the future. However, by the time of the document submission date, the spec mentioned here holds.

Dataset	#T	#S	#P	#O	Type
SWDF	372 k	32 k	185	96 k	real-world
DBpedia	681 M	40 M	63 k	178 M	real-world
WatDiv	1 G	52 M	186	92 M	synthetic

Table 5.1: RDF datasets which were used throughout the evaluation phase. Numbers of distinct triples (T), subjects (S), predicates (P) and objects (O) of each dataset. Additionally, Type classifies the datasets as real-world or synthetic.

server instance to avoid network latency. Data sets held in N-Triples format (`.nt` files) were uploaded to the server and stored on disk.

5.1.3 Bulk Loading

Generally, bulk loading is the activity of loading a large volume of data into a data storage system (database, triple store, etc.) mainly in one call, thus in a relatively small amount of time. There are many use cases in which bulk loading could apply; one example is data import/ export. In this work context, bulk loading is used to load an entire RDF graph into Tentris; thus, it can calculate its memory footprint and the time to load. During the bulk loading experiment, all the three RDF datasets (Sec. 5.1.1) were involved for each variant of the triple stores (compressed, non-compressed). I also expanded the load test to check the space utilization performance of the compressed Hypertrie as the dataset scales. For that, I used generated WatDiv synthetic datasets with increasing scale factors: [10, 100, 1000, 10000]. The bulk loading tests' results were calculated using a set of utility methods inside the Tentris project. Those utilities rely on built-in functions inside C++ language, which interface low-level system services and data to their clients.

Calculating Size

There are many methods to calculate memory utilization of processes or data structures residents. Each method fits a set of use cases and they can vary with accuracy. For the case at hand, I relied on the `proc` virtual file system in the server instance that ran the experiments. Given that, "The `proc` filesystem is a pseudo-filesystem which provides an interface to kernel data structures. It is commonly mounted at `/proc`. Typically, it is mounted automatically by the system, but it can also be mounted manually" [2]. The content of the files inside `proc` has mostely the status of *read-only* as the files are written by the kernel. For each process (with a process id *pid*) currently running in the linux system, there exists a subdirectory `/proc/[`*pid*`]`. Underbeneath each subdirectory `/proc/[`*pid*`]`, there are a set of files and subdirectories disclosing meta-information about process with *pid*. The file `/proc/[`*pid*`]/status` provides status information about the process including various memory consumption information. The information are provided in a human-readable fashion.

After a single bulk load experiment is finished, Tentris access the corresponding status file; i.e. `/proc/slef/status` where the magic symbolic link `self` is automatically evaluated to Tentris' process ID. The field `VmRSS` inside `Status` file provide the current memory utilization of the resident Tentris process including the sizes of resident anonymous memory, resident file mappings and resident shared memory.

[2] https://man7.org/linux/man-pages/man5/proc.5.html

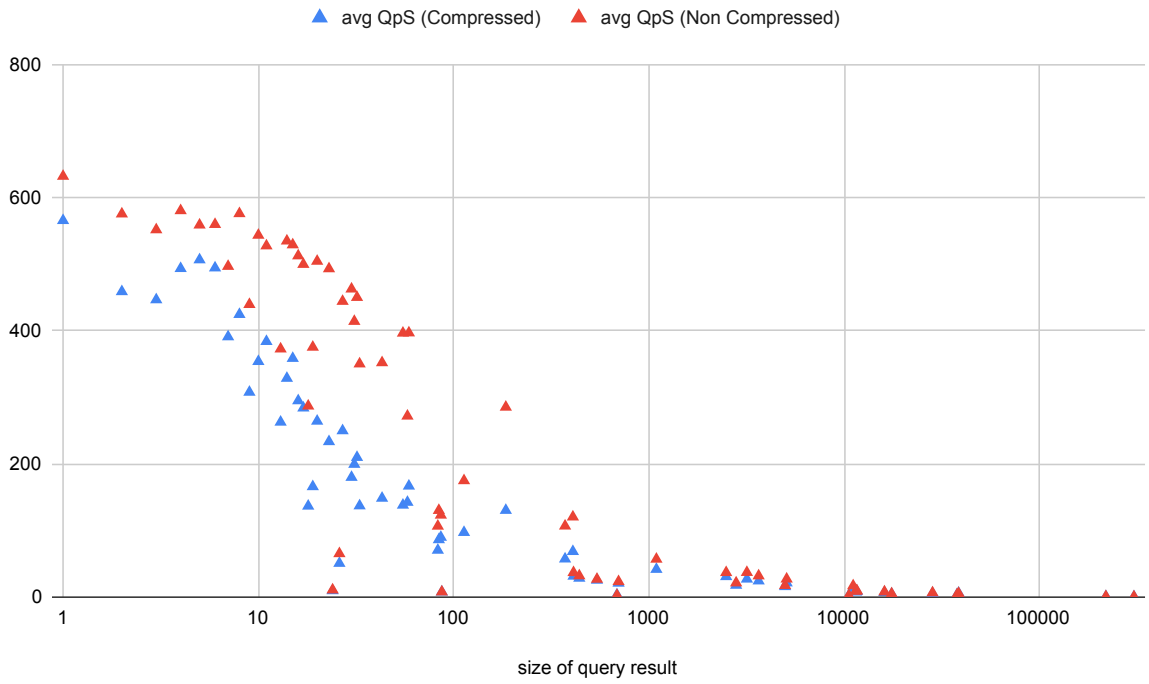
Calculating Load Time

A simple tic-toc approach is followed to calculate the load speed of RDF graph into Tentris. More specifically, two start and end time flags are setup immediately before and after the actual loading of keys. The difference between the two checkpoints is then calculated to capture the load speed. The time checkpoints calculation utilized `steady_clock` instead of `system_clock`. As `system_clock` talks periodically to machine clock to correct itself, it might make minor timing mistakes. As `steady_clock` is independent from the system clock, thus providing more reliable timing insights.

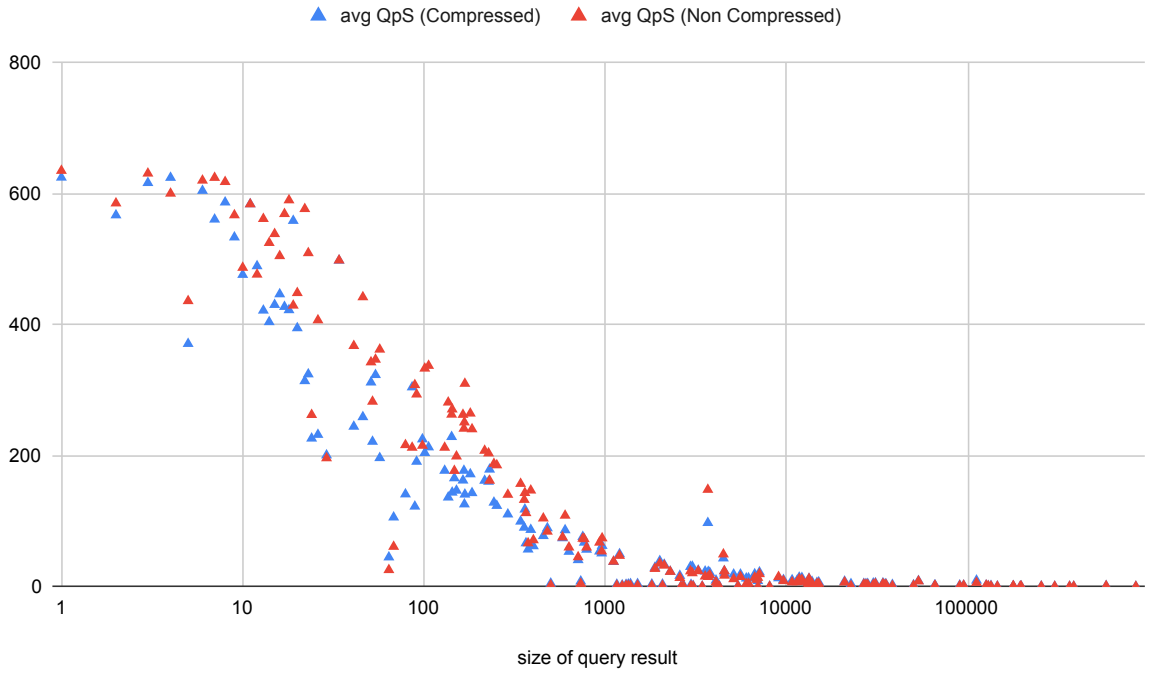
5.1.4 Benchmark Execution

For executing the benchmark, I used the generic SPARQL benchmark execution framework IGUANA v3.0.0-alpha2 [CLSN17]. IGUANA is a benchmark suite for executing benchmarks. It takes a benchmark, namely a data set and a possible list of SPARQL queries/updates, as input. Then it simulates a SPARQL user that pushes a series of queries repeatedly in a stress test scenario to a SPARQL endpoint where the next request is sent immediately after returning the last response. IGUANA can execute both synthetic benchmarks and benchmarks based on real data. As part of its execution, the suite returns information on the respective triple store's different behavioral aspects, such as query processing speed for each query and the query result's size. The framework enables different benchmark execution options and fashions (measure performance of triple stores under updates, parallel user requests, etc ...). As Tentris provides an HTTP-based SPARQL interface, I used the HTTP-based benchmark with one user as a benchmark setup. The suite returned the average response time for each query, and the query result's size to consider later for comparison.

5.2 Results



(a) SWDF



(b) DBpedia

Figure 5.1: Average Queries per Second (avg QpS) by size of the query result considering the two variants of Hypertrie implementations (compressed, non-compressed). Figure 5.1a shows the performance with the SWDF dataset, and figure 5.1b shows the performance with the DBpedia dataset.

6

Conclusion

Parallelizeation Different types of containers Different types of compression methods

Bibliography

[CLSN17] Felix Conrads, Jens Lehmann, Muhammad Saleem, and Axel-Cyrille Ngonga Ngomo. Benchmarking rdf storage solutions with iguana. In *Proceedings of 16th International Semantic Web Conference - Poster & Demos*, 2017.